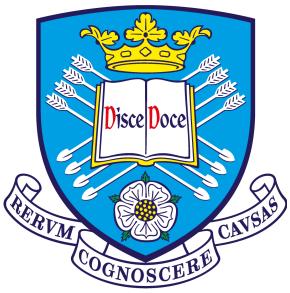
Automated Classification and Repair of Presentation Failures in Responsive Web Pages

Ibrahim Althomali

Supervisor:
Professor Phil McMinn



The University of Sheffield
Faculty of Engineering
Department of Computer Science

MAY 2022

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Abstract

Testing is an important part of software engineering that is employed while developing the layout of a responsively designed web page. This type of design produces a layout that aims to pleasantly responds to the available width within any browser window by rearranging the layout along the way. With a large number of possible screen sizes that a browser may conform to, from mobile phones to large desktop monitors, manually testing the layout for presentation failures can be a time-consuming and error-prone task. With help from the ReDeCheck tool or its re-implementation in Layout DR, the developer can automatically test the web page for five type of presentation failures known as Responsive Layout Failures. Even though ReDeCheck alleviates some of the burdens, the developer must still manually investigate the reported failures and classify them as either a real visible failure that will require a repair or as something that can be ignored. Since the detection phase of the tool uses the underlying structure of the page to infer failures, it can report failures in the structure that are not visible in the rendered page, known as Non-Observable Issues. Finally, after filtering out the real failures the developer must now investigate the root cause and manually repair the layout as fast as possible.

Since any successful software, especially a web page, is continuously updated and improved, there is a never-ending process of testing, investigating the findings of these tests, and finally repairing any issues that are found. In this thesis, I propose and evaluate new ways to further aid the developer, of a responsively designed web page, in this process. This is by (1) automating the classification of three failure types that are known to report non-observable issues, (2) extending it to classify all five type of failures and I reassess the approach on additional subjects, and (3) automating the patch generation process to quickly repair any of the five type of failures. The benefits of my research include a further reduction in the time dedicated to testing for presentation failures, reducing the time needed to repair the failures, and reducing the chance for human-made errors. Thus, increasing the adoption and retention of automated means for testing and repairing the web page.

The research that makes up this thesis concluded that my proposed classification approach achieved a high agreement with the human-made alternative. The findings also showed that for any given failure, the layout can always be repaired automatically. In many cases, up to two alternative patches that successfully remove the failure can be generated using my automated repair technique. In addition, I identified multiple opportunities for future research in this domain.

Dedicated to my mother Muzinah, my father Mohammad, and my whole family.
You are my prosperity.

Acknowledgements

First and foremost, I acknowledge Allah (The proper name of God in Arabic) for his mercy on me, his never-ending sustenance, and generosity that no gratitude or act of worship can satisfy. Bear witness, that I follow Allah's message to his final messenger, Mohammad, the same message carried by all the messengers that came before him "that there is no god but I; therefore worship and serve Me." [92]

After God, the most deserving acknowledgement is to my supervisor without whom this thesis would not be, professor Phil McMinn. Thanks for seeing my potential, recommending this topic, and nurturing the research that makes up my thesis. Then I would like to thank Dr Gregory Kapfhammer for meeting with me on a weekly bases, giving me feedback on my research, and collaborating with us to get the work published. In line, I would like to thank my previous graduate-level advisors and teachers, Dr Mark Grechanik and professor Jianwei Niu, who had a great impact on me and my interest in the area of software testing. I thank them for their inspiration and support during the years I spent learning under them.

My acknowledgements would not be complete without recognizing the impact that my friends and colleagues, in the Software Verification and Testing lab at the University of Sheffield, had on my PhD journey. These are my brothers in arms: Nasser Albunian, Abdullah Alsharif, Islam Elgendy, John Michael Foster, George O'Brien, Benjamin Clegg, and David Paterson. Thanks for sharing your wealth of knowledge, your encouragement, and for making the lab and my life more enjoyable. Moreover, I equally extend the same appreciation for the support and knowledge that I received from my friends outside the lab: Bader Alotaibi, Abdullah Alqahtani, Abdulmonem Alshahrani, and Fahad Algorain. Finally, I also thank Duhaiman Alduhaiman and Hamad Almuqari for their friendship and invaluable training during my time in the industry.

I am delighted to also acknowledge the roots of my happiness during this journey, my sons Mohammad and Abdullah. Furthermore, I share my appreciation for their mother, my wife Arwa, who stood by my side along the way. My sincere gratitude also goes to my brothers and sister, Omar, Khalid, and Maryam for their unyielding support that underpins my life. Finally, I acknowledge and appreciate the financial support that I received from my employer Taif University through the Ministry of Education in Saudi Arabia.

Declaration

I, the author, confirm that the Thesis is my own work. I am aware of the University's Guidance on the Use of Unfair Means (www.sheffield.ac.uk/ssid/unfair-means). This work has not been previously presented for an award at this, or any other, university.

Publications

Various pieces of research presented in this thesis have been previously published in the following peer-reviewed venues:

1. Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. "Automatic Visual Verification of Layout Failures in Responsively Designed Web Pages." In: *International Conference on Software Testing, Verification and Validation (ICST 2019).* [1]

2. Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. "Automated Visual Classification of DOM-based Presentation Failure Reports for Responsive Web Pages." In: *Software Testing, Verification and Reliability (STVR 2021).*

3. Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. "Automated Repair of Responsive Web Page Layouts." In: *International Conference on Software Testing, Verification and Validation (ICST 2022).*

[1] Awarded IEEE distinguished paper award.

Contents

1 Introduction **1**

1.1 Responsive Web Design 2
1.2 Motivating the Research 4
1.3 Thesis Objectives 5
1.4 Thesis Structure and Contributions 6

2 Literature Review **8**

2.1 Software Testing 8
2.2 Software Repair 11
2.2.1 Fault Localization 14
2.2.2 Automated Program Repair 16
2.3 The Web 19
2.3.1 Developing a Web Page 20
2.4 Testing Web Pages 22
2.4.1 Cross-Browser Testing 24
2.4.2 Responsive Design Testing 29
2.4.3 Regression Testing 32
2.4.4 Internationalization Testing 34
2.4.5 General Layout Testing 35
2.5 Repairing Web Pages 36
2.5.1 Repairing Cross-Browser Failures 37
2.5.2 Repairing Internationalization Failures 38
2.5.3 Repairing Mobile-Friendly Issues 41
2.5.4 Generic Layout Repair 44
2.6 Concluding Remarks 46

3 Classifying Non-Observable Issues in Layouts **47**

3.1 Motivating the Research . . . 48
3.2 Detection Prior to Classification . . . 50
3.3 Classifying NOI Failures . . . 52
3.3.1 Summary of Approach . . . 52
3.3.2 Classifying Presentation Failures . . . 52
3.3.3 Identifying the Areas of Concern (AOCs) . . . 53
3.3.4 Analysing the Areas of Concern . . . 55
3.4 Empirical Evaluation . . . 58
3.4.1 Design of Experiments . . . 58
3.4.2 Results of the Experiments . . . 61
3.4.3 Discussion . . . 71
3.5 Concluding Remarks . . . 72

4 Classifying Observable Failures and Reassessing Automated Classification 73
4.1 Motivating the Research . . . 74
4.2 Detection Prior to Classification . . . 75
4.3 Classifying Wrapping and Small-Range Failures . . . 79
4.3.1 Summary of Approach . . . 79
4.3.2 Classifying the Failure Reports . . . 80
4.4 Empirical Evaluation . . . 86
4.4.1 Design of Experiments . . . 87
4.4.2 Results of Experiments . . . 93
4.4.3 Discussion . . . 115
4.5 Concluding Remarks . . . 116

5 Repairing Presentation Failures 119
5.1 Motivating the Research . . . 120
5.2 Detecting Failures . . . 121
5.2.1 Element Collision . . . 122
5.2.2 Element Protrusion . . . 124
5.2.3 Viewport Protrusion . . . 125
5.2.4 Element Wrapping . . . 125
5.2.5 Small-Range . . . 126
5.3 Repairing Failures . . . 127
5.3.1 Summary of Approach . . . 127
5.3.2 DOM Failure Assessment . . . 128

5.3.3 Patch Sourcing and Application . . . 130
5.3.4 Repair Assessment . . . 133
5.4 Empirical Evaluation . . . 135
5.4.1 Design of Experiments . . . 136
5.4.2 Results of Experiments . . . 142
5.4.3 Discussion . . . 151
5.5 Concluding Remarks . . . 152

6 Conclusions and Future Research 153
6.1 Summary of Research . . . 153
6.1.1 Initial Point of Research . . . 154
6.1.2 Classifying Non-Observable Failures (Chapter 3) . . . 154
6.1.3 Classifying Observable Failures (Chapter 4) . . . 155
6.1.4 Reassessing Automated Classifications (Chapter 4) . . . 155
6.1.5 Repairing Presentation Failures (Chapter 5) . . . 156
6.2 Future Research . . . 157
6.2.1 Improving Detection of Failures . . . 157
6.2.2 Improving Classification of Failures . . . 159
6.2.3 Improving Repair of Failures . . . 160
6.3 Final Remarks . . . 162

List of Tables

3.1 Subject web page details . . . 59
3.2 Manual classification . . . 62
3.3 Minimum viewport classifications . . . 63
3.4 Middle viewport classifications . . . 67
3.5 Maximum viewport classifications . . . 68

4.1 Experimental subject web pages. . . . 89
4.2 Verve's results for wrapping failures from the initial subjects. . . . 94
4.3 The manual classifications of failure from subject web pages . . . 95
4.4 Results of classifying small-range failures using horizontal referencing 97
4.5 Results of classifying small-range failures using horizontal-plus-vertical referencing . . . 98
4.6 Results using the minimum viewport and the additional set of web pages 100
4.7 Results using the middle viewport and the additional set of web pages 101
4.8 Results using the maximum viewport and the additional set of web pages 103
4.9 Verve's results for wrapping failures from the additional subjects. . . 104
4.10 Small-range failures with the additional set and horizontal referencing 107
4.11 Small-range failures with the additional set and horizontal-plus-vertical referencing . . . 108
4.12 Small-range failures with the fault-injected set and horizontal referencing . . . 110
4.13 Small-range failures with the fault-injected additional set and horizontal-plus-vertical referencing . . . 111
4.14 Prospective vs. retrospective thresholds . . . 112

5.1 Subject web page details . . . 137
5.2 Automatic wider source-viewport repair results . . . 142
5.3 Failure-free source-viewport . . . 144

5.4 Technique introduced failures 145
5.5 Human study results 146

List of Figures

1.1 Responsive design example 3
1.2 Devices using the Internet 4

2.1 An example HTML file to illustrate it's alignment graph. 27
2.2 An example web page to illustrate an RLG. 30

3.1 Observability of element overlap 49
3.2 Failure wireframe examples 51
3.3 Classification process 53
3.4 Areas of concern 54
3.5 Misclassified by Viser at minimum 65
3.6 Change classification at middle 66
3.7 Disagreement of classifications to agreement 69
3.8 Timing box plots 71

4.1 Viewport protrusion failure snapshots 76
4.2 Element wrapping failure snapshots 77
4.3 Wireframe of a wrapping and small-range failures 78
4.4 Comparing Verve to manual classification 80
4.5 Area of concern for wrapping 81
4.6 The usage of the *horizontal referencing* system to determine AOCs . 83
4.7 The usage of the *vertical referencing* system to determine AOCs . . . 84
4.8 Initial set of subjects with misclassified wrappings 96
4.9 Initial set of subjects with misclassified small-range 99
4.10 Example misclassification of an out of flow element 102
4.11 Additional set of subjects with misclassified wrappings 105
4.12 Example of fault-injected small-range failure 109
4.13 Box plots of Verve's runtime 113

4.14 Example of fault-injected small-range failure 115
4.15 Example subject containing video 117

5.1 Layout DR tool overview 121
5.2 Failure and repair example snapshots 123
5.3 Repair process 132
5.4 Repair snapshots vs source-viewport snapshots 135
5.5 Human study web page 139
5.6 Human study MarkText failure 33 146
5.7 Human study Bower failure 1 147
5.8 Human study ConsumerReports failure 8 148
5.9 Runtime duration 150

Glossary

Snapshot An image capturing part-of or all the graphical output of a program in execution that is ready for rendering on the screen.

1

Introduction

The value and practicality of using a web page are known to billions of people. Individuals, private organizations, and government bodies all rely on them to be the default gateway to communicate and interact with the world. A web page is used to display information, expedite services, and facilitate endless entertainment. Given the important status of a web page, the developers should organize the page in an aesthetically pleasing way and strive to make the content easily accessible. Thus, the presentation of the page should make the content easier to understand and use.

The simplest form of a web page is automatically organized and presented to the visitor using the default style of the browser. Although this basic page will display information, it is not enough. The developer must explicitly add styling directives to colour, shape, and position the content appropriately for a real-world use case. This involves the use of up to three different languages that markup the content, style the page, and make it an interactive experience. Although there is some complexity involved, the developer can make the page look desirable and meet visitors' expectations. This complexity increases when the developer takes into consideration how the page will be presented on devices with different screen sizes.

Since devices with varying screen sizes are capable of accessing the page, the challenge is to present the content in a way that does not require the visitor to scroll the page horizontally regardless of the screen size. This should improve the experience of using the web page because it is simpler to scroll vertically for more content instead of scrolling in both directions. Moreover, the text and graphics should be rendered appropriately as well. The user should not have to zoom in and out of the page to properly view the content. Boiled down, the content presented to the visitor should be based on the available width of the browser. To the point, the page should respond to a smaller screen size of mobile phones differently than for a large screen attached to a desktop.

Historically, a web page was developed for larger screen monitors but with the release of the iPhone in 2007, developers had to take into consideration the increased traffic coming from these smaller devices [90]. The first solution developers adopted early on was to design and maintain two separate web pages, one for mobile devices and the other for larger screen sizes [70]. To achieve this, the traffic coming from mobile devices can be automatically diverted to a page designed for mobile phones. Alternatively, the developer can provide a link for visitors to manually click on to

go to the mobile version of the page. Since the release of the iPhone, this solution has become impractical due to the release of many more devices with varying screen sizes. Nevertheless, a better solution called Responsive Web Design (RWD) has been adopted since then.

1.1 Responsive Web Design

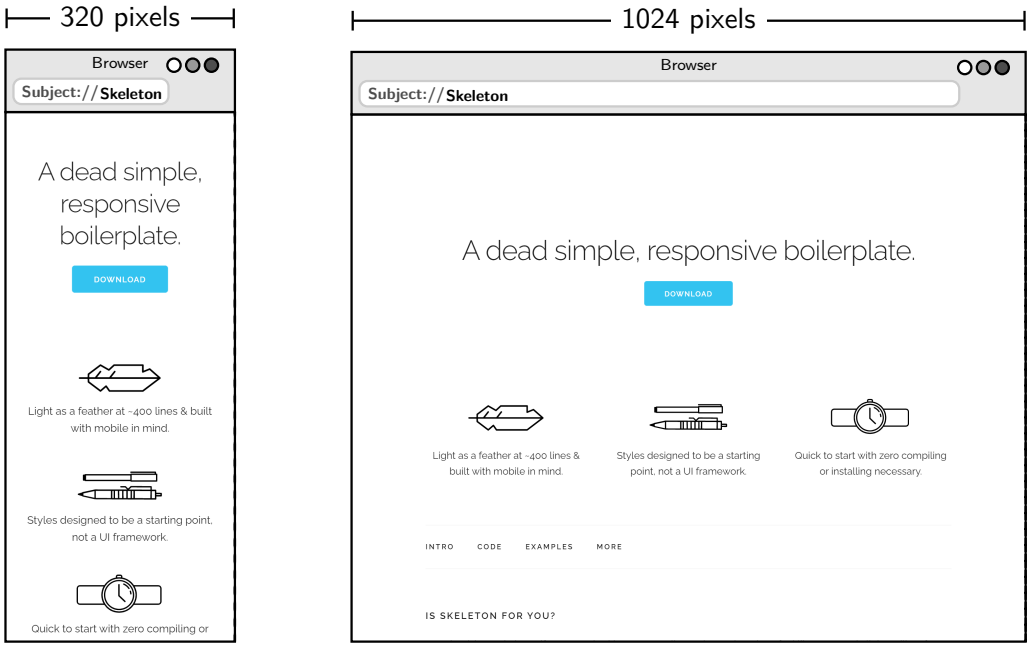
As the technologies behind a web page continue to advance, developers are able to create better web pages. Once the foundational technology became available, responsive web design was coined by Ethan Marcotte in 2010 [69]. This design is able to produce pages that are suitable for presentation on any device. Thus, the page is able to scale and rearrange the content depending on the available display space. This arrangement of the content is referred to as the *layout* of the page. A responsively designed web page can be pleasantly presented on any devices available in the market today and on any other one released in the future.

The key feature enabling responsive design is a styling directive called media queries [70, 90]. These are *media* rules that allow a developer to ask the browser if the size available for display meets some criteria. Furthermore, the rule embodies more styling directives that are automatically triggered when the browser answers yes to the display criteria. To make this rule work in a responsive manner, the developer starts with a default layout based on the expected traffic. For example, the default presentation of the page can for mobile devices if the developer expects the majority of traffic to come from smaller devices. Then, media queries are added to change the layout for larger devices. There can be multiple rules essentially acting as breakpoints that rearrange the layout to better utilize the space available.

The second ingredient of responsive web design is flexibility [90]. To successfully make a page responsive, the elements holding the content of the page will need to scale in between the breakpoints introduced using media queries to rearrange the elements. More precisely, the flexibility should be based on the available horizontal display space within the browser. This means that elements should be styled with flexible units which may extend to text size. Moreover, images as individual elements or as part of the background are also expected to conform or change at certain breakpoints.

Many frameworks are available to assist the developer in creating a responsive web page. These include Bootstrap [13], Foundation [118], and Skeleton [110] just to name a few. Figure 1.1 showcases the Skeleton framework changing the layout in response to the available horizontal space. In part (a), a browser of a mobile phone that is 320 pixels wide visits the Skeleton page. Using any device of this size, the elements vertically align on top of each other due to the lack of horizontal space. In contrast, a browser window of a computer that is 1024 pixels wide has plenty of horizontal space and thus the elements can be positioned next to each other as seen in part (b) of the figure. For a more content-rich layout, the page will need more breakpoints between 320 and 1024 pixels and beyond this range.

Whether the developer is using a framework or a custom-made design, there is a chance that the page will be presented to the visitor in an undesirable way that is contrary to expectations and contrary to the directives set by the developer.



s (a) Mobile phone. (b) The layout for a desktop computers or a laptop.

Figure 1.1: Presents an example of a responsively designed page using two devices, the mobile phone in (a) and the desktop in (b). These images are from the Skeleton CSS framework for responsive design at `http://getskeleton.com/`.

This is known as a *presentation failure* [66]. As with any other piece of software, a responsively designed web page must be tested to see how it will be presented using different devices. Instead of purchasing every device available on the market, the layout of the page can be tested through browser window resizing or using browser built-in tools [90]. By manually resizing the browser window, the developer can investigate how the layout conforms to different window sizes. Alternatively, a browser's built-in feature for inspecting a responsive layout can be used to manually test the responsiveness of the page. They provide additional helpful features like specifying the exact size of the canvas where the browser will render the layout, referred to as the *viewport* size. The viewport size can also be set to known device sizes which can make testing easier. This approach is superior to browser window resizing because the viewport can be explicitly set and because it allows the developer to, optionally, see the breakpoints of the layout on a ruler.

An alternative to manually testing a responsive page is to use the state of the art tool ReDeCheck [124]. This tool can automatically sift through different viewport widths in order to build a graph-based model of the page. This graph encodes the relative position of elements from all the viewports tested by the tool. Using this model and multiple specialized algorithms, the tool can automatically recognize an anomalous position of elements over certain viewports and report five different type of presentation failures specific to responsive web pages called *Responsive Layout Failures* (RLFs).

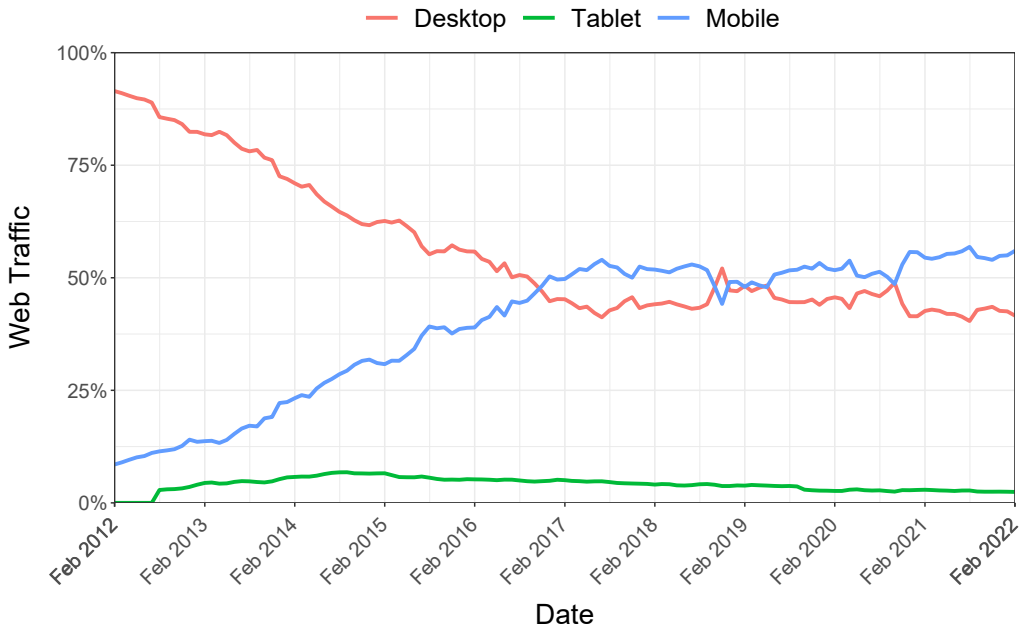


Figure 1.2: Presents the distribution of web traffic among desktops (including laptops), tablets, and mobile devices. From `https://gs.statcounter.com/`.

1.2 Motivating the Research

When the traffic started to come from mobile devices, the value of a mobile-friendly web page was immediately recognized. Visitors not only expected to see a different version of the page made for the smaller mobile device, one study revealed that the majority of participants would not purchase if a mobile-friendly version is not available [36, 46]. Today, over half the web traffic is generated from mobile devices when compared to larger devices [33]. To illustrate the change over the past ten years, Figure 1.2 presents the percentage of web traffic broken down to desktops, tablets, and mobile devices. This data was gathered using 2 million globally positioned sites based on page views, not unique visitors. It is also worthy to note, that the “desktop" figures include laptop devices. This data suggests that there is a real need for pages to handle different devices. Furthermore, a further breakdown based on screen resolution would only magnify this effect [104].

Responsive web design can meet the demand for web pages that conform to any device or screen resolution if programmed correctly. Naturally, producing a device-friendly web page is of little to no use if it fails to present correctly. Therefore, testing the responsiveness of the page is very critical. Moreover, for web pages that frequently update the content, frameworks, libraries, or the style of the web page, testing is a continuous activity. Therefore, automation is the key.

Although the ReDeCheck tool can automatically report failures that are observable in the presentation of a page, it can also capture structural failures that are non-observable in the rendered page. The cause of this problem is that ReDeCheck does not use any visual information in its model of the page. Instead, it uses an interface provided by the browser to determine the visibility and the position of all elements in the page. To overcome this issue, the developer has to read all the

reports generated by the tool and investigate each one to prioritise the ones worthy of attention and repair from the trivial problems. This lack of automation not only undermines the usefulness of the tool, but it can be a time consuming, error-prone, and largely subjective task.

Even with the aid of automated testing and further automation of failure classification, the developer must still repair the presentation failures. Without the repair, testing is essentially useless. It is well established that the longer it takes to identify a failure in software, the higher the cost of repair will be [5]. This is due to the complexity associated with localizing the fault, correcting it, and distributing the fix once the software is in production. With some degree of similarity, a live web page is essentially software in production. Although a presentation failure can be limited to some viewports within a specific browser, the complexity of repair remains the same. The modification required to repair the page may be as simple as updating a single style property or as complex as a redesign of the page [128]. Regardless, automating the repair can either save on the cost of repairing the failure, save the credibility of the web page while the failure is being manually localized and repaired, or both.

The initial point for the research of this thesis begins after the ReDeCheck tool automatically detected the presentation failures of a web page. Thus, the focus of all the research is solely on responsively designed web pages. Two components motivated the research of this thesis. The first is that the ReDeCheck tool does not automatically classify an observable failure that is worthy of repair. The second is that the developer can further benefit from automating the repair of the detected responsive layout failures.

1.3 Thesis Objectives

The goal of the techniques developed and the experiments held for this thesis is to improve the process of testing a responsibly designed layout and to automate the repair of any problems found during the testing process. The intention is, to aid in the development and maintenance of a responsively designed web page by adding more automation. Given a set of presentation failures automatically detected in a responsive layout, the first objective of this thesis is to help the developer prioritize the important failure, worthy of attention and repair, by analysing the layout graphically to automatically classify the reported failures. The second objective of this thesis is to automatically produce repairs that the developer can use to efficiently triage the reported failures. Minimally, these patches should buy the developer valuable time needed to diagnose the problem and create a customized repair. Therefore, the two main objectives were...

1. Develop a technique to automate the classification of the five types of presentation failures found in responsively designed web pages.

2. Develop a technique to automate the repair of the five types of presentation failures found in responsively designed web pages.

1.4 Thesis Structure and Contributions

In this chapter, I have introduced responsive web design which is the core criteria for all the subjects investigated in this thesis. Furthermore, I identified the post automated detection problems that motivated the research of this thesis, these were the classification and repair of presentation failures. The content of the chapters to follow are described next.

Chapter 2: Literature Review – presents a review of the literature on the automated means to detect presentation failures and investigates the automated means of repairing these failures. The chapter begins with software testing and program repair in general and then looks at the different types of web pages and how they are developed. Then, the topic is narrowed down to the testing and repair of presentation failure in web pages.

Chapter 3: Classifying Non-observable Issues in Layouts – investigates the usage of knowledge from multiple graphical layers of the layout to classify non-observable issues. A graphical layer is removed by changing the opacity of certain elements thus revealing what the elements introduced into the canvas. This approach targeted three type of responsive layout failures that were associated with non-observable issues. The key contributions of this chapter are:

1. A technique to automatically classify reports of non-observable issues.
2. An empirical study comparing manual and my automated approaches using 20 web pages which demonstrated that non-observable issues can be classified automatically and effectively.

Chapter 4: Classifying Observable Failures and Reassessing Automated Classification – investigates how to automatically classify the other two failure types associated with only observable failures. For the first of these two failure types, I extended the graphical layers approach used in Chapter 3 to classify it. The second failure type required that I develop a new classification approach. For this, I developed a colour histogram-based comparison approach that uses specialized dissection of images from the layout with pre-determined thresholds to reach a classification. In this chapter, I also employed additional subject web pages to reassess both of my automated classification approaches covering all five failure types. The key contributions of this chapter are:

1. New algorithms to automatically classify two more failures types reported by ReDeCheck.
2. An empirical evaluation comparing the automated classifications to the manual ones for the two new failure types. For the evaluation, I used the same pool of 25 web pages that I drew from the subjects of the previous chapter. Demonstrating that it is possible to automatically and effectively classify the two new failure types.
3. An empirical reassessment of the automated classification of all five failure types over 20 new subject web pages. Demonstrating that the findings from Chapter 3 and Chapter 4 do extend well onto new subject web pages.

Chapter 5: Repairing Presentation Failures – presents an approach to automatically generate up to two CSS patches that are able to remove any of the five type of failures from a problematic layout. The technique relies on borrowing a neighbouring layout from a viewport that does not have the failure in order to override the layout of the problematic viewport. To verify the success of the patches, the problematic viewport is automatically checked to ensure that it is free from that failure. The key contributions of this chapter are:

1. A technique to automatically create up to two patches that repair each presentation failure.
2. An empirical study to evaluate the effectiveness and efficiency of my automated repair strategy using 31 web pages. Demonstrating that any reported failure can be automatically repaired.
3. A human study to evaluate alternative repairs against the original subject with the failure. Demonstrating that humans generally prefer one of the automated repair options over the original page with the failure.

Chapter 6: Conclusions and Future Research – concludes this thesis and presents a summary of the techniques developed and evaluated in each chapter, discloses the known limitations, and provides opportunities for future research.

2

Literature Review

As the standards and the underlying technology of the web continue to evolve, more literature is published to automatically detect and repair the defects that arise in the layout of a web page. These defects can cause the page to be less functional, less informative, or less pleasing to the eye. As a result, it may decrease the credibility [96] of the information and services presented by the page and may reduce a visitor's loyalty [31]. In this chapter, I will review of the current state of knowledge on testing for defects in the presentation of a web page and how they are automatically repaired.

At the start of my review, in the next section, I will be going over how a piece of software is generally tested for defects. Then in Section 2.2, I will be going over how to repair the defects that are detected in software. Then in Section 2.3, I will be describing what a web page is and how it is developed. This is followed by Section 2.4 where I describe how the layout of a web page can be tested for defects. Then in Section 2.5, I will describe how to repair these presentational defects that are detected in the layout of a web page are repaired. Finally, I give my concluding remarks in Section 2.6.

2.1 Software Testing

The testing phase of any software is part of a bigger process referred to as software *verification* and *validation* [111]. Both of these processes check that the software meets the outlined specifications and that the people paying for the software get the functionality that they expect. These two terms were famously described by Boehm [12] as two separate questions. First, verification entails asking if the product is being developed right? In other words, it ensures that the software meets the functional and non-functional requirements already written down. Meanwhile, validation of the software entails asking if the right product is being developed? Thus, it goes beyond what is written down and attempts to meet the customer's real expectations. This is a necessary step because what is written down may not be what the customer really wants.

Along with software testing, this umbrella process, verification and validation, also includes software inspection and reviews. These additional processes look into

the system requirements, design models, as well as the source code. What is distinct about the inspection and review process is that the software does not need to be executed to verify it. More importantly, software inspection has three advantages over testing. First, because inspection is a static process, a single inspection can reveal multiple errors in the system. In contrast, an error can mask other errors during testing. Secondly, an incomplete system can be verified without incurring the additional cost of scaffolding code that would be needed to run the incomplete system for testing. Finally, not only can an inspection reveal defects, it can identify quality concerns that relate to maintainability, portability, and compliance with standards.

The benefits of inspecting software are well known as early as the 1986 paper by Fagan [75]. He reported that up to 60% of errors can be caught by informally inspecting the software [111]. However, software inspection is not a replacement for software testing. It is not an ideal process for discovering timing issues, performance problems, and problems that arise due to unplanned interaction between subsystems. Therefore, testing for defects is the primary way the industry evaluates software that is under development [5].

In practice, testing for defects is a mixture of both manual and automated processes. When manually testing software, the tester executes the code using input data that is crafted by a human and compares the output to what the human expected. Here, a human's wisdom serves as the *test oracle* that is able to judge the correctness of an input-to-output mapping. This type of testing is best suited for difficult to describe or unknown criteria like discovering unwanted side effects in the system. Automating these tests by encoding the oracle into a program that can be executed on demand is a more beneficial approach when a criterion is well defined. By removing the manual labour, the automated tests can be used during the development of the software and easily reused whenever the system is updated without requiring a human the second time. Moreover, there also exists a body of work that investigates partially or entirely removing the human from the testing process. Automating the test oracle can be done implicitly by making assumptions about the system or explicitly by deriving it from documentation or formal specifications [9].

There are three main keywords used to describe defective software which are *failure*, *error*, and *fault* [5]. A software failure, is a defect in the behaviour of the software that is observable externally. In whole, this thesis deals with failures that are observable in the presentation of a web page. On the other hand, a software error is an incorrect assignment to the variables in memory that are used by the software, referred to as the program state. An error is observable internally and may or may not manifest externally as a failure. Finally, a fault is a design mistake in the software introduced by a human. Thus, a fault is the root cause of errors and failures. Another term used frequently and informally to mean a fault, error, or failure is a software *bug*. Meanwhile, the process of searching for the root cause of the defect, the fault, is known as *debugging*.

A consideration to be made during the development of the software is the testing stopping criteria. In other words, how much of the code needs to be exercised or how much of the input space needs to be explored before the software is considered of good quality. This is called the *coverage criteria* of testing [5]. This criteria must be practically achievable. Therefore, it is critical to realize that even for a

small program that takes only three integer variables as input, it is impossible to test all possible input. Thus the real goal is to determine how to best explore the input space with minimal overlap. As a result, this should reduce the cost of testing and improve the quality of the software since the number of tests is not the direct measure of quality.

The coverage criteria of testing can be based on one of the four abstract categories: graphs, logic expressions, syntax descriptions, and the input domain. The most common of the criteria are graphs and more specifically a control flow graph of the source code [5]. Using this model, a graph is constructed based on the execution of the code blocks in the source code. A code block is a contiguous set of instructions that are all executed once the first instruction is executed. These blocks form the nodes of the graph while how the program control is passed between the blocks form the edges of the graph. During testing, the development team can decide how best to cover this graph during testing. For example, they can decide that the test suite must reach every node in the graph or that all edges are executed or that certain paths are taken in the graph. Although the coverage criteria are a vital part of testing, the details of these criteria are not of direct concern to this thesis.

Even though different criteria-based designs for testing exist, a human-based design is still an important factor in the quality of the software. Using domain-based knowledge, a human can capture parts of the system that an automated approach may miss. Therefore, including human-based test designs is important when testing the extremes of the software, *stress testing*. This type of testing explores the input using large and small values, boundary values, invalid values, or generally untypical input. Where the human has an advantage is when using domain-specific knowledge and crafting inputs that an automated approach would miss.

While the software is being developed, testing can be broken down into levels based on functionality. At a micro level, *unit testing* evaluates the smallest grouping of related and contiguous program instructions that are called units. At one level above, *module testing* evaluates the smallest collection of units that are assembled in some abstraction (*i.e.*, class or package) or file. At this level, the whole module is tested in isolation but the interactions between the units that make the module are also tested. One level above is *integration testing* which assumes that modules are working correctly. At this level, the interfaces between modules are evaluated to ensure different modules communicate as expected on both sides. At a macro level, *system testing* evaluates the fully assembled system as a whole against the specifications. Although there is an overlap at each level, system testing evaluates all modules together including the ones adopted off the shelf.

After the software is fully developed comes another evaluation of the software called *acceptance testing*. This type of testing aims to evaluate if the completed software meets the original requirements and does what the customer wanted. This should not be confused to mean that inspection and analysis of customer requirements should be delayed until the software is completed. In fact, the longer a bug goes unnoticed, the harder it is to debug and the more expensive it is to repair. This relationship between time and cost is referred to as the cost-to-change curve [5]. As the gap increases between the time the fault is introduced into the system and the time it is discovered and repaired, so does the cost at an exponential rate. Thus, a problem in the requirements that are written down in non-executable format can

be a great liability if discovered only during acceptance testing.

As software development evolved, demand for a faster turnaround from ideas to artefacts grew. Meanwhile, the liability of mistakes in documentation became less tolerated. From this evolution came *agile* software development. Agile methods do away with the bulk of documentation to favour executable artefacts. To pull this trade-off successfully, one agile method known as *Test-Driven Development* (TDD) uses executable tests as the main way to define the behaviour of the system before it is implemented. Thus the goal of TDD is to implement code that allows a test to pass. Although this is a very beneficial approach that puts testing first, it changes the traditional role of testing from evaluating to defining the software [5]. This change can negatively impact the quality of the software if it is not considered in advanced. Since the tests of TDD are biased toward the normal behaviour of the user and the software, they may miss the testing of abnormal system behaviour.

Regardless of the methodology used to develop the software, the set of tests used to evaluate the code should grow as the system grows. This test set is referred to as the *test suite*. An important role of the test suite is to evaluate the software as code is added, modified, or removed. This is known as *regression testing*. This is especially important when using TDD. The benefits of the test suite reach beyond the original release of the software, it is also repurposed to assess that the system still achieves its original functionality with any future releases.

There is also another level of testing that targets the test suite, thus testing the quality of the tests. This type of testing is referred to as program-based *mutation testing*. It starts by purposefully injecting a fault into the program to produce a mutated version of the program. This is done by changing the valid syntax of the language in the program into another valid syntax. For example, a program with the mathematical statement `a=x-y` can be changed to `a=x+y` in the mutated version. To judge the adequacy of the test suite, the same tests are executed on multiple mutant programs. The more faults that are uncovered in the mutated versions of the program using the test suite, the better it is considered to be. Since tests compare the input and output, they are limited to uncovering faults that propagate into the output as a failure.

An fundamental principle of testing is that testing the software cannot prove that it is free from failures. Instead, testing is done to show the absence of faults, errors, and failures that were considered during testing. Overall, the goal of testing is to reduce the risk of producing low-quality software. An even better goal for testing is to improve the quality as measured in reliability, safety, maintainability, security, and efficiency. More importantly, all this emphasis and effort put into software testing is worthless if the bugs discovered are not repaired.

2.2 Software Repair

With a bug identified in the program, comes the important task of fixing the program. While the testing and debugging process can be methodological or ad-hoc, the actual repair requires good consideration. This is to avoid introducing regression problems and to maintain overall quality as measured in reliability, safety, maintainability, security, efficiency, and readability. Put elegantly by Butcher [19], “There’s

more to a good fix than just making the software behave correctly."

If the bug was found without using the test suite, all tests that executed the statement with the fault in the code are passing. On the other hand, a new test case could be the reason for detecting the bug. Regardless of how it was found, the first step to properly repairing the failure is to know the current state of the test suite. As in, the developer must identify all the passing and all the non-passing tests. Then, the developer must modify the existing tests or add new ones so that the test suite contains at least one non-passing test due to the bug. This is done to show that a bug exists and that it can be reproduced using a test case. This is especially important if TDD is used as the method of development. More importantly, the main job of these non-passing tests is to verify that a change in the program successfully repaired the bug. Moreover, the changes made to the program must not introduce any regression issues. This can be measured by verifying that the previously passing tests do not change over to the set of non-passing tests.

There are two ways of resolving a failure, by repairing the underlying fault or by making the failure go away [19]. The easier of the two is usually not to over investigate the source of the problem and do what it takes to suppress the failure. This is not necessarily an ill-intentioned move to make because there are multiple powers at play. First, there is time pressure to resolve the bugs in order to satisfy the business bottom line (*i.e.*, meet key performance indicators or please internal or external people) or to move to another part of the project that requires implementation. Furthermore, sometimes the root cause is known but suppressing the failure is the easier and safer choice given the time available. If the cause extends to multiple modules of the code or is in the architecture of software, repairing the root cause might take too long or come at a high risk of introducing compatibility problems with older versions. Then it might be a good idea to put it off until enough time is available to fix the root cause of the problem.

The second way to resolve a bug is to analyse the root cause and fix it. A thorough analysis is very important because if the real cause is known, similar failures can be prevented. Even if the root cause in the code is specific to a small number of instructions and it will really repair the fault, it may not be the real cause. To understand the real cause, the developer must investigate how the code ever worked while it contained the failure and investigate why it was not detected any earlier. These questions may uncover bigger bugs that need to be addressed like a security vulnerability. Otherwise, it may also lead to improvements in the development process by adopting better testing coverage or better oversight in design.

Sometimes to repair the real cause, the style and readability of some part of the system needs to be updated. If multiple bugs arise when multiple developers interact with the same unit, module, or subsystem, this part of the code may require *refactoring*. Essentially, refactoring improves the quality of the code without changing its behaviour. By doing so, developers can better understand the code and prevent similar failures from happening. It is important to note that repairing a bug by modifying the code to behave differently should not be done at the same time as refactoring. Even more importantly, refactoring without test cases is not refactoring, it is simply hacking [19]. Refactoring requires the presence of tests in order to verify that the behaviour remains the same after the improvements to the quality of the code are completed.

Regardless of how good the development process is or how experienced the developers are, bugs are to be expected. A clear process must be in place to manage the tracking of known bugs, classifying the severity of the bug, and how a repair is finally accepted. First, there should be a database in place that allows for bugs to be reported and tracked during development and after the software is released. Assigning a severity level of a bug and the repair priority can be tricky. As the database grows, newer bug reports will be classified as more severe than they really are just to put them ahead of the queue for repair [19]. One way to mitigate this is to have it independently classified and prioritized by someone other than the reporter. Finally, to accept a repair, a good strategy is to have the changes reviewed and signed off by another developer.

Estimating the time required to repair a failure is not very difficult once the failure is identified. On the other hand, estimating the time needed to debug the problem is practically impossible to do. This is an important realization if the "no broken windows" policy is used during development [119]. This policy is derived from a behavioural observation made on properties like a car or building [131] due to a sense of abandonment. It was noticed that a property with no windows broken can survive some time without being vandalized but as soon as one window is broken, more participants joined in on the vandalization of the property in a very short amount of time. Of course, developers are hopefully not concerned with crime encouraging more crime but rather low-quality code or broken code that encourages more problems down the road. Therefore, the "no broken window" policy is a good idea but it will come with the added unknown cost of debugging and repair. Nevertheless, one way to get a good sense of the time needed to debug and repair is by statistically analysing the time it took to repair older failures [19].

In the real world, there will be broken glasses. But if the bug database is not stable and continues to grow there are two ways to quickly tackle the problem which are either through real repairs or suppression based repairs. To really repair the failures, a special group can meet and find the most severe and dedicate people and time to repair them immediately. Another way of really repairing the failures is to *bug blitz* the out-of-control database [19]. The goal here is to repair as many bugs as possible with all hands on deck. Using this strategy, the number of bugs repaired is more important than fixing the more severe bugs. Alternatively, the broken glass can be resolved by boarding up the windows. This can be done by entirely removing the segment of code thought to be associated with the failure and scheduling for its re-implementation. Another form of window boarding is to control as much as possible of the problematic segment of the code by introducing an interface around it so that it behaves more appropriately until a proper fix can be made.

The main principle of this section has been that the debugging and repair processes are not trivial and automation can play an important role. Due to the time constraint involved in debugging the failure, suppression of the failure is used in practice as a fix to buy more time. If automated methods are able to really repair or suppress the failure, it will surely help the development team manage the bugs database. One type of automation that aids the developer in the debugging process is *fault localization*.

2.2.1 Fault Localization

Whether a failure is identified while testing the program or unintentionally discovered while using the program, now comes the process of *fault localization*. Essentially, this is the act of considering all the instructions that make up the program in order to narrow them down to a single faulty instruction that caused the failure. Certainly, there is always a chance that multiple faults are responsible for the program's failure. The processing of debugging in this situation is officially known as *Multiple Fault Localization* (MFL) [137]. But a more general classification of the existing fault localizing methods is split into *tradition* and *advanced* techniques [129].

The traditional techniques for localizing a fault are intuitive and include logging, assertions, breakpoints, and profiling. In program logging, manually inserted print instructions are planted in the program to output certain variables in the program. By reviewing the logs, the problem is narrowed down to a segment of the code. Assertions are also used to guard against an incorrect program state. With the assertions added to the program and the state of the assertion falsified, the program terminates. The breakpoints (conditional or not) pause the program in certain states for evaluation and more debugging. Finally, profiling measures the speed performance or memory usage for specialized debugging like for optimization issues. It can help narrow down the code to segments that are performing poorly or causing a memory leak.

More advanced techniques of fault localization are a must for large scale software where traditional methods would not be as useful. These can be subclassified into at least seven main categories [129]. These categories can be based on program slicing, program spectrum, statistical power, program state, machine learning, data mining, and based on different types of modelling. Moreover, these categories are not mutually exclusive and thus can work in combination for improved results. Next, I will briefly describe each of these categories.

Program slicing is about removing parts of the code and keeping only the statements that are associated with a *slicing criterion*. Based on the original definition made by Weiser [130], this criterion is a subset of variables in the program associated with a statement of interest in the program. Thus, slicing aims to exclude the statements that are irrelevant to the outcome of the variables in the subset. Therefore, reducing the search domain needed to find the fault associated with the identified variables. This reduction can be achieved without running the program through a process called *static slicing*. Alternatively, the reduction can also benefit from execution traces of the program and use it to slice the program down in a process called *dynamic slicing*. In static slicing, all the statements that could possibly affect the statement associated with the variables in question are kept in the search domain. This is because it cannot predict the runtime values of certain variables. Alternatively in dynamic slicing, the values are known due to information from runtime. This knowledge allows dynamic slicing to further reduce the statements to the actual path taken in the program. Noteworthy, the output of both types of program slicing approaches limits the number of statements that need to be investigated but does not help prioritize one statement over the other.

The spectrum-based fault localization uses information from the execution path of test cases to determine suspicious segments of the code [112]. The suspicious-

ness of a code segment is determined based on statistical power. Information like which statement or branches were covered by the test cases along with their outcome (non-passing or passing) can provide good statistical insight into the code. Many other parameters can be used to calculate the suspiciousness of the code like complete paths executed, loop-free paths executed, and counting the number of variable definition-usage pairs. Using this method, the output is ranked and the developer should prioritize investigating each statement by its rank.

Another type of fault localization is through statistical debugging. Basically, this is a dynamic analysis that examines the program behaviour during its execution by sampling points in the program. For example, at some point in the program, an instrumented predicate could monitor that an index of an array is less than its length. Predicates can also monitor which branches are taken (`false` or `true`) or the return value from a function. By monitoring these predicates, parts of the program can be isolated for further debugging. Although many runs need to take place for better statistical power, it does not need to monitor all traces of the program. To get the number of runs needed, the instrumented predicates should be light enough to include in the production code released to the users. This type of localization is especially helpful in separating the effects of different bugs and identifying for each bug the associated predicate [55]. The output of this method does rank the suspiciousness of code segments for the developer.

The program state can also be used to localize a failure. As a reminder, a program state is simply the values of variables that make up the program at a particular point in execution. One way to localize faults using this approach is to compare the states of different versions of the same program. Another approach is to change the value of a variable in order to judge if it is responsible for a failure. Alternatively, the states of the program for passing tests can be compared to the states of failing tests. The output of using state-based localization does not explicitly rank the segments of code identified as potentially faulty.

As is the case with many other tasks, machine learning and data mining approaches can be employed to localize faults. Machine learning builds a model based on training data that is used later to evaluate decisions or make predictions. For example, Wong and Qi [133] proposed a back propagation neural network for fault localization. They used the coverage data and the result of each test to train the neural network for it to learn the relationship between the coverage data and the result of the test. Meanwhile, data mining helps analyse a large amount of data to discover new knowledge or patterns. This approach is useful in evaluating the execution traces of a program since it produces too much data. Using data mining, one can ask it to identify the pattern of executed statements that lead to failures [132]. The output from applying machine learning or data mining does rank the suspiciousness of code segments.

Different models can also be used to localize failures. Although a perfect oracle model is ideal, models can also be derived from the code of a program that is known to contain failures. The model-based debugging is achieved in three main steps [71]. First, the code of the program is automatically compiled into a logical model or a constraint satisfaction problem. Second, the model is diagnosed. Third, the result of diagnosing the model is mapped back to a location in the code. An example of this model-based debugging is to model the dependency between statements in the

program whether statically or dynamically. Alternatively to these dependency-based models, value-based models that represent data flow can also be used to localize faults but are more computationally intensive [132]. The output of using model-based fault localization does not explicitly rank the potentially faulty segments.

Many of the fault localization techniques simplify the problem of debugging by assuming that a single fault is the cause of failure [132]. Contrary to this assumption, there was a study based on real-world projects that concluded, that it is often the case that a failure is triggered by multiple faults that are spread out in a large software system [43]. Furthermore, another study by Lucia et al. [58] concluded that faults are not really localized in the program. They found that 67% of severe faults are not associated with a single statement and 42% of severe faults are not localized to a single method. This realization gave rise to many publications in the area of multiple fault localization.

There are three ways to localize multiple faults [137], *one-bug-at-a-time* (OBA), *parallel debugging*, and *multiple-bugs-at-a-time* (MBA). In the OBA approach, a developer iteratively debugs and repairs each fault until all failures are resolved. Using OBA, each suspected statement in the program can be ranked and repaired in order of suspiciousness. In parallel debugging, the faults are split into fault-focused clusters depending on their suspiciousness score and by profiling the execution of test cases. As a result, more than one fault can be repaired simultaneously by multiple developers. Finally, the MBA approach aims to tackle most or all the faults in a single debugging iteration. This strategy can improve the fault detection rate and reduce debugging time. Furthermore, the computational cost of clustering in the alternative parallel technique is saved using the MBA method.

There are additional caveats to fault localization associated with the test cases and the test suite used, as discussed by Wong et al. [132]. In their survey, they noted that localization techniques that use multiple failing and multiple passing test cases are superior to the techniques that use only one passing and one failing test case. Some techniques assume, for effectiveness, that there are multiple test cases from each class (non-passing and passing) available which is not always the case. Nevertheless, using only one representative test case of each class can help reach a better-detailed root cause. Another related issue is that of test coverage. A test suite with low coverage may hinder the fault localization ability of many techniques. Moreover, some research also adds that using the entire test suite is not ideal for fault localization and thus either reduction or prioritization of test cases are in order. Along the same line, the sequence and number of test cases executed are important if changes in the order or number of test cases executed results in a difference in the outcome of the test cases. Finally, although fault localization techniques do not explicitly localize omission faults, they may raise suspicion of other code and states that are implicitly caused by the missing code.

2.2.2 Automated Program Repair

The journey of problems and solutions that lead to the topic of automated program repair started with emphasizing the need to test the program for failures. Once a failure is revealed, the debugging process attempts to localize one or more faults that are causing the failure. Then, automation is expected to repair the program

by modifying the program source code in a way that satisfies a correctness criterion. In automated program repair, the program is considered to be corrected once the test cases that were once non-pass due to the fault are now passing after the code is modified.

There is plenty of published literature about automated program repair. To get a sense of attention this topic received, one living review of automated program repair by Monperrus [80] accounted for 175 references in the year 2018 and 367 references in the year 2021. A definition of automated program repair was provided in a recent paper, by Goues et al. [39], which gave an overview of the state-of-the-art techniques in automated program repair. They defined the repair process as an implicit search over the space of changes to the source code. In their paper, they broke down the automated program repair techniques into two high-level categories, *heuristic-based* repairs and *constraint-based* repairs. Within these categories, when a machine learning technique is applied, they referred to it as *learning-aided* repair. Next, I will describe each of these three types of automated repair.

The heuristics-based method of program repair iterates over a search space of syntactical changes to the program using a generate-and-test methodology. In each iteration, the Abstract Syntax Tree (AST), which represents the parsed source code using a tree-like structure, is modified to produce a patched version of the code. Even though a fault is localized and known, many possible mutants of the program need to be naively made and checked if heuristics are not employed. To overcome this problem, the edits to the program can be limited to insertion, replacement, or deletion per statement or higher-level grouping of statements called a block. The inserted and replaced code can also be derived from other code located at a different part of the program.

Goues et al. [39] suggested that this trust in other parts of the same program is based on the *plastic surgery* [44] and the *competent programmer* [32] hypotheses. The first hypothesis uses plastic surgery as an example indicating that the solution to the problem can come from another part of the body. Meanwhile, the second hypothesis suggests that a programmer is competent and thus produces a program that is very near to being semantically correct. In other words, the search for the semantically correct syntax is not far from the current syntax. Overall, both of these theories suggest that code elsewhere in the program is correct and can be used to patch the fault or that the search for a proper repair is not far off.

Automated program repair techniques vary in how they explore for new patches [39]. One approach is to use genetic programming heuristics that change the code in the direction of the solution. This guidance is done by using a fitness function. For example, the number of test cases that pass can be used as an indicator of a good direction in the search space. Underneath the hood, different mutation strategies based on heuristics can provide additional effectiveness. Moreover, some techniques randomly sample solutions but restrict the depth of edits to only one for efficiency.

After a solution is generated, the next step is to check if the fault is repaired and that no new faults were introduced. To do this, the new version of the program is validated by counting the number of passing tests. Therefore, the test suite is executed every time a solution is generated to validate its correctness or to judge how close it is to being correct or being wrong in order to guide the next phase

of generation. If the search space is very big, this can come at a high cost on performance. To overcome this limitation [39], techniques either reduce the number of test cases, sample test cases, or prioritize the execution of test cases that are more likely to fail first.

Aside from the heuristics-based repair approaches, the second way to automatically repair a program is by creating a *repair constraint* for the patch to satisfy [39]. These repair constraints are typically derived using symbolic execution. This type of execution analyses the program assuming symbolic values for input rather than real values in order to cause parts of the program to execute. The solution to this repair constraint can be achieved through constraint solving or search-based techniques. Thus, the distinction is in formulating the repair constraint and not the how it is solved.

The final distinction between alternative automated program repair techniques is whether they are learning-enabled or not. These learning-based techniques apply learning in one of three ways depending on when the learning is incorporated into the repair process [39]. The first way, as done by Long and Rinard [57], is to learn a model of the correct code from a corpus of successful patches made by humans available in open-source repositories. This approach relies on a fundamental principle or hypothesis that correct code from all types of software share universally correct properties once the syntactical differences are abstracted away.

Similarly, the second way learning is incorporated into the repair process is by sourcing templates that can be used for a code transformation by inferring them from patches made by humans. For example, the work of Long et al. [56] infers AST-to-AST transformation templates based on how the human-made patch changed the faulty code into the corrected code. In turn, the template is used to transform new faulty code into correct code. Another learning-based repair strategy is used in fixing *common programming errors* that arise due to the developer's lack of experience in a particular programming language, as done by Gupta [40]. For example, the trained model can fix missing closing brackets or braces, incompatible operators, and missing declarations. In this strategy, the compiler of the language is used as an oracle to validate the patch before suggesting the patch to the user.

When building or judging an automated program repair approach, the attributes that should be considered are *scalability*, *repairability*, and *quality* of the repairs [73]. The scalability attribute is important because it judges if the repair approach can be applied to a sizeable real-world program. Then, the repairability attribute helps judge if the technique can repair a significant amount of failures and a broad spectrum of failure types. Finally, a good quality repair makes less change to the original program, reduces the amount of deleted code, and is more likely to be accepted by a human developer. Another good quality measure is to verify that the generated repairs are functionally-equivalent to a human-made repair instead of judging quality by simply passing the required test cases.

There are two interesting factors of automated program repair techniques that may limit their effectiveness. This is the correctness criteria used to evaluate the repairs and the original program size that is undergoing a repair. The majority of work in the area uses the test suite as the main measure of correctness [39]. This may result in an over-fitting problem because the generated repair aims to simply

pass the test suite. Two extreme examples, are the deleting of an entire segment of code or changing variables in the program to constants just to pass the the test suite. There are alternative criteria that have been used in the literature to measure correctness including human judgment, crowdsourcing evaluation, comparing the generated repair to a developer-made patch, and measuring the performance of the repaired program on a given workload. The size of the program may also be an enabling or hindering factor for some repair approaches that create a patch from the same program based on the plastic surgery theory. One limitation of taking code from elsewhere in the program is that a small program may not have sufficient patching resources to fix the problem. Thus, these approaches should be reserved for repairing larger programs.

Now that I have reviewed software testing, fault localization, and approaches to automate the repair of programs in general, the next sections narrow down the scope to the literature surrounding the topic of web pages. Even though the general topics covered thus far are important, the research problems addressed in this thesis are limited to the Web domain. More specifically, it is the front-end graphical interface of web pages that is most relevant to this thesis.

2.3 The Web

The Web that pervasively entered our lives was invented by Sir Tim Berners-Lee in 1989. He intended to develop an information-sharing system that makes it easier for his colleagues to share information across the Internet. The system was named the "World Wide Web" which later became known as the Web [10]. This system was composed of a Web server, Web client, Hypertext Transfer Protocol (HTTP), Universal Resource Identifier (URI), and Hypertext Markup Language (HTML). The HTML language was used to describe a document or page containing hypertext links. Web servers simply held these HTML pages (web pages) ready to be shared with others. The Web client was developed to create, edit, and browse these pages. While HTTP described how the Web client (the Browser) and Web server would communicate over the Internet. Finally, URI was used as the scheme to address these pages. In 2016, Sir Berners-Lee received the ACM A.M. Turing Award for his contributions that underpin our invaluable Web [29].

Since its inauguration, the Web continues to develop under the standards set by Berners-Lee and his colleagues at the World Wide Web Consortium (W3C) [122]. Both the browsers and the web developers are expected to conform to HTTP, Cascading Style Sheets (CSS), and JavaScript Web API standards set by the W3C [121]. Developers use the HTML language to describe the structure of a web page's content whereas the CSS language is used to describe the presentation of the web page. Although the browser has default settings, CSS can be used to describe the colours, fonts, and how the layout should be rendered by the browser. Moreover, CSS can describe how the web page should adapt its presentation to respond to different browser widths. To be more specific, it responds to the size of the area where the page will be rendered known as *viewport size*. This allows the developer to plan on presenting the page on multiple devices and hence *responsively design* [123] the web page using CSS. Finally, JavaScript Web API is an important standard that governs how the JavaScript language communicates with the browser in order to make the

page interactive and the content dynamic.

The most fundamental JavaScript Web API standardized by the W3C is the Document Object Model (DOM) interface. The DOM is a representation of the web page which can be used by scripts to dynamically access and modify content, structure, and the style of the HTML document [82]. The document is programmatically represented, via the DOM interface, in an object-oriented fashion while the hierarchical structure of HTML is represented using a tree data structure. Therefore, each item in the HTML document is represented as an element object and as a node in the tree structure. Using the DOM interface, the developer can make a web page interactive by creating event listeners. These events can be fired by a script or triggered by a user action like a mouse click. Fundamentally, HTML, CSS, and JavaScript are the basic building blocks used to develop a web page.

2.3.1 Developing a Web Page

A growing list of front-end frameworks and libraries exist to assist in the development of web pages. These tools facilitate easier coding, improve performance, and assist in the design of a web page. Yet, the simplest way to start developing a professional web page is to use a standard text editor. The code that is needed to turn any content into a web page on the Internet is simple but may grow in complexity as more features are added. A web page can be developed using only the HTML language. The marked-up content, using HTML, is sufficient to be properly rendered by the browser. Beyond that, the web page and its content can be brought to life using CSS and JavaScript. The design of the web page can also be made responsive by programming how the content should be resized and rearranged based on the viewport size. On the back-end, code can be integrated with the front-end to better manage the content of the page.

The developer has two core strategies when making a web page. The developer can either make a static or dynamic web page. A static web page consists of a file that does not change whether it is being viewed on the client's browser or being stored on the server. On the other hand, a dynamic page introduces new or updates the content of the web page, based on some event, by running the relevant client-side or server-side code. Both of these strategies, static and dynamic, have their advantages and specific application.

Static Web Pages

Coding a static web page is the original and most basic form of web development [24]. A static web page is most often defined by the features it lacks [95]. The definition of a static page entails that the rendered content and links of the page are not dependent on data stored elsewhere like in a database. Since all the data is directly coded in the web page file, this type of web development is best suited for content that does not need to be updated frequently. When using a static site, any new content can be accessed using the links provided within the current page thereby requesting a new web page.

In the early days of the Web, content was manually coded into a static web page and was published using the File Transfer Protocol (FTP) [95]. Any modifications

had to be manually inserted into a local copy of the web page and later transferred to replace the old file on the server. As the number of web pages on a site grew, the complexity of managing the links between pages also grew. Therefore, anyone who wanted to make a static website required technical know-how in coding the content, proper link management, and the process of deployment on the server.

Although static pages were not very popular, a new trend toward automatically generating static websites has emerged. The `StaticGen.com` [113] website confirms this trend and lists the top static site generators that automate the process of turning content into static pages. Facilitated by automated tools, the motivation behind this trend is better speed and security [24]. The performance of a static page is more efficient since there is no need for server-side code or database access in order to send the file to the client. They are easily cached since the web page file stored on the server contains all of the content needed. The removal of the database and server-side code also makes the web page more secure by reducing vulnerabilities like the threat of a *database injection* attack. In this attack, the data sent to the server, for processing by the database, is modified to do something unexpected. This allows the attacker to act like someone else, retrieve private customer data, modify important data like a bank balance, or give the attacker administrator privileges.

Dynamic Web Pages

The main advantage of a dynamically designed page is that the content can be updated without manually re-coding the web page file. This is achieved by storing content in a database and using a back-end programming language to manage packaging the latest content and links into a web page structure. Another advantage over the statically designed web pages is that all links are always up-to-date without the need to fix any broken links. This approach also allows non-technical people to manage the content of the web page by using content management software that directly interacts with the database. On the downside, the database, back-end language, and management software come at the cost of lower performance and the need for frequent software updates.

In Zemmetti's book [138], he gives a good historical perspective of the development of dynamic web pages. In the early days, developers used Common Gateway Interfaces (CGIs) to request new content from the server which made the web page dynamic. Unfortunately, a CGI was limited to one request at a time and the server suffered from performance issues as the number of requests increased. Other dynamic content technologies like Microsoft's Active Server Pages (ASP) and Sun's Java Servlets were later introduced but the server still had to do all the work.

The introduction of JavaScript reduced the workload of the server and empowered the page on the client-side to modify the web page. Soon developers switched from developing dynamic web pages to building *web applications* using HTML and JavaScript. With time, this naturally evolved into something called Single-Page Applications (SPAs). Essentially, this type of page limited the number of web pages for the entire site to just one [81]. This comes with the additional cost of increased complexity in coding the SPA.

A new frontier for the web is something called Progressive Web Applications (PWAs). This type of page attempts to look and feel like a native application of the

underlying device. The main motivation for this trend is increased usage of native mobile phone applications as substitutes for web pages. To reduce the difference from a native application, a progressive web application has a set of three new features that give it better accessibility [109]. The first feature is a home-screen button that provides access to the PWA without having to open a browser and type a URL. The second feature allows access to the PWA while offline. The visitor of a PWA, can use the set of cached content while being offline. The third feature is the ability to send a notification to the device even when the browser is closed. The main advantage of PWA over native applications is that the user does not need an application store to start using the application.

Whether the developer decided to build a static or dynamic web page, it must be tested to ensure that content is presented as the developer expected. The presentation layer, or Graphical User Interface, is the reason why web pages, native applications, and even the operating system are easy to use. This type of interface presents the user with a rendered representation of some content in memory using, typically, a two-dimensional monitor. The graphical rendering is both informative and allows input devices, like the mouse, to interact with the software. Therefore, maintaining and testing the correctness of this presentation layer is fundamental to all modern software.

Asserting that the web page is properly presented in the browser is an important part of the development and maintenance phases. This testing can be manually achieved but as the development of the web page grows more complex, there should be automated measures in place to aid the developer in testing the page. The significance and challenges of this task are described next.

2.4 Testing Web Pages

A browser is a commonly used piece of software that presents web pages using a graphical user interface. It is mainly designed to graphically render web pages but has evolved to include a larger set of file types. Ideally, the content of a web page is presented to the user as intended by the original developer of the web page. Coding the content into the web page and designing the layout of the content can be a complex task. The developer may use multiple languages including HTML, CSS, and JavaScript to build the content and the layout of a web page that will later be presented using a web browser. Adding to the complexity, the developer should take into account the common standards applied by different browsers and how different monitor sizes will affect the presentation of the web page.

With a large number of browser vendors and monitor sizes to consider, developing a web page can be a very challenging task. Whether due to programming complexity or human error, a fault may seep into the code. Even if the developer is a competent programmer and takes all of the variables into consideration, a browser may deviate from the standards intentionally for exclusive features or unintentionally by misimplementing the standards. To this point, although there are standards set for the DOM interface, Mozilla warns that “care must be exercised when using them” [82] because many browsers extend the standards. The developer may also deviate, intentionally or unintentionally, from the standards while utilizing certain features.

For any of these reasons, the page may fail to present as expected.

A failure in the layout can make the web page less appealing, less informative, or as severe as being less functional. Appeal damage is a purely cosmetic issue caused by an unintended rearrangement of HTML elements or a change in colours. The second level of severity is when elements are visually distorted leading to the loss of valuable information. One scenario where this can occur is when an element is partially or fully outside the scrollable space and therefore cannot be rendered. Another scenario is when two elements overlap in two-dimensional display space and the information of the overlapped element is overwritten. The failure severity is escalated if an event is associated with the overlapped element. Since the overlapped element is hidden, the event cannot be triggered leading to a loss of functionality.

Collectively, the layout problems that cause loss of appeal, information, or functionality are referred to using the umbrella term *presentation failures* [66]. Because a browser's graphical interface presents the page by laying out the content, these problems are also referred to as *layout failures*. In the literature, the terms *failure* [6], *error* [28], *bug* [42, 114], *fault* [26], or *issue* [26] are also used in reference to a layout to mean a presentation failure. Further dissections of these terms are also used in the literature to refer to a failure depending on how it was detected. For example, if two different browsers are used to detect the failure, the term *cross-browser* is used. If a newer version of the page is compared with an older one, the keyword is *regression*. If an alternative display language is used (*e.g.*, Arabic), the term *internationalization* is used. If an alternative viewport is used, the word *responsive* is added to refer to a layout failure.

Multiple approaches were proposed in the literature to automatically test for a presentation failure. These approaches relied on either a visual analysis of the layout using snapshots, structural analysis based on the DOM of the page, or a combination of visual and structural analysis. Regardless of the underlying approach, the techniques proposed were implemented into tools that made the best use of the technology available at the time of development. As browsers and web languages improve over time, more advanced techniques were proposed. The approaches to detect cross-browser failures are described next in section 2.4.1. Then in section 2.4.2, the approaches to detecting responsive layout failures in the literature are discussed. While sections 2.4.3 and 2.4.4 describe regression failures and internationalization failures respectively.

Many other tools and services that test the design of webpages are available to the public. Some of these tools target responsively designed web pages ([93], [94]) and mobile webpages [78] by visually rendering the site at multiple widths to allow the developer to do a proper investigation. Others target cross-browser issues ([17], [30]) that test the webpage under different environments to report discrepancies to the developer. Other tools also include monitoring for both cross-browser and regression issues [79]. This literature review is limited in its ability to investigate the underlying technology for such commercial services and tools due to the lack of transparency and documentation.

2.4.1 Cross-Browser Testing

The early days of the web saw fierce browser competition. Their goal was not to ensure that pages worked on a competitor's browser, instead, it seemed the opposite was happening. This time is referred to as the browser war days. During this time, it was common to display a message stating that the web page is best suited for some browser vendor. Nevertheless, the developer had to take into consideration how the page will be presented on different browsers. To this day, differences between browsers are expected and the page should be tested for cross-browser problems.

A cross-browser failure is a presentation failure that arises when comparing the presentation of a web page in two different browsers. They are referred to as Cross-Browser Issues (XBI) [26] or Cross-Browser Incompatibility [27, 28, 74]. More specifically, a cross-browser failure is defined to be the difference in a web page's visual appearance or behaviour in different browsers. They may result in a minor cosmetic issue or a critical change in the functionality of the web page.

According to Choudhary et al. [26] there are two types of XBI; layout and functional issues. These issues could arise due to several underlying causes. First, it could arise because the browser is non-compliant by not following the common standards. Meaning that the browser did not fully implement the standard or has mistakes in the implementation of the standards. In another case, a browser issue can arise due to the usage of additional non-standard features of a browser that is not available in other browsers. Alternatively, the issue could be raised due to the differences in the default styling of browsers. The web page may also have issues due to the local resources not being available, such as a browser plug-in or font. The final possible underlying cause is a syntactically incorrect page. This could be an issue because different browsers may resolve the syntax error differently which may propagate visually as a cross-browser issue.

Six main techniques and tools have been proposed in the literature to detect XBIs that I review here. The first proposed approach, WebDiff, introduces automation to the process of testing a web page for XBIs. The second approach, CrossT, is specialized in detecting functional XBIs in web applications. The third approach, CrossCheck, improves on previous approaches and uses a machine learning technique to detect layout and functional XBIs. Then, the more successful X-PERT approach introduces *alignment graphs* to detect XBIs. At the same time, the tool BrowserBite also aimed to improve the approach used by CrossCheck in its own way. Then came the X-Check tool which improves on top of the X-PERT tool. The implementation of each approach is explained in more detail next.

The tool WebDiff was developed by Choudhary et al. [26] to automatically identify cross-browser issues in web applications. At the time, existing commercial tools were limited to showing side by side views of the page thus requiring the developer to manually find any issues. WebDiff is based on differential testing [72] which is an approach that exposes potential bugs by looking at differences in two or more comparable systems. More specifically, the tool finds differences by comparing the DOM and visual appearance of a web page in two browsers to detect any issues.

WebDiff automatically finds cross-browser issues by doing the following. First, it opens the web page under test in multiple browsers and chooses one browser to serve as the reference to be compared to. Then, DOM-based information about the elements is gathered from each browser and a snapshot of each browser is also taken. To identify the

elements with dynamically changing content which should be ignored, a second snapshot is taken and compared with the first snapshot. To force these dynamic elements to match in the snapshots, they are changed to a common colour so that they match visually. In the next step, the locations of all elements are matched up to find differences between the browsers.

To visually analyse the snapshots, a *colour histogram* [15] is used to group the colours of each element into predefined ranges. Therefore, the same colour distribution is only possible if the two renderings are similar. Any small shift in the position of elements is ignored by using the Earth Mover's Distance [98] metric. Here, it is used to ignore slight changes, due to a small shift, between the images and to check for perceptual similarity. Finally, the HTML tags of the elements are reported to the developer for further investigation.

Another tool developed by Mesbah and Prasad to detect cross-browser issues is called CrossT [74]. The process behind this tool consisted of two main steps. The first step starts by opening multiple browsers and loading the web application. Then the applications are simultaneously crawled in order to capture their behaviour as a finite-state machine, referred to as a *navigation model*. To do this, they used the Crawljax tool which crawls and triggers behavioural events in the web application. In the navigation model, every screen observable by the end-user is a state and a user action represents a transition between the states. Finally, in the last step, these navigation models are isomorphically compared.

As input, CrossT takes a URL, a list of browsers, and a list of HTML elements to include or exclude in the run. After the navigation models are collected, they are compared at a *trace-level* and *screen-level*. First, at the trace-level, the difference in the sequence of events or user actions are checked. Then, at the screen-level, the internal DOM representations are checked for differences. At this level, case sensitivity, white spaces, attribute order, and more heuristics are ignored by CrossT to reduce the false positive rate. During the run, a plug-in is used to take a snapshot of each state reached. Then a report is generated for each pair of mismatched screens which includes two snapshots, a DOM printout with highlighted differences, and a sequence of user actions to reach the screen.

As an improvement to existing cross-browser testing tools, Choudhary *et. al.* [27] unified two complementary tools, WebDiff [26] and CrossT [74], in the hope of better XBI detection. Combined, a comprehensive technique emerges from WebDiff ability to focus on visual XBIs and CrossT's ability to detect functional XBI issues. This new solution was implemented in a tool called CrossCheck. The authors also improve on the previous visual analysis approach by using machine learning and new image processing metrics.

The new machine learning applied is a classification approach which proved to be far superior to the custom-made heuristics previously used in WebDiff. The classifier is first trained on a data set that was generated using ten web applications where each XBI was manually labelled true if it was substantially different. The classifier is provided to CrossCheck as input along with the subject URL and a choice of two browsers. Because the classifier is part of the input, it must be available before running CrossCheck.

The classifier uses a *decision tree* classification method with *features* that are picked by Choudhary et al. [27] based on their personal experience with XBIs. The first feature is *size difference ratio* and is used to detect differences in the size of two elements. The second feature *displacement* captures the Euclidean distance between the position of two elements. Third, the minimal area of the two elements is used by the classifier as a

threshold for other features. The fourth is a boolean feature that detects if there is a difference in the text of two DOM leaf nodes. The final feature measures the distance between colour histograms, as used in WebDiff [26] without the need to use the Earth Mover's Distance [98].

The technique used in CrossCheck is comprised of three phases. In the first phase, the two browsers are simultaneously crawled and user-action events (*e.g.*, mouse click) are triggered to build a navigation model. Again, this model is a finite-state machine where each state is a screen and the transitions are the events causing a change in the screen. For each state, the DOM is recorded and a snapshot is taken. The tag name and event type are also recorded as a label for each transition. In the second phase, the two models generated in the first phase are compared for equivalence. The mismatched transitions represent trace-level mismatches while the matching pair of states are used to compare the DOMs for a mismatch by comparing each node. The DOM nodes that are mismatched are recorded as a possible XBI. Then, the nodes that were matched are visually compared using the snapshots taken in the first phase. Then, the features needed by the classifier are computed for each of the matched DOM nodes. In the third phase, all the mismatches from the model are analysed and the resulting XBIs are clustered and reported.

Another advancement in cross-browser testing by Choudhary et al. [28] is the X-PERT tool. It was developed to detect XBIs based on a crawl-and-compare approach. A goal of this tool is to provide a comprehensive solution by combining existing differencing techniques with new ones. The crawling and comparing features of X-PERT are identical to that of the previous tool CrossCheck [27]. Contrary to the approach of CrossCheck, the technique implemented in X-PERT is derived from a study of XBIs in the public domain; an improvement over previous techniques that largely depended on intuition. The results of the XBI real-world study lead to the identification of three types of XBIs.

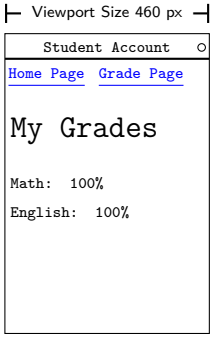
The X-PERT tool is capable of detecting three type of failures. These are *structural*, *content*, and *behaviour*. The first type, *structural*, refers to a change in the layout. An example of this type is when the relative alignment of HTML elements is rearranged. The second type, *content*, refers to the differences noticed when comparing the same element across different browsers. This type of failure is further divided into two sub-problems, *text-content* and *visual-content*. An example of the first type was given by the authors as an element that renders different text in different browsers. The second type refers to the visual aspect of an element like a difference in the style of the text. The final type of XBI, *behaviour*, is a category of XBIs where a difference in the functionality of a component is observed.

The novelty of X-PERT lies in observing and comparing the structure of the page for XBIs. The tool achieves this by using a graph-based model of the page called the *alignment graph*. The alignment graph is a directed graph that represents a *parent-child* and *sibling* relationships between HTML elements. A parent-child relationship is first determined by the child's XPath being a prefix of the parent's XPath. Additionally, the rectangular coordinates of the child element must fall within the bounds of the parent rectangle. As for the sibling relationship, it is established between elements that share the same parent. Moreover, other geometrical relationships are added to the model based on the relative position of the elements. Meaning, that a child element can be horizontally aligned left-of, above-of, or centre-justified within the parent element. The technique further breaks down more relative positions to describe vertical positioning and to describe sibling relationships.

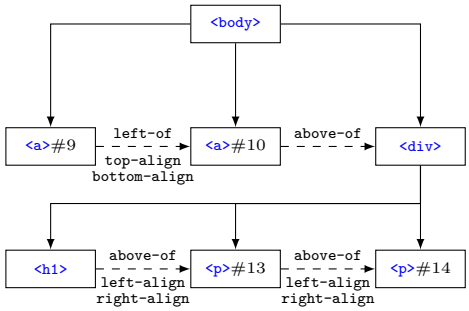
An illustration of an alignment graph is given in Figure 2.1. In this example, a simple HTML file is shown in part (a) while the rendering of the file is shown in part (b). Importantly, the alignment graph of the page can be seen in part (c). In the graph, the

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4        <title>Student Account</title>
5        <style> @media (max-width: 450px)
6        {a { display: block;}} </style>
7      </head>
8    <body>
9      <a href="index.html">Home Page</a>
10     <a href="grades.html">Grades Page</a>
11     <div>
12       <h1>My Grades</h1>
13       <p>Math: 100%</p>
14       <p>English: 100%</p>
15     </div>
16   </body>
17 </html>
```

(a) Simplistic HTML file.



(b) Web page.



(c) Alignment graph.

Figure 2.1: An example HTML file to illustrate it's alignment graph.

parent relationship is represented using solid edges while a dashed line represents a sibling relationship. While only a single parent or sibling edge exists between any two nodes, some sibling edges are omitted from the graph to avoid cluttering. Using an alternative browser, X-PERT would generate another alignment graph and compares the two for differences.

To begin testing a page using the X-PERT tool, the URL is required as input and a choice of two browsers. Then the browsers are simultaneously crawled and compared. Instead of comparing all DOM elements, only the leaf elements are compared for *visual-content* XBIs. This is an improvement to the adapted features of CrossCheck in order to reduce the false positive rate. Once an alignment graph is constructed, it provides the needed insight into the layout. This information is used to properly compare different layouts by isomorphically comparing two graphs to detect any structure XBIs. X-PERT then generates an HTML formatted report containing a list of the failures detected. The developer can now assess these reports using the side-by-side snapshots included in the report.

The usage of a graph-based model to represent the relative layout of elements, as done in X-PERT, went on to inspire more work on the detection of presentation failures. More specifically, it inspired the development of similar graphs that capture differences across viewport sizes [124, 125], languages [3], and versions of a responsive web page [6].

As was the goal of the X-PERT tool, a tool named BrowserBite [100, 101, 107] aimed to reduce the high false positives produced by previous tools like CrossCheck and WebDiff. Unlike the X-PERT tool, BrowserBite uses a pure image processing approach that intentionally does not rely upon the DOM to detect XBIs. The BrowserBite tool with its basic configuration without any machine learning is also claimed to perform just as well as the X-PERT tool [101]. Furthermore, BrowserBite was available as a patent-pending testing service [18] until the end of the year 2017. The tool is comprised of four main phases. These are snapshot capturing, image segmentation, image comparison, and machine learning classification.

With a URL and a browser chosen as a baseline reference, the tool begins by capturing a snapshot of the page using the reference browser and another snapshot for each testing-browser. Besides the URL, the input to the tool should include other runtime configurations like the operating system to use while testing. This is used to spin off a virtual machine with the runtime environment needed to test the page. The captured images are then broken down into smaller *Regions of Interest* (ROIs) by focussing on intensity change rather than a colour change. This is an attempt to mimic what a human visual system

would focus on [59]. The image processing is broken down using greyscale conversion, corner detection, dilation, and block analysis in order to create the ROIs. To overcome issues with comparing ROIs with different sizes the dilation parameter is changed as needed. The images are then compared using image coordinates and *raw moments* that describe the image geometry like its orientation. After pairing ROIs from the reference browser to the ROIs from the testing-browser, the images are compared for size, visibility, content, and font style differences. Without using machine learning, the tool reports these differences to the developer.

To reduce the number of false positives that the basic BrowserBite may produce, machine learning is used to train the tool on what is safe to ignore. The authors of the tool used two classifier variations, a binary and a quaternary classifier. For the binary classification, each XBI is either a true positive or a false positive. Meanwhile, the quaternary classifier breaks the classes into levels of severity where the lowest is a false positive followed by a minor, major, and critical issue. As with all manual classifications, subjectivity should be noted especially as more classes are added. The authors also used two classifier methods, decision tree and neural networks only to find out that the neural network outperformed the decision tree method.

Building on the state of the art tool X-PERT, is a tool created by He et al. [45] that was named X-Check. Like the BrowserBite tool, X-Check aimed to reduce the false positives reported by the X-PERT tool. The main shortcoming of X-PERT in testing is that it only captures user-initiated events (*e.g.*, a mouse click). Since the same web page is tested in multiple browsers, the appropriate state of the page must be compared across the browsers. This means that the same sequence of events should be fired to match the execution sequence of the reference browser. Since JavaScript has more than just user-created events including time-based events and asynchronous server responses, X-PERT may come to a wrong conclusion. To solve this problem, the X-Check tool uses a record and replay strategy for testing web pages.

To capture the non-deterministic events like a browser response, the X-Check tool integrated the Mugshot [76] JavaScript library. This tool allows individuals and automated tools to trace the execution of a webpage. Unlike the alternative Jalangi [108] execution trace tool, Mugshot is not an extension of a specific browser and thus is browser-independent. To capture the user-initiated events, the tool adds a listener to the root `window` element to record all events. For timing and server request multiple JavaScript functions are interposed with a wrapper function to monitor their execution.

Similar to the X-PERT tool, X-Check uses the alignment graph [28] to capture structural changes in the layout of different browsers. To analyse the pages visually, the tool uses the Chi-Squared [16] histogram distance method to evaluate differences in colours of matched elements in the snapshots. Finally, to detect any text-based changes, the tool compares the text content of DOM nodes for differences.

The output of the tool is a list of XBIs and the name of the page where the failures were detected. The tool also outputs snapshots with highlights where a difference was found. As a result of their empirical evaluation, the X-Check tool was found to outperform the X-PERT tool especially in reducing the number of false positives in highly interactive web applications.

Although each of the cross-browser testing tools was shown to help the developer test for cross-browser issues, they have their limitations. This includes partial manual intervention, false positive reports, duplication in failure reports, and some techniques that required manual verification of the reported failures. For example, CrossT required a user to specify which parts of a web page to include and exclude in the detection. Meanwhile,

a classifier is needed as input to CrossCheck before it can be used to detect failures. As for the tools WebDiff, CrossT, and CrossCheck, they all produce a high false positive rate with duplications in the reports. For the performance of the BrowserBite tool, it will depend on the training set and the manual classification used. Furthermore, these tools and both X-PERT and X-Check all required a reference browser to be used as an oracle for the "correct" layout. Furthermore, all the techniques of this section were limited to the detection of presentation failures and thus required a human to manually repair the reported failures.

2.4.2 Responsive Design Testing

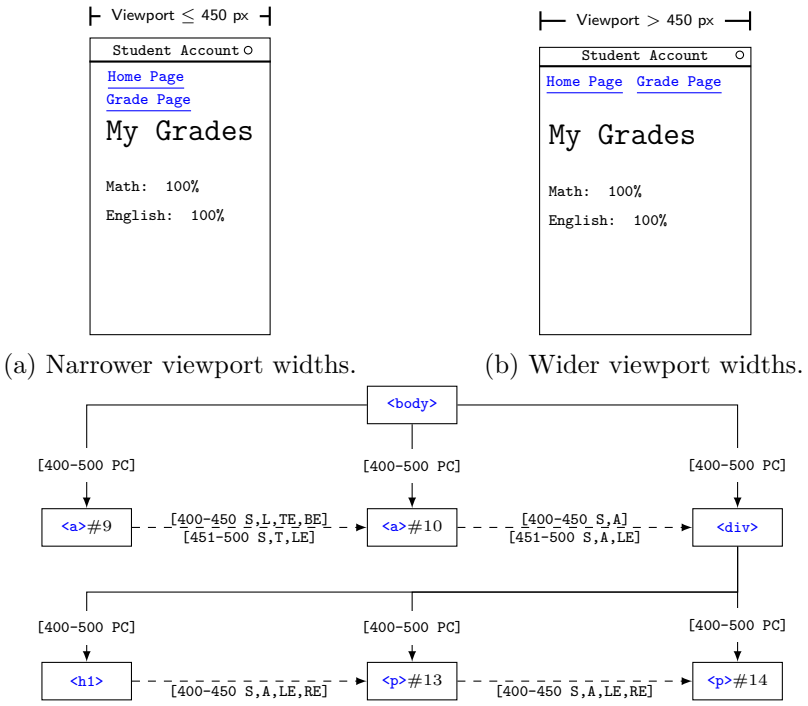
Responsively designed web pages present a different challenge to testing the presentation of a web page. These web pages are developed to conform the layout of a web page in a way that better presents the page depending on the available browser window width. By prohibiting a design that requires horizontal scrolling, the visitor would vertically scroll for more content. Two terms that generalize the presentation failures that exist for this type of design. These are *Responsive Layout Failures* (RLFs) and *Visibility Faults* (VFs). It is worthy to note that the research of this thesis solely revolves around responsively designed web pages and their RLFs.

Responsive layout failures are presentation failures that are observed in responsively designed web pages across different widths using the same browser. A goal of a responsively designed web page is to provide an appropriate layout depending on the current browser width. With a large range of possible browsers widths, testing the design of responsive web pages for presentation failures is a challenging task. Walsh et al. [124] developed ReDeCheck to tackle this challenge without the need for an explicit oracle. At the time, developers of responsively designed web pages mainly relied on a manual effort to detect RLFs. More specifically, ReDeCheck was developed to automatically detect five types of RLFs that were defined by Walsh et al. [124].

The RLF detection technique of ReDeCheck [124] relies on comparing HTML elements of a web page to each other and across different viewport widths. More specifically, the position of two elements is compared in two consecutive viewports to identify any differences in their relative positions. To make this inference, the ReDeCheck tool uses a graph model to represent the layout of a web page across different viewport widths, referred to as a Responsive Layout Graph (RLG). The original idea of modelling the layout of a web page as a graph stems from *alignment graphs* introduced in X-PERT [28] to handle cross-browser failures.

A key feature of the RLG that improves on the alignment graph is the incorporation of multiple viewport widths. The RLG model is built by automatically quarrying the DOM for the HTML elements that make up the page and retrieving their coordinates. Moreover, the visibility and relative alignment of each element is recorded as the width of the browser is changed. In the graph, HTML elements form the nodes while relationships between the elements form the edges of the graph. These relationships between the elements like parent-child, sibling, and relative alignments are not based on the HTML hierarchical structure. Instead, they are calculated based on their coordinates. Along with the relationship descriptors, edges are also labelled with an inclusive range of viewports where the relationship holds true. The nodes of the graph also carry information about the range of viewports where the element was visible.

Figure 2.2 illustrates a responsive layout and its RLG model using the same HTML file from Figure 2.1 (a). In this example, the range of viewport widths visited for testing



(a) Narrower viewport widths.

(b) Wider viewport widths.

(c) Responsive Layout Graph.

Figure 2.2: An example web page to illustrate an RLG.

begins from 400 pixels to 500 pixels wide. The responsive design of this page sets the links to vertically align on smaller viewport sizes. For viewports larger than 450 pixels wide, the page is designed to automatically switch the links into horizontal alignment. This is achieved by using the `@media` CSS query as seen in the HTML file. With this rule, the page will be rendered on smaller viewport sizes as shown in part (a) and for larger viewport sizes as shown in 2.2 (b).

The RLG graph shown in Figure 2.2 (c) denotes the HTML elements of the page as rectangular nodes, parent-child relationships "PC" using solid edges, and sibling relationships "S" as dashed edges. More layout attributes that describe the relative layout of the elements to each other are shown in the graph. These include above-of "A", left-of "L", top-of "T", left-edge aligned "LE", right-edge aligned "RE", top-edge aligned "TE", and bottom-edge aligned "BE". The graph also shows the inclusive numerical values of the viewport range where the relationships or layout attributes hold true. Omitted from the graph, to avoid cluttering, is the visibility constraint of each node that denotes where the element is found to be visible. This should be an entry to each node `[400-500 VC]` because all the nodes are visible throughout the testing range. Some sibling edges are also omitted to make the graph easier to read. Since the alignment of the second link (Grade Page) changes within the testing range, the graph captures this change within the sibling edges.

In addition to the RLG model, Walsh et al. [124] also defined five type of responsive layout failures. The first type is an *element collision* which occurs when two elements overlap each other unintentionally. The second type, *element protrusion*, occurs when an element is no longer confined within the area of its intended parent element. A *viewport protrusion* occurs when an element protrudes outside of the main HTML element of the page for displaying content, the `body` element. The fourth type of failure is called an *element wrapping* and occurs when an element breaks its row formation and wraps into a new line. The fifth and final type, a *small-range* failure, is a layout of two elements that

occurs within a small number of viewports in contrast to their layout in other viewports. This number was experimentally defined to be five or fewer viewports. What is noteworthy here is that by definition, a small-range failure can be a duplicate report of any of the other four failure types that occur in five viewports or less. Nevertheless, they aim and can capture other important failures that are associated with the breakpoints of the responsive design.

With the RLG model, the ReDeCheck tool uses a set of specialized algorithms to detect the five type of RLFs. First, an element collision is detected by finding an overlap attribute "O" on a sibling edge in a narrower viewport but not at a wider viewport. The detection of an element protrusion is also achieved by looking for the overlap attribute but must also find a relationship change from sibling to parent-child in the immediately wider viewport. A viewport protrusion is detected in a slightly different manner and depends on the fact that all visible elements should have at least one parent-child relationship except for the `body` element which has no parent in an RLG. ReDeCheck looks for an element that is labelled as visible in the model but with a parent-child relationship that does not fully cover the visibility range. To detect a wrapping failure, the tool first finds at least two siblings with the alignment right-of or left-of. Furthermore, other siblings should exist with above-of or below-of alignments. To report it as a failure, the alignments of the siblings with above-of or below-of must change to right-of or left-of in the adjacent wider viewport. Finally, small-range failures are detected by first finding alignment attributes that are only applicable in less than five viewports. In order to report it as a failure, the approach verifies that a different alignment attribute exists between the same two nodes in the immediately wider and narrower viewports. Using ReDeCheck, the example web page of Figure 2.2 would not report any responsive layout failures.

The ReDeCheck tool tests each responsively designed web page over a configurable testing range of viewports, $\{test_{min}..test_{max}\}$. In previous experiments and the ones conducted for this thesis, 320 pixels is used for $test_{min}$ and 1400 pixels for $test_{max}$. The tool traverses this testing range to capture how the layout changes over different viewports. Optionally, a customized stepping size can be configured to increase the efficiency of the tool. A stepping size of 60 was shown to be ideal by Walsh. If any change occurs while stepping over certain viewports, a binary search is used to find the exact viewport where the change occurred. Once the viewports in the testing range are visited and the RLG model is built, the detection algorithms are executed and any failures found are reported with their failure range, $\{fail_{min}..fail_{max}\}$. Since ReDeCheck uses a purely DOM-based approach, some of the reported failures can be DOM-based structural problems that do not manifest visually in the layout. These are known as non-observable issues.

Another term used in the literature to mean a presentation failure of responsively designed web pages is a *visibility fault*. These faults are incurred due to the dynamic changes of the layout that result in a change in the functionality of the web page. An example is an interaction or event in the page over two different viewport widths that produces two different layouts with an accessible link in one layout but not the other. In this scenario, one layout has more functionality than the other which could be caused by a change in the CSS property of an element. One possibility is that the change led to a new back-to-front display order of the elements in the layout and thus the link is inaccessible. Ryou and Ryu [99] proposed a technique, implemented in VFDetector, to automatically detect loss of functionality due to layout changes in responsively designed web pages. The change may be triggered by a user's interaction with the web page or a change in the width of the browser causing a rearrangement of the layout. In comparison with previous work, ReDeCheck [124] does not consider JavaScript and therefore cannot detect layout issues that are a result of a user's interaction with the web page.

The VFDetector technique detects visibility faults by following three steps. Furthermore, the tool assumes that the initial states of the web page across different widths are comparable since no events have been triggered explicitly. In the first step, the tool identifies pairs of layouts that should have consistent functionality. Second, VFDetector finds the subset of HTML elements that need to be compared. These are HTML elements that are intended to be visible in both layouts and have one of 26 types of user events associated with them. An example of this is an HTML button element with a click event listener that is intended to be visible in both layouts. Furthermore, the visibility of each element is categorized as either *absent*, *full-cover*, *partial-cover*, *off-screen*, or *normal* depending on its position in the back-to-front display order. The third and final step is to inspect the visual status of HTML elements and identify any visibility faults.

The two type of visibility faults detected by VFDetector are *inconsistency fault* and *covered error*. To determine the type of visibility fault, VFDetector compares the visibility of the same element in both layouts. If the element has a full-cover visibility in only one layout and hence the event handler cannot be executed, it is referred to as a covered error. Otherwise, if there is a difference in visibility other than full-cover visibility, then it is referred to as an inconsistency fault.

The VFDetector tool is able to detect visibility faults by using a DOM API wrappers to monitor and collect information about the registration of event handlers. This is achieved by using a proxy that intercepts a web page request and injects the DOM API wrappers. The proxy is also used to prevent the web page under test from loading a different page. The tool then calculates the back-to-front order to determine the visibility of elements that were recorded by the wrappers. Specifically, the visibility is calculated based on the HTML structure, coordinates, and the size of elements. After that, only the visible elements that do not have full-cover visibility are triggered until all dynamically changed layouts have been reached and recorded or a timeout occurs. In their experimental evaluation of the tool, the timeout was set to 30 minutes. The tool then prints a report that provides a *replay description* which specifies the width of the browser, sequence of events to reach the layout, and the elements exhibiting the visibility faults.

Both the VFDetector and ReDeCheck tools have their limitations. While ReDeCheck is able to detect five type of RLFs, it is limited to the initial state of the web page across different viewport widths. Although VFDetector overcomes this limitation of testing only the initial state, it is limited to the layout failures of elements that have an event associated with them. Moreover, both of these tools are restricted to a DOM-based analysis and do not take into consideration the final rendering of the web page. This created the need to distinguish between observable and non-observable failures. Another limitation or potential advantage that ReDeCheck does not utilize is that the overall frequency of alignment is not monitored for insight (*e.g.*, suspiciousness) to infer if the change was intended or not. Furthermore, all the reports generated by both these tools need to be manually verified by a human and manually repaired to fix the problems detected.

2.4.3 Regression Testing

When the underlying HTML structure of a web page is modified for improvements and updates, new layout failures could be unintentionally introduced [60]. An example of this is when a page is refactored in order to move to a newer HTML version or when replacing deprecated HTML tags. If the intention of the changes made was not to alter the layout, the previous version of the web page can serve as a reference to verify that no new presentation failures were introduced. This process of re-testing the software after a recent modification is called regression testing [5]. The tool FieryEye was introduced by

Mahajan et al. to solve the problem of detecting failures in a refactored web page. FieryEye uses image-processing and probabilistic techniques to detect presentation failures. A developer can benefit by using this technique to ensure that any structural changes made do not manifest to a change in the appearance of the web page.

The first and most important input to the FieryEye is a snapshot of the previous version of the web page to serve as a reference that shows how the web page should appear after modification. With the snapshot, a list that excludes certain regions of the web page must also be provided as input. These regions should exclude elements with dynamically changing content from the analysis because the visual change of the element can increase the rate of false positives. Finally, a URL of the web page to be tested must also be provided in order to detect possible layout failures.

FieryEye starts by analysing the web page under test against the reference to find any visual differences. The actual analysis follows the same approach used in their previous tool WebSee [67, 68]. In that tool, the authors used Perceptual Image Differencing [136] to mimic the human notion of similarity when comparing images. The change in visual appearance is then used as input to the probabilistic model that should map the failure to possible faults. The faults that cause a presentation failure are due to either an incorrect value in an HTML attribute or a CSS property. Since there are many types of attributes and properties, only the subset that could lead to a visual change were manually identified from the HTML and CSS standards. This subset of attributes and properties are reported as possible causes of presentation failures to the developer. The report provides the possible causes as a ranked list mapped to a potentially faulty HTML element.

The model that is used to map the failures to possible causes is based on conditional probability. The information gained by injecting faults into the web page under test is used to build the model. The injections are made by systematically modifying properties that influence the visual appearance of an element to see if there is a visible change. By observing the generated data samples and learning the correlations, the authors of the tool were able to build their probabilistic model. Although this model is based on manufactured faults, the distribution of real faults can improve the model and the results.

The ranking of all possible causes is achieved by calculating the probability that a property or attribute caused the failure given a specific *visual symptom*. The visual symptoms are predicates grouped into abstraction; they are colour, visibility, size, text appearance, decoration style, and the difference in pixels. These predicate conditions are used to categorize an image into specific groups. For example, if a colour is added or removed from the visual rendering of an element then the specific predicate in the colour group is flagged as true. Similarly, a visibility predicate is flagged true if the height or width of the element is greater than zero which indicates that it is visible. The authors identified 24 of these predicates and abstracted them into the 7 groups. As a result of this ranking, the evaluation of the authors showed that the developer needed to inspect 8 possible causes before reaching the right one.

In addition to detecting failures that arise during HTML migration to a newer version, responsively designed web pages require special regression consideration. For example, a developer may modify certain CSS properties of a web page to alter the layout at a specific browser width. The modification may appear as intended in the desired viewport size but it can also manifest into layout failures in alternative viewport widths. To monitor for these unintended changes, Walsh et al. developed the original ReDeCheck to detect regression-based layout failures in responsively designed web pages [6].

This version of ReDeCheck originally introduced a graph-based model that captures HTML elements and their relative positions across a range of viewport sizes, known as

the RLG. The RLG is an improvement to the *alignment graph* first developed by Choudhary et al. [28] which was limited to a fixed viewport size. This improvement allows for the modelling of the visibility, width, and relative alignment of elements across a range of viewports. To detect any layout failures, ReDeCheck first extracts the RLG of the old version and the new version of the web page. Then a pairwise comparison of the models is made for visibility, width, and relative alignment of the DOM elements. Any differences are then reported to the developer.

Interesting to note, that during the evaluation of the ReDeCheck regression testing mode, referred to as ReDeCheck-RM [126], the page is mutated using eight operators. In essence, these operators modify specific CSS and HTML of the page in an attempt to introduce synthetic regression failures. The target CSS modifications were to, property value (*e.g.*, `20px` to `15px`), property unit change(*e.g.*, `%` to `px`), media query feature (*e.g.*, `min-width` to `max-width`), media query breakpoint (*e.g.*, `min-width:700px` to `min-width:702px`). While the target HTML modification was to the text of an element (*i.e.*, increase and decrease length) and the HTML class attribute by adding, removing, and replacing a class-name that is used to specify the style of a specific element.

Both of the tools ReDeCheck and FieryEye had some limitations. In the case of FieryEye, the tool had limited automation because the developer had to manually distinguish regions of the web page to exclude from testing. A problem with the ReDeCheck tool was that it detected both intended and unintended changes to the web page. Furthermore, both of the tools assumed and required a correct version of the web page to be available as a reference, the previous version of the page. This means that the tools will not detect any failures that existed in the older version of the page. Importantly, the developer is left with the manual task to identify false positives from the reported layout failures and has to manually repair all the true positive failures.

2.4.4 Internationalization Testing

The layout of a web page is typically designed with the text-based content of the page set to some default language. Supporting multiple languages is possible through the use of isolated "need-to-translate" strings located in a resource file or through translation APIs. Both of which allow the web page to support thousands of languages but at the risk of introducing presentation failures. These failures occur when the HTML elements visually distort while accommodating the alternative text length, width, and height. With a large number of supported languages, manually checking the layout for each language can quickly become an unfeasible solution. Alameer et al. [2, 3] termed the distortions that occur while elements expand, contract, or move to accommodate the translated text as Internationalization Presentation Failures (IPFs).

The tool GWALI, developed by Alameer et al. [3], was developed to automatically detect these failures. To do this, the tool builds a *layout graphs* model of the web page in the default language and compares it against a second model of the same page under a different language. The layout graph represents the relative position and visual relationships of text elements and HTML tags of the web page. The nodes of the two layout graphs are mapped and the differences are calculated using specialized parameters determined through experimentation to generate a list of potential failures. Then, each node is given a suspiciousness score using three heuristics that ignore a certain level of change based on a threshold value set by the authors. Finally, the list is ranked before outputting an ordered list of IPFs.

Although not as efficient as GWALI, other non-IPF tools can be adapted to detect

possible IPFs including X-PERT, FLB, and WebSee as evaluated by the authors. The reason for the efficiency of GWALI is that it is specialized in text elements. On the other hand, this also limits the ability of the tool to detect other type of layout failures. Another limitation of the tool is that it assumes that the web page displaying the default language is free from layout failures. Thus it is used to serve as the correct reference layout. Another vulnerability of the tool is that the false positive rate is highly dependent on the threshold set for acceptable changes. Finally, the task of verifying the reported IPFs and the process of repairing them is a manual task left for the developer.

2.4.5 General Layout Testing

The testing tools that I have reviewed thus far were easily classified into a specific category of presentation failures. Some tools are simply more generic in their approach to testing the layout of a web page. These generic tools do not require a reference browser, viewport, language, nor an older version of the page to detect the failures. This flexibility gives them broader applicability but hinders their ability to compete as well as other specialized tools.

In the year 2009, Michael Tamm [114] released a tool under the name "Fighting Layout Bugs" (FLB) to mitigate and detect layout issues. The technique uses HTML validation, CSS validation, and image processing to detect layout issues. By validating the HTML of a web page, the tool essentially prevents different browsers from guessing the correct structure of the page which may lead to layout issues. Furthermore, HTML validation prevents CSS from using the possibly incorrect HTML structure. The validation of CSS is as also important to have a consistent rendering of a web page and mitigate any issues. Nevertheless, validating HTML and CSS does not eliminate all layout issues.

Michael Tamm also saw the importance of visually analysing the web page to detect layout issues. His solution ensures that pixels associated with text do not overlap with the location of pixels that represent graphical edges. First, the text pixels are identified by using a JQuary [47] to dynamically change the colour property of all HTML text to black and a snapshot is taken. Then, the colour of all the text in the document is changed to white and another snapshot is taken. By comparing the change in the two snapshots, the pixels associated with text-based content are identified. The tool then needs to identify the pixels that are located in any vertical or horizontal edges. It starts by setting the colour of all text to be transparent and takes a snapshot. Then all vertically and horizontally neighbouring pixels are checked for high contrast to detect the graphical edges of the page. With this information, the tool can detect any text that overlaps an edge and reports it as a layout issue. Furthermore, it is able to report any text that is in low contrast with its surrounding pixels as a layout issue.

Another generic method for testing the layout was proposed by Tanno and Adachi [115] to make the process of manually classifying the reported failures (differences in the layout) more efficient. Their method is independent of the underlying runtime environment. Instead, all that is required is a reference snapshot assumed to be correct and another snapshot to test. It does not matter whether it came from two different viewports, two browsers, or the same application running on two devices.

The approach of Tanno and Adachi aimed to detect differences between two images that are due to addition, omission, movement, or scaling of content. It works by first matching the size of the two images that were provided as input for comparison. Then it segments the image into rectangular areas then extracts and matches features in each rectangle. To segment the image into rectangular regions, the process involves the use of a Sobel filter [15] for targetting larger gradation differences and using the Canny edge

detector [25] to determine edges. Moreover, the KAZE [4] method is used to investigate image features within the matched rectangular areas. This results in finding a difference in the rectangular units rather than pixel units.

The output of the Tanno and Adachi method is a list of differences presented using a graphical interface. The developer can use this interface to manually classify each of the findings as either an acceptable difference or not. Since the aim of the approach is efficiency, the developer can also add custom rules in order to reduce sensitivity. For example, the developer can add a rule to ignore differences of 20 pixels or less in movement and up to 3% scaling. Although helpful, this approach is still mostly manual.

Another general-purpose specifications-based approach to test the layout of the page was suggested by Hallé et al. [42]. Their approach provides a tester with a declarative language that is able to express constraints that uphold known layout specifications or prevent a specific failure from recurring. This makes regression testing an ideal use case for their approach. Nevertheless, it is not solely for regression testing. The language was implemented into a tool called Cornipickle that is able to make expressions about the DOM and CSS properties. A prototype of the tool provided a sufficient grammar for testing the layout of a web page which includes the ability to express events or temporal operators. Moreover, the grammar can be extended by the user to make it more readable. This means that the tester can add new definitions to the vocabulary of Cornipickle which will be read by the interpreter and handled appropriately. This is in done in an attempt to improve the readability and adoption of the language that gets its inspiration from Cucumber [8, 134].

So long as the tester expresses the correct layout or what contradicts a correct layout, Cornipickle can catch many type of presentation failures. The authors of the tool identified six variations of *disruption* that occur to the layout and four *behaviour*-based layout bugs. The layout disrupting bugs involve the element position, element size, element visibility, number of elements in the page, resource files used to style the page, and encoding problems. The behavioural bugs include a change in the position due to some event, inactivity of buttons, lack of update to the page, and incorrect updates to the content of the page. Nevertheless, expressing these failures is a manual task that may not be as useful as automated tools that can detect what the tester misses.

The Cornipickle language is not as familiar nor as competitive as basic test-running tools like Cypress [49], Selenium [106], Puppeteer [91], or Playwright [35]. More importantly, the last thing the developer needs is another language on top of HTML, CSS, and JavaScript. It is reasonable to also expect that the developer is already using other languages, pre-processors, and libraries that make the job easier or safer like TypeScript [120], Sass [103], or BootStrap [13]. On the other hand, the readability of the Cornipickle language allows the process of testing the web page to reach beyond the technically savvy tester. In theory, other business affiliates or stakeholders with minimal technical knowledge can also participate in testing by writing constraints using the human-readable language of Cornipickle.

2.5 Repairing Web Pages

Although the published literature presented multiple automated techniques to test a web page for different types of presentation failures, there were fewer techniques that automatically repair them. The work was primarily focused on three areas, the repair of cross-browser issues, internationalization failures, and mobile-friendly problems. Importantly, no previous research that I reviewed attempted to automatically repair responsive

design failures. As such, one of my contributions in this thesis, discussed in Chapter 5, has been the automated repair of responsive design failures. Next, I will review what has been proposed to automatically repair presentation failures in web pages.

2.5.1 Repairing Cross-Browser Failures

While many techniques have been proposed to detect cross-browser issues, discussed in Section 2.4.1, there were far fewer attempts to automatically repair these type of failures. More specifically, there was only two approach that aided in the repair of XBIs implemented into the tools XFix and X-Diag. While XFix can automatically generate patches that repair the layout, the X-Diag tool is designed to debug the issue and find the root cause.

To repair the layout failures that arise in different browsers known as structural XBIs[28], Mahajan et al. [62, 63] employed a search-based technique to reach the CSS values that will repair the XBI from the browser exhibiting the issue. This process begins by finding the layout XBIs using the X-PERT tool. Included in the output of this XBI detection tool is the misalignments resulting from comparing two alignment graphs. Moreover, the report also includes the two elements involved in the failure which will be the target elements for repair. The authors manually determined CSS properties that have the potential of dealing with the reported misalignment. Thus, one or more of these properties are assumed to be the root cause of the misalignment.

A search-based technique aims to explore a large solution space in a rewarding way in terms of efficiency or effectiveness. The search of XFix for an optimal solution used the Alternative Variable Method (AVM) [50, 52] which consists of an *exploratory* and a *pattern* step. In the case of XFix, the exploratory step makes a small increment to the CSS value in the positive or negative direction. Then, based on the feedback about the exploratory move from a *fitness* function, the pattern step makes exponential leaps in the same direction. The fitness function used by XFix is based on quantifiable pixel differences between the position and size of the elements in the reference browser and the browser exhibiting the failure. Specifically, they used a combination of differences in the size of the target element, location of the target element, and the location of the elements neighbouring to the target element.

Once the optimal value is found or the limit of attempts is reached, the CSS value concluded by the first phase of searching is considered to be a *fix*. This search is repeated for each element reported as problematic and all the CSS properties identified by the authors to have a potential effect on the XBI (*e.g.*, the `margin` and `padding` properties). This results in the output of multiple competing fixes that are able to repair the failure. Therefore, a second phase of searching is used to repair as many XBIs as possible using a subset of all the fixes. This is important since in combination, different fixes can either work better with other fixes, break other fixes, or create more XBIs. Thus, this second phase of searching for the best combination of fixes aims to minimize the XBIs reported by the X-PERT tool.

In their evaluation of XFix, the authors found that it is able to repair 86% of the failures reported by X-PERT and up to 99% of the human observable failures that are reported by X-PERT. The performance of this patch searching process ranged from 43 seconds to 110 minutes with a mean of 30 minutes. This cost was split 67% to searching for fixes and 32% to searching for the best combination of fixes. To save time, the authors suggested parallelization for future work but I believe that the position and size criteria as used in AVM can be better tuned. More specifically, the target position and size are

known in advance and potentially could be used while searching for the fix.

In semi-competition with the XFix tool is the X-Diag tool that was proposed by Xu et al. [135]. While XFix automatically generates CSS that is able to resolve the issue, the X-Diag tool automatically localizes the failure to a specific JavaScript DOM API call, CSS property, or HTML attribute. Nevertheless, the process of repairing the problem is still a manual task using the X-Diag tool.

To detect the XBIs that will be diagnosed by X-Diag, the authors used the X-Check tool. This tool utilizes an approach similar to X-PERT's in the detection of XBIs with the addition of monitoring JavaScript events using Mugshot. In the X-Diag tool, the Jalangi [108] tool is also used for stronger monitoring and localization of JavaScript code. These events are replayed in a reference browser and a testing browser to appropriately compare the page across different states of the application.

The first root cause analysis done by X-Diag is on calls made to the DOM during the execution of the application. This is achieved by instrumenting the JavaScript code using Jalangi. While code is executing, it is monitored for differences in the returned values during replayed of events or for calls that return an exception. The second level of root cause analysis is done on CSS properties. This is done by collecting all the declared CSS properties from the codebase and comparing them to the final values applied by the browser using the `getComputedStyle()` function across different browsers. Finally, the third level of root cause analysis is on the HTML element tags and attributes across different browsers. This is achieved by checking for syntax errors in the declaration of the HTML tag and by checking the value of HTML attributes across different browsers. It is worthy to note that the root cause analysis of the tool stops at the first level that is able to detect a difference. Thus only one root cause is reported.

During the evaluation of the X-Diag tool, they found it to have 89% precision and 88% recall in debugging visible failure when compared to human debugging. In terms of performance, the analysis revealed that the tool's runtime ranges from 3.34 to 11.83 seconds with a median of 7.95 seconds. In comparison, the majority of XBIs took a human over half an hour to debug.

Both of XFix and X-Diag specialize in helping the developer resolve the XBIs detected in the page. Although the authors of X-Diag suggest that the tool can be adapted to work with the X-PERT tool, it is still limited to fixing cross-browser issues as is the case for the XFix tool. For the X-Diag tool, the actual fix is still a manual process since it does not suggest a solution to the root cause. Meanwhile, the XFix tool is limited to suggesting CSS based patches and does not directly resolve any JavaScript-based faults in the code.

2.5.2 Repairing Internationalization Failures

For a web page that is designed to address different audiences, the language of the text-based content is translated to the language of the target audience. Because the page was originally designed using a particular language, the translated text may not fit as well as the untranslated text into the design of the page. This is due to the nature of languages as some translations are longer or shorter than the original language. This expansion or contraction can lead to text overflowing, element movement, text wrapping, and other general misalignments in the page. The layout failures that occur due to this change are known as Internationalization Presentation Failures (IPFs).

To automatically repair a layout failure that occurs due to a change in the language of the text-based content, Mahajan et al. [64] proposed a combination of a similarity

clustering approach and a search-based approach. This combination aims to find a set of appropriate values of predefined CSS properties that eliminate the IPF from the subject web page. This technique was implemented into a tool called IFix which uses the GWALI tool to automatically detect and report IPFs. More details of the detection technique followed by GWALI were described in Section 2.4.4.

To avoid a repair that is over-fitted to a target element, the repair should include related neighbouring elements. For example, a repair of a menu item that leaves it extremely larger or smaller than the other menu items is not ideal. Instead, all the menu items should give way to maintain the same style across all menu items. This is where IFix employed the clustering technique DB-SCAN [34] with a customized distance function. In the case of IFix, the distance function is used to measure style metrics instead of the location of an element. A benefit of using this technique is that each element can only belong to a single cluster and does not require the number of clusters to be determined beforehand. For repair, all the elements of the page are first clustered and then the clusters containing the GWALI reported elements are picked as the target for the repair. The style metrics defining the distance function include, as binary differences between any pair of elements, the height, width, edge alignment match (*i.e.*, top, right, bottom, and left), and tag name match. The style metrics also included, as a ratio, explicitly defined and matched CSS values and the XPath similarity using Levenshtein distance [54].

Before the search for the ideal values that will repair the failure begins, an initial population of repairs is generated using the target clusters. More specifically, the repair aims to resolve three CSS properties of each cluster which are the `width`, `height`, and `font-size`. The initial repair values are calculated based on the expansion rate of the text before and after translation. Then, the Alternative Variable Method [50, 52] is used to fine-tune these values. In addition to the AVM method, the values are mutated using a randomly picked value from the Gaussian distribution from the previous value. This is to explore areas of the solution space that may not be reachable using AVM alone. From this population of candidate repairs, a fitness function guides the search to pick the best repairs based on two components. First, the amount of layout change that quantifies the dissimilitude of two layouts is measured based on pixel differences that arise when comparing two *layout graphs*. The second is a measure of change due to applying the solution. Specifically, the squared percentage of change to each CSS value as compared with the original is used to penalize too much change in the layout. In combination, these two components help IFix pick the best set of candidate repairs. This process is terminated if the time expires or no improvements to the solutions can be made over 20 iterations.

The evaluation of IFix compared three different modes of the tool that reduce its main features. Thus, the full featured tool is compared to a mode that runs without clustering and another mode using an unguided random search. The results suggested that the full mode is far superior to the other modes averaging a 98% reduction in IPFs. Furthermore, it was found that the tool takes anywhere between 73 seconds up to 17 minutes to complete with an average runtime of 4 minutes. Noteworthy is that the clustering feature is a time-saving feature as well. Finally, a human study revealed that 64% of participants preferred the automatically repaired version of the page.

Although the results of IFix were largely promising, there were three limitations. First is the length of time needed to produce a repair which can take up to 17 minutes in certain cases. Second is the low quality of repairs because they were found to be less readable and less aesthetically pleasing. This is mainly because it focuses the repair on the cluster of elements involved in the failure. As a result, this can leave them in proportion to other elements in the page less aesthetically pleasing and less readable. To overcome

these limitations, Alameer et al. [1] suggested the usage of a constraint solver to repair IPFs as implemented in CBRepair.

The CBRepair tool works in three steps; extract relative relationships, convert relationships to constraints, and solve the constraints to repair the failure. Some example relative relationships between HTML elements are top-of, top-aligned, and contained-by. The first strategy to find the relationships that need to be modelled by the tool is based on the differences in alignment as reported by GWALI, thus the two failing elements are reported as out of alignment. The second approach is to find the lowest common ancestor, in the DOM, of the two elements reported by GWALI as failing. This is a simple heuristic to ensure that the style of similar elements is also modelled in the system. Overall, for the identified elements, the correct relative relationships are those formed in the reference untranslated page.

The second step of CBRepair is to convert the relative relationships into linear constraints. Here the constraints describe the correct relationship between two elements as dictated by the layout of the elements in the untranslated page. For example, if in the untranslated page the top edge of two elements (`e1` and `e2`) are aligned, the constraint that should be upheld in the repaired version would be `e1_top = e2_top`. Besides these relative alignments, the tool also generates constraints on the font size to prevent the system from reducing the text size which can cause readability issues. It does this by making sure the width and height of any text content are equal to a constant value in the repaired version as derived from the untranslated page. In other words, when adjusting an element size the repair must keep the original font size.

The variables of this constraint-based system will represent the CSS properties that have an effect on the size and position of the elements in the page as defined by W3C Box Model [14]. In this model there are four boxes which help the browser render the element starting with the innermost box for *content*, then a *padding* area, then a *border* area, and finally the outermost box the *margin* area. This results in 16 variables per element since each side of the boxes can be specified separately; top, right, bottom, and left.

The final step of CBRepair is to solve the constraints in order to produce a repair. Each repair represents new values for the content, padding, border, and margin that resolve the IPF. For this task, the tool uses Google's OR-Tools [86] to solve the constraints of the system. This results in multiple solutions that are able to resolve the issues but CBRepair must decide which one is the best. By using Linear Programming, the tool sets an objective function to pick a preferred solution from multiple solutions by favouring the ones that minimize any change to the original values based on the untranslated page.

The CBRepair tool was evaluated against the IFix tool for effectiveness, efficiency, and quality. The results showed that CBRepair reduces the failures by 65% while IFix outperformed with a 98% reduction. The root cause for the disadvantage was determined by the authors to be the lack of font size reduction in CBRepair. In terms of efficiency, CBRepair had an average runtime of 13 seconds while the IFix tool had a 5 minute average. Finally, in a human study that evaluate the quality of the repairs generated by both tools, the CBRepair repairs were determined to be more readable, more attractive, and more similar to the original page than the repairs made by IFix.

In addition to CBRepair's attempt to resolve the readability problem of the IFix generated repairs (due to the over reduction of font size), a newer version of the tool named IFix^{++} [65] aimed to do the same. Although CBRepair solved the problem by preserving the original font sizes, the IFix^{++} tool keeps it as a last resort in resolving the internationalization failure. Arguably, this is a better choice since CBRepair did not perform as good as the original IFix in terms of the number of repairs.

Because IFix^{++} is an improvement to the original IFix, much of the underlying technique is the same. This includes the usage of GWALI to generate failures and the usage of clustering to find elements that are similar in style. The improvements were mainly in adding new CSS properties during the search for a repair and the prioritization of properties other than the `font-size`. The original tool used only the `width`, `height`, and `font-size` properties in repairing the failure. In the new tool, the W3C Box Model [14] inspired the use of the `margin` and `padding` properties. This is an addition of eight properties split into the four sides of the Box Model; top, right, bottom, and left. Furthermore, they intentionally excluded `letter-spacing`, `word-spacing`, `font-weight`, and the four sides of the `border` property due to its limited effect in resolving an IPF.

Before the search begins, the new tool IFix^{++} initializes a value for these eleven CSS properties as done in the original tool. The only difference is that the new tool has eleven instead of three properties. Then as done in the first tool, AVM is employed to search for new values and mutation is also added to diversify the solution space. During the production of a repair, related sides of the Box Model are taken into consideration to maintain the aesthetics of an element. For example, if the left side is modified then the right side is also modified by the same amount. To prioritize the changes to each CSS property, a preference weight is used to multiply the percentage of change caused by each property. More specifically, the least weight was given to the `padding` and `margin` properties, then ten folds that weight was given to the `width` and `height` properties, then four times that weight was given to the `font-size` property. This leads the fitness function to prefer the repair with the least changes to the page using the lower weights. The search terminates after 20 iterations or if no improvements occur after 2 iterations.

The new version IFix^{++} was evaluated against the original IFix for effectiveness, efficiency, and quality. As the underlying technique in both tools is the same, both tools were able to reduce the number of IPFs by about 94% on average. In terms of efficiency, IFix^{++} required 23% more runtime than the original IFix. This is to be anticipated since the solution space is much larger using eleven properties when compared to only three properties. Finally, a human study determined that the quality of IFix^{++}'s repairs were more legible and visually appealing than the IFix repairs.

All three tools IFix, CBRepair, and IFix^{++} demonstrated success in repairing IPFs but had some limitations. The most obvious is their limited applicability to only internationalization failures. Furthermore, they all require the use of a reference page that is assumed to have a correct layout, namely the untranslated page. This makes it harder to adapt these tools to repair other type of failures. Nevertheless, the underlying problem solved using a cluster-based, constraint-based, or search-based can be retrofitted to other type of layout failures.

2.5.3 Repairing Mobile-Friendly Issues

There exists a set of mobile usability issues that arise due to the lack of planning for devices with smaller screens. These issues include improper viewport configuration, the page doesn't fit the limited width of the device, the text in the page is not readable, usage of incompatible plug-ins, and the inappropriateness of clickable elements in terms of size and spacing between the elements for a mobile device that mainly relies on touch screen input [11, 77]. It is critical for pages to be mobile friendly since the search engine requests that originate from a mobile device take into consideration the mobile-friendliness score when ranking the pages.

To resolve these issues, the page can be made responsively designed, use dynamic

serving, or make a separate page for mobile devices [83]. The responsive design solution provides all the rules and settings to the browser for it to execute the proper design for that device. Meanwhile, a dynamically served design is one where the appropriate design for the specific device is completed by the server. Finally, the last solution is to make an entirely separate page with its own URL for mobile devices. Although these solutions provide a page that is designed for a mobile page, they may still contain mobile-friendly issues.

To reduce the number of mobile-friendly issues, Mahajan et al. [61] proposed a technique to automatically repair three type of mobile-friendly problems. These were issues with *font sizing*, *tap target spacing*, and *content sizing* which should be limited to the viewport width. This leaves out two type of problems that are not handled by their approach. First, for viewport configuration problems, it is simply a one time task of adding the `meta` viewport tag to resolve the issue. Therefore, it is not a hard manual task to claim automation benefits from. The second type of problem not handled is related to browser plug-in issues like accommodating Flash-based content. This type of problem is rightly treated as out of scope since a layout modification will not resolve the issue. Mahajan and his colleagues proposed a technique, implemented in MFix, that builds a graph model of the layout and uses the constraints encoded into the model to improve the mobile-friendliness of the page while minimising the amount of layout change incurred on the page.

The technique implemented in MFix has three phases, *segmentation*, *localization*, and *repair*. In the segmentation phase, similar elements that require simultaneous and proportional adjustment of values are grouped into segments. To achieve this, they employed a clustering-based partitioning algorithm originally proposed by Romero et al. [97]. This algorithm starts with each leaf element of the DOM tree in its own segment and gradually merges two segments based on the number of hopes and a threshold to the lowest common ancestor. The algorithm terminates when no further mergers are possible. Thus, the results of this phase are segments that group elements in the page which should be repaired together.

The localization phase of MFix consists of two parts. First, it needs to localize the faulty HTML elements and then it needs to identify the faulty CSS properties. To identify the HTML element and the type of failure, MFix uses Google's Mobile Friendly Test (GMFT) [77] which localizes the problem to a few HTML elements and describes the type of problem. To target all the elements that need a repair, MFix uses the segment associated with the reportedly problematic elements. For example, if GMFT reports an HTML element with font size issues, MFix will find the associated segment and target all the text nodes of the segment for repair.

To localize specific CSS properties that require the repair and apply values to maintain the style dependencies between different elements, MFix uses a *property dependence graph*. This graph is built based on CSS inheritance rules and style similarity between different elements. The goal of this graph is to maintain the original proportion of values for stylistically similar elements which are the segments identified in the first phase of the tool.

To improve the mobile-friendliness of the page, MFix aims to reduce the amount of change incurred on the layout while making improvements to the mobile-friendliness score of the page. To do this, the tool builds two graph models from the page. The first one represents the relative alignments of segments and the other represents the alignments of elements in each segment. Then, the graphs of the original page are compared against the graphs of the repaired page to identify how many relationships have changed. Ini-

tially, the GMFT tool suggested improvements that are used by MFix with customized modifications. For example, if GMFT suggested that the font should be 16 pixels, MFix adds a constant equal to 14 pixels. To test more values, they used a Gaussian distribution around the original values. The goal of which is to make as many small changes as possible throughout the segment rather than a few large changes to a few elements. The property dependence graph is used here to apply different values to the target HTML elements based on the established ratios in the dependence graph. To score the friendliness of the page, the APIs of the GMFT tool and Google's PageSpeed Insights (PSI) [38] tool are used. Finally, a media query is added to limit the application of the patch to the viewports associated with mobile devices. For performance, the tool used Amazon Web Services to parallelize these computations.

In the evaluation of MFix, they found that it was able to improve mobile-friendliness by 33% on average with 36 out of 38 subject web pages passing GMFT after an MFix patch is applied. The MFix tool will cost anywhere from 2 minutes to 10 minutes per subject with an average runtime of 5 minutes per repair. The quality of the repairs was judged in a human study which revealed that participant preferred 26 out of 38 repairs over the original page. Overall, the study also revealed that the repaired versions were more aesthetically pleasing and 17% of the repairs had improved readability.

One limitation of MFix is that it does not consider the structural symmetry of the page when making repairs. As such, Azmain and Ganguly [7] proposed and evaluated improvements to the MFix tool. The first upgrade to the tool is in its segmentation phase. During this phase, the authors opted to use the VIPS [23] algorithm which combines DOM structure information with visual cues to segment the page. According to the authors, the tool is also able to detect symmetry problems as well as resolve them. This process is largely unclear but involves using balance and proportion metrics. Where the balance score measures the distance to the closest headline and the nearest call to action button among other measurements. The proportion metric is calculated by dividing the height over the width of each element, then they are averaged for the right side of the page and the left side which are later subtracted from each other to reach a final score for the page. Then a search-based approach is employed to balance the symmetry score and the mobile-friendliness score.

Unfortunately, the approach of Azmain and Ganguly is not clearly described but the examples showcased illustrated impressive improvements over the MFix tool. As a result of their own evaluation, 88% of human participants preferred the symmetric solution over the MFix repair. The validity of their work is not clear.

Another work by Le-Cong et al. [53] also saw the opportunity to improve MFix by adding two new search-based alternatives that take into account usability and aesthetics. In their work, they use the same segmentation and localization features of MFix and the novelty of their work is in the repair phase of the tool. By using the Particle Swarm Optimization (PSO) [51] algorithm over a random search, it can improve the values used for the repair over multiple iterations. The authors used this algorithm because of its "high quality" solutions but also noted that it can come at a high cost in terms of runtime. Therefore, they also studied the use of the Tabu [37] search algorithm to optimize for time. For both algorithms, the initial solution values are the ones suggested by the GMFT tool.

When the Le-Cong version of the tool was compared with the original MFix, the PSO algorithm significantly outperformed both the Tabu algorithm and the MFix tool in improving the usability score. Moreover, both PBO and Tabu search algorithms significantly outperform MFix in aesthetics. More precisely, PSO was better than Tabu for 36 out of 38 subjects in mobile-friendliness and 30 out of 38 in terms of aesthetics. Suggesting that

it is a better fit than the original MFix tool.

Although MFix and the upgraded versions of it do not require or use a correct layout as a reference to detect presentation failures, they did rely on Google's PSI and GMFT to score the page. This can be considered an advantage but it will limit its usage to the three type of mobile-friendly problems identifiable by these tools. Thus it is not as beneficial in repairing cross-browser issues, internationalization failures, or even responsive design failures.

2.5.4 Generic Layout Repair

The approaches that I reviewed so far can automatically repair a presentation failure but they are specialized in repairing either cross-browser issues, internationalization failures, or mobile-friendliness issues. This specialization restricts their usefulness to resolve other types of failure. Nevertheless, there were also public tools and some literature that was not concerned with the type of failure, but rather a general approach to repair or mitigation of presentational defects.

A generic approach to verification of the layout and repairing any violations was presented by Jacquet et al. [48]. Their approach uses Mixed Integer Linear Programming (MILP) and IBM's CPLEX software to solve constraints about the page in order to generate a repair. This is a declarative system which assumes that the constraints are known in advance and manually identified. The overall aim of the system is to produce a"hot fix" (a repair that suppresses the failure) in less than two seconds. To achieve this, the type and number of constraints are reduced, the number of elements that need to be looked at by the solver to generate the solution are also minimized, and no re-rendering of the page is used to test the generated repair.

In this Jacquet system, four type of constraints are applied. These were *alignment*, *inclusion*, *disjointness*, and *non-decreasing sizes* constraints. The layout constraints like alignment, inclusion, and disjointness are expressed on pairs of elements. They include horizontal alignment, vertical alignment, containment (inclusion), and separation (disjointness). For the solver not to go for the choice of setting the width or height of an element to 0, the non-decreasing size constraint is added. Moreover, this constraint adds a time-saving property because not all elements should be affected by the change. Instead, part of the DOM tree is affected which is referred to as the *zone of influence*. More precisely, the parent and siblings are affected recursively in one direction up the DOM tree.

To prevent the solver from producing a layout that is drastically different from the original layout, an objective function is also defined for the solver. To represent the amount of change the layout is undergoing, the change in position of each element is calculated on the x and y axis. The overall change in the layout is the cumulative change in the position of all elements. Thus, for the solver to reduce the amount of change to the layout, this value is minimized by the objective function.

The Jacquet prototype tool has two phases, the *detection* and *correction* phase. In the detection phase, the DOM is traversed to produce a unique identifier for each element, their unique positions in the coordinate system, and the set of constraints for that elements. This is used to generate an input model for the solver to solve. In the correction phase, the resulting solution outputted by the solver is applied to the layout and expected to repair the issue without any feedback. To achieve this, the `position` CSS property is set to the `absolute` value in order to force the element to be fixed into the position generated by the constraint solver relative to another element with the same value for the property.

In order to expand an element as part of the solution, the `width` and `height` properties are modified while taking into consideration the `padding` and `border` size.

In their evaluation of the Jacquet tool, they used real-world pages and developed a helper tool PageGen [88] to create synthetic DOMs from a real DOM with the ability to create synthetic presentation failures. This helped them generate 100 synthetic DOM trees with failures and variable number of elements (2 to 10,450). The purpose of which is to stress the tool in order to evaluate its efficiency. The overall results showed that it can resolve the issues in a matter of seconds but it comes with critical limitations. First, the input requires the developer to manually provide complete constraints which ensure that their satisfaction equates to the correct layout. Their "hot fixes" are also limited since they modify the original code drastically in a way that does not work for any other viewport, browser, or translation of the page which makes the patch impractical. Furthermore, the repairs can never reduce the sizes of elements which is not a realistic criterion for a repair. Without this constraint on size, the performance advantage is lost.

A more powerful framework that uses constraint solving to verify the correctness of a layout and also repair the violation raised was presented by Panchekha [89]. Their framework, Cassius, is a formalization for a considerable portion of CSS semantics. It can efficiently reduce the problem to the theory of quantifier-free linear real arithmetic to allow the use of Satisfiability Modulo Theories (SMT) solvers. It can be used to build other tools that automate the verification, debugging, and synthesis of CSS. Essentially, it is a declarative implementation of a browser's layout engine. Compared to the standard layout engine, Cassius can similarly compute a layout from an HTML document and a CSS stylesheet. With superiority, Cassius can also compute a CSS stylesheet from an HTML document and a mock-up of the desired layout.

As proof of the capabilities of Cassius, the authors built a verifier, debugger, and CSS synthesizer in a matter of a few days per build. An example job for the verifier is to ensure that a given layout contracts or expands as intended for all viewports. The debugger can then be used to localize the violations, as identified by the verifier, to parts of a stylesheet. Then, the synthesizer can automatically repair the problematic constraints to enforce the intended properties. As mentioned previously, the synthesizer can also work in the opposite directions to produce a stylesheet.

During the evaluation of the Cassius framework, it is first measured with 2075 tests relevant to the conformance to standards. During this testing, the Mozilla Firefox browser is used as an oracle since it is known to pass these tests. Only six failed due rounding errors on the part of Firefox, which are technically not observable. The framework was also tested to see if it rejects invalid rendering which resulted in a 99.3% success rate. Only failing due to the CSS standards that were not implemented in Cassius like calculating the font metrics which determine the height of a text-line.

For evaluation of the three tools built using Cassius, they used five famous web pages including the Amazon web page. To evaluate the verifier, they used it to verify that no text box or link element `<a>` overlap for any viewport in the range 800 to 1920 pixels. For evaluating the debugger, they added non-satisfiable assertions which negate the position and size of randomly selected boxes. To evaluate the synthesizer, random expressions of some rules were purposefully left with primitive holes. These experiments revealed that the verification took 2 to 12.2 seconds to complete, debugging took 0.7 to 5.5 seconds, and synthesis took, in the best case, minutes to complete 25 expression holes. The main reason for the delay in synthesis is due to usage of CSS selectors that match many elements. Another finding is that the debugger was able to narrow the problem to anywhere between 2 to 7 CSS rules and 4 to 10 CSS properties.

There were also more generic approaches to preventing layout failures from happening, in a sense repairing the fault before it propagates to a failure. Generally, they help prevent any type of failure but are better fitted to preventing cross-browser issues. These are mainly split into validating tools, CSS normalizing libraries, or CSS resting libraries. First, the HTML and CSS validation tools [41, 84, 102, 116, 117] aim to remove syntax issues, based on W3C standards, from the codebase before it is ever deployed. If they are not corrected before deployment, different browsers may render the page differently depending on how they handle the specific syntax error. Then, some libraries that aim to normalize the CSS [85] so that it is applied equally in all browsers. This approach overrides the default styles set by a browser with a new common style. Finally, there are CSS resetting libraries [20, 21] that aim to unstyle the browser by removing the styles usually added by the browsers. These basic approaches are undoubtedly helpful in preventing layout issues from occurring.

2.6 Concluding Remarks

Since the development of the web, the need to test a layout for failures started becoming apparent. The growing features of the web and growing dependency on web services made testing for presentation failures a worthy problem to solve. Furthermore, testing only reduces the problem of finding failures in the page. Without repairing these failures, testing serves no purpose. In this chapter, I have reviewed what has been proposed to test and repair the layout failures that occur in a web page.

Many techniques, implemented into tools, have been proposed in the literature and successfully shown to automatically test the layout of a web page for presentation failures. They were mainly divided into specialized testing for cross-browser issues, regression failures, internationalization failures, and responsive design failures. First of which, cross-browser testing, aimed to identify failures in the layout when an alternative browser is used. Regression testing aimed to automatically detect the failures in the layout based on an older version of the layout. Internationalization testing aimed to test the layout for correctness after the text-based content of the page is translated using the untranslated page as a reference. Most relevant to my work in this thesis, automated responsive design testing was developed to detect layout failures that occur between alternative viewports width.

Although an automated technique to test for responsive layout failures exists [124–126, 128], it leaves two manual tasks for the developer to complete. First, since the testing technique does not use any graphical information about the layout to detect the failures, it may report DOM structural issues that are not observable in the rendered page. These non-observable issues are less of a priority in terms of repairing them. To find the real observable failures, the developer must manually inspect these reported failures and manually filter out the ones requiring a repair. This leads me to the second task that must be manually done, the repair of these failures. As described in this chapter, the development team may not have the luxury of time or possess the skills needed to fully analyse the root cause and manually repair the failure. Fundamentally, these two limitations are the basis of the work of my thesis. First, to automatically classify the reported responsive layout failures as observable, non-observable, and false positive and second to develop a technique that automatically repairs these failures. In the next chapter, I begin with the problem of automatically classifying non-observable failure reports.

3

Classifying Non-Observable Issues in Layouts

Although the state of the art tool for responsive layout testing, called ReDeCheck, assists a developer in automatically finding potential presentation failures, many of the reported failures are not visually evident in the layout. Some of which may be false positive reports while the others are only a problem in the underlying structure, the DOM, of the layout. These structural issues that are not reflected in the rendered layout, known as non-observable issues [124], may require the attention of the developer but are less important than the observable failures. To distinguish these apart, the developer using ReDeCheck must manually scrutinize many reports generated by the tool to identify observable from non-observable failure reports. This is a potentially time consuming, error-prone, and in some instances a subjective task. This chapter presents a technique that automatically analyses images of the layout to identify the non-observable failure reports.

The technique that I present in this chapter is able to automatically classify each report generated by ReDeCheck into either a true positive, non-observable issue, or a false positive report. Since there were only three types of presentation failures that were associated with non-observable issues in the previous research done by Walsh et al. [124], the technique presented in this chapter covers these three out of the five that the ReDeCheck tool is capable of reporting. They are known as *element collision*, *element protrusion*, and *viewport protrusion* responsive layout failures. To classify them, the technique works by pealing back graphical layers of the layout, in the live web page, and compares specific pixels across different layers to reach a classification of the failure report. For evaluation of the approach, I used the manual classification made in the original research of Walsh et al. as a baseline to evaluate the effectiveness and efficiency of the automated approach.

The chapter begins by explaining the problem followed by a brief overview of background information. Then the technique is explained and empirically evaluated.

The key contributions of this chapter are:

1. A technique to automatically classify reports of non-observable issues.
2. An empirical study of manual and automated approaches using 20 web pages. Demonstrating that non-observable issues can be automatically and effectively classified.

3.1 Motivating the Research

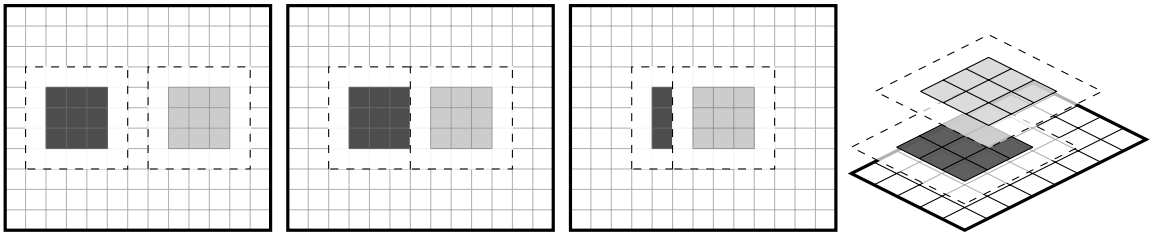
A developer of a responsively designed web page must plan and build the structure of a web page, that will later form the layout, in a way that is ideal for display in the available width of the screen for any given device. Due to the large number of devices with different screen sizes that may visit the web page, manually inspecting the layout to make sure that it is presented as originally planned for all these devices is a difficult task. To be more precise, it is the space available for rendering the web page within the browser window, referred to as the *viewport*, is what the developer must take into consideration. While inspecting how the layout adapts to different viewport widths, the developer is testing the layout for a failure to display as expected. This is referred to as a *presentation failure* and more specifically for responsively designed web pages, it is known as a *Responsive Layout Failure* (RLF).

To assist the developer, the tool ReDeCheck automates the testing of a layout across multiple viewport widths in order to raise concerns about potential presentation failures. Although beneficial, the tool uses no visual information from the page to find these failures. Instead, it relies on the underlying structure of the page to detect these failures and assumes that the rendered layout reflects the underlying issue. Due to this fact, there are three possible classifications that a report may fall under. The first and most helpful outcome is a *true positive* failure that is observable in the rendered layout. The second is a *non-observable issue* (NOI) for reports with no visual evidence of a problem but do have an underlying issue. Finally, a *false positive* report is one where no issue is evident visually nor is it evident in the underlying structure of the layout. To get a sense of how many NOIs are to be expected, during a manual classification made by Walsh et al. [124] of 118 reports from 26 subject web pages there was a total of 83 NOI reports. That is to say, if the developer was more concerned about true positive reports, 70% of the reports are not a priority to investigate or repair.

To aid in visualizing the NOI problem, four wireframes are presented in Figure 3.1 that depict alternative layout positions of two HTML elements. In all four wireframes, the background of the page is coloured in white. Meanwhile, the elements are coloured in a light and dark grey colour to distinguish them apart. Moreover, both elements have a white border that matches the background colour. In part (a) of the figure, the elements are not overlapping each other and are one pixel apart from each other with no layout failure. In part (b), a layout failure occurs because the elements are now slightly overlapping when they should not. Since the overlapping pixels are coloured in white, the overlap is essentially non-observable. In another scenario presented in part (c), the area of overlap between the two elements is much greater and is observable due to the colour differences noticeable within the overlapping area. To clarify this further, part (d) of the figure depicts a layered view of the overlap.

Considering the different layouts presented in Figure 3.1, the ReDeCheck tool would not distinguish between the non-observable and observable overlap scenarios showcased in parts (b) and (c) and would generate a report in either case. One of the algorithms implemented into the tool infers a problem in the structure of the layout if at some viewport width the elements were not overlapping but in a narrower viewport, the tool finds that they do overlap. This algorithm detects one of five types of responsive layout failure known as an *element collision*. It is left to the developer to manually investigate all the reports generated by the ReDeCheck tool to weed out all non-observable issues that lack sufficient evidence to justify a repair.

Although the NOI structural issues reported by ReDeCheck are not a priority for repair, they could require the attention of the developer. Ideally, they should not be



(a) No overlap. (b) Non-observable. (c) Observable. (d) Layered view.

Figure 3.1: Wireframes depicting the observable and non-observable overlap of two HTML elements. The elements are coloured in light and dark grey with a white border that is the same colour as the background. In part (a) the elements are not overlapping while in (b), (c), and (d) they are. As only pixels of the same colour overlap in part (b) it is non-observable while in (c) it is. Part (d) shows a layered view of the overlap.

dismissed as false positives. A false positive report is expected to remain as such even if the developer decided to change the colour of elements associated with the false positive report. On the other hand, a simple colour modification of the elements may change an NOI classification into a true positive one. Therefore, the NOI reports could require a repair but are of less priority than the observable issues raised by the tool.

An important consideration is the relationship between the viewport size and the severity of the presentation failure. Any failure reported by ReDeCheck conveniently includes the range of consecutive viewports $\{fail_{min}..fail_{max}\}$ where the failure manifests. This range, referred to as the *failure range*, starts at the $fail_{min}$ viewport which is the minimum point of the range and ends at the $fail_{max}$ viewport is the maximum point of the range. At the $fail_{max}+1$ viewport, the viewport is big enough to accommodate the layout and does not have the same failure. While at the $fail_{min}-1$ viewport, the same failure is mitigated as a result of the layout adjusting to either a developer made rule or the browser default behaviour. One exception is when the $fail_{min}-1$ viewport was not tested and therefore the failure manifests there as well. Within the failure range, the severity of the failure is expected to become worse as the viewport size becomes narrower. Going back to the example layouts presented in Figure 3.1, this would mean that it could be the case that part (b) is from the $fail_{max}$ viewport where the collision starts to occur. Furthermore, it would also mean that part (c) is from a viewport nearer to $fail_{min}$ of the failure range as the collision becomes more severe. This means that a manual investigation of two different viewports from the same failure range would lead to two different outcomes about the same failure.

To alleviate the burden on the developer, automation can be used to weed out non-observable issues and therefore improve the usability of the ReDeCheck tool and help pave the way for automated repair of true positive failures. Since multiple viewports may need to be investigated for a large number of reported failures, this can be a time consuming and error-prone task using the manual approach. In this chapter, I present an approach to automatically peek into multiple graphical layers of a layout in order to classify non-observable issues for three responsive layout failure types that were associated with NOIs by Walsh et al. [124]. But first, I will give an overview of these three failure types and how they are detected.

3.2 Detection Prior to Classification

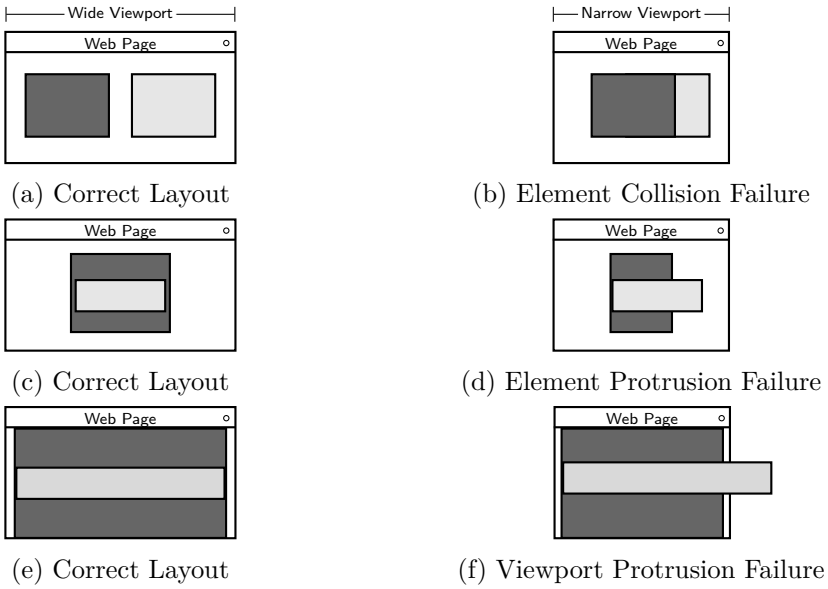
Before I present my automated technique to classify non-observable failures in a web page, it is essential to understand the different types of failures that are subject to the classification. For responsively designed web pages, the tool ReDeCheck can automatically detect five types of presentation failures. It is able to achieve this by first visiting a range of viewport widths to extract information about the layout of each viewport. Then, it uses the information extracted to build a graph-based model of the web page called the Responsive Layout Graph (RLG). Based on the coordinates of each element and relative to the others, the RLG model groups elements into a common container called the *parent*. Furthermore, other alignments are determined between elements that share the same parent, called *siblings*. One example of an alignment is whether or not the coordinates of two siblings *overlap* each other. Using the RLG model, the ReDeCheck tool uses a set of algorithms to detect five types of failures specific to responsively designed web pages that are referred to as Responsive Layout Failures (RLFs).

The intuition behind the detection algorithms of ReDeCheck is based on the changes observed between consecutive viewport widths as the size decreases. At the wider viewport width, the elements of the page may have the space needed to be presented as intended by design. Meanwhile, at a narrower viewport, the space is constrained and hence may force certain HTML elements in the layout into a failed state. With the improperly laid out elements identified at the narrower viewport, the only step left for the algorithm to do is to measure the range of viewports where the elements are improperly positioned. From the five types of RLFs detectable by ReDeCheck, only three types were found to output reports of non-observable issues during the manual classification made by Walsh et al. [124]. These types, which are the focus of this chapter, are the *element collision*, *element protrusion*, and the *viewport protrusion* failures.

Figure 3.2 depicts six wireframes showing three correct layouts paired with three improper layouts at a different viewport showcasing the three failure type. Each of the wireframes illustrates a web page with two HTML elements coloured in either light or dark grey. The correct layouts occur at some wide viewport width and are showcased in parts (a), (c), and (e) of the figure. While the wireframes in parts (b), (d), and (f) showcase the same HTML elements in a failed state of the layout at some narrower viewport width. The algorithms implemented in ReDeCheck use this difference in the relative position of the two elements across the wider and narrower viewport to detect and report element collision, element protrusion, and viewport protrusion failures that are explained next.

element collision – for elements that are laid out next to each other without any overlap by design, they are expected to remain separated regardless of the viewport size. An example of this layout can be seen in Figure 3.2 (a) where the two elements are laid out side by side. The space available to maintain some distance without any overlap becomes less possible as the viewport width decreases. If the overlap occurs in the layout of a narrower viewport, as seen in part (b), an element collision failure is detected due to the new overlap of the siblings. The collision is not solely a visual problem, interacting with any functionality associated with the element covered by the other is no longer possible with the hidden portion. An element collision report generated by ReDeCheck would include the failure type, the XPaths of the two elements in collision, and the failure range spanning the observed overlap.

element protrusion – While building the structure of a web page that makes up the layout, some elements are created to organize and contain other elements of the page. For example, the dark grey coloured element in Figure 3.2 (c) is designed to contain the lighter



(a) Correct Layout (b) Element Collision Failure

(c) Correct Layout (d) Element Protrusion Failure

(e) Correct Layout (f) Viewport Protrusion Failure

Figure 3.2: Three wireframe examples of the types of RLFs reported by ReDeCheck and automatically classified by Viser. The figures to the left-hand side illustrate a responsively designed web page with a correct layout while the figures to the right-hand side depict a type of responsive layout failure.

grey element. As the viewport width becomes smaller, some containers may not be able to accommodate all of the elements that they contain. As a consequence, one or more of the contained elements may protrude the boundary of their intended container. This is the case for the light grey element depicted in the layout shown in part (d) of the figure. In this scenario, an algorithm implemented in ReDeCheck would observe that the light grey element overlaps an element that was its container at the wider viewport but is no longer fully contained by it at the narrower viewport. Therefore, it would generate an element protrusion report that includes the failure type, XPaths of the elements involved, and the failure range where the overlap was observed.

viewport protrusion – in a responsively designed web page, it is allowed and expected for the content of the page to exceed the viewport height and therefore the user would vertically scroll to view more content. On the contrary, content is not allowed to exceed the viewport width. For illustration, Figure 3.2 (e) shows that the light grey element is correctly contained within the available viewport width and its parent. Nevertheless, as the viewport width becomes smaller, the available horizontal space may not be sufficient enough to accommodate elements that are laid out horizontally by design or a very wide element. In my simplified example in Figure 3.2 (f), the extra-wide light grey coloured element exceeded the viewport. In reality, the portion exceeding the viewport would not be visible to the user except after scrolling. In a worst-case scenario when there is no scroll bar available, the element would be unreachable. With a high degree of similarity to an element protrusion failure, a viewport protrusion failure occurs when the element exceeds the boundary of the viewport. A specialized ReDeCheck algorithm uses the main `body` HTML element as the boundary of the viewport. The ReDeCheck tool would report the failure type, the two elements involved, and the range of viewports where the element is protruding the viewport.

3.3 Automatically Classifying Non-Observable Issues

Given a set of presentation failures reported by the ReDeCheck tool, the approach that I will present in this section is able to distinguish and classify the solely structural issues of the layout that are not observable visually. Since the underlying technique used in ReDeCheck detects presentation failures between pairs of elements, the classification technique that I developed investigates only the two elements reported by the detection tool for failure. I implemented this approach into a tool named "Viser" (VISual VerifiER) that classifies each report generated by ReDeCheck into either a non-observable issue, true positive, or a false positive report. I begin the section with a summary of the approach. Followed by a comparison between the manual and the automatic approach before I explain the details behind my automated RLF classification approach.

3.3.1 Summary of Approach

Since ReDeCheck only uses the structure of the layout and does not use any visual information about the web page to detect failures, it may report issues that are non-observable to the human eye in the layout. To automate the classification of these failures, I implemented a tool, Viser, that peeks into multiple graphical layers of the layout in order to compare them for an invalid colour change. This analysis takes place in a region of the layout that is identified using the positions of the faulty elements reported by ReDeCheck. I refer to this region as the Area Of Concern (AOC). When handling any failure report, the first step to classification is to identify the AOC. Then the tool captures multiple snapshots of the graphical layers within the AOC. These layers are uncovered by alternately hiding one or both of the faulty elements, using CSS, from view to reveal the layer rendered behind. The intuition is that an element that introduces content should not be unintentionally overwritten by other content in the page. Here, the introduction of content can be simplified to a colour change of a specific pixel in the AOC made by a different layer. By comparing pixels across multiple layers in the area of concern, Viser is able to distinguish the reports of non-observable issues from the true positive observable presentation defects in the layout.

3.3.2 Classifying Presentation Failures

The failures reported by ReDeCheck can be one of three classifications which are a True Positive (TP), Non-Observable Issue (NOI), or a False Positive (FP) report. In the case of a false positive report, the ReDeCheck tool is simply wrong. Meaning that an inspection using the DOM of the layout would not be able to corroborate the same structural defect raised in the report. Obviously, there should also be no visible defects associated with the reported elements since the structural integrity of the layout is in order. In the case of a non-observable issue, an inspection of the layout would corroborate the structural problem with the reported elements but with no recognizable presentation defects in the layout. Finally, a true positive report would have both a structural and a visible defect in the presentation of the layout.

The fundamental steps needed to classify a reported presentation failure are common to both the manual and the automated approach. Figure 3.3 compares both the automated process implemented in Viser and the human-based manual process. Both processes start after the developer runs ReDeCheck on a web page to detect and report presentation failures. Assuming failures are found, each report will include information about the failure type, the XPaths of the HTML elements involved, and the failure range of viewport

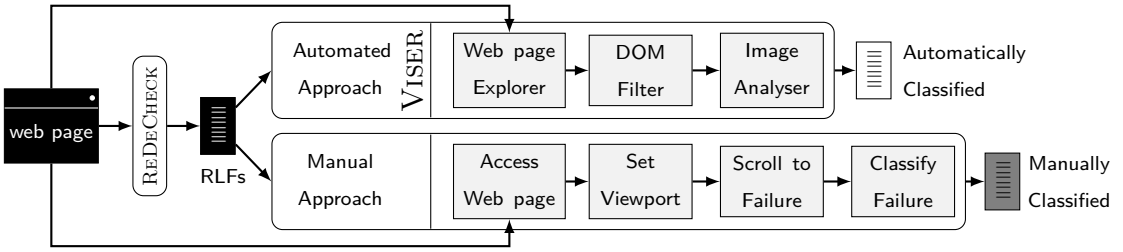


Figure 3.3: The high-level architecture of the Viser tool for the automatic classification of layout failures. Along with the external input sources, this figure also shows the alternative manual process steps that require a human expert.

widths where the failure occurred in the form of a lower bound and an upper bound $\{fail_{min}..fail_{max}\}$. To classify the reported failures, the first step is to read and understand the report. Then the web page must be loaded into a browser to begin the investigation. To view the presentation failure, the viewport width of the browser must be set to one of the viewports reported in the failure range. If the content of the page exceeds the viewport size, searching the page by scrolling to the location of the faulty element may be required. Finally and most significantly, the structural and visual analyses of the layout for the alleged failure are required in order to reach a classification verdict.

The automated approach to classifying a reported failure using Viser starts with the *Web page explorer* component as seen in Figure 3.3. This stage is responsible for launching the browser, loading the web page, setting the appropriate viewport width, and locating the faulty elements in the layout. Then Viser investigates ReDeCheck's report by examining the DOM of the layout during the *DOM Filter* stage. This investigation will either corroborate the structural problem or reject it and classify the report as false positive. Any report of a presentation failure that passes this stage is either a true positive report or a non-observable issue. For this final distinction in classification, the *image analyser* component goes to work.

The key feature behind Viser's automated classification is the image-based analysis of the reported presentation failure. This analysis involves investigating a specific region of the web page associated with reported failure that I refer to as the Area of Concern (AOC). The AOCs are rectangular areas derived from the coordinates of the elements involved in the reported layout failure. On a graphical level, the tool expects any associated presentation defect in the layout to be in the AOC. The goal of the analysis is to conclude if the graphical layers within the AOC have an observable true positive layout failure. For example, if the report involves a transparent element with no content but is not in its proper structural position in the layout, the analysis should yield an NOI classification. In the following section, I describe how Viser identifies these AOCs for different RLF types followed by the details of how the image analysis and final classification are completed.

3.3.3 Identifying the Areas of Concern (AOCs)

Calculating an AOC depends on three sources of information. They are the coordinates of the faulty elements, the layout *scenario* of the faulty elements relative to one another, and the type of responsive layout failure being handled. First, the coordinates of the elements involved in the failure are retrieved using the DOM. Logically, any presentation defect caused by the faulty elements is expected to be apparent in the area where the elements are rendered in the page. Therefore, the AOC is limited to the area spanning

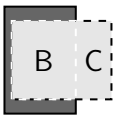
	Layout scenario		
			
	Contained	**Overlapped**	**Separated**
Element Collision	A	B	-
Element Protrusion	-	B, C	D
Viewport Protrusion	-	B, C	D

Figure 3.4: For purposes of identifying the area of concern (AOC) that requires analysis in the layout, the positioning of any two elements relative to one another are generalized into three scenarios. The scenarios *contained*, *overlapped*, and *separated* are illustrated using two elements coloured in light and dark grey. For each scenario, the AOCs are identified with the letters A, B, C, and D for the three failure type.

the coordinates of both elements or a portion of it.

The layout scenario indicating how the two elements are laid out relative to one another is the second source of information needed to calculate an AOC. For this, the positions of both elements are generalized into three layout scenarios as seen in Figure 3.4. They are the *contained*, *overlapped*, and *separated* layout scenarios. In all three scenarios of the figure, the two faulty elements are coloured in a light and a dark grey to distinguish them apart. In the contained scenario, the coordinates of one element fully lie within the coordinates of the other. Alternatively, in the overlapped scenario, there is only a partial overlap of both elements and no single element fully contains the other. Finally, in the separated scenario there is no overlap of the coordinates.

Along with the coordinates and the layout scenario, the final information needed to calculate the AOC is the failure type. Depending on the type and the scenario, as seen in Figure 3.4, Viser calculates an AOC differently. Since there are three scenarios and three types of failures, there is a total of nine possible outcomes that are represented as a matrix in the figure. Looking at an element collision in the contained scenario first, The AOC is equal to the coordinates of the contained element represented with the light grey colour and labelled A. Again with an element collision but in an overlapped scenario, the AOC is restricted to the coordinates of both elements that are overlapping and labelled B. Finally, for an element collision report in the separated scenario, there is no AOC to analyse since the elements are not in collision. Instead, the report is rightly treated as false positive.

Since both an element protrusion and a viewport protrusion involve an element protruding its container, an AOC is calculated in the same matter for both type of failures. For these two type of failures in all three scenarios, the dark grey coloured element in Figure 3.4 depicts the container element while the lighter grey element depicts the reportedly protruding element. In the case of a contained scenario for these two type of failures, there is no protrusion occurring by the definition of the scenario. Hence, this will be correctly treated as a false positive report. In the overlapped scenario, two AOCs are calculated to carry out a different analysis depending on the AOC. The first is the area labelled B where the coordinates of both elements overlap. The second AOC in the overlapping scenario, labelled C, is equal to the non-overlapping coordinates of the protruding element. Finally, for the separated scenario, the AOC is equal to the entire coordinates of the protruding element.

3.3.4 Analysing the Areas of Concern

Once an AOC is calculated by VISER for a failure report, the *image analysis* component of the tool peeks into the underlying graphical layers of the AOC to determine if the content of an element has been overwritten by other content in the page or if it was written out of position. Since the elements of a layout are stacked on top of each other in the rendered web page, CSS can be used to reveal an element by changing the stacking order. An illustration of the stack or layers was presented earlier in Figure 3.1 (d). The actual image-based analysis of the presentation failure is made on snapshots of these layers captured by the VISER tool. More specifically, the pixels in the AOC are compared for a colour difference across different layers. If the tool finds a difference between the graphical layer, the failure is deemed observable and classified as a true positive report. Otherwise, the failure is a non-observable issue.

To reveal an underlying graphical layer in the layout of the web page, the VISER tool changes the `opacity` property of the faulty elements described in the report. This CSS property once applied to an element using the value `0`, would make the element and all of its descendant elements invisible. Thereby, revealing the other elements that were stacked lower in the rendering stack of the layout. A special characteristic of this property is that the position occupied by the element is reserved for it regardless of its visibility. In other words, changing this property does not change the layout since the space is always reserved for the element. Any change to the layout may compromise the integrity of the classification. An added benefit of using the `opacity` property to reveal hidden layers of the layout is that it is a browser-independent method.

To analyse the presentation of the web page, snapshots must be taken of the layout. Unfortunately, the canvas where the layout of a web page is rendered by the browser is limited to the viewport size. Therefore if the content of the web page is larger than the viewport size, only a portion of the layout equal to the size of the viewport will be visible at any one time. Changing the viewport size in order to view more content may affect the size and position of elements and therefore compromise the integrity of the classification due to a change in the layout. Therefore, capturing a snapshot of an AOC that exceed the size of the viewport requires multiple steps. The first step is to scroll the page and capture the required portion of the layout alternatively. Then, the individual portions are assembled into their proper positions by VISER.

There is a challenge that arises when using scrolling to reach different parts of the layout. When the maximum scrolling distance allowed by the browser is reached for that specific page, any portion of an element that goes beyond this distance is not included in the snapshot. To evaluate the failure properly, the full AOC must be included in the snapshot. Because basic scrolling will not resolve this case, the VISER tool takes a *best effort* approach to modify the position of the element in order to bring it into view for the snapshot. By calculating the coordinates value that goes beyond the maximum scrolling distance, the tool can pull the element only as much as needed for the snapshot. To achieve this, the tool sets the margin properties of the element to a negative value and therefore offsetting its original position to be fully within the rendered page. Importantly, the portion of the element that is visible using scrolling alone is captured and evaluated first before applying the best effort approach that modifies the layout only when necessary.

Prior to analysing the AOC for a presentation failure, the VISER tool automatically filters out the false positive reports. To do this, the tool relies on the semantics of the failure type to corroborate the report at the time of classification. If it cannot be corroborated, then the report is classified as a false positive. The tool accomplishes this by first setting the viewport to a viewport where the failure was reportedly found. Then the coordinates

Algorithm 1 Top-level VISER algorithm

INPUT: Two HTML elements, *back* and *front*, and the failure type, *ft*.
OUTPUT: TP if the RLF is deemed observable, NOI if it is not.

```
1: procedure VISER(back, front, ft)
2:     scenario ← GETSCENARIO(back, front)
3:     if scenario = contained then
4:         AOC ← GETCONTAINEDAOC(back, front)                ▷ AOC = A (Figure 3.4)
5:         return THREELAYERSANALYSIS(back, front, ft, AOC)
6:     if scenario = overlapped then
7:         AOC ← GETCONTAINEDAOC(back, front)                ▷ AOC = B (Figure 3.4)
8:         return THREELAYERSANALYSIS(back, front, ft, AOC)
9:     if scenario = separated then
10:        AOC ← GETDETACHEDAOC(back, front)                 ▷ AOC = D (Figure 3.4)
11:        return TWOLAYERSANALYSIS(front, AOC)
```

of the elements involved in the failure are retrieved from the DOM of the web page. Depending on the semantics of the failure type, the coordinates are assessed for truth. For an element collision type of failure, it is classified as a false positive report if the coordinates of the two elements do not overlap. For an element protrusion or a viewport protrusion failure, a false positive classification is reached when the coordinates of the reportedly protruding element are found to be within the coordinates of the container element. These false positive layout scenarios for each failure type do not require an AOC as indicated previously in Figure 3.4 with the - sign.

After filtering out all the false positive reports, Algorithm 1 outlines the top-level procedure followed by VISER to classify all other reports. The overall objective of this procedure is to calculate the AOC needed for analysis and route it to one of two specialized procedures for the analysis. The procedure starts by identifying the scenario using GETSCENARIO() in line 2. Depending on the layout scenario of the failure being handled by the algorithm, one or both of the Algorithms 2 and 3 are used to analyse the AOC. In the case of a *contained* layout scenario, the AOC labelled A in Figure 3.4 covering the overlap of the two elements is passed to Algorithm 2, see lines 3 - 5. Similarly in the *overlapped* layout scenario, the overlapping portion of the element labelled B in the figure is routed to Algorithm 2, see lines 6 - 8. Finally, in the case of a *separated* layout scenario, the AOC labelled D in the figure that is equal to the coordinates of the protruding element is passed to Algorithm 3, see lines 9 - 11.

Once an AOC is calculated and control is passed to Algorithm 2, the next step for the tool is to capture snapshots of the AOC for usage in the analysis of the failure. This procedure specializes in AOCs that require analysis of snapshots taken from three layers. The first snapshot captures the layer of content hidden behind both elements involved in the failure. These are the elements identified in the report and passed in as input to the procedure. The second snapshot captures the layer with the "back" element visible which is stacked and hidden behind the "front" element. The third and final snapshot captures the top layer with the "front" element visible. To do this, the algorithm starts by making both elements transparent using the MAKETRANSPARENT procedure in order to reveal the deepest layer in lines 2 to 3. Then the SNAPSHOT procedure captures a snapshot and saves it to *imgNoElements* in line 4. It is important to note that the SNAPSHOT procedure captures a specific part of the page defined by the AOC. Then the RESTORE procedure is used to restore the opacity of the "back" element to its original value and a snapshot of it is saved into *imgBack* in lines 5 and 6. For the final snapshot in lines 7 and 8, the opacity of the "front" element is restored and a snapshot is saved into *imgFront*.

Algorithm 2 Image analysis for three layers of an AOC

INPUT: Two HTML elements, *back* and *front*, the failure type, *ft*, the AOC *AOC*.
OUTPUT: TP if the RLF is deemed observable, NOI if it is not.

```
1: procedure ThreeLayersAnalysis(back, front, ft, AOC)
2:     back ← makeTransparent(back)
3:     front ← makeTransparent(front)
4:     imgNoElements ← snapshot(AOC)
5:     back ← restore(back)
6:     imgBack ← snapshot(AOC)
7:     front ← restore(front)
8:     imgFront ← snapshot(AOC)
9:     if imgNoElements ≠ imgBack ∧ imgNoElements ≠ imgFront then
10:        if ft = element_collision then
11:            return TP
12:        if ft = element_protrusion ∨ ft = viewport_protrusion then
13:            AOC ← getDetachedAOC(back, front)          ▷ AOC=C (Figure 3.4)
14:            return TwoLayersAnalysis(front, AOC)
15:    return NOI
```

With the three snapshots of the AOC captured and the control remaining in Algorithm 2, the images are compared in line 9 of the algorithm for differences. If there are no differences found between the images, the failure report is classified as a non-observable issue regardless of the failure type and the end of the procedure is reached. On the other hand, if a difference between the images is found, the next step depends on the failure type currently being handled by the algorithm. In the case of an *element collision* failure type, the failure is deemed observable and therefore is classified as a true positive in line 11. In the case of an *element protrusion* or a *viewport protrusion* type of failure, Viser must analyse another AOC to reach a final classification decision. This AOC covers the portion of the element protruding from the container element that is labelled C in Figure 3.4. For this, the AOC and the control are then passed to Algorithm 3.

Once the control finally reaches Algorithm 3 from either of the two other algorithms, the alternative image analysis begins. This algorithm specializes in analysing an AOC with only a single element in its coordinates. This would be the area without overlap between the two elements corresponding to the labels C and D in Figure 3.4. More specifically, the two images needed are of the layer hidden behind the element and an image of the element itself. To this end, the algorithm starts by making the element transparent and captures a snapshot of the AOC and saving it into *imgNoElement*, see lines 2 and 3. Then in lines 4 and 5, the opacity of the element is restored to its original value and a snapshot of the AOC is saved to *imgFront*. Next, the two images are compared for differences in line 6. If there is a difference, Viser classifies the report as a true positive failure. Otherwise, the failure is classified as a non-observable issue. With this, the presentation failure report is automatically classified by Viser for a specific viewport chosen to do the classification.

To classy any failure, the Viser tool must first pick a viewport width from the failure range to carry out the classification within. If the failure range is long enough, it is reasonable to expect the layout to change or vary throughout the range. At the very least, the layout becomes more compact and there may be less space for the element to spread apart as the viewport size decreases. Therefore, the outcome of the classification may differ from viewport to viewport. The developer can configure which viewport Viser uses to automatically classify the reported failures but the default value is set to the minimum of the failure range, $fail_{min}$. It is reasonable to expect the failure to be more noticeable in the narrower viewports as opposed to the wider viewports with less constraint on space.

Algorithm 3 Image analysis for two layers of an AOC

INPUT: An HTML element, *front*, and the AOC *AOC*.
OUTPUT: TP if the RLF is deemed observable, NOI if it is not.

```
1: procedure TwoLayersAnalysis(front, AOC)
2:     front ← makeTransparent(front)
3:     imgNoElement ← snapshot(AOC)
4:     front ← restore(front)
5:     imgFront ← snapshot(AOC)
6:     if imgNoElement ≠ imgFront then
7:         return TP
8:     return NOI
```

3.4 Empirical Evaluation

In this section, I investigate the effectiveness and efficiency of automatically classifying reported presentation failures using Viser. To this end, I used the same set of web pages used in the previous evaluation of ReDeCheck [124]. Moreover, the manual classifications of that study were adopted as a baseline to compare with Viser's automated classifications. The empirical evaluation focused on answering these three research questions:

Research Question One – *Can* Viser *automatically distinguish non-observable issues from true positives and how does it compare to manual classification?* To answer this question, I compare Viser's results using the default setting of performing the image analysis at the minimum viewport of the failure range to the results of manual classifications.

Research Question Two – *Within the range of viewport where a presentation failure is reported, which viewport has the best chance of matching the manually set classifications and which has the best chance of revealing a true positive failure report?* To answer this question, I used Viser to classify each failure at three points in the failure range: the *minimum* or narrowest from the first research question, the *middle* or halfway point of the range, and the *maximum* viewport of the range. At each of these viewport widths, the manual classifications are compared with Viser's classifications to reach a conclusion.

Research Question Three – *How long does* Viser *take to classify a presentation failure?* In this question, I investigate how efficient Viser is to run and if it is a practical addition to ReDeCheck for layout testing.

The design of the experiments set forth to answer these research questions are explained next.

3.4.1 Design of Experiments

In this section, I will identify the subject web pages used in the experiments and their details, the runtime environment used to build Viser and used to run the tool during the experiments, the methodology followed to answer each of the research questions, and finally disclose any known threats to the validity of the results and any mitigating steps taken to reduce these threats.

Table 3.1: The details of the web pages used in the experiments of this chapter.

Web Site Name	URL	Number of HTML Elements	Number of CSS Declarations
3MinuteJournal	3minutejournal.com	80	5499
AirBnb	airbnb.com	1470	9890
BugMeNot	bugmenot.com	42	658
CloudConvert	cloudconvert.com	908	6731
ConsumerReports	consumerreports.org	1042	8007
CoveredCalendar	coveredcalendar.com	148	8414
DaysOld	daysold.com	66	2930
Dictation	dictation.io	195	8271
Duolingo	duolingo.com	856	4260
Honey	joinhoney.com/install	461	7903
HotelWiFiTest	hotelwifitest.com	359	6746
Mailinator	www.mailinator.com	280	8697
MidwayMeetup	midwaymeetup.com	86	4147
PDFescape	pdfescape.com	179	1954
PepFeed	pepfeed.com	343	7276
Pocket	getpocket.com	664	6607
TopDocumentary	topdocumentaryfilms.com	411	1501
UserSearch	usersearch.org	866	3900
WhatShouldIReadNext	whatshouldireadnext.com/search	112	2314
WillMyPhoneWork	willmyphonework.net	782	6576
Total		9350	112281

Subject Web Pages

The subjects used in the experiments of this chapter were selected from a set of web pages used to evaluate ReDeCheck's effectiveness at detecting five types of responsive layout failure by Walsh et al. [124]. Their study featured a total of 26 subject web pages that were not all included in the experiments of this chapter. Since the purpose of the approach implemented in Viser aims to automatically identify and classify non-observable issues, only the three failure types that reported non-observable issues are handled by Viser. Therefore, the pool of subjects that did not report an element collision, element protrusion, or a viewport protrusion failures were not included in the set of subjects used to evaluate Viser. An additional subject, StumbleUpon, was also excluded from the set of web pages used in this chapter because it failed to load correctly during preparations for the experiments. Most likely, the tool used to save an offline version of the page was not successful at saving all of the resources required by the page. Since the web page is no longer online to load the required resources from, the page must be omitted from the study.

These subjects, used in this chapter, were previously selected in a random fashion by Walsh et al. using the `randomusefulwebsites.com` website which has changed to `discuvver.com`. I downloaded these subjects and used them without modification from the repository cited in their paper `github.com/redecheck/example-webpages`. Although the repository contains 26 web pages, only the 20 eligible subjects were used in the experiments of this chapter. The name of each web page used and other information about each subject are listed in Table 3.1. Collectively, these subjects had a total of 117 presentation failure reports.

Runtime Environment

To evaluate Viser, I matched the execution environment of my experiments to that of the original ReDeCheck evaluation experiments to the best extent possible. This was needed to reduce a threat to validity avoiding discrepancies in the results that might arise due to differences in the experimental setup between the two evaluations. Therefore I ran Viser on an iMac with 8GB of RAM, running OS version 10.13 and using Firefox browser

version 46. This is the same machine that was used in the original evaluation. Similar to ReDeCheck, the Viser tool uses Selenium WebDriver [105] to launch and control the web browser in order to load the web pages and take the snapshots needed in order to classify the failures. Finally, to match the browser settings used in the original evaluation, Viser disabled the scrollbars associated with the browser window and fixed the viewport height to 1000 pixels as done in the original research.

Methodology

The answers to the first two research questions rely on comparing the human-made manual classifications with Viser's automated classifications for agreement. To clarify the disagreement, I grouped the reasons behind these differences into the *subjective*, *obscured*, and *misclassified* categories. Where the subjective category is for a difference in classification that is due to a minor visual change detected by Viser but would be imperceptible to a human. The obscured category is for edge cases where other than the reported failing elements are used to conclude a classification about the failure. In other words, the category is used when a conclusion about either of the two approaches is obscured because each approach used a different portion of the page to classify the failure. Finally, a misclassification categorization is due to an incorrect outcome from either of the two approaches.

RQ1 Methodology – To answer RQ1, I used Viser to automatically classify all 117 presentation failure reported from the set of 20 subjects. Specifically for this question, Viser was configured to use the minimum viewport width, $fail_{min}$, to classify each of the reports. The possible classification outcome can be either false positive (FP), non-observable issue (NOI), or true positive (TP). An FP is a result of the ReDeCheck tool being wrong while a TP is the result of attesting to a visually evident presentation failure. The NOI classification is reached when a structural problem is corroborated but does not manifest visual evidence in the presentation of the layout. I then cross-checked if Viser's classifications agreed with the manual classification as previously decided in the original study by Walsh et al. [124]. Along with calculating the percentage of agreement, I investigate the differences behind the classifications.

RQ2 Methodology – To answer RQ2, I used Viser to automatically classify two more viewports spanning the failure range to reach a total of three classifications per failure report. These are at the minimum (*i.e.*, the lower bound), middle, and maximum (*i.e.*, the upper bound) viewports from the reported failure range. The methodology of the first research question was repeated to classify the failure at the middle, $fail_{mid} = floor((fail_{min}+fail_{max})/2)$, and the maximum, $fail_{max}$. While running this experiment, Viser helped discover a defect in ReDeCheck which caused the tool to be inaccurate at determining the upper bound of the failure range for 35 viewport protrusion failures. This defect is due to an efficiency feature of ReDeCheck that samples viewports instead of visiting all viewports when testing for layout failures. The feature uses an interval to skip over a configurable number of viewports to save time. The tool then performs a binary search when needed between these interval points. To overcome this defect, I used an interval setting of 1 to force the ReDeCheck tool to visit every viewport in the testing range of 320 - 1400 pixels wide. Moreover, the findings of this workaround were used to correct the failure ranges of the 35 viewport protrusions.

RQ3 Methodology – To answer RQ3, I recorded the time it takes for Viser to classify each of the 117 failures at the minimum viewport of the reported failures range, $fail_{min}$. I repeatedly run this experiment a total of 30 times to get a reliable estimate of the

running time of VISER. This is to minimize the chance of an implicit effect from the operating system or other software that may impact the execution time of the experiments. Moreover, the machine was not explicitly used during these experiments.

Threats to Validity

The findings of my research rely on the baseline manual classifications adopted from Walsh et al. [124] that were used to measure the effectiveness of VISER's classifications. Therefore, the validity of the results depends on accurately matching the manual classification with the automatic classification produced by VISER. Since the manual classification did not include the XPath of elements involved in the failure, I matched the failures using the snapshots available. These snapshots, combined with the type of failure, range, and name of the web page enabled me to confidently perform this matching.

Another threat to validity is the possibility of mistakes in the implementation of the VISER tool. To control this threat, I configured VISER to keep a record of all the images used to analyse and classify each of the reported failures. Furthermore, VISER also outputs a record of the coordinates of each of the elements involved in the failure. I consulted these records during the examination of the classifications that differed between the manual and automated VISER approach. Thereby, raising confidence that the tool operated correctly. More importantly, to support the replication of the experiments in this chapter, I made the VISER tool publicly available at `https://github.com/redecheck/viser`.

3.4.2 Results of the Experiments

Answer to RQ1 – the manual classifications performed by Walsh et al. [124] were used to measure the outcome of VISER's automated classifications. For completeness, these manual classifications are broken down in Table 3.2. Notable in the table for both the element collision and element protrusion failures, a large majority were classified as non-observable issues. More specifically, the element collision failures had 24 NOI classifications and only 7 TPs. Meanwhile, the element protrusion failures had 36 NOIs and 3 TPs. For the viewport protrusion failures, about half were assigned the non-observable classification with 23 as NOI and 24 as TP. It is also worthy to note that there were no false positives recorded during the manual classification of this set of presentation failures.

For this research question, VISER used the minimum viewport width of each reported failure range to automatically classify the failure. The results of this experiment are furnished in Table 3.3. To make comparing manual and automatic classification easier, the manual classifications from Table 3.2 are repeated in this table as the denominator of the ratio values. The totals of these ratios across all the subjects are shown in the row titled "Agreement with manual". The overall results of this table show an 86.3% level of agreement between the manual and VISER's classifications as shown in the "Agreement per viewport" row of the table. On a per failure type level, the element collision type had the highest agreement amongst the three failure types with a 93.5% agreement level as shown in the row labelled "Agreement per failure type". Out of the 117 classifications, there were 16 failures in disagreement between the automated and manual approaches for the minimum viewport. Next, I will provide more details about the disagreements between the classifications.

A total of 9 disagreements fall into the subjective category. These failures had a small number of pixels that were changed as a result of the presentation failure and therefore were labelled by VISER as TPs. Nevertheless, these changes are imperceptible to the

Table 3.2: The manual classification of presentation failures from a prior study [124]. In this table "TP", "NOI", and "FP" respectively denote true positive, non-observable issue, and false positive. The "Element Collision", "Element Protrusion", and "Viewport Protrusion" columns correspond to the failure types of Figure 3.2.

	Manual Classifications									
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
3MinuteJournal	-	1	-	-	2	-	8	-	-	11
AirBnb	-	1	-	-	4	-	-	4	-	9
BugMeNot	-	-	-	1	3	-	2	-	-	6
CloudConvert	1	-	-	-	-	-	-	-	-	1
ConsumerReports	-	7	-	1	3	-	9	3	-	23
CloudConvert	-	-	-	-	-	-	-	3	-	3
DaysOld	-	-	-	-	-	-	-	1	-	1
Dictation	-	-	-	-	-	-	-	1	-	1
Duolingo	-	1	-	-	-	-	2	2	-	5
Honey	-	-	-	-	8	-	-	2	-	10
HotelWiFiTest	-	-	-	-	-	-	1	-	-	1
Mailinator	-	1	-	-	-	-	-	-	-	1
MidwayMeetup	1	-	-	-	1	-	-	1	-	3
PDFescape	-	-	-	1	5	-	1	3	-	10
PepFeed	4	3	-	-	2	-	1	1	-	11
Pocket	-	2	-	-	3	-	-	-	-	5
TopDocumentary	-	7	-	-	4	-	-	-	-	11
UserSearch	-	1	-	-	-	-	-	-	-	1
WhatShouldIReadNext	-	-	-	-	-	-	-	2	-	2
WillMyPhoneWork	1	-	-	-	1	-	-	-	-	2
Total	7	24	-	3	36	-	24	23	-	117
Total per failure type		31			39			47		-

human eye. In one case, the total AOC affected was only two pixels in width and a little more in height but yielded no human observable failure in the layout. Suffice it to say, all nine were subjectively labelled as NOI during original manual classifications due to these minor changes in the layout. An avenue for future work lies in advancing the automated classification approach of Viser to take the number of pixels changed and the degree of colour change into account when analysing failures.

There were also 2 disagreements that I categorized as obscured. Both of these failures were reported from the ConsumerReports subject and were manually classified as TPs. The Viser tool disagreed and classified them as NOIs. While Viser's analysis was correct for the elements identified in the failure report made by ReDeCheck, there is also a visual defect in the layout near the AOC that is noticeable by a human. I attribute this problem to the imprecise reporting made by ReDeCheck rather than a problem with the classification approach followed by Viser.

Among the disagreements in classifications are an additional 2 reports that were misclassified by Viser. The first is an element protrusion failure that comes from the PDFescape subject and was misclassified because the parent or container element had the `overflow` property set to the value `hidden`. This caused the reportedly protruding content or element to not be rendered by the browser and hence can't be "seen" by Viser in the captured snapshots. Logically, this led Viser to classify it as an NOI because it does not have knowledge beyond what is apparent in the snapshot of the current viewport. Meanwhile, the content missing from the page is really a TP failure report as correctly classified by the manual approach due to what the human knows from other viewports.

Table 3.3: The results of Viser's classifying 117 presentation failures reported by ReDeCheck using the minimum viewport of the reported failure range.

	Minimum Viewport									
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
3MinuteJournal	-	1/1	-	-	2/2	-	8/8	-	-	11
AirBnb	-	1/1	-	-	1/4	3/-	2/-	2/4	-	9
BugMeNot	-	-	-	1/1	3/3	-	2/2	-/-	-	6
CloudConvert	1/1	-	-	-	-	-	-	-	-	1
ConsumerReports	1/-	6/7	-	1/1	3/3	-	9/9	3/3	-	23
CoveredCalendar	-	-	-	-	-	-	-	3/3	-	3
DaysOld	-	-	-	-	-	-	-	1/1	-	1
Dictation	-	-	-	-	-	-	-	1/1	-	1
Duolingo	1/-	-/1	-	-	-	-	2/2	2/2	-	5
Honey	-	-	-	-	8/8	-	-	2/2	-	10
HotelWiFiTest	-	-	-	-	-	-	-/1	1/-	-	1
Mailinator	-	1/1	-	-	-	-	-	-	-	1
MidwayMeetup	1/1	-	-	-	1/1	-	1/-	-/1	-	3
PDFescape	-	-	-	-/1	6/5	-	3/1	1/3	-	10
PepFeed	4/4	3/3	-	-	2/2	-	1/1	1/1	-	11
Pocket	-	2/2	-	-	3/3	-	-	-	-	5
TopDocumentary	-	7/7	-	-	4/4	-	-	-	-	11
UserSearch	-	1/1	-	-	-	-	-	-	-	1
WhatShouldIReadNext	-	-	-	-	-	-	-	2/2	-	2
WillMyPhoneWork	1/1	-	-	-	1/1	-	-	-	-	2
Total	9	22	-	2	34	3	28	19	-	117
Agreement with manual	7/7	22/24	-	1/3	32/36	-	22/24	17/23	-	-
Agreement per failure type	93.5 %			84.6 %			83 %			-
Agreement per viewport	86.3 %									-

See Figure 3.5 (a) for a snapshot of the layout with the presentation failure and added borders around the elements reported by ReDeCheck associated with failure.

The failure from PDFescape reveals one limitation of the basic heuristics implemented in Viser when dealing with this edge case. Considerably, this limitation extends to the detection of layout failures as well. Here, the hiding of the overflowing content may be intended as the layout makes a change from the wider menu to the more compact "burger" menu icon. This is an icon with three horizontal lines that look like the buns and meat of a burger that is used for devices with a small screen. For clarity, the wider menu of the page can be seen in Figure 3.5 (b). I believe the source of the problem to be premature hiding of the wider menu or delinquency in displaying the alternative compact menu. Therefore, I leave it for future experimentation to improve Viser in order to track content across different viewports or to modify the `overflow` property which is expected to improve the classification in this case but may also negatively impact other cases.

The second failure misclassified by Viser is a viewport protrusion that comes from the HotelWiFiTest subject. Here, ReDeCheck detected an element containing a large portion of the page's content overflowing the viewport width. Although the overflowing content was not hidden from view, it did not align properly with the main menu header due to the overflow. Since the Viser tool focuses on the reported elements and does not account for other misalignments in the page, the failure was labelled by the tool as an NOI failure. Worse than the alignment, this failure requires that the user horizontally scrolls the page in order to view the content. In retrospect, a viewport protrusion classification should take into account the scrolling that is required by the user to view the overflowing content even if it is an NOI. In which case, the failure should be escalated to a TP classification.

Future improvements of the algorithm should treat the visual content written outside the viewport that requires horizontal scrolling as a true positive presentation failure. See Figure 3.5(c) for a snapshot of the layout with the presentation failure.

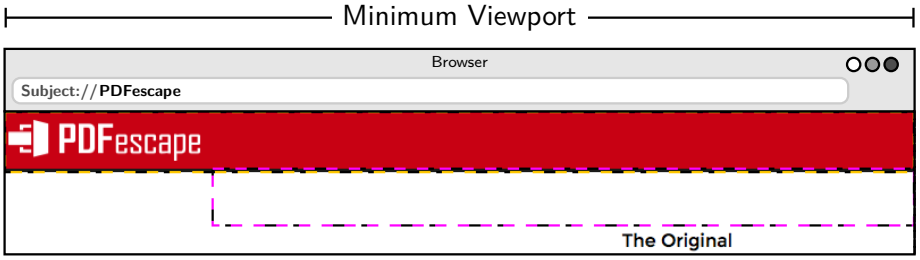
Finally, there were a total of 3 failures that were misclassified by Walsh et al. during their manual classification. For these three element protrusion failures, the ReDeCheck reported elements with no associated defects in the layout. Most likely, they were assumed to be NOIs in error when classifying these failures. Meanwhile, the Viser tool automatically classified these failures as FPs since the elements mentioned in the ReDeCheck report did not even cause a protrusion in the underlying layout structure. These reports were filtered and classified as false positives without the need for visual analysis. Based on the DOM readings, I also investigate the coordinates of the elements and found them to be free from failure as well. Ultimately, I believe the root cause of this misclassification to be a defect in ReDeCheck 's collection of DOM information when constructing the RLG.

Conclusion for RQ1 – the Viser tool demonstrates a high agreement of 86.3% with the human-made manual classification using the minimum viewport of the reported failure range.

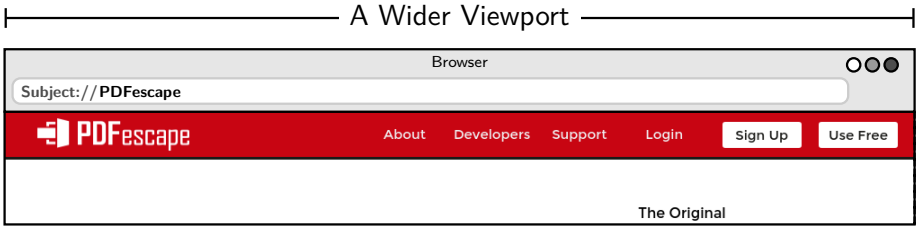
Answer to RQ2 – Table 3.4 and Table 3.5 respectively show Viser's results when it is configured to classify the middle and maximum viewport of the reported range of viewports where the failure was detected. Combined with the results of the minimum viewport covered in the previous research question, it is clear that Viser's classifications can vary across the three viewports spanning the reported failure range depending on the chosen viewport for classification. These results show that Viser is more likely to agree with the manual classifications at the minimum viewport of the range of the reported failure. Compared to an agreement of 86.3% at the minimum viewport, the agreement for the middle and maximum viewports of the range respectively drop to 83.8% and 78.6%. This also extends to matching true positives where at the minimum viewport there were 30 failure reports out of 34 matched. While in the middle viewport, the number of matches dropped to 26 true positives and 19 at the maximum viewport. As was done for the minimum viewport in the answer to the previous research question, the disagreements between the manual and Viser's classifications are discussed next with references to specific subjects highlighting the key trade-offs between the approaches.

Classifications of the middle viewport - From the set of classifications agreed at the minimum viewport, 4 failures had their classifications automatically changed by Viser at the middle viewport. All of these four cases were of the viewport protrusion type. Of which, the first two come from the PDFescape and PepFeed subjects. Both of these failures were correctly reclassified by Viser as NOIs. Furthermore, it would be a misclassification to use the TP manual classification for this viewport. It is also important to reiterate that both approaches correctly classified these two failures as TPs at the minimum viewport. In effect, for these two failures, the observability of the defect in the presentation of the layout varied depending on the viewport width chosen for classification. As the viewport width expanded, elements had enough space to spread apart and become a non-observable issue at the middle viewport. Therefore, the judgment made during the manual analysis of the failure does not hold for the entire range. Importantly, Viser can automatically detect these differences in observability.

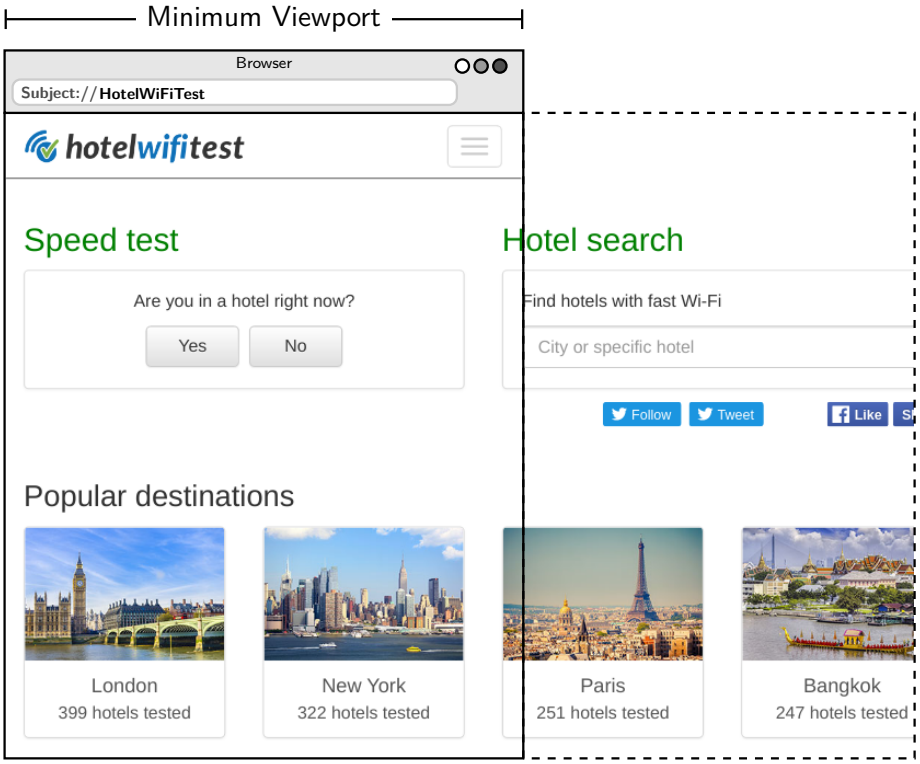
The two failures correctly reclassified by Viser are showcased in Figure 3.6. In the figure, the snapshots taken at the minimum viewport are showcased in part (a) for the



(a) The element protrusion failure misclassified by VISER as a non-observable failure. The yellow dashed line highlights the coordinates of the container element while the maroon dashed line shows the coordinates of the protruding element. The content missing here is the menu as shown below.

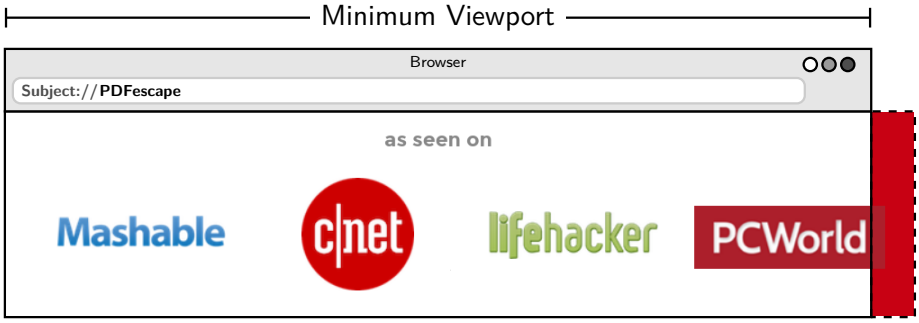


(b) Captured from a wider viewport before the presentation failure occurred. In this viewport, the full menu is displayed.

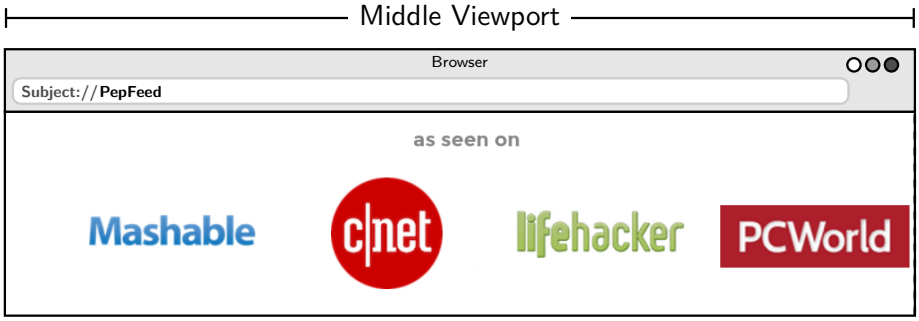


(c) The viewport protrusion failure that was misclassified by VISER as a non-observable issue. Although there is no missing content or lost functionality, the user is required to horizontally scroll in order to view the content beyond the browser's size that is surrounded by a dashed line in the figure.

Figure 3.5: The figure presents two failure reports that were misclassified by VISER as non-observable issues. The first of which comes from the PDFescape subject and is showcased in parts (a) and (b) while the second comes from the HotelWiFiTest subject and can be seen in part (c).

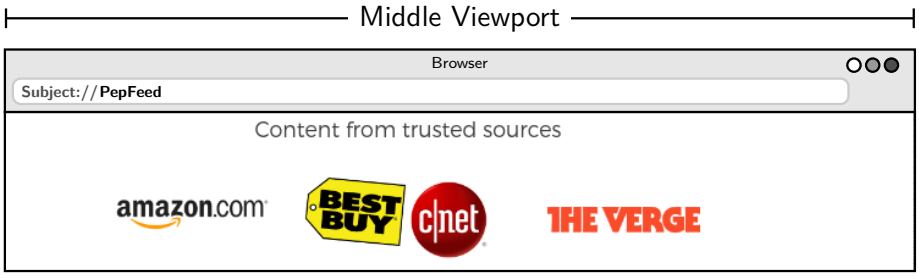


(a) At the minimum viewport, the element containing the “PCWorld" is protruding the viewport.



(b) At the middle viewport, the “PCWorld" logo is not visisibly protruding the viewport.

(c) At the minimum viewport, the element containing the “Amazon" is protruding the viewport.



(d) At the middle viewport, the “Amazon" logo is not visibly protruding the viewport.

Figure 3.6: The figure presents two failure reports that were reclassified by Viser as non-observable issues in the middle viewport. The first of which comes from the PDFescape subject and is showcased in parts (a) and (b) while the second comes from the PepFeed subject and can be seen in parts (c) and (d).

Table 3.4: The results of Viser's classifying 117 presentation failures reported by ReDeCheck using the middle viewport of the reported failure range.

	Middle Viewport									
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
3MinuteJournal	-	1/1	-	-	2/2	-	6/8	2/-	-	11
AirBnb	-	1/1	-	-	1/4	3/-	2/-	2/4	-	9
BugMeNot	-	-	-	1/1	3/3	-	2/2	-/-	-	6
CloudConvert	1/1	-	-	-	-	-	-	-	-	1
ConsumerReports	-	7/7	-	1/1	3/3	-	9/9	3/3	-	23
CoveredCalendar	-	-	-	-	-	-	-	3/3	-	3
DaysOld	-	-	-	-	-	-	-	1/1	-	1
Dictation	-	-	-	-	-	-	-	1/1	-	1
Duolingo	1/-	-/1	-	-	-	-	2/2	2/2	-	5
Honey	-	-	-	-	8/8	-	-	2/2	-	10
HotelWiFiTest	-	-	-	-	-	-	-/1	1/-	-	1
Mailinator	-	1/1	-	-	-	-	-	-	-	1
MidwayMeetup	1/1	-	-	-	1/1	-	1/-	-/1	-	3
PDFescape	-	-	-	-/1	6/5	-	2/1	2/3	-	10
PepFeed	4/4	3/3	-	-	2/2	-	-/1	2/1	-	11
Pocket	-	2/2	-	-	3/3	-	-	-	-	5
TopDocumentary	-	7/7	-	-	4/4	-	-	-	-	11
UserSearch	-	1/1	-	-	-	-	-	-	-	1
WhatShouldIReadNext	-	-	-	-	-	-	-	2/2	-	2
WillMyPhoneWork	1/1	-	-	-	1/1	-	-	-	-	2
Total	8	23	-	2	34	3	24	23	-	117
Agreement with manual	7/7	23/24	-	1/3	32/36	-	18/24	17/23	-	-
Agreement per failure type	96.8 %			84.6 %			74.5 %			-
Per inspection point	83.8 %									-

failure from PDFescape and in part (c) for the PepFeed subject. For this viewport, the agreed-upon true positive classification is appropriate since the "PCWorld" and "Amazon" logos are protruding from the viewport in each case. Meanwhile, both failures at the middle viewport are non-observable issues as seen in parts (b) and (d) for PDFescape and PepFeed respectively. In both cases, the logos that were visibly protruding from the viewport are no longer protruding at the middle viewport. Since the underlying structural issues remain to be true at the middle viewport, these failures should not be classified as false positives.

Moving on to the other two classifications in disagreement, both were a misclassification by Viser. Both these viewport protrusion reports come from the 3MinuteJournal subject. Initially, at the minimum viewport, both classification approaches agreed that the failure is a TP. However, in the middle viewport, Viser changed its classification to NOI. In this case, the content of the page overflows the viewport and requires a user to scroll the page horizontally. Because Viser does not consider scrolling as a problem when classifying a failure, this led to misclassification by the tool. This scenario is essentially identical to that of the HotelWiFiTest subject discussed as part of the answer to the first research question and showcased in Figure 3.5 (c). Again, this scrolling effect that breaks the RWD principle should be addressed as part of future work.

Classifications of the maximum viewport - there were a total of 7 disagreements associated with the maximum viewport that previously agreed with the manual classification at the minimum and middle viewports. Only at the widest viewport in the reported failure range did Viser conclude a different classification. All seven were true positive viewport protrusion failures at the smaller viewports. Since the viewport is wider at the maximum of the range, the space is sufficient for the previously visible defects in the layout to

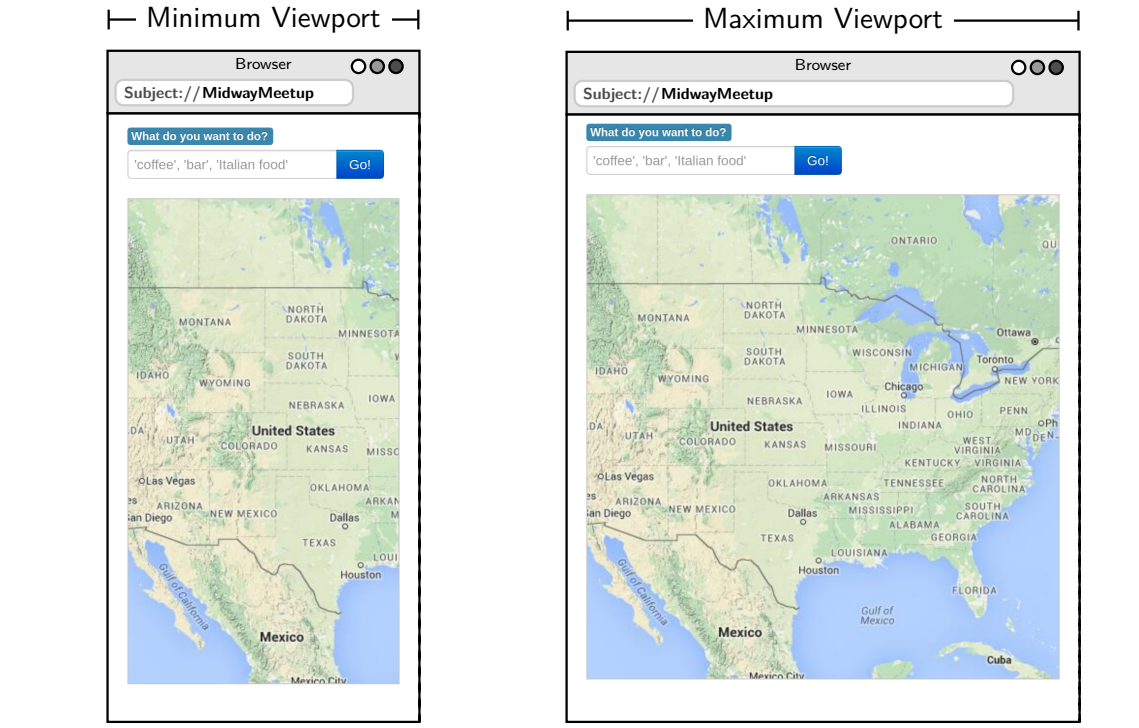
Table 3.5: The results of Viser's classifying 117 presentation failures reported by ReDeCheck using the maximum viewport of the reported failure range.

	Maximum Viewport									
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
3MinuteJournal	-	1/1	-	-	2/2	-	-/8	7/-	1/-	11
AirBnb	-	1/1	-	-	1/4	3/-	2/-	2/4	-	9
BugMeNot	-	-	-	1/1	3/3	-	2/2	-/-	-	6
CloudConvert	1/1	-	-	-	-	-	-	-	-	1
ConsumerReports	-	7/7	-	1/1	3/3	-	8/9	4/3	-	23
CoveredCalendar	-	-	-	-	-	-	-	3/3	-	3
DaysOld	-	-	-	-	-	-	-	1/1	-	1
Dictation	-	-	-	-	-	-	-	1/1	-	1
Duolingo	1/-	-/1	-	-	-	-	2/2	2/2	-	5
Honey	-	-	-	-	8/8	-	-	2/2	-	10
HotelWiFiTest	-	-	-	-	-	-	-/1	1/-	-	1
Mailinator	-	1/1	-	-	-	-	-	-	-	1
MidwayMeetup	1/1	-	-	-	1/1	-	-	1/1	-	3
PDFescape	-	-	-	-/1	6/5	-	2/1	2/3	-	10
PepFeed	4/4	3/3	-	-	2/2	-	-/1	2/1	-	11
Pocket	-	2/2	-	-	3/3	-	-	-	-	5
TopDocumentary	-	7/7	-	-	4/4	-	-	-	-	11
UserSearch	-	1/1	-	-	-	-	-	-	-	1
WhatShouldIReadNext	-	-	-	-	-	-	-	2/2	-	2
WillMyPhoneWork	1/1	-	-	-	1/1	-	-	-	-	2
Total	8	23	-	2	34	3	16	30	1	117
Agreement with manual	7/7	23/24	-	1/3	32/36	-	11/24	18/23	-	-
Agreement per failure type	96.8 %			84.6 %			61.7 %			-
Per inspection point	78.6 %									-

become non-observable issues. Six out of the seven originated from the 3MinuteJournal subject and one came from the ConsumerReports subject. These cases are all similar to the failures that went from TPs to NOIs at the middle viewport as seen in Figure 3.6.

An additional, notable, change in classification involved a viewport protrusion failure from the 3MinuteJournal subject that was classified as a false positive report by Viser at the maximum viewport. For this failure, Viser agreed with the TP manual classification at the minimum viewport and disagreed in the middle viewport with an NOI classification. An investigation of the issue revealed that the ReDeCheck tool reported a protrusion of a single pixel in width based on the coordinates read from the DOM while Viser coordinates calculated to a non-protrusion. It is worthy to note that the two tools use alternative approaches to retrieving DOM coordinates. The ReDeCheck tool relies on a customized JavaScript script adopted from another tool that is injected into the web page while the Viser tool uses Selenium's built-in methods to retrieve the readings and consequently reported no protrusion in this instance.

Agreement after disagreement from minimum to maximum - The differences in classification discussed so far looked at newer disagreements as the viewport expanded. On the contrary, there are 2 cases where Viser originally disagree but later agreed with the manual classification at a wider viewport. The first is a viewport protrusion failure from the MidwayMeetup subject showcased in Figure 3.7 parts (a) and (b). This failure report involved an image of a map that presents most of the United States and happens to protrude the viewport size based on the coordinates retrieved from the DOM. Although there is an observable part of the map missing, as "noticed" by Viser, subjectively this should really be a non-observable issue that could also be considered a non-failure to begin with.



(a) VISER classified as TP. (b) VISER classified as NOI.

(c) VISER classified as TP. (d) VISER classified as NOI. (e) VISER classified as NOI.

Figure 3.7: The figure showcases two presentation failures that VISER initially disagreed with the manual classification on but at a wider viewport agreed with the manual classification. The first is a viewport protrusion from the MidwayMeetup subject seen in parts (a) and (b). The second is an element collision that comes from the ConsumerReports subject and is showcased in parts (c), (d), and (e).

The second case is an element collision failure that comes from the ConsumerReports subject. Again, the human that manually classified the failure was able to dismiss the report and classify it as a non-observable issue. While the tool, Viser, classified it as a TP in the minimum viewport and as an NOI in the middle and maximum viewports. The failure is showcased in Figures 3.7 (c), (d), and (e) for visual reference. Largely unnoticeable, the collision occurs in the middle of the page towards the lower portion of the blue region. The only noticeable difference is the missing lower portion of the blue region in the snapshot of part (c). At the minimum viewport, part (c), the bottom black border of the white banner in the middle of the blue region is in collision with the content positioned below the blue region. Therefore, it was classified by Viser as a TP failure. At the middle and maximum viewport, parts (d) and (e), only a transparent portion of the element is in collision with the lower content. Hence, it was classified as an NOI by Viser.

The key findings from these two cases are that, ideally, some observably missing content should be dismissed. One way to do this with Viser is to look at a winning classification over all three viewports. Furthermore, the `overflow` property is a key feature that may be able to distinguish an acceptable overflow and avoid reporting it in the first place as seen in the case from the MidwayMeetup subject.

Conclusion for RQ2 – Viser is more likely to agree with a manual classification when is set to use the minimum viewport width from the reported failure range. Nevertheless, the classification of larger viewports in the reported range is the only way to ensure the consistency of the classification throughout the range. The changes in classification, as the viewport widens, can be explained by: (1) the fact that the observability of the failure changes as the space expands in wider viewports; (2) the role of subjectivity over "minor" observable issues; and (3) that a small number of misclassifications were made by Viser. Notably, almost all the failures involved in the change of classifications were exclusively of the viewport protrusion type.

Answer to RQ3 – To analyse how long it takes to automatically classify a responsive layout failure, the runtime of the Viser tool was recorded across 30 runs. Figure 3.8 shows box plots that visualize the runtime of the tool for element collision, element protrusion, and viewport protrusion failures. The figure reveals that the time to classify a viewport protrusion failure has a "longer tail", resulting in a slightly higher median runtime. This effect is attributed to the extra work that Viser does to move elements into view for a snapshot and to investigate AOCs that may be larger than the viewport. Nevertheless, no major distinction can be made between different failure types. Across all failure types, the tool took a median of 0.79 seconds and a mean of 0.91 seconds to automatically classify a presentation failure. Importantly, the time to load the web page and resize the browser were excluded from measurement as this cost would be shared by any other technique, whether manual, semi-automated, or automated. All of the recorded times account for the overhead of finding the reportedly problematic elements, checking the structure of the layout, taking the required snapshots, classifying the failure, and writing all diagnostic images to disk.

Conclusion for RQ3 – On average, Viser took under a second to classify a presentation failure. Arguably, a manual classification of the same failure is expected to take longer than a second. This suggests that using Viser is practical and efficient.

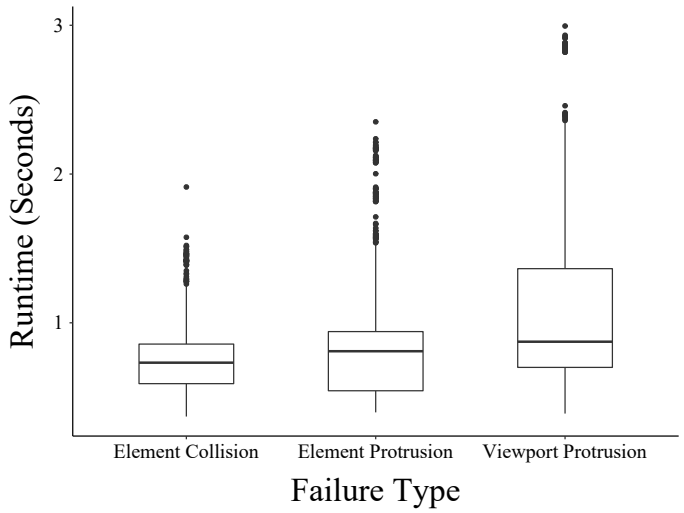


Figure 3.8: VISER's execution time across all of the 117 presentation failures and 30 trials for each type of layout failure. In these plots the bottom and top whiskers show the minimum and maximum data values excluding outliers, while the box itself represents the inter-quartile range, the middle line represents the median value, and the circles are outliers.

3.4.3 Discussion

The results from the empirical study suggest that VISER is a good automated alternative to the manual classification of presentation failures reported by the REDECHECK. In this section, I will examine a few open points for discussion and opportunities for improving VISER in greater detail.

Questioning Observability – is not always discernible whether the reported presentation failure is an observable visual defect in the layout of the web page or not. This makes the final manual classification decision somewhat subjective. Essentially, the task requires an observer to recognize a difference between what is visually expected and what is visually apparent. I found that the previously published manual classifications by Walsh et al. [124], which I used in the experiment of this chapter, employed "exemptions" based on the severity of a change in the layout. For instance, consider an element A and an element B with n overlapping pixels. In this case, a human would decide whether the n pixels of overlap are negligible and if the overall aesthetics remain satisfactory. Both of these criteria are not easily defined and remain to a great extent subjective. It may also depend on the content that each element brings to the layout. Nevertheless, this is an avenue that can be studied in future work. For example, it may be useful to measure the number of changed pixels, determine if a colour change is visible to the human eye, or introduce general heuristics that are concerned with the overall AOC size.

It is also worthy to note that I had concerns with the previously published manual classifications used in the empirical experiments of this chapter that I was not aware of prior to using them. After a deeper examination of the manual classification, I found that some classifications were neither confined to the type of failure nor the XPaths reported. This may be considered a bias or illustrates the subjectivity and difficulty of the task. For instance, an element was reported protruding out of an ancestor container, which is an NOI, but was manually classified as a TP due to a simultaneous protrusion out of the immediate container. Although a visual defect is apparent in the layout, the reports were not specific to the elements involved in the defect. Therefore, reclassifying some

of these manual classifications may be justifiable but I refrained from "tampering" with the benchmark classification not to introduce any further sources of bias from my end. Moreover, a reclassification would not tackle the underlying subjective nature that is inherent in the process of manually classifying a presentation failure.

Since all of the previous research that I reviewed in the area of testing web page's for presentation failures used a manual approach to visually confirm the reports from their prototype tools, the accuracy and consistency of the manual approach will influence, positively or negatively, the research outcomes. Although I did not experimentally study the output of other testing tools and alternative types of presentation failures, clearly from the results of the experiments carried out in this chapter, there are consistency and efficiency benefits associated with the automation of the classification process.

Revealing Layers – using the `opacity` CSS property is one way to reveal the graphical layers needed to analyse and classify the presentation failures without making Viser browser dependent. An alternative strategy is to manipulate the `visibility` property of an element. However, descendant HTML elements can override the inheritance of this property. This means that Viser would have to traverse the DOM tree in order to check and set the `visibility` property of all descendant HTML elements. One negative aspect of this approach is the added implementation complexity and execution time overhead.

3.5 Concluding Remarks

While responsive web design allows a developer to build layouts for a variety of devices with different screen sizes, the developer still needs to check for presentational problems in the web page across different layouts for different devices. Even though the ReDeCheck tool automatically checks a web page for presentation failures in a responsively designed web page, the developer must manual verify ReDeCheck's failure reports. This manual task can be time-consuming, imprecise, and error-prone. As such, in this chapter, I presented a new technique to automatically classify the element collision, element protrusion, and viewport protrusion failures reported by ReDeCheck. Implemented into the tool Viser, this automated technique adjusts the `opacity` property of the HTML elements in an area of concern looking for a visible difference to distinguish non-observable layout issues from observable ones.

Using the results of the manual classification of presentation failures from a previously published paper as a baseline, this chapter's experiments showed that Viser's automatically generated classifications agreed with the manual ones 86.3% of the time. The results also demonstrate that Viser is more likely to agree with a manual classification and reveal a true positive failure when it is set to analyse a web page at the minimum viewport of the failure range that was reported by ReDeCheck. With Viser taking less than a second in runtime to classify a presentation failure, the empirical results suggest that Viser is a good alternative to the manual classification.

Unfortunately, the Viser tool is limited in its ability to automatically classify the failures reported by ReDeCheck because it only handles three out of five failure types that can be reported by the tool. These three failure types were associated by Walsh et al. [124] with non-observable issues and thus handled by Viser. This leaves two more failure types that were not associated with NOIs which a developer must manually classify before attempting to repair them. This problem motivated me to extend automation to the two new failure types in order to alleviate the developer from their burden. Furthermore, I wanted to know how well the findings of this chapter extend to new subjects. This research is the focus of the next chapter.

4

Classifying Observable Failures and Reassessing Automated Classification

In the previous chapter, I presented a technique that automatically classifies three types of presentation failures reported by the layout testing tool ReDeCheck. Moreover, this technique was specifically developed to solve the problem of identifying and classifying non-observable issues from the reported failures. Only three out of five types of presentation failures were associated with non-observable issues based on a published experiment conducted by the creators of ReDeCheck, Walsh et al. [124], where they manually classified the reported failures. These were the element collision, element protrusion, and the viewport protrusion type of failures that Viser can automatically classify, as presented in the previous chapter. In this chapter, classification is extended to the *element wrapping* and *small-range* failure types that were not associated with non-observable issues.

Although the results from the experiments of the previous chapter were positive, they still warranted further work. Fundamentally, the technique was limited by the fact that Viser did not support the classification of two types of presentation failures reported by ReDeCheck. Since Viser cannot classify failures pertaining to either an incorrect element wrapping or the sporadic rearrangement of element at a small number of viewport widths, this chapter introduces "Verve" (Visual classifiEr for ResponsiVe tEsting), a tool that automatically classifies all five types of presentation failures detected by ReDeCheck. To do this, the strategy behind Viser that manipulates the opacity of HTML elements is extended to classify element wrapping failures in the new tool Verve. Furthermore, the new tool employs a new approach that uses a histogram-based image comparison technique to classify the reported failure involving layout mistakes at a small number of viewport widths, the small-range failures.

This chapter begins by explaining the problem that Verve was created to resolve. Then, a brief overview is given of how the two new failure types are detected. To follow is an explanation of the techniques implemented in the new tool Verve. Finally, the empirical evaluation of Verve and the results are presented.

The key contributions of this chapter are:

1. New algorithms to automatically classify the *element wrapping* and *small-range* failures types reported by ReDeCheck.
2. An empirical evaluation comparing the manual and automated classification of *element wrapping* and *small-range* failures using the 25 subjects from the previous chapter. Demonstrating that the two new failure types can be automatically and

effectively classified.

3. An empirical reassessment of the automated classification of all five failure types, as implemented in Verve, over 20 new subject web pages. Demonstrating that my automated classification technique does extend well onto new subject web pages.

4.1 Motivating the Research

The problem presented in this chapter is similar to the problem that was presented in Section 3.1 of the previous chapter. Although the ReDeCheck tool helps the developer of a responsively designed web page test the layout as it changes to befit a different viewport width of the browser, the classification of the reported presentation failures is a task left for the developer. Therefore, the developer must read the reports and inspect the live web page to determine if there is a valid concern. The result of the inspection could lead the developer to classify the failure as either a true positive or a false positive report. Where a true positive is labelled when an observable defect is found in the presentation of the layout while a false positive is the result of a ReDeCheck mistake.

The problem solved in the previous chapter relates to the identification and classification of reports that have a measurable issue in the structure of the web page but are visually non-observable in the rendered layout. This is a class of failure reports known as Non-Observable Issues (NOIs). The main reason for their existence is due to the fact that the tool uses the DOM to detect failures and does not gather or analyse information about the rendered layout. Therefore, the reports of the DOM-based ReDeCheck tool may report NOIs. This class of reports was associated with the element collision, element protrusion, and viewport protrusion failure types reported by ReDeCheck. Even though the element wrapping and small-range failure types were not associated with NOIs in previous research, the results of ReDeCheck must still be checked for false positive reports. A human may come to this conclusion if the tool made a mistake or the issue was so minor it is imperceptible to the eye.

To illustrate the complexity of classifying these reported failures, Figure 4.1 shows a failure that is both an NOI and true positive failure at a different viewport. This is a real example of a viewport protrusion failure from the SB-Admin-2 subject found in the viewport range 320-379 during the experiments of this chapter. At a viewport width of 379 pixels as seen in part (a), there is no apparent problem in the presentation of the web page. Nevertheless, the DOM-based coordinates of the container of the top-right profile image is protruding out of the main `body` element. Therefore, at this maximum viewport size of the reported failure range, it should be classified as a non-observable issue. At the middle of the range, 349 pixels as seen in part (b), the horizontal space available is more confined and the profile image is now protruding off to the right edge of the page. At a minimum viewport of the failure range, 320 pixels as seen in part (c), the profile image is entirely missing from view and is no longer accessible.

Another real example of a presentation failure, of a different type, is shown in Figure 4.2. This failure is an element wrapping from the SB-Resume subject that was detected in viewports 1056-1121 pixels during the experiments of this chapter. One viewport wider than the maximum of the failure range, viewport 1122 pixels as seen in part (a), at the bottom right of the snapshot there are twelve icons of programming languages and tools that are properly aligned in a single row. At this viewport, there is no failure since all the icons are aligned in a row. While at the immediately narrower viewport of 1121 pixels as seen in part (b), the last icon wraps into a new row and is therefore flagged by ReDeCheck as a presentation failure. Although slightly less problematic than the loss of

functionality seen in the previous example, this may break the intended design.

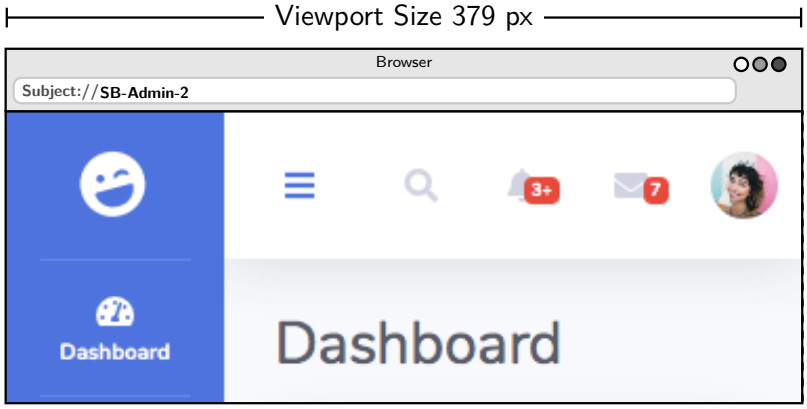
Without automating the classification of element wrapping and small-range types of failures raised by ReDeCheck, the user must still manually sift through the reports to classify these two failure types since Viser does not support them. Doing so, manually, over many reports generated by the tool can be time-consuming and error-prone. During one such case, Walsh et al. [124] et al. had to manually classify 209 presentation failure reports specific to these two type of failures from 25 subjects. Moreover, they found 47 of these reports to be false positives. In other words, about 22% of the failures reported by the tool were of no good to the developer while the other 78% are all true positives according to their manual classifications. In an extreme case, the PepFeed subject had only 2 true positive reports out of 16 reported presentation failures. Automating the classification of this large number of reports that may require the inspection of multiple viewport widths for each failure report will pass on more benefits to the developer. This is done by reducing the required labour and time to test the layout and eliminating the chance for a human-made error during the classification. Thus improving the quality of the reports that are finally handled by the developer.

4.2 Detection Prior to Classification

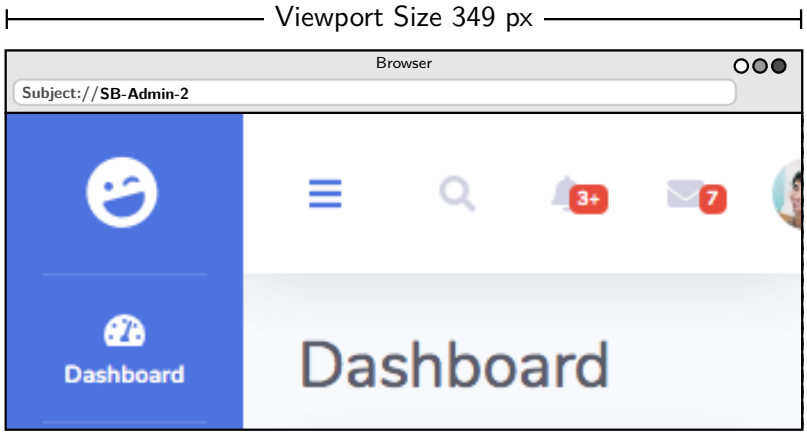
As explained in Section 3.2 from the previous chapter, the ReDeCheck tool detects presentation failures by building a Responsive Layout Graph (RLG) model of a web page [125]. An RLG captures the behaviour of the layout of the web page as it responds to a change in viewport width [127]. For each viewport width, the relative alignment of the visible HTML elements with respect to one another (e.g., "above-of", "right-of", "contained-by") are represented in the model. To construct the RLG, ReDeCheck gathers the required information using the DOM after instructing a desktop browser to navigate to a web page and change the viewport widths to values from a specified testing range, $\{test_{min}..test_{max}\}$. This viewport testing range typically starts with a narrow width of 320 pixels that mimics the width of a mobile phone and extends to a more spacious width of 1400 pixels corresponding to a desktop computer. The class of presentation failures found by the ReDeCheck tool are referred to as responsive layout failures (RLFs).

In the previous chapter, the process of detecting three types of responsive layout failures was explained. Briefly, the first type called *element collision* is detected when two elements, that share the same container element, do not overlap at one viewport but later overlap in a narrower viewport. Therefore, the range of viewports where the overlap occurs is reported as the failure range. The second called *element protrusion* is detected when at some viewport, the area of one element is contained within the container element but at a narrower viewport width protrudes the area of the previously determined container element. At the smaller viewport, none of the two elements fully overlap the other. For this type, the failure range is equal to viewport widths where there is only partial overlap between the elements. With high similarity, a *viewport protrusion* failure is detected when an element protrudes the main container element, the `body` element. Hence, it will not be contained by any other element. A real-world example was showcased in Figure 4.1. All of these failures may lead to undesirable aesthetics or cause a loss of functionality for important links, input fields, or buttons if they are unreachable or obscured due to the presentation failure.

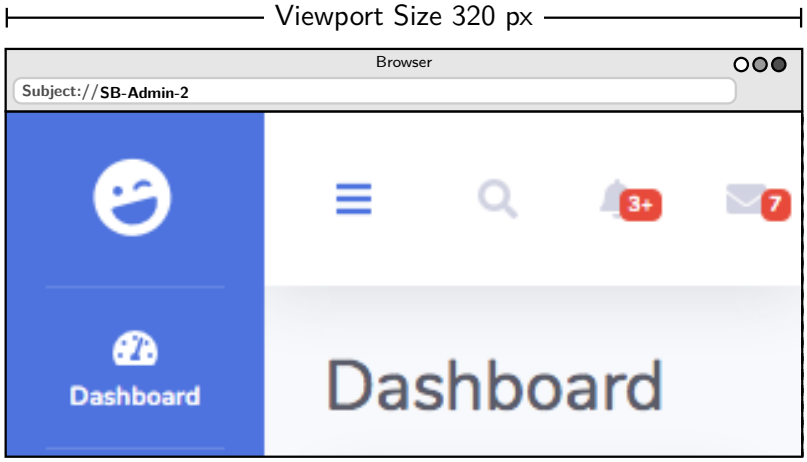
The first of the new failure types handled by Verve is referred to as an *element wrapping* failure. This type of failure occurs when, by design, a group of HTML elements that are aligned to appear together in a single row formation can no longer remain side



(a) NOI Viewport Protrusion

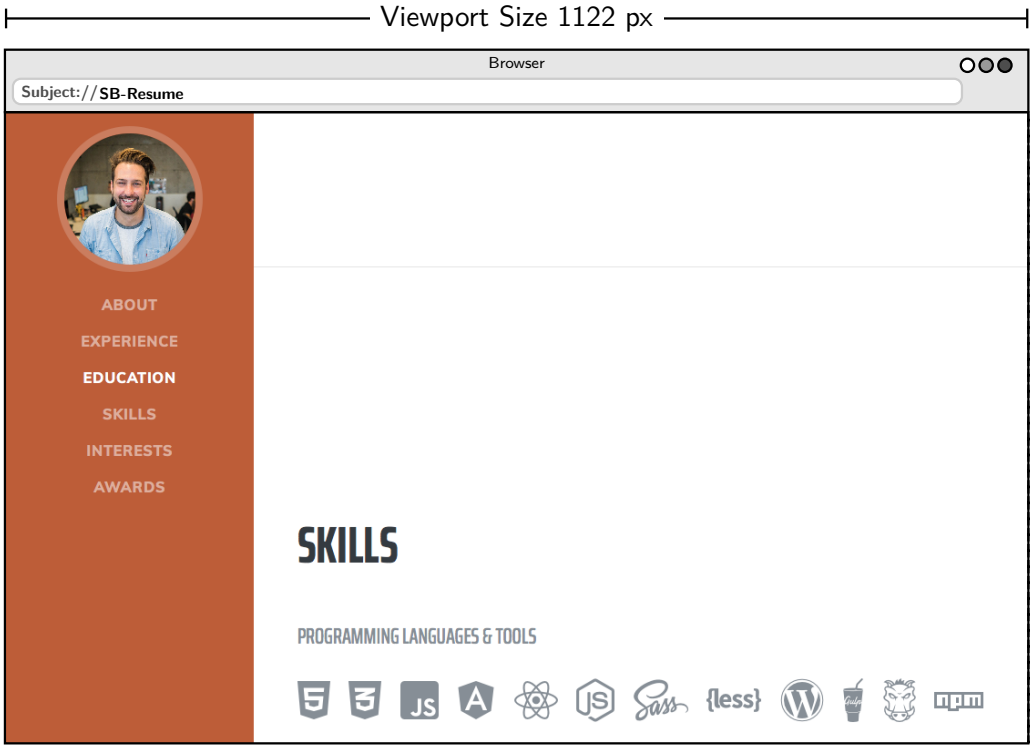


(b) TP Viewport Protrusion

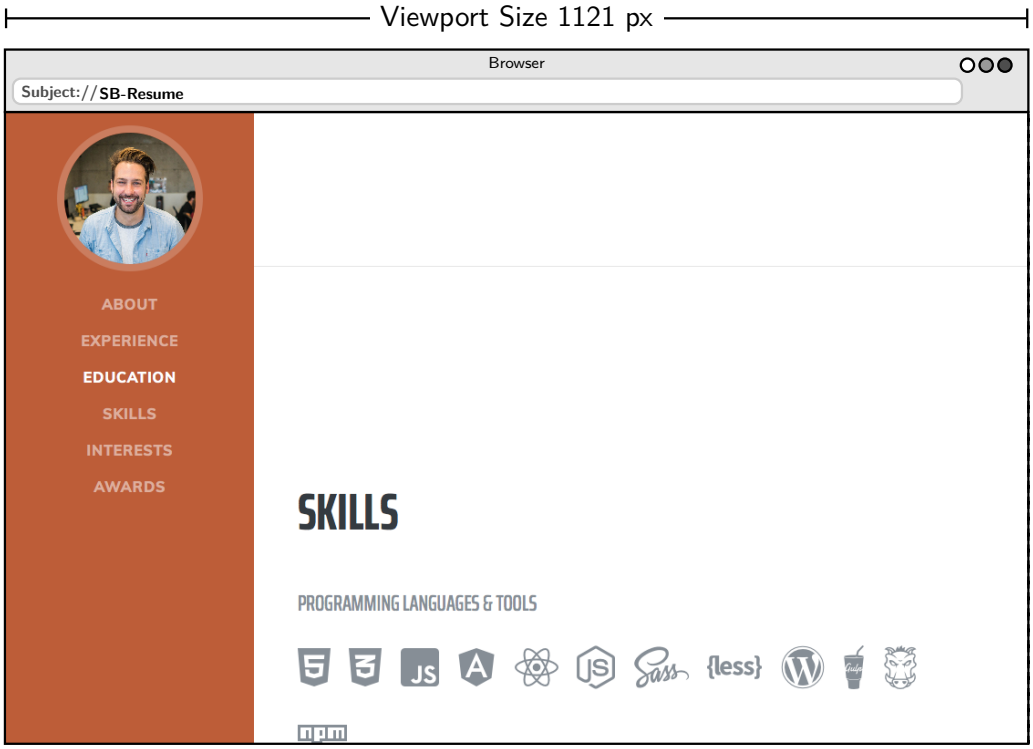


(c) TP Viewport Protrusion

Figure 4.1: Three snapshots of the SB-Admin-2 web page that capture a viewport protrusion failure at the maximum of the reported failure range of 320 – 379 pixels, in (a), and at the middle of the range in (b), and at the minimum of the range in (c), as reported by ReDeCheck and correctly classified by Verve.

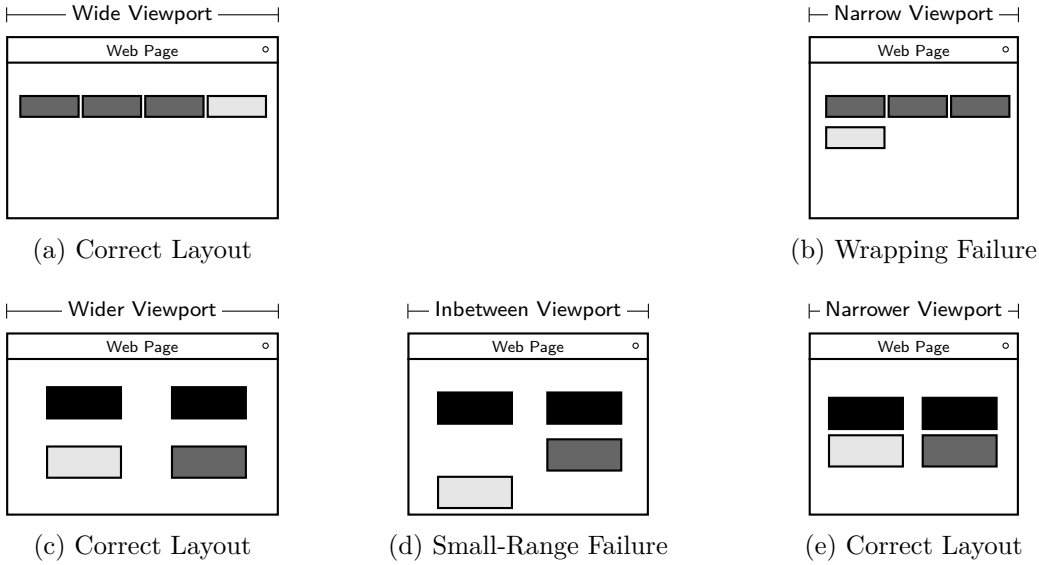


(a) No Layout Failure



(b) Wrapping Layout Failure

Figure 4.2: Two snapshots of the SB-Resume web page that capture its layout before a wrapping failure occurs, in (a), and a wrapping failure with the range of 1056–1121 pixels in (b), as reported by the ReDeCheck and correctly classified, without human intervention, as a true positive by Verve.



(a) Correct Layout (b) Wrapping Failure

(c) Correct Layout (d) Small-Range Failure (e) Correct Layout

Figure 4.3: Wireframe examples of the *element wrapping* and *small-range* responsive layout failures. The actual failures are represented in part (b) for the wrapping failure and part (d) for small-range failure while the other parts of the figure represent a correct layout at a different viewport.

by side. Although the formation of elements is feasible at a wide viewport width, the narrower viewports may not be wide enough for the formation to persist. This scenario is represented in the wireframe example of Figure 4.3 parts (a) and (b). The elements in dark and light grey colours are properly aligned in a row formation at the wider viewport width as seen in part (a). While at a narrower viewport width, illustrated in part (b), the light grey element is forced to wrap below the darker grey elements that remain in a row formation. In this scenario, the wrapping failure is caused by a confinement in the horizontal space of the smaller viewport width.

To detect an element wrapping failure, the ReDeCheck tool first iterates over the alignments between any pair of elements that share the same *parent* in order to determine if they are in a row. A parent element in the RLG model is another element with the most confined coordinates that contain the coordinates of the element in question. To determine row formation, the tool uses the existence of the alignment "right-of" without the existence of an "above-of" or "below-of" alignments to identify if two or more elements are in a row. If at a smaller viewport width one of the row forming elements wraps and establishes the "above-of" or "below-of" alignment, the tool reports the wrapped element as a failure in the layout. A real-world element wrapping example found during the experiments of this chapter was presented in Figure 4.2. Along with the list of XPaths of the elements making up the row, the ReDeCheck tool also reports the range of viewport widths where the failing element wrapped below the row.

The second type of layout failure automatically classified by the new Verve tool is the *small-range* failure. This type of failure report indicates a layout that occurs anomalously for only a small number of consecutive viewport widths. Figure 4.3 (c), (d), and (e) illustrate a wireframe example of this failure type. For the majority of viewport widths (i.e., parts (c) and (e)), a total of four web page elements are correctly laid out in a 2×2 grid formation. Yet, over a small number of viewports in between as seen in part (d), the light grey element to the button left of the figure falls out of alignment with the others.

Unlike the other four type of responsive layout failures detected by ReDeCheck that

are likely to be caused by a lack of available space to layout the elements as the viewport tightens, a small-range presentation failure is more likely to be a programming mistake. More precisely, an improper usage of media queries in the web page may be the cause of this type of failure. For example, the media query "`@media (max-width: 768px) {...}`" may be used by the developer to display a layout specific to viewport widths of up to 768 pixels. The media query "`@media (min-width: 768px) {...}`" may be added to display a layout specifically tailored for viewport widths that are bigger than 768 pixels. However, since the viewport ranges defined by both of these media queries are inclusive, both will be activated at the 768 pixels viewport width. Potentially, this will lead to a problem in the layout due to the simultaneous activation of two competing sets of rules when only one set was intended to be active at a time. This type of failure is more difficult for the developer to detect manually since it occurs only at a few viewport widths.

To detect small-range failures, the ReDeCheck tool uses the signature of the failure characterized by a temporary change in relative alignment of elements that suspiciously occur at only a few viewport widths. To do this, the tool iterates over pairs of elements investigating the range of their relative alignments. This is done to find an alignment between a pair of elements that holds true for only a predetermined number of viewports. During the experiments of Walsh et al. [124], the threshold of 5 viewports was used to indicate a potential small-range failure. If the two elements have different alignments at both the immediately narrower and wider than the consecutive viewports falling within the threshold, the two elements are reported as a small-range failure. The change in alignments observed by the algorithm is also included in the report as well as the range of viewport where the temporary re-arrangement occurred. A failure report of this type may also be a duplication of any of the other four type of failure if the failure range happens to be five viewports or less.

4.3 Classifying Wrapping and Small-Range Failures

Given a set of wrapping and small-range presentation failures reported by the ReDeCheck tool, the automated approach that I am presenting in this section aims to classify the reports to identify true positive failures. To achieve this, the prototype Verve automatically examines the underlying structure of the layout and captures images of the layout to analyse the failure. Next, I will give an overview of the approach implemented in Verve followed by details of how this is achieved for an element wrapping and a small-range failure.

4.3.1 Summary of Approach

The technique followed by Verve is able to automate the classification of all failures reported by ReDeCheck using images captured from the layout. Prior to capturing and analysing images, the tool first corroborates the reported failure using a DOM-based reading at the time of classification. This step is important in order to filter out false positive reports generated by ReDeCheck that may have been raised in error or as a limitation of the detection algorithms. Thereby leaving only non-observable issues and true positive reports to be distinguished and classified. After filtering all false positives, the tool needs to calculate the area of the layout that will be graphically analysed. To do this, the coordinates of the elements involved in the failures are used to limit the image analysis to a region of the web page where the failure occurred. This region, referred to as the Area of Concern (AOC). Then, the layout in the AOC is captured using snapshots and

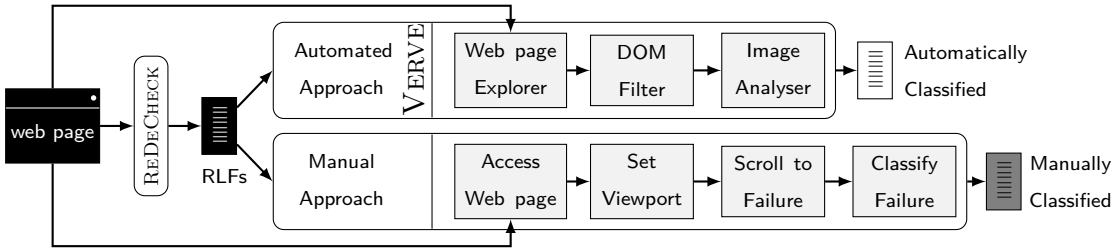


Figure 4.4: The high-level architecture of the Verve tool for the automatic classification of layout failures. Along with the external input sources, this figure also shows the alternative manual process steps that require a human expert.

analysed in one of two ways depending on the failure type. The first approach compares different graphical layers of an AOC to classify element wrapping failures, as done by Viser and described in the previous chapter for element collision, element protrusion, and viewport protrusion

The second image analysis approach of Verve uses colour histograms in order to classify small-range failure reports. Unique to this type of failure, a colour histogram of the AOC is generated for three viewport widths associated with the reported failure range $\{fail_{min}..fail_{max}\}$. By definition, the failure range of all small-range reports is limited to five viewport widths. Therefore, little layout change is expected between these viewports and hence only one viewport is sufficient to represent the failure. This is the $fail_{min}$ viewport corresponding to the minimum viewport within the failure range. A snapshot of the AOC is captured with the viewport set to $fail_{min}$ and is used to create a colour histogram. Then two comparison snapshots of the AOC from the immediately narrower, $fail_{min}-1$, and immediately wider viewports, $fail_{max}+1$, are captured for comparison against the one captured using viewport $fail_{min}$. Depending on a predetermined threshold value, Verve uses the result of differencing the colour histograms to automatically identify true positive presentation failure reports.

4.3.2 Classifying the Failure Reports

After a developer runs ReDeCheck in order to test the layout of a web page, two possible failures that the tool may detect and report is an element wrapping and small-range failures. If a report is generated, it will state the responsive layout failure type (e.g., element wrapping), the range of viewport widths for which the failure was deemed to occur (in the form of a lower and an upper bound such as $\{fail_{min}..fail_{max}\}$), and the XPaths of the HTML elements involved. If Verve is not used, the developer must manually decide what to do with these reports. The lower portion of Figure 4.4 illustrates the steps the developer needs to do in order to investigate the reports generated by ReDeCheck. This involves loading up the web page, setting the viewport width of the browser to one within the reported failure range, manually identifying the elements involved and scrolling to the failure if necessary, and finally deciding if the report is a true positive or not.

As depicted in the top portion of Figure 4.4, the Verve tool automates these steps. The *Web Page Explorer* component first opens the browser, sets the viewport width, and locates the faulty elements automatically. Then the Verve tool cross-checks ReDeCheck's results by examining the DOM in the *DOM Filter* step. For this, Verve checks that each element reported has a physical size (i.e., its width and height are not zero), and that they can be reached, if not initially present, by scrolling the web page. More specifically,

Figure 4.5: The area of concern (the element marked "E") for a wrapping failure.

Verve checks the coordinates of the bounding box of each element and treats negative coordinates as unreachable since the browsers do not scroll to negative coordinates nor render its content. Furthermore, coordinates that are greater than the maximum position that can be scrolled to are also deemed to be off the page. The tool Verve makes further checks at this stage depending on the responsive layout failure type as will be detailed in the following subsections. Any reports that do not pass these checks are classified as false positives. Otherwise, Verve proceeds to the *Image Analyser* component for visual analysis of the failure.

The *Image Analyser* step investigates specific regions of a web page referred to as *areas of concern* (AOCs). AOCs are specific to each type of responsive layout failure. An AOC is a rectangular area within the page that pertains to the elements involved in a layout failure. This is the area where graphical content is suspected to have been inadvertently overwritten by other content on the page or to have been written out of its designated position. The *Image Analyser* component ultimately intends to determine if this is the case (i.e., the presentation failure produces a visible and observable defect). For example, if the misplaced element has no content and is transparent, the failure will not be detectable by a human and so a failure report produced by ReDeCheck will likely be of little concern to the developer of the web page.

The remainder of this section describes how Verve identifies AOCs for the two new failure types not described in the previous chapter, and how the image analysis of the AOC can automatically classify the failure reports.

Element Wrapping Failures

To classify an element wrapping report, the Verve tool re-purposes the graphical-layer revealing technique used to classify element collision, element protrusion, and viewport protrusion failure reports explained in the Section 3.3.4 of the previous chapter. Briefly, the approach uses the CSS `opacity` property in order to reveal the lower graphical layer hidden behind the rendering of an HTML element. By comparing the lower layer against the top layer, the tool can infer if an element is overwriting other content of the page or if the element is written out of position. Prior to any graphical analysis taking place, the Verve tool begins by filtering out any false positive reports in the *DOM Filter* phase.

Specifically for wrapping failures, the *DOM Filter* phase of Verve makes a check of the DOM to check three criteria. First, it ensures that there are at least three elements in the original "row" (as per the original ReDeCheck algorithm [124]). Second, it checks that the wrapped element is below all other row elements and that all row elements are of valid size for display with a width and height greater than zero. If the criterion is not met, then Verve concludes that it is a false positive report. Otherwise, Verve proceeds to the image analysis phase in order to determine if the report being handled is true positive or just a non-observable issue.

The *Image Analyser* component of Verve treats a wrapping failure report as if it was another "separated" scenario that was illustrated in Figure 3.4 and discussed in the previous chapter. Therefore, the AOC of wrapping failure that requires visual analysis is equal to the entire coordinates of the wrapped element as shown in Figure 4.5. The

algorithm that handles wrapping failures, described in Algorithm 4, takes this AOC and passes it to the TwoLayersAnalysis procedure of Algorithm 5. In turn, the algorithm will check if the wrapped element is visible and therefore returns TP. Otherwise, it will return NOI if the wrapped element is fully transparent. For this type of presentation failure, a transparent element could be introduced by the developer to add padding or used as a placeholder for future content that has not made it into the page yet.

Algorithm 4 Classification of wrapping failures

INPUT: The wrapped element *wrapped*.
OUTPUT: TP if the RLF is deemed observable, NOI if it is not.

```
1: procedure ClassifyWrappingFailures(wrapped)
2:     AOC ← getWrappingAOC(wrapped)                ▷ AOC = E (Figure 4.5)
3:     return TwoLayersAnalysis(wrapped, AOC)
```

Algorithm 5 Image analysis for two layers of an AOC

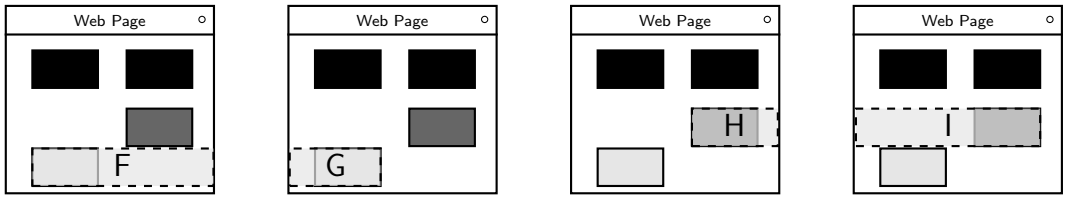
INPUT: An HTML element, *front*, and the AOC *AOC*.
OUTPUT: TP if the RLF is deemed observable, NOI if it is not.

```
1: procedure TwoLayersAnalysis(front, AOC)
2:     front ← makeTransparent(front)
3:     imgNoElement ← snapshot(AOC)
4:     front ← restore(front)
5:     imgFront ← snapshot(AOC)
6:     if imgNoElement ≠ imgFront then
7:         return TP
8:     return NOI
```

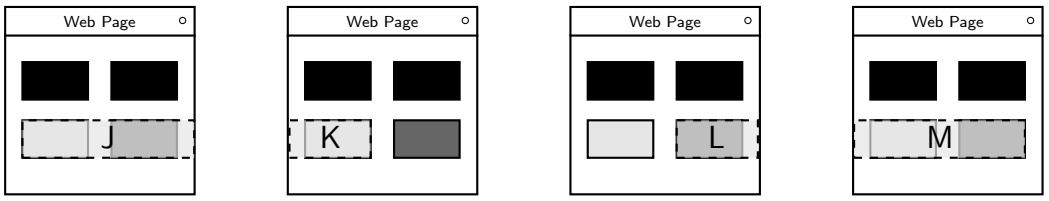
Small-Range Failures

As described in Section 4.2, the detection of small-range failures and their effect on the layout are altogether different from the other failure types. Unlike other failure types, The small-range algorithm intends to catch mistakes in the coding of media queries rather than the problems associated with the lack of horizontal space available within a particular viewport width. More specifically, it is expected to be caused by a programming mistake in the logic of a media query resulting in some media rules being activated when they should not be. The interaction between different CSS properties of one or more HTML elements may create an anomalous layout for a few viewports while some rules are inadvertently activated.

Because small-range failures represent general element misalignments with positions that are difficult to characterize in advance, this type of failure requires a new classification approach that is different from the approach used to classify the other four failure types. Instead of examining different graphical layers, the new approach, implemented in Verve, attempts to measure the level of visual disturbance caused by the misalignment in the layout. If the visual difference between a snapshot of the failure is above a predetermined threshold when compared with snapshots of the web page at viewport widths on either side of the reported viewport failure range, $fail_{min}-1$ or $fail_{max}+1$, Verve flags it as a true positive failure. If the visual difference is negligible and therefore under the experimentally determined threshold, the report is flagged as a false positive by Verve. Nevertheless, like other failure types, an area of concern must be calculated to do the image comparison.



(a) A layout of a viewport where a small-range failure occurs (referred to as a *failure* viewport)



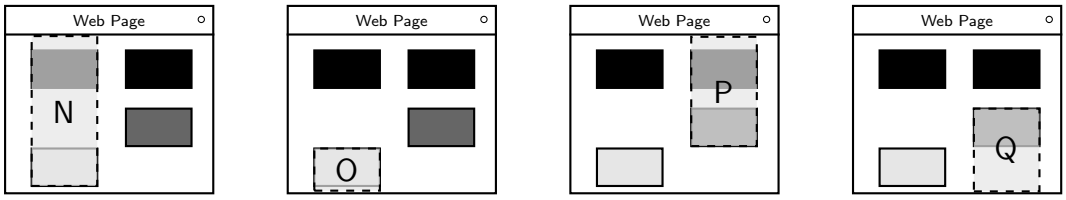
(b) The immediately narrower viewport without a failure (referred to as a "comparison" viewport)

Figure 4.6: Showcasing the *horizontal referencing* system for identifying the areas of concern (AOCs) for visual or graphical scrutiny using a small-range responsive layout failure involving two distinct HTML elements, represented by the light grey and dark grey boxes. Each AOC from the failure viewport, part (a), is labelled with a letter and has a counterpart AOC labelled in the comparison viewport directly below it.

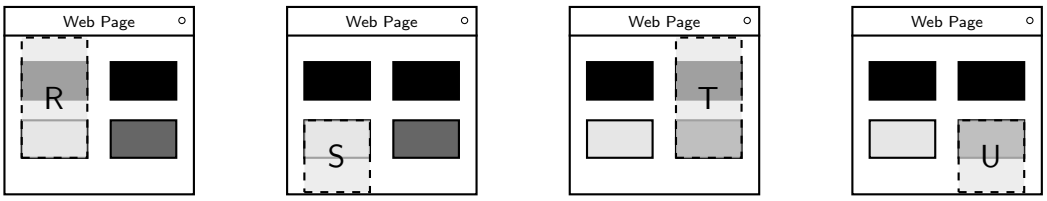
As with the other failure types, VERVE automatically identifies regions of the web page (i.e., the areas of concern) for graphical scrutiny. The difference with other failures is that VERVE not only identifies AOCs using a viewport containing the failure from the failure range $\{fail_{min}..fail_{max}\}$ as reported by REDECHECK, instead it also uses the viewports $fail_{min}-1$ and $fail_{max}+1$ that are at either side of the failure range. I refer to these two additional viewports as the *comparison* viewports and any viewport from the failure range as the *failure* viewport. On a technical level, VERVE compares multiple AOCs from the failure viewport to their counterparts from a comparison viewport in order to ascertain if there is a visual difference and assign an appropriate classification.

The REDECHECK tool reports small-range failures as pairs of elements whose relative alignment has changed for a small number of viewport widths. To illustrate this, Figures 4.3(c), (d), and (e) depict a light grey element that moved from the left of the dark grey element to its bottom-left and therefore REDECHECK will report both of these elements. In this case, Figure 4.6 depicts the same scenario but with the AOCs labelled. In part (a) of the figure, the small-range failure from the *failure* viewport is presented. While in part (b) of the figure presents the narrower *comparison* viewport, $fail_{min}-1$, where the anomalous alignment is no longer evident.

There are many different ways in which a page can be dissected into AOCs in order to capture any visual disturbance associated with the reported failure. Since the problem requires measuring the misalignment in the relative position of an element, the content of the page in a given direction of an element is used by VERVE as a reference for its position. For example, if element Y is to the right-hand of element X in the comparison viewport but not in the failure viewport, an AOC covering the area to the right of element X will capture this change in the relative position of the two elements. Since a change in the layout is expected for different viewports in a responsively designed web page, the VERVE tool relies on the two comparison viewports nearest to the failure range to reach a conclusion. Using the same example, if the element Y is also not on the right-hand side



(a) A layout of a viewport where a small-range failure occurs (referred to as a *failure* viewport)



(b) The immediately narrower viewport without a failure (referred to as a "comparison" viewport)

Figure 4.7: Showcasing the *vertical referencing* system for identifying the areas of concern (AOCs) for visual or graphical scrutiny using a small-range responsive layout failure involving two distinct HTML elements, represented by the light grey and dark grey boxes. Each AOC from the failure viewport, part (a), is labelled with a letter and has a counterpart AOC labelled in the comparison viewport directly below it.

of X in the second comparison viewport, then Verve treats it as an expected responsive design change.

There are at least two possible positional referencing systems that arise from this example. The first is the *horizontal referencing* system which is illustrated in Figure 4.6. This system dissects AOCs using the coordinates of the bounding boxes of each reported element and extended out horizontally to either the right or left edge of the page at each viewport width (e.g., F and G). This results in two AOCs per element, four AOCs for each viewport, and a total of 12 AOCs for all three viewports. To reduce the complexity of the illustrated example, the wider comparison viewport with similar AOCs was omitted. A naturally alternative positional referencing system, illustrated in Figure 4.7, would be the *vertical referencing* system with AOCs extending out from the reported element vertically to the top or the bottom of the page (e.g., N and O). Combining the referencing style of both systems creates a *horizontal-plus-vertical referencing* system covering all four directions from the element that is seeking a positional reference. For the experiments of this chapter, I implemented the horizontal referencing and the horizontal-plus-vertical referencing systems into Verve and evaluated them in Section 4.4.

Using either the horizontal referencing or horizontal-plus-vertical referencing options, Verve's classification algorithm for small-range failures creates a colour histogram of each AOC using snapshots of the page. That is a histogram of the number of pixels with a certain colour. Then, each colour histogram from a failure viewport is compared with a corresponding histogram of an AOC from a comparison viewport. An example pair of corresponding AOCs from Figure 4.6 are the AOCs labelled F and J. Algorithm 6 shows the steps Verve follows to classify a failure using AOCs identified with the horizontal-plus-vertical referencing option. Alternatively, the steps of the algorithm with the horizontal referencing option set are the same but omit the steps associated with the "vertical" AOCs (e.g., lines 7–11 and 17–21).

The first part of the small-range algorithm is dedicated to calculating the AOCs needed

Algorithm 6 Classification of small-range failures

INPUT: Two HTML elements, *element1* and *element2* involved in the failure; the failure viewport *failureVP*; the narrower comparison viewport *narrVP*; the wider comparison viewport *widerVP*; and finally, the colour histogram distance metric *metric*.
OUTPUT: TP if the RLF is deemed observable, FP otherwise.

```
1: procedure ClassifySmallRangeFailures(element1, element2, failureVP, narrVP, widerVP, metric)
2:     ▷ Get horizontal AOCs from failure viewport
3:     element1RightFailureAOC ← getRightAOC(element1, failureVP)        ▷ AOC = F (Figure 4.6)
4:     element1LeftFailureAOC ← getLeftAOC(element1, failureVP)          ▷ AOC = G (Figure 4.6)
5:     element2RightFailureAOC ← getRightAOC(element2, failureVP)        ▷ AOC = H (Figure 4.6)
6:     element2LeftFailureAOC ← getLeftAOC(element2, failureVP)          ▷ AOC = I (Figure 4.6)
7:     ▷ Get vertical AOCs from failure viewport
8:     element1TopFailureAOC ← getTopAOC(element1, failureVP)            ▷ AOC = N (Figure 4.7)
9:     element1BottomFailureAOC ← getBottomAOC(element1, failureVP)      ▷ AOC = O (Figure 4.7)
10:    element2TopFailureAOC ← getTopAOC(element2, failureVP)            ▷ AOC = P (Figure 4.7)
11:    element2BottomFailureAOC ← getBottomAOC(element2, failureVP)      ▷ AOC = Q (Figure 4.7)
12:    ▷ Get horizontal AOCs from narrower (comparison) viewport
13:    element1RightNarrAOC ← getRightAOC(element1, narrVP)              ▷ AOC = J (Figure 4.6)
14:    element1LeftNarrAOC ← getLeftAOC(element1, narrVP)                ▷ AOC = K (Figure 4.6)
15:    element2RightNarrAOC ← getRightAOC(element2, narrVP)              ▷ AOC = L (Figure 4.6)
16:    element2LeftNarrAOC ← getLeftAOC(element2, narrVP)                ▷ AOC = M (Figure 4.6)
17:    ▷ Get vertical AOCs from narrower (comparison) viewport
18:    element1TopNarrAOC ← getTopAOC(element1, narrVP)                  ▷ AOC = R (Figure 4.7)
19:    element1BottomNarrAOC ← getBottomAOC(element1, narrVP)            ▷ AOC = S (Figure 4.7)
20:    element2TopNarrAOC ← getTopAOC(element2, narrVP)                  ▷ AOC = T (Figure 4.7)
21:    element2BottomNarrAOC ← getBottomAOC(element2, narrVP)            ▷ AOC = U (Figure 4.7)
22:    ▷ Get vertical and horizontal AOCs from wider (comparison) viewport
23:    element1RightWiderAOC ← getRightAOC(element1, widerVP)
24:    ...
25:    ▷ Match up the AOCs for colour histogram analysis
26:    setsOfAOCs ← {(element1LeftFailureAOC, element1LeftNarrAOC, element1LeftWiderAOC),
27:                  (element1RightFailureAOC, element1RightNarrAOC, element1RightWiderAOC),
28:                  ...}
29:    ▷ Compare the colour histograms for each set of AOCs
30:    for each (failureAOC, narrAOC, widerAOC) ∈ setsOfAOCs do
31:        histogramForFailureAOC ← getColourHistogram(snapshot(failureAOC, failureVP))
32:        histogramForNarrAOC ← getColourHistogram(snapshot(narrAOC, narrVP))
33:        histogramForWiderAOC ← getColourHistogram(snapshot(widerAOC, widerVP))
34:        distanceNarrower ← getDistance(metric, histogramForFailureAOC, histogramForNarrAOC)
35:        distanceWider ← getDistance(metric, histogramForFailureAOC, histogramForWiderAOC)
36:        distance ← min(distanceNarrower, distanceWider)
37:        if SatisfyThreshold(metric, distance) then
38:            return TP
39:    return FP
```

for analysis as seen in lines 2–24. These AOCs are then matched up into sets of three from the three viewports involved as seen in lines 25–28. Next, a loop iterates over each set of AOCs for comparison in lines 29–38. Within the loop, in line 31, the SNAPSHOT procedure creates an image of the AOC by cropping a snapshot taken from the failure viewport which is then passed to the GETCOLOURHISTOGRAM procedure that returns a colour histogram of the cropped image. This is repeated for the narrower and wider comparison viewports in lines 32 and 33 with each histogram saved into a different variable.

With the three histograms ready for analysis, the algorithm then uses a metric to compute a *distance* between a histogram of the failure viewport against that of a narrower or wider comparison viewport in lines 34 and 35 respectively. The algorithm then, in line 36, chooses the smaller of the two distances for the analysis. This means that VERVE will use the layout of the comparison viewport that better resembles the failure viewport. The intention is to overcome any intended layout changes programmed in the responsive design of the web page. Thus, allowing the tool to distinguish the difference between the failure occurring or not while overcoming unrelated but intended changes in the layout.

If this distance is above a threshold, as determined by the SATISFYTHRESHOLD procedure in line 37, VERVE will conclude that the visual disturbance to the presentation of the web page is sufficient to classify the reported failure as a true positive, see line 38. Otherwise, VERVE will classify the report as a false positive once the loop iterates over all sets of AOCs, terminating the algorithm in line 39.

To compute the colour histogram of an image and to calculate the distance between any two histograms, VERVE integrated and outsourced these tasks to the publicly available OpenCV [87] tool. Since there were five distance metrics available in OpenCV, I employed all five distance measures in order to evaluate the better metric for the web page layout use case as part of the experiments of this chapter. Namely, these were the *Bhattacharyya Distance*, *Chi-Square*, *Alternative Chi-Square*, *Correlation*, and *Intersection* [16]. The *Kullback-Leibler Divergence* measurement was not included in this thesis because images from different viewports with different sizes may output negative values. Thus, the results of *Kullback-Leibler Divergence* may not be fairly compared against other distance measure. For the thresholds required by each metric, I determined it in an automated fashion during preliminary experimentation (This will be explained in more detail in Section 4.4.1). These predetermined thresholds are also evaluated for suitability using the subjects in the empirical study of this chapter, which I will introduce and discuss next.

4.4 Empirical Evaluation

To empirically evaluate VERVE's new small-range and element wrapping failure automated classification algorithms for effectiveness and efficiency, I conducted experiments using VERVE to classify the presentation failures found in the web pages used in the previous evaluation of REDECHECK by Walsh et al. [124], henceforth referred to as *initial set*, as done in the previous chapter for the other three failure types. Furthermore, a new set of subjects henceforth referred to as the *additional set* is introduced in this chapter that was not used in the prior chapter to study VISER. The aim of the new subjects is to evaluate how well VERVE's performance extends into the new set of subject web pages and compare it with the previous results. All of the experiments were conducted to answer the following research questions:

Research Question One: – *Can* VERVE *automatically classify wrapping failure reports when compared with the human-made manual classifications?* To answer this question, I employed the original set of web pages used to assess the REDECHECK tool, called the initial set, in order to analyse for agreement between VERVE's classification against the human-made manual classifications specifically for wrapping failures. This *initial set* of subjects was also used in the experiments of the previous chapter to analyse the VISER's ability to classify element collision, element protrusion, and the viewport protrusion failures. See Section 3.4.1 of the previous chapter for the subjects used for that purpose.

Research Question Two: – *How do* VERVE*'s automatically classified small-range failure reports compare with the manual classification and which colour histogram distance metric performed the best and which referencing approach is better?* To answer this question, I employed the *initial set* of subjects to analyse the level of agreement between VERVE's automated classifications and the manual classifications specifically for small-range failure reports. Moreover, I also analyse which of the five histogram distance methods is the most effective at increasing the agreement. Finally, I compare both the horizontal referencing and horizontal-plus-vertical referencing alternative approaches implemented into VERVE.

Research Question Three: – *Reassessment of automated classifications.*

(a) *Does* Verve *effectively classify collision, element protrusion, and viewport protrusion failure reports for the new responsive web pages and how does it compare with results from the initial set of subjects?*

(b) *Does* Verve *effectively classify wrapping failure reports for the new responsive web pages and how does it compare with results from the initial set of subjects?*

(c) *Does* Verve *effectively classify small-range failure reports for the new responsive web pages and how does it compare with results from the initial set of subjects?*

Since this question aims to reassess the performance and therefore is essentially a repeat of the first two research questions in both this chapter and the prior one, the same methodology was followed to reach a conclusion about Verve's automated classifications when using a new set of subjects referred to as the *additional set* of subjects. Furthermore, the results are compared between the two sets for major degradation or improvement in efficacy.

Research Question Four: – *Does* Verve *efficiently classify the failure reports?* To determine if Verve operates fast enough to support its practical use in the testing of responsive web pages, I recorded the time the tool takes to perform failure report classification for all of the subjects.

The design of the experiments set forth to answer these research questions are explained next.

4.4.1 Design of Experiments

In this section, I will identify the subject web pages used in the experiments and their details, the runtime environment used to build Verve and used to run the tool during the experiments, the methodology followed to answer each of the research questions, and finally disclose any known threats to the validity of the results and any mitigating steps taken to reduce these threats.

Subject Web Pages

For the experiments of this chapter, I used a total of 45 web pages with 469 presentation failures to be automatically classified by Verve as part of its empirical evaluation. This total comes from two sets of subjects. The first set, called the *initial set*, is the same set of subject web pages used to evaluate Verve's (Viser's, before rebranding) element collision, element protrusion, and viewport protrusion classification algorithms. Moreover, this is the same set of web pages used to evaluate ReDeCheck's effectiveness by Walsh et al. [124]. Its usage is now extended to evaluate the element wrapping and small-range classification algorithms introduced into the tool for this chapter. To reassess the performance of Verve on the initial set of subjects, I further evaluated all of Verve's algorithms on 20 new subjects, called the *additional set* of pages.

The initial set of subjects that is comprised of 25 web pages and made available by Walsh et al. [124] are listed in Table 4.1(a). I took these subjects, without modification, directly from the repository cited in Walsh et al.'s paper (`github.com/redecheck/example-webpages`). However, as I previously disclosed in Section 3.4.1, although 26 subjects were used in their study, the StumbleUpon subject was no longer usable since some of the resources required by the page are no longer hosted online. Nevertheless, the initial set has a total of 326 failures reported by ReDeCheck for the evaluation of Verve. This figure includes the 117 failures used to evaluate Viser in the previous chapter. For the

manual classifications of these 326 failures that are used as a benchmark against VERVE's automated classification, I used the classifications made by Walsh et al. [124] that are publicly available at `redecheck.org/issta17`.

The additional set of subjects used in the experiments of this chapter is the 20 web pages shown in Table 4.1(b). This set is made up of a collection of seven real web pages and 13 example web pages for themes. Although these Bootstrap-based example web pages are not meant to be hosted as is, they demonstrate features for different types of web pages. For instance, among these 13 themes is a full-featured template for web dashboards, advertising agencies, and art portfolios. Moreover, these web pages are maintained in popular GitHub repositories and publicly hosted in the "Blackrock Digital" organization's site at `github.com/blackrockdigital`. Their repositories are also regularly updated, frequently forked, and starred. This gives a strong indication that the responsive web development community views them as useful templates for usage in their own websites. The popularity of these themes can be illustrated by noting that at the time of using subjects, the SB-Admin-2 subject was created by 14 contributors who made 19 releases to the project that has over 4,400 forks and 7,300 stars. I was able to collect this additional set of subjects by searching the public GitHub repositories for demonstration based responsive web pages. As for the seven real web pages, they were derived from another pool of pages used in Walsh's PhD thesis [128] for further evaluation of improvements to the REDECHECK tool. Collectively, this set has 20 web pages with a total of 143 presentation failures reported by REDECHECK.

Runtime Environment

As done with the runtime setup of VISER, see Section 3.4.1, I attempted to match the environment of the original evaluation of REDECHECK by using the same machine once again for evaluating VERVE. This is to avoid any discrepancies in the results that are due to differences in the experimental setup for the evaluation of the three tools. Thus I ran VERVE on an iMac with 8GB of RAM, running OS version 10.13 and using version 46 of the Firefox browser. Similar to REDECHECK and VISER, the new tool VERVE uses Selenium WebDriver [105] to interact with the web browser that is set to a fixed viewport height of 1000 pixels and set to render the web pages without scrollbars. To support the automated classification of small-range failures, I integrated OpenCV [87] version 3.2 into VERVE. This is to allow for the generation and comparison of colour histograms of images captured from the layout under investigation.

Methodology

Throughout Section 4.4.2 where I answer the research questions, I will discuss instances where VERVE disagrees with the manual classification in three categories: *subjective*, *obscured*, and *misclassified* failures. For *subjective* disagreements, VERVE classified the failure report as a true positive but based on a human's subjective judgement of the failure, the report was dismissed. This is because the visual discrepancy caused by the failure is not substantial and therefore only amounts to a few pixels or it is imperceptible to the human eye. This subjectivity may also carry over to the *obscured* disagreement. For the *obscured* disagreements, the REDECHECK tool reported and VERVE subsequently analysed different HTML elements to those actually causing the visual anomaly. Therefore, the final judgement is obscure since both approaches can be considered correct. For the *misclassified* disagreements, either VERVE or the human classification was deemed incorrect in my final analysis. These three categories were also described in Section 3.4.1 and used during

Table 4.1: Experimental subject web pages.

(a) The initial set of web pages.

Web Site Name	URL	HTML Elements	CSS Declarations
3-Minute-Journal	3minutejournal.com	80	5408
AccountKiller	accountkiller.com/en	344	4685
AirBnb	airbnb.com	1470	9964
BugMeNot	bugmenot.com	42	654
CloudConvert	cloudconvert.com	908	6494
Consumer-Reports	consumerreports.org	1079	8330
Covered-Calendar	coveredcalendar.com	148	8324
Days-Old	daysold.com	66	2800
Dictation	dictation.io	195	8290
Duolingo	duolingo.com	856	4150
Honey	joinjoney.com/install	461	7999
HotelWifiTest	hotelwifitest.com	359	6833
Mailinator	mailinator.com	280	8729
MidwayMeetup	midwaymeetup.com	86	4072
Ninite	ninite.com	642	4091
Pdf-Escape	pdfescape.com	180	2041
Pepfeed	pepfeed.com	343	7341
Pocket	getpocket.com	664	6416
RainyMood	rainymood.com	89	112
RunPee	runpee.com	438	14788
TopDocumentary	topdocumentaryfilms.com	411	1521
UserSearch	usersearch.org	866	3590
WhatShouldIReadNext	whatshouldireadnext.com/search	112	2224
WillMyPhoneWork	willmyphonework.net	782	6572
ZeroDollarMovies	zerodollarmovies.com	247	11228
Total		11148	146656

(b) The additional set of web pages.

Web Site Name	URL	HTML Elements	CSS Declarations
EatThisMuch	eatthismuch.com	807	12318
Forvo	forvo.com	584	19722
GMapStreetViewPlayer	brianfolts.com/driver	268	5617
RetailMeNot	retailmenot.com	1336	2823
HoursOf	hoursof.com	1258	9513
SB-Admin-2	startbootstrap.com/themes/sb-admin-2	360	6656
SB-Agency	startbootstrap.com/previews/agency	420	6303
SB-Business-Casual	startbootstrap.com/previews/business-casual	56	4438
SB-Clean-Blog	startbootstrap.com/previews/clean-blog	93	6160
SB-Coming-Soon	startbootstrap.com/previews/coming-soon	43	5969
SB-Creative	startbootstrap.com/previews/creative	135	6318
SB-Freelancer	startbootstrap.com/previews/freelancer	284	6064
SB-Grayscale	startbootstrap.com/previews/grayscale	116	6120
SB-Landing-Page	startbootstrap.com/previews/landing-page	130	6146
SB-New-Age	startbootstrap.com/previews/new-age	127	6649
SB-One-Page-Wonder	startbootstrap.com/previews/one-page-wonder	68	4424
SB-Resume	startbootstrap.com/previews/resume	176	5924
SB-Stylish-Portfolio	startbootstrap.com/previews/stylish-portfolio	143	6363
SimilarSites	similarsites.com	478	10268
Tiiime	tiii.me	80	847
Total		6962	138642

the evaluation of VISER. As done in the previous chapter, I discuss the reasons for the disagreement and identify the subject web page while outlining possible future work to improve VERVE, if any work is needed.

RQ1 Methodology – To answer RQ1, I used VERVE to classify each of the 14 wrapping failures detected in the initial set of 25 web pages. Given the failure viewport range of each responsive layout failure, in the form $\{fail_{min}..fail_{max}\}$, I instructed VERVE to inspect and automatically classify the reported wrapping failures at $fail_{min}$ referred to as the *minimum* viewport, $fail_{mid}= floor((fail_{min}+fail_{max})/2)$ referred to as the *middle* viewport, and at $fail_{max}$ referred to as the *maximum* viewport of the failure range. For each of these viewports, the tool outputs an automatic classification for each reported failure as either a TP, an NOI, or an FP.

I then checked whether VERVE agreed with the human manual classifications of the reported failure as decided in the original study by Walsh et al. [124]. Where it can be either a true positive (TP, an observable failure), non-observable issue (NOI), or false positive (FP, no failure). Here, the FPs are reports by REDECHECK that do not exhibit an issue visually in the rendering of the web page nor in its internal DOM representation. On the other hand, NOIs are reports that are not visually problematic but have a confirmable issue in the underlying DOM structure. Finally, TPs, are failures that manifest visually in the rendered layout and in the underlying DOM structure. To reach a conclusion, I calculated the percentage of agreement between VERVE and the manual classification and investigated any differences in classifications.

RQ2 Methodology – To answer RQ2, I used VERVE to automatically classify the 195 small-range failures detected in the initial set of web pages. Since small-range failures are restricted to a range of 1–5 viewports, only the minimum viewport, $fail_{min}$, is used to automatically classify the failure. While the viewports immediately narrower and wider than the small-range failure are used as comparison points as discussed in Section 4.3.2. These two viewports respectively correspond to the $fail_{min}-1$ and $fail_{max}+1$ values of the failure range. To establish the best histogram comparison measure, I used the five of the methods available in the OpenCV library. More specifically, these were *Bhattacharyya Distance*, *Chi-Square*, *Alternative Chi-Square*, *Correlation*, and *Intersection* [87]. For the *Bhattacharyya Distance*, *Chi-Square*, and *Alternative Chi-Square* metrics, the lower the distance value, the better the match, with zero representing a perfect match. The *Correlation* and *Intersection* metrics, however, use a higher-is-better score. *Correlation*'s score is bounded at one, while *Intersection*'s score is unbounded. So that I could easily and consistently compare the results using each metric, I wrote a wrapper function around *Correlation* and *Intersection* to convert its result to a lower-is-better score, as is the default for the other metrics. The wrapper for *Correlation* inverts its score, while the wrapper for *Intersection* normalizes its score and inverts its result.

In order for VERVE to conclude that it has detected enough of a visual disturbance to classify a small-range failure as TP, I needed to set a TP threshold for each of the five metrics. To establish the thresholds, I made a preliminary run of VERVE to find the AOCs, create a colour histogram for each AOC, and output the *distance* between all pairs of histograms. Throughout this process, each pair of histograms was sourced from the failure viewport and a comparison viewport, as explained in Algorithm 6. To automate the process and ensure its correctness, I implemented a helper tool called THRESHOLDFINDER and used it to establish a TP threshold for each of the five measures with the ultimate goal of maximizing accuracy. That is, the goal of THRESHOLDFINDER is to automatically find the thresholds that will maximize agreement with the manual classification. This tool takes the distances as input and uses them as candidate thresholds along with ± 0.01 of

each distance. Alternating through each candidate threshold, ThresholdFinder automatically classifies all failures in the set based on the candidate threshold. The resulting classifications assigned by ThresholdFinder are then compared to the manual classification to calculate the accuracy for each candidate threshold. When matching against the manual classification, ThresholdFinder used a balanced score for both classes, namely TP and FP. Ultimately, the tool reported the threshold with the maximum accuracy from the set of candidate thresholds. Whenever multiple candidate thresholds may achieve the same accuracy, ThresholdFinder reports the threshold in the middle position of the identified set.

With two alternative approaches implemented in Verve to classify small-range failures, *horizontal referencing* and *horizontal-plus-vertical referencing*, this required that I establish two sets of thresholds, one for each approach. Therefore, I ran ThresholdFinder twice using the failures reported from the initial set of web pages, once for each approach. I refer to both these sets of thresholds determined before running the real experiments as the *prospective* thresholds. To establish whether these thresholds are more generally applicable, the prospective thresholds are also used by Verve to automatically classify the small-range failure from the additional set of web pages as part of RQ3(c). While the prospective thresholds were determined using the initial set of subjects, I used ThresholdFinder again to determine *retrospective* thresholds using all 45 web pages after running the experiments. As part of RQ3 (c), the retrospective thresholds are used to weigh in on the overall performance of Verve as compared to when the tool used the prospective thresholds. This should solidify or correct any findings that may be influenced by a change in the thresholds retrospectively.

RQ3 Methodology – To answer RQ3, I assessed how well all five failure type classification algorithms implemented in Verve extend to the 20 subject web pages in the additional set. Similar to the methodology of the research questions involving all but the small-range failure type, I set Verve to investigate and classify the $fail_{min}$, $fail_{mid}$, and $fail_{max}$ viewports of each reported failure range. For small-range failures, I once again employed all five histogram methods and used the ThresholdFinder-established *prospective* thresholds from RQ2 to classify the failures reported from the additional set. Unlike the previous experiments, for this new set of web pages, I manually classified the failures produced by ReDeCheck in advanced to be able to compare them with the classifications automatically produced by Verve.

To control a validity threat arising from the lack of true positive small-range failures in the additional set of pages reported by ReDeCheck, I manually injected faults into each page from the additional set to create small-range failures suitable for analysis as part of this research question. To complete this task, I manually selected a target element that seemed likely to cause a layout failure if displaced. Then I made two changes to the coding of the web page. First, I manually injected a single media query rule into each page, with the viewport range of 992–995 pixels wide. Secondly, I added CSS rules nested within this query pertaining to the `margin-left` or `margin-top` property of the chosen element to cause it to be offset from its original position only within the range of the media query. Both of these modifications were intended to produce a layout failure limited to a small number of viewport widths that ReDeCheck would report as a small-range failure. I picked the range of 992–995 pixels since the majority of the additional set of subjects had a manually programmed breakpoint in the CSS at 992 pixels, making that position in the viewport range a realistic position where a small-range defect may occur in practice.

RQ4 Methodology – To answer RQ4, I ran Verve to classify all 469 failures from both sets that are comprised of 45 web page subjects. Instructing the tool to examine

each report at the minimum viewport of the failure range reported by ReDeCheck, $fail_{min}$, and recording the time it takes for Verve to classify each failure. I repeated this process 30 times for each failure to obtain a reliable estimate of Verve's running time and to minimize the chance of effects that might be caused by, for example, the underlying operating system hosting the experiments.

Influencing the runtime of Verve is an added delay of 200 milliseconds that is used for all web pages and experiments in this chapter. This is to allow the HTML elements to load and "settle" into their final location to support, for instance, any programmed temporary transitional effects. This delay time is repeated whenever Verve resizes the browser, scrolls the web page, or changes the opacity of an element. Meaning, that multiple delays may be introduced throughout the entire classification process. Therefore, I recorded Verve's execution times with and without this added delay. Importantly, I excluded the time it takes to load the web page and resize the browser as this cost is shared by any technique, whether manual, semi-automated, or automated. All of the recorded times account for the overhead of finding the offending HTML elements, classifying the failure with Verve, and writing to disk all the diagnostic images of the web pages.

Threats to Validity

One threat to the validity of this chapter's results is the extent to which they generalize to other web pages. However, care was taken to ensure the overall set of subjects was diverse in terms of functionality and complexity. Meaning, that they should represent a wide variety of web pages. As Tables 4.1 (a) and (b) show, the subjects vary considerably in complexity from 42 to 1470 HTML elements and from 112 to 19722 CSS declarations. The functionality and responsive layout of the chosen web pages also differ substantially; from SB-Admin-2, a template for back-end administrative portals; to DaysOld, providing calendar features; and AirBnb, supporting international e-commerce corporations.

Another threat related to the pool of subjects used in my experiments is that the additional set of subjects did not contain any small-range failures and only had false positive reports. Although it is difficult to find candidate subjects for ReDeCheck that contain detectable presentation failures, it is much more difficult to find small-range failures as evident by the final set of subjects used in the experiments. Nevertheless, the ReDeCheck tool detects and reports many false positives which will require manual or automated classification in order to realize that these are in truth false positives. To control the threat of the missing true positives, I manually injected code to create this failure in the new subjects as detailed in Section 4.4.1. Even though these failures are synthetic, and so may not be representative of all real-world small-range failures, they exemplify the concerns that practising web developers have about this type of layout defect [22].

As with the experiments of the previous chapter, the validity of this chapter's experiments depended on accurately matching the manual classifications of ReDeCheck's failure reports with the automatically produced classifications. Since the manual classifications, involving the initial set of subjects, from the experimental evaluation of ReDeCheck [124] did not include the XPath of offending elements, the failures were manually matched using the available snapshots. These snapshots, combined with the type of failure, range, and name of the web page enabled me to confidently perform this matching. For the additional set of subjects, I undertook the manual classification process of the failures found in the additional set. Since the conclusions of the manual process are inherently subjective, I made my classification publicly available for independent evaluation at

`verve-tool.github.io`. Importantly, many of the failures reported by ReDeCheck for the additional set of subjects had very little subjectivity. Therefore, limiting this validity threat since they were easy to classify.

A further threat to validity is a defective implementation of the Verve tool, which would compromise its classification of the responsive layout failures reported by ReDeCheck. To control this threat, I programmed and configured Verve to keep a record of all the images used to evaluate each responsive layout failure. Moreover, I programmed Verve to also maintain a record of the coordinates of each offending element. I consulted these records during the examination of all classifications in disagreement. Thus helping me verify that the prototype operated correctly.

In relation to the threat surrounding the implementation of the tool, Verve's usage of the Firefox web browser, the Selenium testing tool, and the OpenCV library may be an additional threat. Although the tool is limited to one browser, Firefox is a popular browser that is frequently used for responsive web page testing and therefore a good option for ensuring that the results are representative of the real world. I am also confident that Selenium did not compromise Verve's correctness because, in addition to my error-free experiences when using Selenium, this web automation framework has a large and active community that is committed to reporting and fixing defects. Similarly, defects in the OpenCV library are also a validity threat that may compromise this chapter's experimental results. Even though OpenCV is a widely used library for image analysis, I addressed this concern by manually confirming OpenCV's analysis of a few selected images and by consulting the manuals of OpenCV for its proper usage when needed (e.g., [16]).

Finally, since the runtime results for RQ4 are subject to the interference of background operating system processes, I ran all of the experiments 30 times to reduce the chance of them influencing the recorded timing. Importantly, to support the replication of this chapter's experiments and to further control all the aforementioned validity threats, I made the Verve tool, its documentation, and the scripts needed to run the experiments all available in a GitHub repository at `github.com/verve-tool/verve`. To further support the confirmation of this chapter's results, I have made a screenshot of every ReDeCheck failure report, its manual classification, and details about Verve's automatic classification available at the `verve-tool.github.io` site.

4.4.2 Results of Experiments

Answer to RQ1 – For wrapping failures, Table 4.2 shows the results of using Verve to classify the reports of wrapping failures, produced by ReDeCheck, from the initial set of 25 web pages. Moreover, it shows the agreement with the manual classifications that were classified by Walsh et al. [124] broken down to the subject web page level. The overall results show that the two classification approaches agree 78.6% of the time for all viewports used by the tool for classification. These were the minimum, middle, and maximum values from the reported viewport failure range. In the table, the result from each viewport is described in a separate column accordingly labelled. It should also be noted that the table describes Verve's classification in the numerator position and the manual classification in the denominator position. For completeness, Table 4.3 gives the full manual classification results from the original Walsh et al. study in table (a). For ease of comparison, my manual classification of the failures reported from the additional set of subjects, that will be used to answer RQ3, are introduced in table (b).

Out of a total of 14 reported failures, Verve disagreed with the manual analysis for three element wrapping failures. Two of these failures come from the Duolingo subject.

Table 4.2: VERVE's results of automatically classifying the wrapping failures reported from the initial set of subjects using three viewports from the failure range. As explained in Section 4.4.1, these three viewports are the minimum, middle, and maximum values of the reported failure range. In this table "TP", "NOI", and "FP" respectively denote a true positive, non-observable issue, and false positive.

	Wrapping									
	Minimum			**Middle**			**Maximum**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
3-Minute-Journal	-	-	-	-	-	-	-	-	-	-
AccountKiller	1/2	-	1/-	1/2	-	1/-	1/2	-	1/-	2
AirBnb	2/2	-	-	2/2	-	-	2/2	-	-	2
BugMeNot	1/1	-	-	1/1	-	-	1/1	-	-	1
CloudConvert	-	-	-	-	-	-	-	-	-	-
Consumer-Reports	-	-	-	-	-	-	-	-	-	-
Covered-Calendar	2/2	-	-	2/2	-	-	2/2	-	-	2
Days-Old	-	-	-	-	-	-	-	-	-	-
Dictation	-	-	-	-	-	-	-	-	-	-
Duolingo	2/-	-	-/2	2/-	-	-/2	2/-	-	-/2	2
Honey	-	-	-	-	-	-	-	-	-	-
HotelWifiTest	-	-	-	-	-	-	-	-	-	-
Mailinator	-	-	-	-	-	-	-	-	-	-
MidwayMeetup	-	-	-	-	-	-	-	-	-	-
Ninite	1/1	-	1/1	1/1	-	1/1	1/1	-	1/1	2
Pdf-Escape	-	-	-	-	-	-	-	-	-	-
Pepfeed	1/1	-	-	1/1	-	-	1/1	-	-	1
Pocket	-	-	-	-	-	-	-	-	-	-
RainyMood	-	-	-	-	-	-	-	-	-	-
RunPee	-	-	1/1	-	-	1/1	-	-	1/1	1
TopDocumentary	-	-	-	-	-	-	-	-	-	-
UserSearch	1/1	-	-	1/1	-	-	1/1	-	-	1
WhatShouldIReadNext	-	-	-	-	-	-	-	-	-	-
WillMyPhoneWork	-	-	-	-	-	-	-	-	-	-
ZeroDollarMovies	-	-	-	-	-	-	-	-	-	-
Total failures	11	-	3	11	-	3	11	-	3	14
Agreement with manual	9/10	-	2/4	9/10	-	2/4	9/10	-	2/4	-
Per inspection point	78.6 %			78.6 %			78.6 %			-

In both cases, the HTML elements are text-based links that form several rows in the footer of the page. For reference, Figure 4.8 (a) showcases a snapshot of the footer being described. Clearly, the developer intended for these elements to naturally wrap around as the viewport size gets smaller, even if this would single out an element on its own row. Therefore, the manual analysis classified these failures as false positives. On the other hand, VERVE reported them as true positives. Justifiably, this difference is categorized as a misclassification by VERVE. Since VERVE correctly detected a visual change in the layout but was semantically incorrect, the shortcoming can also be attributed back to the wrapping failure detection algorithm implemented in REDECHECK.

The third and final classification of a wrapping failure in non-agreement comes from AccountKiller subject. This failure is showcased in Figure 4.8 (b) using the maximum viewport from the reported failure range. Here, the Twitter social media button was detected to have wrapped below a row of 15 other elements. This failure was manually classified as a true positive while being automatically classified as a false positive by VERVE. As part of the DOM checking phase used to filter out false positives before analysing the AOCs, see Section 4.3.2, VERVE concluded that the REDECHECK reportedly wrapped element is not below all the 15 row elements. More specifically, it was not completely positioned below the container of the Google Plus social media icons sitting between the third, from the left-hand side, square icon of a white envelope and the Pinterest

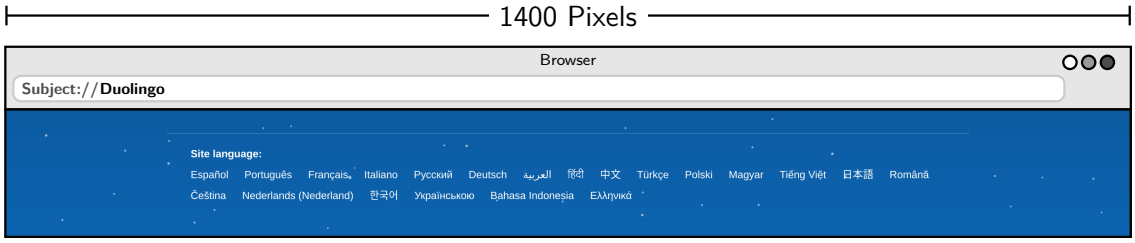
Table 4.3: The manual classification of all failure reports used in the experiments.

(a) Manual classification of failures from the initial set of web pages from a published study [124].

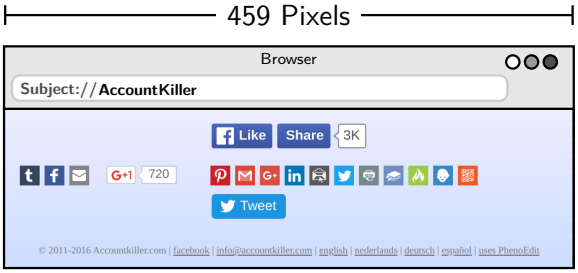
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			**Wrapping**			**Small-range**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
3-Minute-Journal	-	1	-	-	2	-	8	-	-	-	-	-	-	-	1	12
AccountKiller	-	-	-	-	-	-	-	-	-	2	-	-	147	-	5	154
AirBnb	-	1	-	-	4	-	-	4	-	2	-	-	-	-	2	13
BugMeNot	-	-	-	1	3	-	2	-	-	1	-	-	-	-	-	7
CloudConvert	1	-	-	-	-	-	-	-	-	-	-	-	1	-	-	2
Consumer-Reports	-	7	-	1	3	-	9	3	-	-	-	-	-	-	1	24
Covered-Calendar	-	-	-	-	-	-	-	3	-	2	-	-	-	-	-	5
Days-Old	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	1
Dictation	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	1
Duolingo	-	1	-	-	-	-	2	2	-	-	-	2	-	-	1	8
Honey	-	-	-	-	8	-	-	2	-	-	-	-	-	-	3	13
HotelWifiTest	-	-	-	-	-	-	1	-	-	-	-	-	-	-	2	3
Mailinator	-	1	-	-	-	-	-	-	-	-	-	-	-	-	2	3
MidwayMeetup	1	-	-	-	1	-	-	1	-	-	-	-	-	-	-	3
Ninite	-	-	-	-	-	-	-	-	-	1	-	1	-	-	-	2
Pdf-Escape	-	-	-	1	5	-	1	3	-	-	-	-	-	-	-	10
Pepfeed	4	3	-	-	2	-	1	1	-	1	-	-	2	-	14	28
Pocket	-	2	-	-	3	-	-	-	-	-	-	-	-	-	3	8
RainyMood	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RunPee	-	-	-	-	-	-	-	-	-	-	-	1	-	-	5	6
TopDocumentary	-	7	-	-	4	-	-	-	-	-	-	-	-	-	2	13
UserSearch	-	1	-	-	-	-	-	-	-	1	-	-	-	-	-	2
WhatShouldIReadNext	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-	2
WillMyPhoneWork	1	-	-	-	1	-	-	-	-	-	-	-	2	-	-	4
ZeroDollarMovies	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2	2
Total failures	7	24	-	3	36	-	24	23	-	10	-	4	152	-	43	326
Total per failure type		31			39			47			14			195		-

(b) The manual classification of the failures reported from the additional set of subject web pages.

	Element Collision			**Element Protrusion**			**Viewport Protrusion**			**Wrapping**			**Small-range**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
EatThisMuch	-	5	-	-	6	-	1	1	-	-	-	1	-	-	2	16
Forvo	-	-	-	-	3	-	-	-	-	2	-	2	-	-	29	36
GMapStreetViewPlayer	-	-	-	-	2	-	-	-	-	-	-	-	-	-	-	2
HoursOf	-	-	-	-	1	-	-	1	-	-	-	-	-	-	-	2
RetailMeNot	2	-	-	-	30	-	-	-	-	-	2	4	-	-	15	53
SB-Admin-2	-	-	-	-	-	-	1	-	-	-	-	-	-	-	1	2
SB-Agency	-	4	-	-	8	-	1	-	-	-	-	3	-	-	-	16
SB-Business-Casual	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SB-Clean-Blog	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SB-Coming-Soon	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SB-Creative	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SB-Freelancer	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SB-Grayscale	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	1
SB-Landing-Page	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	1
SB-New-Age	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SB-One-Page-Wonder	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SB-Resume	-	1	-	-	-	-	-	-	-	2	-	-	-	-	-	3
SB-Stylish-Portfolio	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	4
SimilarSites	-	-	-	-	-	-	4	-	-	-	-	-	-	-	1	5
Tiiime	-	-	-	-	1	-	-	-	-	-	-	-	-	-	1	2
Total failures	2	10	-	-	51	-	8	2	-	4	2	11	-	-	53	143
Total per failure type		12			51			10			17			53		-



(a) The snapshot shows the footer at the bottom of the Duolingo subject using the widest viewport tested by ReDeCheck. From this wide viewport all the way to the narrowest, 320 pixels, the collection of text-based elements never formed a single row and some of the elements naturally wrap as the viewport gets smaller.



(b) A snapshot from the AccountKiller subject captured at the widest viewport where the failure occurred, known as the maximum viewport. In this viewport, the Twitter social media button wraps below the other 15 social media elements that ReDeCheck deemed to be in a row. This failure was manually classified as true positive while Verve classified it as a false positive.

Figure 4.8: The figure presents two subject web pages with failure reports that were misclassified by Viser. The first subject, Duolingo, is showcased in part (a) while the second subject, AccountKiller, can be seen in part (b).

square icon. This effect is not apparent in the rendered layout but is true at the lower DOM structure level. This misclassification by Verve can be overcome by introducing into the tool a sufficient distance-based tolerance for elements not quite below the row. Subjectively, this can also be considered an acceptable formation for the elements and hence be labelled as a false positive report.

Conclusion for RQ1 – Verve consistently classified wrapping failures throughout the reported failure range. Therefore, a single viewport chosen for classification should be sufficient for automatic classification. Importantly, Verve achieves 78.6% agreement with the manual analysis for wrapping failures with the initial set of subjects. Further work on both ReDeCheck and Verve is required to improve the analysis of the three cases where the manual analysis disagreed with Verve.

Answer to RQ2 – Table 4.4 summarizes the results for automatically classifying small-range failures using the horizontal referencing approach with the five distance metric integrated into Verve. The table also compares Verve's classification with the manual classification for agreement. As for the threshold values used in the experiment, they are reported in the headings against each metric using the ϵ symbol. These values were automatically established during preliminary experimentation using the ThresholdFinder tool as explained in the methodology section.

The results show that the level of agreement can be as high as 97.4% using the *In-*

tersection method or as low as 87.2% with the *Correlation* method. A close second to the top position was the *Chi-Square* method performing at 96.4% followed by 93.3% using the *Alternative Chi-Square* method. In fourth place comes *Bhattacharyya Distance* with 91.8% followed by the lowest performer *Correlation*. All of which performed very well when compared, solely based on numbers, to the results of automatically classifying wrapping failure seen in answer to the first research question.

For the highest performer, *Intersection*, there were a total of five classification disagreements to blame for the imperfection in the results. From these five, VERVE misclassified two as true positives and three as false positives. Nevertheless, two of these failures, from the CloudConvert and WillMyPhoneWork subjects, happen to be duplicate reports of element collision failures that VERVE was able to successfully classify with its element collision algorithm. Suggesting, that when a small-range failure is also an instance of another type of failure, it is better to use VERVE's other classification algorithms specifically tailored for that type of failure. I expect the efficacy of the tool to improve, for future work to prove, if the small-range histogram method is reserved for situations where there is no duplication in the reporting of the failure.

Table 4.4: The results of using VERVE to classify the small-range failures detected in the initial set of web pages while configured to use the horizontal referencing approach described in Section 4.3.2. This table features five histogram comparison measures and uses ϵ to denote the threshold value used for each method.

	Small-range horizontal referencing										
	Bhattacharyya Distance $\epsilon = 0.20$		**Chi-Square** $\epsilon = 0.11$		**Alternative Chi-Square** $\epsilon = 0.14$		**Correlation** $\epsilon = 0$		**Intersection** $\epsilon = 0.09$		
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	**Total**
3-Minute-Journal	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
AccountKiller	137/147	15/5	147/147	5/5	147/147	5/5	128/147	24/5	147/147	5/5	152
AirBnb	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
BugMeNot	-	-	-	-	-	-	-	-	-	-	-
CloudConvert	-/1	1/-	1/1	-	-/1	1/-	-/1	1/-	-/1	1/-	1
Consumer-Reports	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Covered-Calendar	-	-	-	-	-	-	-	-	-	-	-
Days-Old	-	-	-	-	-	-	-	-	-	-	-
Dictation	-	-	-	-	-	-	-	-	-	-	-
Duolingo	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Honey	1/-	2/3	3/-	-/3	3/-	-/3	3/-	-/3	2/-	1/3	3
HotelWifiTest	1/-	1/2	1/-	1/2	2/-	-/2	-	2/2	-	2/2	2
Mailinator	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
MidwayMeetup	-	-	-	-	-	-	-	-	-	-	-
Ninite	-	-	-	-	-	-	-	-	-	-	-
Pdf-Escape	-	-	-	-	-	-	-	-	-	-	-
Pepfeed	-/2	16/14	5/2	11/14	5/2	11/14	-/2	16/14	2/2	14/14	16
Pocket	-	3/3	-	3/3	-	3/3	-	3/3	-	3/3	3
RainyMood	-	-	-	-	-	-	-	-	-	-	-
RunPee	1/-	4/5	-	5/5	3/-	2/5	-	5/5	-	5/5	5
TopDocumentary	-	2/2	-	2/2	1/-	1/2	-	2/2	-	2/2	2
UserSearch	-	-	-	-	-	-	-	-	-	-	-
WhatShouldIReadNext	-	-	-	-	-	-	-	-	-	-	-
WillMyPhoneWork	2/2	-	2/2	-	2/2	-	2/2	-	-/2	2/-	2
ZeroDollarMovies	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
Total	142	53	159	36	163	32	133	62	151	44	195
Agreement with manual	139/152	40/43	152/152	36/43	151/152	31/43	130/152	40/43	149/152	41/43	-
Agreement per measure	91.8 %		96.4 %		93.3 %		87.2 %		97.4 %		-

For the results based on the alternative horizontal-plus-vertical referencing approach, see Table 4.5. The thresholds determined specifically for this approach are also reported as ϵ values in the table. Once again, *Intersection* outperformed the other four metrics with only three failure misclassifications resulting in a 98.5% agreement outcome. Ranking in the second position with an agreement of 97.9% are the *Bhattacharyya Distance* and

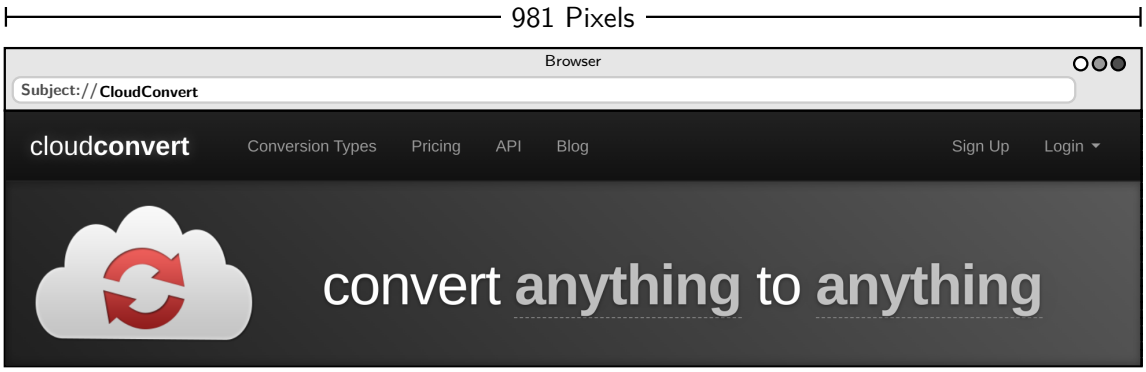
Correlation measures after misclassifying a total of four failures each. Then comes the *Chi-Square* measure which had an agreement of 97.4% after misclassifying five responsive layout failures. With all measures achieving a high agreement with the manual classification, the lowest was *Alternative Chi-Square* at 96.9%, with a total of six misclassifications.

Table 4.5: The results of using Verve to classify the small-range failures detected in the initial set of web pages while configured to use the horizontal-plus-vertical referencing approach. Here, ϵ denotes the threshold value used in the experiment.

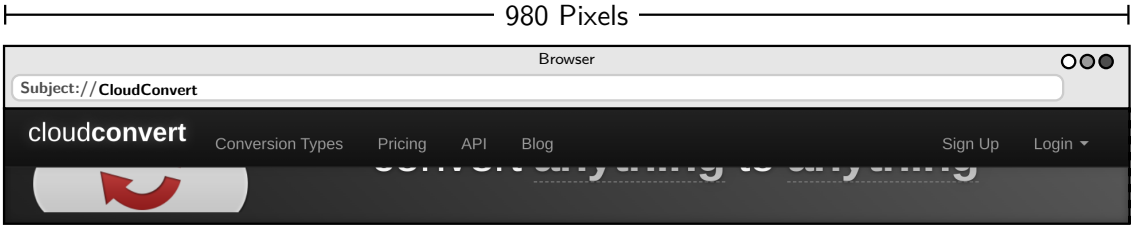
	Small-range horizontal-plus-vertical referencing										
	Bhattacharyya Distance $\epsilon = 0.23$		**Chi-Square** $\epsilon = 0.46$		**Alternative Chi-Square** $\epsilon = 0.82$		**Correlation** $\epsilon = 0$		**Intersection** $\epsilon = 0.15$		
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	**Total**
3-Minute-Journal	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
AccountKiller	147/147	5/5	147/147	5/5	147/147	5/5	147/147	5/5	147/147	5/5	152
AirBnb	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
BugMeNot	-	-	-	-	-	-	-	-	-	-	-
CloudConvert	-/1	1/-	-/1	1/-	-/1	1/-	-/1	1/-	-/1	1/-	1
Consumer-Reports	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Covered-Calendar	-	-	-	-	-	-	-	-	-	-	-
Days-Old	-	-	-	-	-	-	-	-	-	-	-
Dictation	-	-	-	-	-	-	-	-	-	-	-
Duolingo	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Honey	-	3/3	2/-	1/3	1/-	2/3	3/-	-/3	-	3/3	3
HotelWifiTest	1/-	1/2	-	2/2	-	2/2	-	2/2	-	2/2	2
Mailinator	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
MidwayMeetup	-	-	-	-	-	-	-	-	-	-	-
Ninite	-	-	-	-	-	-	-	-	-	-	-
Pdf-Escape	-	-	-	-	-	-	-	-	-	-	-
Pepfeed	-/2	16/14	2/2	14/14	-/2	16/14	2/2	14/14	2/2	14/14	16
Pocket	-	3/3	-	3/3	-	3/3	-	3/3	-	3/3	3
RainyMood	-	-	-	-	-	-	-	-	-	-	-
RunPee	-	5/5	-	5/5	-	5/5	-	5/5	-	5/5	5
TopDocumentary	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
UserSearch	-	-	-	-	-	-	-	-	-	-	-
WhatShouldIReadNext	-	-	-	-	-	-	-	-	-	-	-
WillMyPhoneWork	2/2	-	-/2	2/-	-/2	2/-	2/2	-	-/2	2/-	2
ZeroDollarMovies	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
Total	150	45	151	44	148	47	154	41	149	46	195
Agreement with manual	149/152	42/43	149/152	41/43	147/152	42/43	151/152	40/43	149/152	43/43	-
Agreement per measure	97.9 %		97.4 %		96.9 %		97.9 %		98.5 %		-

Shedding light on a limitation of both referencing approaches, implemented in Verve that are used to classify failures via histogram distances, is the only small-range failure reported from the CloudConvert subject. Although a duplicate element collision failure report was correctly classified by Verve, the small-range failure was misclassified using all five metrics of the horizontal-plus-vertical referencing approach and five out of six from the horizontal referencing approach. This layout failure, seen in Figure 4.9 (b), results in the loss of content from the header of the page as a consequence of collapsing under the top menu of the page. It is important to note that the elements reported as failing, the header and the top menu, span the entire viewport width of the page and are positioned at the top of the page. Therefore, there is nothing to the right or left of the elements to capture for referencing, using either approach, and minimal to none on the top. Although lots of content exists below both elements for referencing, it is basically the content of the entire page. Hence defeating the purpose of using it for referencing the position of the elements relative to other content in the page. In the future, Verve can potentially overcome this edge case by first catching any case that lacks referencing and breaking down the only direction available into smaller AOCs.

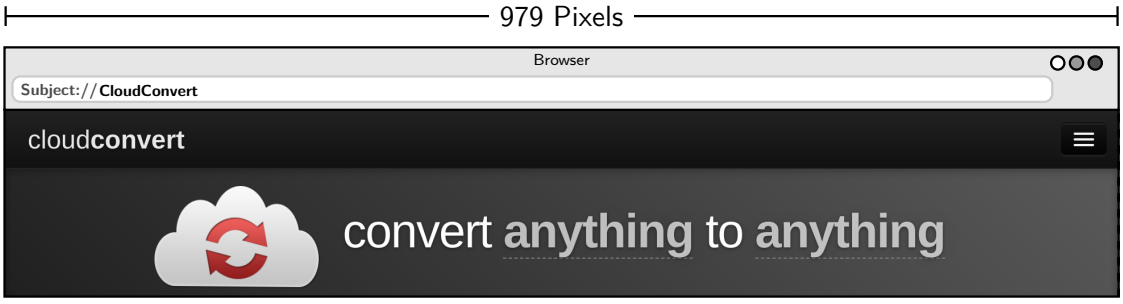
Since the threshold values produced by ThresholdFinder were tuned to the optimal agreement level for both of the referencing approaches implemented Verve, the two can



(a) Captured at one viewport wider than the failure range, the snapshot shows the header of the CloudConvert web page with no failure.



(b) Captures a layout failure in the header of the CloudConvert web page that was detected by ReDeCheck twice, resulting in an element collision report and a duplicate small-range failure report.



(c) Captured at one viewport narrower than the failure range, the snapshot shows the header of the CloudConvert web page with no failure.

Figure 4.9: The figure showcases three snapshots from the CloudConvert web page capturing the layout before the failure occurs at a wider viewport in (a), the failure in (b), and after the failure occurs at the narrower viewport in (c). The failure occurs in only a single viewport and was reported as collision failures and a small-range failure.

be compared for performance. It is clear from the results that the horizontal-plus-vertical referencing approach is the better performer since it achieved the highest agreement level of 98.5% using *Intersection*. More importantly, it virtually scraped the need to carefully choose a specific histogram method with no more than a 1.6% difference between the top and lowest-performing metric. Nevertheless, this referencing approach uses more images and thus requires more processing time and memory than the horizontal referencing approach which performed just as good using *Intersection* or even *Chi-Square*.

Conclusion for RQ2 – When configured to use the horizontal referencing approach, VERVE can classify small-range failures with up to a 97.4% agreement with the manual classification using the *Intersection* distance metric. Even better is the horizontal-plus-vertical referencing configuration which was able to achieve 98.5% agreement using *Intersection* and this configuration is the best for use in VERVE.

Table 4.6: The results from using VERVE for element collision, element protrusion, and viewport protrusion failures of the additional set of web pages after inspecting the "Minimum" of the reported failure range.

	Minimum									
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
EatThisMuch	-	5/5	-	1/-	5/6	-	2/1	-/1	-	13
Forvo	-	-	-	-	3/3	-	-	-	-	3
GMapStreetViewPlayer	-	-	-	-	2/2	-	-	-	-	2
HoursOf	-	-	-	-	1/1	-	1/-	-/1	-	2
RetailMeNot	2/2	-	-	-	30/30	-	-	-	-	32
SB-Admin-2	-	-	-	-	-	-	1/1	-	-	1
SB-Agency	-	4/4	-	3/-	5/8	-	1/1	-	-	13
SB-Business-Casual	-	-	-	-	-	-	-	-	-	-
SB-Clean-Blog	-	-	-	-	-	-	-	-	-	-
SB-Coming-Soon	-	-	-	-	-	-	-	-	-	-
SB-Creative	-	-	-	-	-	-	-	-	-	-
SB-Freelancer	-	-	-	-	-	-	-	-	-	-
SB-Grayscale	-	-	-	-	-	-	1/1	-	-	1
SB-Landing-Page	-	-	-	-	-	-	-	-	-	-
SB-New-Age	-	-	-	-	-	-	-	-	-	-
SB-One-Page-Wonder	-	-	-	-	-	-	-	-	-	-
SB-Resume	-	1/1	-	-	-	-	-	-	-	1
SB-Stylish-Portfolio	-	-	-	-	-	-	-	-	-	-
SimilarSites	-	-	-	-	-	-	4/4	-	-	4
Tiiime	-	-	-	-	1/1	-	-	-	-	1
Total	2	10	-	4	47	-	10	-	-	73
Agreement with manual	2/2	10/10	-	-	47/51	-	8/8	0/2	-	-
Agreement per failure type	100 %			92.2 %			80 %			-
Per inspection point	91.8 %									-

Answer to RQ3 (a) – Table 4.6 presents the results from running VERVE on the element collision, element protrusion, and viewport protrusion failures reported by REDECHECK for the 20 web pages in the additional set of subjects. This table also reports the agreement between the results of VERVE and the manual classification that I performed for the additional set. While this table gives the results from running VERVE at the minimum viewport of the failure range, Tables 4.7 and 4.8 respectively present the results from using the tool at the middle and maximum viewports. It is important to note that all of the

manual classifications from Table 4.3 (b) are shown as the denominator of ratio values in all the result tables.

Table 4.7: The results from using Verve for element collision, element protrusion, and viewport protrusion failures of the additional set of web pages after inspecting the "Middle" of the reported failure range.

	Middle									
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
EatThisMuch	-	5/5	-	1/-	5/6	-	2/1	-/1	-	13
Forvo	-	-	-	-	3/3	-	-	-	-	3
GMapStreetViewPlayer	-	-	-	-	2/2	-	-	-	-	2
HoursOf	-	-	-	-	1/1	-	1/-	-/1	-	2
RetailMeNot	2/2	-	-	-	30/30	-	-	-	-	32
SB-Admin-2	-	-	-	-	-	-	1/1	-	-	1
SB-Agency	-	4/4	-	3/-	5/8	-	1/1	-	-	13
SB-Business-Casual	-	-	-	-	-	-	-	-	-	-
SB-Clean-Blog	-	-	-	-	-	-	-	-	-	-
SB-Coming-Soon	-	-	-	-	-	-	-	-	-	-
SB-Creative	-	-	-	-	-	-	-	-	-	-
SB-Freelancer	-	-	-	-	-	-	-	-	-	-
SB-Grayscale	-	-	-	-	-	-	1/1	-	-	1
SB-Landing-Page	-	-	-	-	-	-	-	-	-	-
SB-New-Age	-	-	-	-	-	-	-	-	-	-
SB-One-Page-Wonder	-	-	-	-	-	-	-	-	-	-
SB-Resume	-	1/1	-	-	-	-	-	-	-	1
SB-Stylish-Portfolio	-	-	-	-	-	-	-	-	-	-
SimilarSites	-	-	-	-	-	-	4/4	-	-	4
Tiiime	-	-	-	-	1/1	-	-	-	-	1
Total	2	10	-	4	47	-	10	-	-	73
Agreement with manual	2/2	10/10	-	-	47/51	-	8/8	0/2	-	-
Agreement per failure type	100 %			92.2 %			80 %			-
Per inspection point	91.8 %									-

At the minimum and middle viewports, there were six failures in disagreement while at the maximum viewport there were an additional three classifications in disagreement. The six at the minimum and middle viewports were all automatically classified by Verve as true positives, whereas during my manual classification I categorized them as non-observable issues. After analysing the differences, I concluded that all six were a misclassification by Verve. Two of which were viewport protrusion failures from the EatThisMuch and HoursOf subjects which had only a few pixels changed that are not visible to the human eye. Three were protrusion failures from the SB-Agency web page which were misclassified due to a shortcoming of applying Algorithm 2 on failures. This algorithm fails in these cases because it does not make an exception for minor changes involving a few pixels, the degree of colour change, nor does it consider other elements that may be involved. For this particular case, a container with other elements that overlap by design with the reported protruding element requires a more sophisticated approach to classify them. I plan, as part of future work, to investigate if additional heuristics would improve, or negatively influence, the overall results.

The sixth and final disagreement comes from the EatThisMuch web page which pertains to an out of flow HTML element that was falsely reported as protruding. At the lower bottom right-hand corner of the viewport, the out of flow element is always available for the user at this position, thus it is *floating* around the page. As the viewport size changes, it continues to float with different elements coincidentally falling into position be-

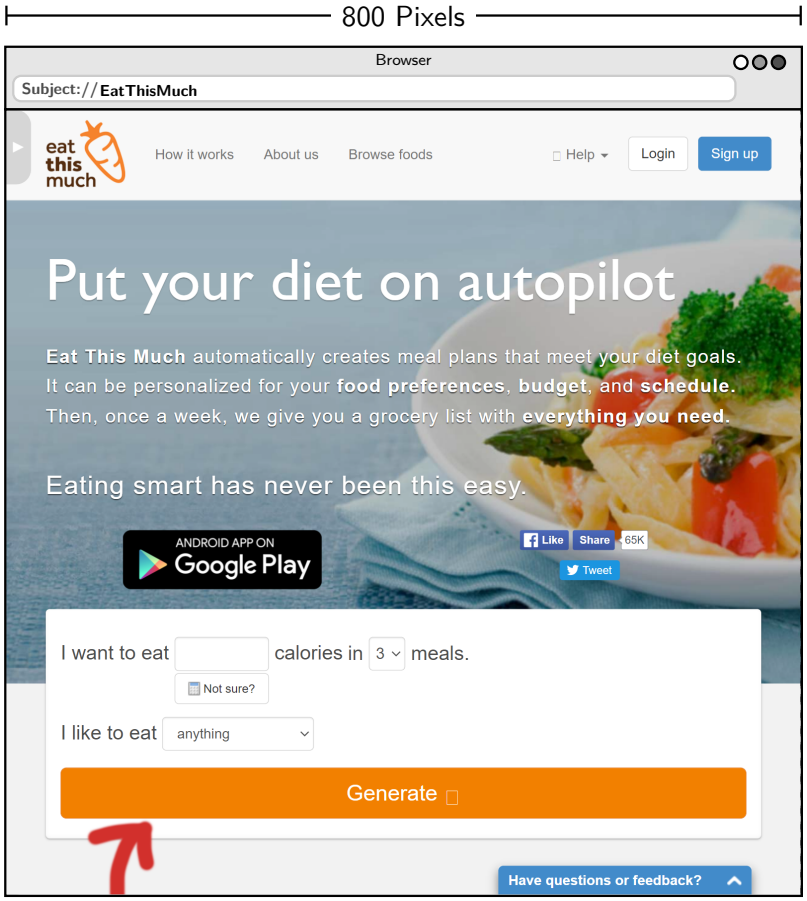


Figure 4.10: A snapshot from the EatThisMuch subject which contains an out of flow element at the bottom right-hand corner of the viewport. This element remains positioned in the same place, relative to the viewport size, to make it easier for a visitor of the page to provide feedback or ask a question regardless of the scrolling position, essentially floating around the page.

hind it. Since there is no logic implemented in ReDeCheck to prevent floating elements from being associated with non-floating containers, it may falsely assume a non-floating container to be the parent of a floating element. Although ReDeCheck should not have reported this as a failure, Verve's visual approach cannot correctly classify it as a false positive either. Furthermore, during my manual classification, I also misclassified it as a non-observable issue.

For the results from the maximum viewport, seen in Table 4.8, there were an additional three viewport failures in disagreement with the manual classification. Verve classified these failures from SB-Admin-2, EatThisMuch, and SB-Grayscale as non-observable issues while I manual classification them as true positives. These failures are more visible at the minimum viewport. This affirms the findings from RQ2 of the previous chapter, that Verve is more likely to agree at the minimum viewport where the failure is most severe. Figure 4.1 shows snapshots of the viewport failure from SB-Admin-2, illustrating how the failure is observable (TP) at the narrower inspection points and becomes non-observable (NOI) at a wider inspection point.

Compared to the results of Verve using the initial set of subjects, the new results from the additional set of subjects showed an improvement in agreement from 86.3% to

Table 4.8: The results from using Verve for element collision, element protrusion, and viewport protrusion failures of the additional set of web pages after inspecting the "Maximum" of the reported failure range.

	Maximum									
	Element Collision			**Element Protrusion**			**Viewport Protrusion**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
EatThisMuch	-	5/5	-	1/-	5/6	-	1/1	1/1	-	13
Forvo	-	-	-	-	3/3	-	-	-	-	3
GMapStreetViewPlayer	-	-	-	-	2/2	-	-	-	-	2
HoursOf	-	-	-	-	1/1	-	1/-	-/1	-	2
RetailMeNot	2/2	-	-	-	30/30	-	-	-	-	32
SB-Admin-2	-	-	-	-	-	-	-/1	1/-	-	1
SB-Agency	-	4/4	-	3/-	5/8	-	1/1	-	-	13
SB-Business-Casual	-	-	-	-	-	-	-	-	-	-
SB-Clean-Blog	-	-	-	-	-	-	-	-	-	-
SB-Coming-Soon	-	-	-	-	-	-	-	-	-	-
SB-Creative	-	-	-	-	-	-	-	-	-	-
SB-Freelancer	-	-	-	-	-	-	-	-	-	-
SB-Grayscale	-	-	-	-	-	-	-/1	1/-	-	1
SB-Landing-Page	-	-	-	-	-	-	-	-	-	-
SB-New-Age	-	-	-	-	-	-	-	-	-	-
SB-One-Page-Wonder	-	-	-	-	-	-	-	-	-	-
SB-Resume	-	1/1	-	-	-	-	-	-	-	1
SB-Stylish-Portfolio	-	-	-	-	-	-	-	-	-	-
SimilarSites	-	-	-	-	-	-	4/4	-	-	4
Tiiime	-	-	-	-	1/1	-	-	-	-	1
Total	2	10	-	4	47	-	7	3	-	73
Agreement with manual	2/2	10/10	-	-	47/51	-	5/8	0/2	-	-
Agreement per failure type	100 %			92.2 %			50 %			-
Per inspection point	87.7 %									-

91.8%. It is worthy to note that there were only 73 reports from the additional set for element collision, element protrusion, and viewport protrusion failures while there were 117 reported from the initial set. Nevertheless, a trend was observed in both sets for the agreement level across the three failure types and all three viewports. The trend showed that Verve tends to agree most over element collision failures, then element protrusion, followed by viewport protrusion failures with the least agreement. Given that the distribution of reported failure varied in both sets and that the element collision failure achieved 100% agreement in the additional set, this suggests that future improvements should focus on the automated classification of element protrusion and viewport protrusion failures.

Conclusion for RQ3 (a) – For the element collision, element protrusion, and viewport protrusion failures reported by ReDeCheck on the 20 pages in the additional set of subjects, Verve's automatic classification frequently matched the manual classification. Notably, it achieved a 91.8% agreement with my manual classifications, an increase from the 86.3% agreement using the initial set of subjects with independently classified failures.

Answer to RQ3 (b) – For wrapping failures, Table 4.9 tallies the classifications outputted by Verve when the 20 subjects from the additional set are provided as input to the tool. The table further compares them with the manual classifications for agreement resulting in a 64.7% agreement between automated and manual classifications. Moreover,

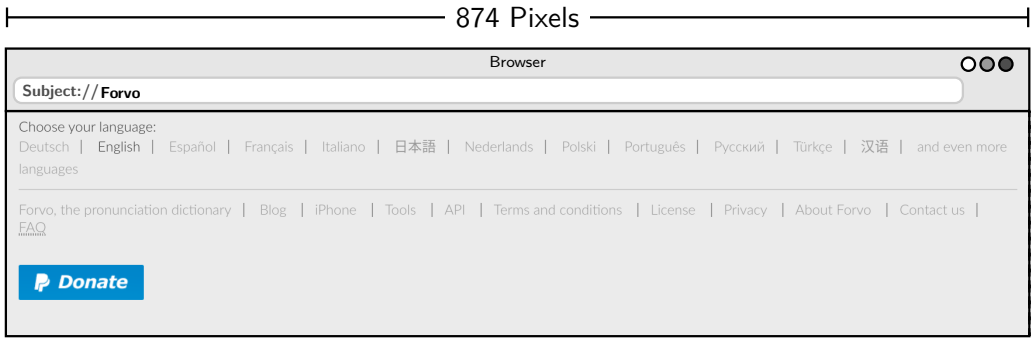
all 17 wrapping failures reported from this set were consistently classified by Verve and agree with the manual analysis at all three of the viewports used in the experiment (i.e., minimum, middle, and maximum). This result supports the intuition, and findings of RQ1, that a single viewport is sufficient for classifying a wrapping failure.

Table 4.9: Results when using Verve to classify wrapping failures for the additional set of web pages.

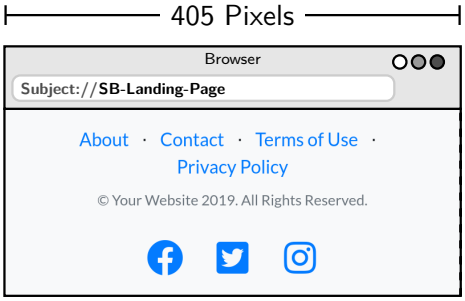
	Wrapping									
	Minimum			**Middle**			**Maximum**			
	TP	NOI	FP	TP	NOI	FP	TP	NOI	FP	**Total**
EatThisMuch	-	-	1/1	-	-	1/1	-	-	1/1	1
Forvo	4/2	-	-/2	4/2	-	-/2	4/2	-	-/2	4
GMapStreetViewPlayer	-	-	-	-	-	-	-	-	-	-
HoursOf	-	-	-	-	-	-	-	-	-	-
RetailMeNot	-	2/2	4/4	-	2/2	4/4	-	2/2	4/4	6
SB-Admin-2	-	-	-	-	-	-	-	-	-	-
SB-Agency	3/-	-	-/3	3/-	-	-/3	3/-	-	-/3	3
SB-Business-Casual	-	-	-	-	-	-	-	-	-	-
SB-Clean-Blog	-	-	-	-	-	-	-	-	-	-
SB-Coming-Soon	-	-	-	-	-	-	-	-	-	-
SB-Creative	-	-	-	-	-	-	-	-	-	-
SB-Freelancer	-	-	-	-	-	-	-	-	-	-
SB-Grayscale	-	-	-	-	-	-	-	-	-	-
SB-Landing-Page	1/-	-	-/1	1/-	-	-/1	1/-	-	-/1	1
SB-New-Age	-	-	-	-	-	-	-	-	-	-
SB-One-Page-Wonder	-	-	-	-	-	-	-	-	-	-
SB-Resume	2/2	-	-	2/2	-	-	2/2	-	-	2
SB-Stylish-Portfolio	-	-	-	-	-	-	-	-	-	-
SimilarSites	-	-	-	-	-	-	-	-	-	-
Tiiime	-	-	-	-	-	-	-	-	-	-
Total failures	10	2	5	10	2	5	10	2	5	17
Agreement with manual	4/4	2/2	5/11	4/4	2/2	5/11	4/4	2/2	5/11	-
Per inspection point	64.7 %			64.7 %			64.7 %			-

There were a total of 6 out of the 17 reported wrapping failures where Verve did not agree with my manual classification. Although I manually classified these failures as false positives, Verve concluded that they were true positive reports. Of the six, two are from the Forvo subject, three are reported from SB-Agency, and one from the SB-Landing-Page. As noticed in RQ1, although the ReDeCheck tool correctly detected an element that does wrap around to a new row when inspected later visually, some of these cases should be exempt from being reported as a failure. In other words, some elements are expected and allowed to wrap. Since Verve does not feature any artificial intelligence that allows it to distinguish these cases, it results in a misclassification. Nevertheless, many can be prevented during the detection phase of ReDeCheck. For example, the detection can introduce a condition to check that at some smaller viewports that the elements are fully rearranged to stack vertically. Otherwise, it may be an acceptable behaviour that was omitted from the design intentionally. Alternatively or in addition, it can also avoid reporting wrappings that contain only text-based links.

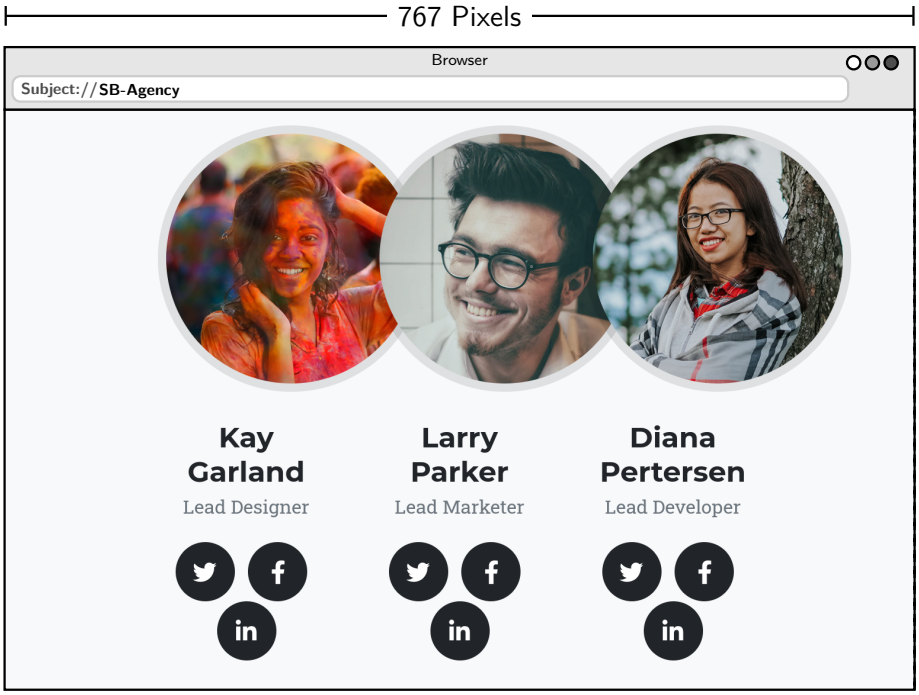
As seen with a few disagreements arising from the initial set of subjects, again in the additional set, three failures had textual links that do, in fact, wrap but should not be considered failures. Two of these failures were from the Forvo web page and one was from the SB-Landing-Page. For reference, Figure 4.11 part (a) shows a screenshot of the first failure reported from the Forvo subject and part (b) of the figure showcases the failure from the SB-Landing-Page. In part (a), an element containing the link labelled “FAQ”



(a) Captured using the maximum viewport of the failure range, the snapshot shows the footer of the Forvo web page that has multiple links forming a row except for the wrapped “FAQ" link.



(b) Captured using the maximum viewport, the “Privacy Policy” is reported as a wrapping failure.



(c) Captured using the maximum viewport and the SB-Agency subject where there were three separate wrapping failures reported by ReDeCheck. In each case, the circular Instagram social media icon is reported to have wrapped below the other two icons.

Figure 4.11: The figure presents three subject web pages with element wrapping failure reports that were misclassified by Viser as true positives. Snapshots capturing the failures from Forvo, SB-Landing-Page, and SB-Agency are respectively showcased in parts (a), (b), and (c).

is reported to have wrapped. The second failure from Forvo reports the same element wrapping at a smaller viewport but now forming two rows above it instead of one. In the case of the SB-Landing-Page failure, part (b), the link labelled "Privacy Policy" has also wrapped into a new row. Evidently, these three element wrapping reports do not warrant a repair nor the attention of the developer.

Finally, there were three failures from SB-Agency in disagreement with VERVE. All three failures reported a wrapping of a social media icon from a set of three circular icons. For reference, Figure 4.11 (c) shows these three Instagram icons simultaneously wrapping three different rows as reported by REDECHECK. Even though the awkward wrapping of icons would normally be a layout failure, this one includes three icons that wrap in an aesthetically pleasing fashion which leaves the layout largely unchanged. Although the developer may want to stop this from occurring, I am more inclined to think that this is an intentional design feature. This leads me to conclude that these reports fall under subjective disagreements.

Conclusion for RQ3 (b) – For wrapping failures, the agreement between the manual classification and the one reported by VERVE dropped from 78.6% to 64.7% when moving from the initial to additional subjects. Yet for both sets of subjects, the effectiveness of VERVE is expected to improve if the detection algorithms of REDECHECK are updated with new conditions that allow some row aligned elements to wrap without penalty.

Answer to RQ3 (c) – REDECHECK reported a total of 53 small-range failures from the additional set of web pages that were all false positives. Suggesting how rare small-range failures occur in comparison to the other failure types or that REDECHECK's detection algorithm may need improvements. Regardless of the reason, to properly evaluate VERVE on true positive reports, I had to manually inject small-range failures and run REDECHECK on this *fault-injected* additional set of pages. For details about the fault injection procedure that I followed see Section 4.4.1.

Using the fault-injected set, REDECHECK was able to detect 96 more small-range failures. I manually classified all of these failures as true positives except for a single failure that did not pertain to the target element of the injection, which is a false positive. This failure from HoursOf had an element positioned in the middle of its container at both viewports nearest to the reported failure range. Meanwhile, at the viewports reported having the failure, the element was very close to the middle but not identified as such. Hence, no visual disturbance was observed in the layout. Next, I will begin by investigating the results of the unmodified additional set of subjects containing only false positives with no synthetic failures.

Table 4.10 gives the results from using VERVE on the small-range failures reported for the additional set of web pages using horizontal referencing. This table shows that *Correlation* obtained the highest level of agreement with a 94.3% match, which is different from the findings of RQ2 where I concluded that *Intersection* was the best performer for the initial subjects. Yet, for the additional set of subjects, *Intersection* came in second place with 67.9% agreement by misclassifying 17 of the 53 failures as true positives. Noteworthy, all of these 17 were also misclassified using *Bhattacharyya Distance*, *Chi-Square*, and *Alternative Chi-Square*. Two of these 17 were also misclassified by *Correlation*.

A closer examination of these 17 disagreements revealed that 12 were reporting a single textual link that changed its own position in reference to other textual links and its position within the element containing it. These links list the available languages

Table 4.10: The results from using VERVE to classify small-range failures detected in the additional set of web pages using the horizontal referencing approach and featuring five histogram comparison measures. In this table, ϵ denotes the threshold value used for histogram comparison.

	Small-range horizontal referencing										
	Bhattacharyya Distance $\epsilon = 0.20$		**Chi-Square** $\epsilon = 0.11$		**Alternative Chi-Square** $\epsilon = 0.14$		**Correlation** $\epsilon = 0$		**Intersection** $\epsilon = 0.09$		
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	**Total**
EatThisMuch	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
Forvo	16/-	13/29	25/-	4/29	26/-	3/29	-	29/29	13/-	16/29	29
GMapStreetViewPlayer	-	-	-	-	-	-	-	-	-	-	-
HoursOf	-	-	-	-	-	-	-	-	-	-	-
RetailMeNot	5/-	10/15	6/-	9/15	9/-	6/15	3/-	12/15	4/-	11/15	15
SB-Admin-2	1/-	-/1	1/-	-/1	1/-	-/1	-	1/1	-	1/1	1
SB-Agency	-	-	-	-	-	-	-	-	-	-	-
SB-Business-Casual	-	-	-	-	-	-	-	-	-	-	-
SB-Clean-Blog	-	-	-	-	-	-	-	-	-	-	-
SB-Coming-Soon	-	-	-	-	-	-	-	-	-	-	-
SB-Creative	-	-	-	-	-	-	-	-	-	-	-
SB-Freelancer	-	-	-	-	-	-	-	-	-	-	-
SB-Grayscale	-	-	-	-	-	-	-	-	-	-	-
SB-Landing-Page	-	-	-	-	-	-	-	-	-	-	-
SB-New-Age	-	-	-	-	-	-	-	-	-	-	-
SB-One-Page-Wonder	-	-	-	-	-	-	-	-	-	-	-
SB-Resume	-	-	-	-	-	-	-	-	-	-	-
SB-Stylish-Portfolio	-	4/4	2/-	2/4	3/-	1/4	-	4/4	-	4/4	4
SimilarSites	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Tiiime	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Total	22	31	34	19	39	14	3	50	17	36	53
Agreement with manual	-	31/53	-	19/53	-	14/53	-	50/53	-	36/53	-
Agreement per measure	58.5 %		35.8 %		26.4 %		94.3 %		67.9 %		-

of the Forvo subject as seen in Figure 4.11 (a). Essentially, this change in position is a wrapping of the textual link that only happens within four viewport widths, a small range. This suggests or affirms that the small-range failure detection algorithm implemented in REDECHECK can be oversensitive to minor changes resulting in the reporting of many failures that are really about a single element.

Still using the unmodified additional set of web pages, Table 4.11 gives the results of VERVE while configured to use the horizontal-plus-vertical referencing approach. Consistent with the findings of RQ2, *Intersection* ranked the highest for this set of web pages achieving a 73.6% agreement with the manual classifications. Therefore, misclassifying 14 out of 53 reported small-range failures. Similarly, the other five measures were in disagreement over the same 14 failures as the top performer. In the second rank comes *Alternative Chi-Square* and *Correlation* both achieving 71.7% agreement after misclassifying one failure more than *Intersection*. The next measure, *Chi-Square*, had a 66% agreement due to a total of 18 failures that were misclassified. Finally, the lowest-ranked measure, *Bhattacharyya Distance*, misclassified six more failures when compared to *Intersection* achieving only a 60.4% agreement with the manual classifications.

Before getting into the results of the fault-injected set of pages, Figure 4.12 showcases snapshots of the SB-Business-Casual subject featuring one of the synthetic small-range failures. This failure occurs at the viewport width range of 992–995 pixels, where I originally injected the fault. Parts (a) and (c) of the figure show the web page rendered at the 991 and 996 pixel viewport widths, respectively. These are the comparison viewports

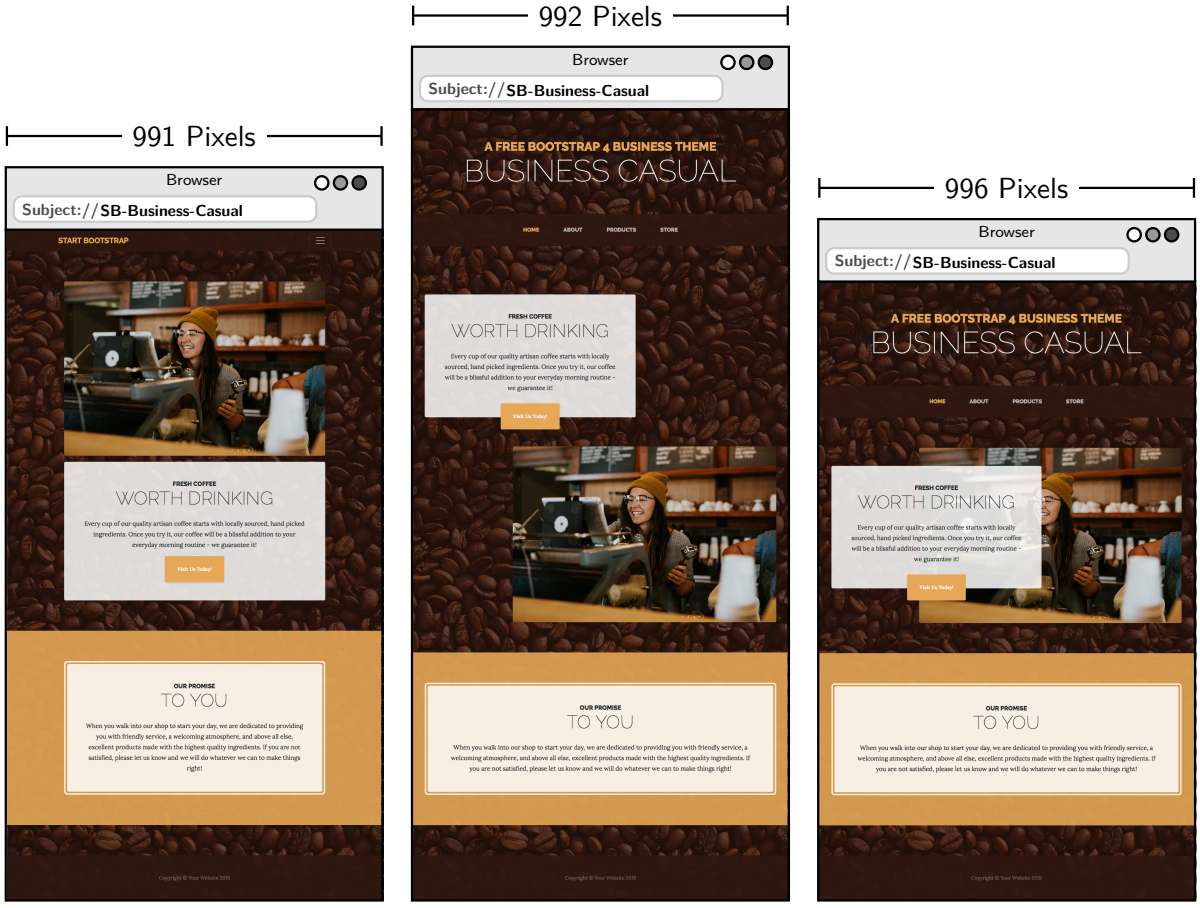
Table 4.11: The results from using Verve to classify small-range failures detected in the additional set of web pages using the horizontal-plus-vertical referencing approach and featuring five histogram comparison measures. In this table, ϵ denotes the threshold value used for histogram comparison.

	Small-range horizontal-plus-vertical referencing										
	Bhattacharyya Distance $\epsilon = 0.23$		**Chi-Square** $\epsilon = 0.46$		**Alternative Chi-Square** $\epsilon = 0.82$		**Correlation** $\epsilon = 0$		**Intersection** $\epsilon = 0.15$		
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	**Total**
EatThisMuch	-	2/2	-	2/2	-	2/2	-	2/2	-	2/2	2
Forvo	16/-	13/29	12/-	17/29	12/-	17/29	12/-	17/29	12/-	17/29	29
GMapStreetViewPlayer	-	-	-	-	-	-	-	-	-	-	-
HoursOf	-	-	-	-	-	-	-	-	-	-	-
RetailMeNot	4/-	11/15	2/-	13/15	2/-	13/15	3/-	12/15	2/-	13/15	15
SB-Admin-2	1/-	-/1	-	1/1	-	1/1	-	1/1	-	1/1	1
SB-Agency	-	-	-	-	-	-	-	-	-	-	-
SB-Business-Casual	-	-	-	-	-	-	-	-	-	-	-
SB-Clean-Blog	-	-	-	-	-	-	-	-	-	-	-
SB-Coming-Soon	-	-	-	-	-	-	-	-	-	-	-
SB-Creative	-	-	-	-	-	-	-	-	-	-	-
SB-Freelancer	-	-	-	-	-	-	-	-	-	-	-
SB-Grayscale	-	-	-	-	-	-	-	-	-	-	-
SB-Landing-Page	-	-	-	-	-	-	-	-	-	-	-
SB-New-Age	-	-	-	-	-	-	-	-	-	-	-
SB-One-Page-Wonder	-	-	-	-	-	-	-	-	-	-	-
SB-Resume	-	-	-	-	-	-	-	-	-	-	-
SB-Stylish-Portfolio	-	4/4	4/-	-/4	1/-	3/4	-	4/4	-	4/4	4
SimilarSites	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Tiiime	-	1/1	-	1/1	-	1/1	-	1/1	-	1/1	1
Total	21	32	18	35	15	38	15	38	14	39	53
Agreement with manual	-	32/53	-	35/53	-	38/53	-	38/53	-	39/53	-
Agreement per measure	60.4 %		66 %		71.7 %		71.7 %		73.6 %		-

bordering the failure range. At the narrower bordering viewport of 991 pixels wide, shown in part (a), an image element of a barista in a coffee shop is positioned above a block of text with a white background. At the wider viewport of 996 pixels, these two elements are overlapping by design. Part (b) of the figure shows the failure at the viewport width of 992 pixels, with these two elements positioned abnormally. Additional, unrelated changes to the layout start at the 992 pixels viewport width and remains for wider viewports. These include a larger logo in the header of the page and an expanded menu that are part of the design for larger viewports. Verve was able to correctly classify this failure as a true positive report using all five distance metrics for both the horizontal referencing and the horizontal-plus-vertical referencing approach.

Table 4.12 presents the results from running Verve on the set of synthetic faults using the horizontal referencing approach. From the result, my analysis revealed that 27 failures were misclassified by Verve as false positives using all five histogram measures. The top-performing measure, *Chi-Square*, with 67.7% agreement had an additional four misclassifications. A close second was *Alternative Chi-Square*, which achieved 64.6% agreement by misclassifying three more than *Chi-Square*. In third place, with only a 43.8% agreement are *Bhattacharyya Distance* and *Correlation*. Finally, the poorest performing measure was *Intersection*, at 29.2% agreement. This result contradicts the *Intersection* measure's first-place performance with the initial set, and its position as the second-ranked in the non-fault-injected additional set of web pages.

Still using the fault-injected set of web pages, Table 4.13 presents the results of using



(a) Narrower comparison. (b) Failure viewport. (c) Wider comparison.

Figure 4.12: Three snapshots of the SB-Business-Casual web page that capture its layout before a small-range failure occurs, in (a), and a synthetic small-range failure in the range of 992–995 pixels in (b), and after the layout failure in (c), as reported by the ReDeCheck and correctly classified, without human intervention, as a true positive by the Verve tool using all five metrics and both referencing approaches.

Verve to classify the failures from this set while configured to use the horizontal-plus-vertical referencing approach. The results show that all five of the measures were in disagreement with the manual classification for 12 out of the 96 failures reported. The top performer for this set was *Chi-Square* with a 79.2% agreement and a total of 20 failures in non-agreement. In the second place, *Correlation* achieved a 68.8% agreement with 30 misclassifications. It is followed in rank by *Bhattacharyya Distance* with 65.6%, *Intersection* with 55.2%, and *Alternative Chi-Square* with 45.8% agreement. The most interesting result is that *Intersection* comes in fourth place even though it is the top performer for the subjects in the initial set and unmodified additional set.

The results from the initial set, additional set, and the fault-injected set of subjects do not clearly answer, in isolation, what is the best histogram distance metric to use nor the better referencing strategy. More importantly, the *prospective* thresholds that were tuned using the initial set of subjects may not be optimal for use in the later sets of subjects thus reducing the possibility of making a strong recommendation. This deliberate omission was made to answer if the thresholds values can extend to different sets of subjects using the same referencing approach but newer subjects. To conclusively answer these questions,

Table 4.12: The results from using Verve to classify small-range failures detected in the fault-injected additional set of web pages using the horizontal referencing approach and featuring five histogram comparison measures. In this table, ϵ denotes the threshold value used for histogram comparison.

	Small-range horizontal referencing										
	Bhattacharyya Distance $\epsilon = 0.20$		**Chi-Square** $\epsilon = 0.11$		**Alternative Chi-Square** $\epsilon = 0.14$		**Correlation** $\epsilon = 0$		**Intersection** $\epsilon = 0.09$		
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	**Total**
EatThisMuch	3/3	-	3/3	-	3/3	-	3/3	-	3/3	-	3
Forvo	-/9	9/-	5/9	4/-	5/9	4/-	-/9	9/-	-/9	9/-	9
GMapStreetViewPlayer	4/4	-	4/4	-	4/4	-	2/4	2/-	4/4	-	4
HoursOf	-/2	3/1	-/2	3/1	-/2	3/1	-/2	3/1	-/2	3/1	3
RetailMeNot	-/8	8/-	-/8	8/-	-/8	8/-	-/8	8/-	-/8	8/-	8
SB-Admin-2	12/12	-	12/12	-	12/12	-	12/12	-	12/12	-	12
SB-Agency	5/5	-	5/5	-	5/5	-	5/5	-	-/5	5/-	5
SB-Business-Casual	1/1	-	1/1	-	1/1	-	1/1	-	1/1	-	1
SB-Clean-Blog	-/5	5/-	3/5	2/-	3/5	2/-	3/5	2/-	-/5	5/-	5
SB-Coming-Soon	3/3	-	3/3	-	3/3	-	3/3	-	3/3	-	3
SB-Creative	-/6	6/-	-/6	6/-	-/6	6/-	-/6	6/-	-/6	6/-	6
SB-Freelancer	7/7	-	7/7	-	7/7	-	7/7	-	-/7	7/-	7
SB-Grayscale	-/5	5/-	-/5	5/-	-/5	5/-	-/5	5/-	-/5	5/-	5
SB-Landing-Page	2/4	2/-	2/4	2/-	2/4	2/-	-/4	4/-	-/4	4/-	4
SB-New-Age	2/2	-	2/2	-	2/2	-	-/2	2/-	2/2	-	2
SB-One-Page-Wonder	-/3	3/-	3/3	-	3/3	-	-/3	3/-	-/3	3/-	3
SB-Resume	1/4	3/-	4/4	-	4/4	-	4/4	-	1/4	3/-	4
SB-Stylish-Portfolio	-/5	5/-	3/5	2/-	-/5	5/-	-/5	5/-	-/5	5/-	5
SimilarSites	-/6	6/-	6/6	-	6/6	-	-/6	6/-	-/6	6/-	6
Tiiime	1/-	-/1	1/-	-/1	1/-	-/1	1/-	-/1	1/-	-/1	1
Total	41	55	64	32	61	35	41	55	27	69	96
Agreement with manual	41/95	1/1	64/95	1/1	61/95	1/1	41/95	1/1	27/95	1/1	-
Agreement per failure type	43.8 %		67.7 %		64.6 %		43.8 %		29.2 %		-

I have taken two steps. First, I aggregated the results from the three sets of subjects to investigate the overall results. Second, I tuned the *retrospective* thresholds using all failures reported from the aggregated set, using the ThresholdFinder tool, in order to assess the performance of the *prospective* thresholds and make a final recommendation.

Table 4.14 presents the agreement for all failures reported from the 45 pages used in the study including the synthetic ones. Moreover, the table re-presents the agreements of the three sets, in isolation, for comparison. The table also provides both the *prospective* thresholds used in the experiments and the *retrospective* thresholds used to assess for possible improvements in agreement. The overall top-performing measure for all web pages using the *prospective* threshold was *Chi-Square* for both the horizontal referencing approach and the horizontal-plus-vertical referencing approach. While the horizontal referencing approach achieved 79.1% agreement, the better performing approach, horizontal-plus-vertical referencing, achieved 87.5% agreement. With exceptionalism, using the *retrospective* thresholds and horizontal referencing, *Chi-Square* was the top performer once again with an agreement of 80.8%. As for the better performing approach, horizontal-plus-vertical referencing, *Chi-Square* was an almost top performer with 87.5% agreement only second to the *Intersection* method which achieved an 87.8% agreement.

The results definitively showed that horizontal-plus-vertical referencing was the superior approach used for automated classification. This is a logical outcome since it uses more images than the horizontal referencing approach, to reference a change in the position of an element. The results further definitively showed that the lower values of the *retrospective*

Table 4.13: The results from using Verve to classify small-range failures detected in the fault-injected additional set of web pages using the horizontal-plus-vertical referencing approach and featuring five histogram comparison measures. In this table, ϵ denotes the threshold value used for histogram comparison.

	Small-range horizontal-plus-vertical referencing										
	Bhattacharyya Distance $\epsilon = 0.23$		**Chi-Square** $\epsilon = 0.46$		**Alternative Chi-Square** $\epsilon = 0.82$		**Correlation** $\epsilon = 0$		**Intersection** $\epsilon = 0.15$		
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	**Total**
EatThisMuch	3/3	-	3/3	-	-/3	3/-	3/3	-	3/3	-	3
Forvo	5/9	4/-	7/9	2/-	-/9	9/-	9/9	-	9/9	-	9
GMapStreetViewPlayer	2/4	2/-	-/4	4/-	-/4	4/-	2/4	2/-	2/4	2/-	4
HoursOf	-/2	3/1	-/2	3/1	-/2	3/1	-/2	3/1	-/2	3/1	3
RetailMeNot	6/8	2/-	6/8	2/-	-/8	8/-	-/8	8/-	-/8	8/-	8
SB-Admin-2	12/12	-	12/12	-	12/12	-	12/12	-	12/12	-	12
SB-Agency	5/5	-	5/5	-	5/5	-	5/5	-	5/5	-	5
SB-Business-Casual	1/1	-	1/1	-	1/1	-	1/1	-	1/1	-	1
SB-Clean-Blog	3/5	2/-	4/5	1/-	-/5	5/-	4/5	1/-	-/5	5/-	5
SB-Coming-Soon	3/3	-	3/3	-	3/3	-	3/3	-	3/3	-	3
SB-Creative	4/6	2/-	4/6	2/-	4/6	2/-	6/6	-	4/6	2/-	6
SB-Freelancer	7/7	-	7/7	-	7/7	-	7/7	-	5/7	2/-	7
SB-Grayscale	-/5	5/-	-/5	5/-	-/5	5/-	-/5	5/-	-/5	5/-	5
SB-Landing-Page	2/4	2/-	2/4	2/-	2/4	2/-	2/4	2/-	2/4	2/-	4
SB-New-Age	2/2	-	2/2	-	2/2	-	1/2	1/-	2/2	-	2
SB-One-Page-Wonder	2/3	1/-	3/3	-	3/3	-	-/3	3/-	-/3	3/-	3
SB-Resume	1/4	3/-	4/4	-	-/4	4/-	4/4	-	-/4	4/-	4
SB-Stylish-Portfolio	3/5	2/-	5/5	-	3/5	2/-	5/5	-	3/5	2/-	5
SimilarSites	-/6	6/-	6/6	-	-/6	6/-	-/6	6/-	-/6	6/-	6
Tiiime	1/-	-/1	1/-	-/1	1/-	-/1	1/-	-/1	1/-	-/1	1
Total	62	34	75	21	43	53	65	31	52	44	96
Agreement with manual	62/95	1/1	75/95	1/1	43/95	1/1	65/95	1/1	52/95	1/1	-
Agreement per failure type	65.6 %		79.2 %		45.8 %		68.8 %		55.2 %		-

thresholds provided better classifications using all metrics except for *Correlation* which was already at the minimal possible value. Another exception is *Chi-Square*, where the thresholds showed less sensitivity to a change in subjects and thus did extend well with little to no change in the threshold values. With its excellent performance, whether using the *prospective* or *retrospective* thresholds, the *Chi-Square* is the best histogram distance metric to apply using Verve.

Conclusion for RQ3 (c) – For small-range failures, the *Chi-Square* method showed superior performance regardless of referencing approach being used. More importantly, the established thresholds for *Chi-Square* were ideal to use and generalized well showing minimal to no possible improvements. Finally, the horizontal-plus-vertical referencing approach is the better classifier using either threshold and regardless of the histogram measure applied in the experiments. Therefore, Verve should be configured, by default, to use the horizontal-plus-vertical referencing approach in combination with the *Chi-Square* measure.

Answer to RQ4 – Verve took 4.08 seconds, on average, to automatically classify each responsive failure with a median value of 3.57 seconds. Figure 4.13 (a) showcases a box plot for the runtime for each failure type with the added 200ms delay. Verve included this delay to allow the web page to complete any transitional visual effects and for all elements to settle into their final positions in the layout. The delay is repeated any time

Table 4.14: The results from using Verve's two alternative approaches to small-range classification, the horizontal referencing approach and the horizontal-plus-vertical referencing approach. This table re-presents the results from applying the *prospective* thresholds used in the study over the three sets: the initial set, the additional set, and the fault-injected additional set. The table also newly presents the combined results of all sets of subjects; it also presents the *retrospective* thresholds and the results from applying them.

				Resulting Classification Agreement				
		Threshold Values		Initial Set	Additional Set	Fault-injected Set	All Sets Combined	
Approach	**Measure**	*prospective*	*retrospective*	*prospective*	*prospective*	*prospective*	*prospective*	*retrospective*
Horizontal-plus-vertical	Intersection	0.15	0.08	98.5 %	73.6 %	55.2 %	82.6 %	87.8 %
Horizontal-plus-vertical	Chi-Square	0.46	0.46	97.4 %	66.0 %	79.2 %	87.5 %	87.5 %
Horizontal-plus-vertical	Correlation	0.00	0.00	98.0 %	71.7 %	68.8 %	85.8 %	85.8 %
Horizontal-plus-vertical	Bhattacharyya Distance	0.23	0.19	98.0 %	62.3 %	65.6 %	83.4 %	85.5 %
Horizontal-plus-vertical	Alternative Chi-Square	0.82	0.30	96.9 %	71.7 %	45.8 %	78.8 %	84.6 %
Horizontal	Chi-Square	0.11	0.05	96.4 %	35.9 %	67.7 %	79.1 %	80.8 %
Horizontal	Alternative Chi-Square	0.14	0.05	93.3 %	26.4 %	64.6 %	75.0 %	78.8 %
Horizontal	Bhattacharyya Distance	0.20	0.05	91.8 %	58.5 %	43.8 %	73.3 %	78.5 %
Horizontal	Intersection	0.09	0.02	97.4 %	67.9 %	29.2 %	73.8 %	76.5 %
Horizontal	Correlation	0.00	0.00	87.2 %	94.3 %	43.8 %	76.2 %	76.2 %

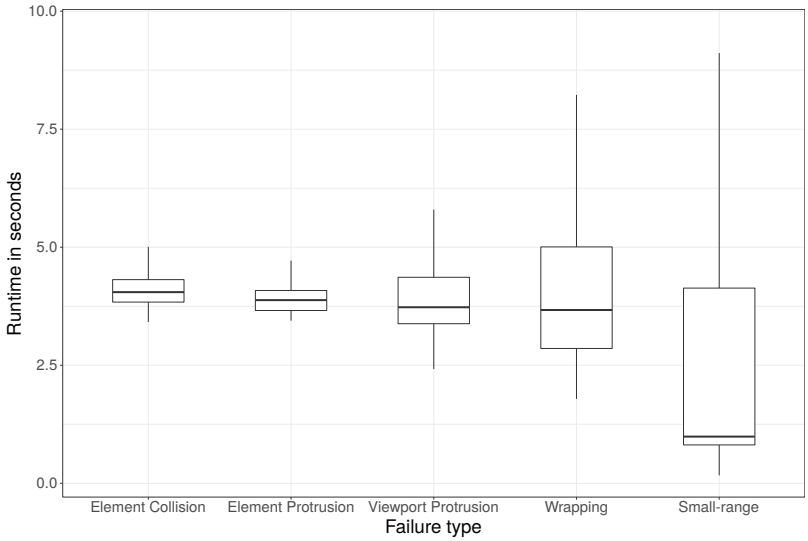
the page is loaded or when the viewport, opacity, or scrolling position is changed. For a more detailed discussion of the reasons behind adding this delay, see the methodology for this research question in Section 4.4.1.

The runtime for the element collision, element protrusion, viewport protrusion, and wrapping failures had similar medians/means of 4.05/5.25, 3.88/3.98, 3.73/4.01, and 3.67/4.21 seconds, respectively, while small-range failures had a median value of 0.99 and a mean value of 3.92 seconds. As shown in Section 4.3.2, the algorithm for classifying small-range failures is fundamentally different from the other four. Since it requires snapshots of multiple AOCs across different viewports it will need additional runtime to complete these steps. It also has additional processing time for of answering the research questions associated with small-range failures. This time includes identifying the better referencing approach and the best histogram metric. Thus, the runtime for a real use case is expected to be better once Verve is configured to use only the recommended, from RQ3, horizontal-plus-vertical referencing approach and *Chi-Square*.

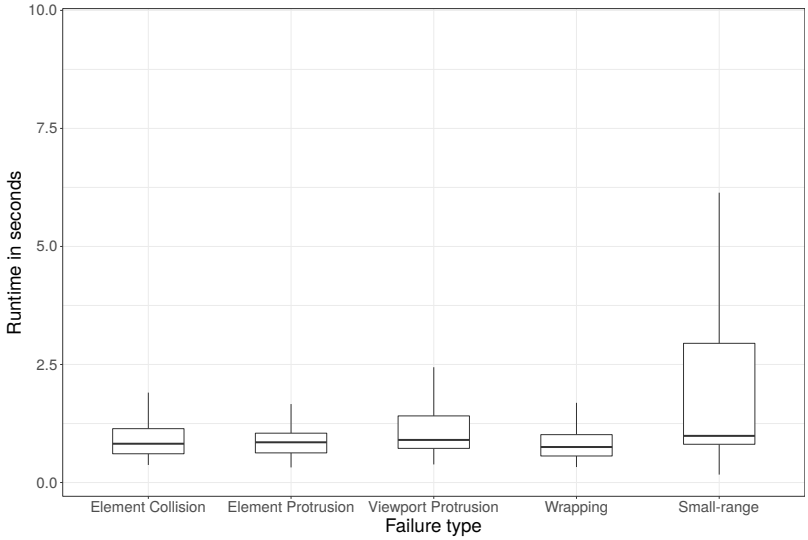
From the failure types that use the `opacity` strategy to classify failures, namely element collision, element protrusion, viewport protrusion, and element wrapping failures, the most notable box plot is the wrapping box plot which is comparability taller. One main distinction of a wrapping failure report is that at least three elements are reported by ReDeCheck while all others always report two. Since I was not objectively optimizing Verve for runtime, the effect of the delay used to scroll to each element reported is visible on the box plot. In the future, I plan to alleviate this oversight by making improvements to the implementation of Verve.

To better understand the execution times, it is important to note that I designed Verve to reuse snapshots where possible. The main reason for this is that many failures share the same viewports required for analysis. Therefore, avoiding the recapture of the same viewport can reduce the number of scrolling delays required. To bring this into perspective, the AccountKiller web page from the initial set of subjects had 147 small-range failure reports in the range of 476–480 pixels wide. Even though this causes a longer execution time for the first failure to request the snapshots from this range, the time to retrieve a failure-specific AOC from the saved snapshots and the time to calculate all five histogram metrics are within the processing time of the relevant failure.

Although the execution time of Verve is normally stable, some subjects had execu-



(a) VERVE's runtime using a 200 milliseconds delay for opacity, viewport, and scroll changes.



(b) VERVE's runtime using no added delay.

Figure 4.13: VERVE's execution time in seconds across all of the 469 presentation failures and 30 trials using the horizontal-plus-vertical referencing approach. In these box plots, the bottom and top whiskers show the minimum and maximum data values excluding outliers, while the box itself represents the inter-quartile range and the bold middle line represents the median value.

tion times that are far outliers from the median value. For instance, when VERVE was configured to use the added delay setting, there was one trial that ran for 795.4 seconds when classifying an element collision failure reported from the TopDocumentary subject. Importantly, this was the only outlier from the 30 trials, as evident by the fact that the execution time for the other 29 trials was between 4.01 and 4.58 seconds. To ensure that the performance trends in the graphs are not skewed by these rare outliers, Figure 4.13 presents box plots that omit the values that are far outliers. To exclude the outliers from the plots without removing any other data points, I used the default option of the *ggplot2* package in the R language for statistical computation.

To further investigate the runtime of VERVE without the influence of the important but optional time delay, I temporarily disabled it and re-ran the timing experiments. Using this no-delay configuration, VERVE took, on average, 2.24 seconds to classify a reported failure with a median value of 0.91 seconds. Notably, without the delay for opacity or scrolling VERVE performed better on average. The opacity delay is used to ensure that the element is fully opaque or transparent prior to taking a screenshot. On the other hand, the scrolling delay is used to move the visible portion of the page, dictated by the viewport height and width, to an AOC in order to capture a snapshot. The larger the AOC, in proportion to the viewport, the more scrolling is required to capture the images needed for analysis by VERVE and thus the greater the number of delays that are needed in the delay-based configuration of the tool.

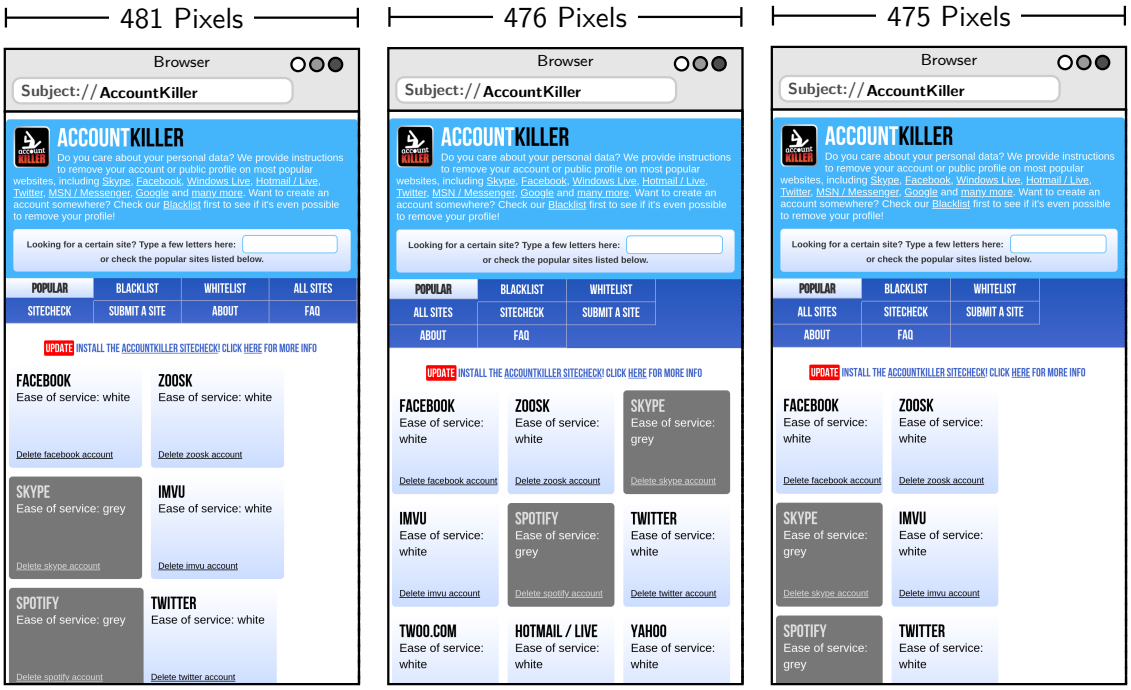
A closer inspection of VERVE's automated classification results without the added time delay provided evidence of why it was important to introduce it in the first place. Without the delay, VERVE classified three failures from BugMeNot, Ninite, and RetailMeNot inconsistently. This was for two reasons. First, when VERVE sends an instruction, through Selenium to the web browser, to change an element's opacity, sometimes it does not fully complete before a snapshot is taken. Thus, asynchronously capturing an image of the web page while in an unintended state may result in an incorrect classification. The second reason is that the rectangle coordinates retrieved from the DOM may vary if the element has not settled into its final position. This occurs due to a transitional effect of an element or possibly due to an optimization feature that speeds up the rendering of content that is available while other content is loading.

Figure 4.13 (b) showcases the runtime of each failure type in a box plot with no added delay. For element collision, element protrusion, viewport protrusion, and wrapping failures the median/mean values were 0.82/1.10, 0.85/1.01, 0.90/1.22, and 0.75/0.89 seconds, respectively. For small-range failures, the median was 0.99 and the mean was 3.29 seconds. Similar to the runs of VERVE with the added delay, the most notable difference between all five failure types is the runtime of the small-range failures which is fundamentally different from the other approaches. Furthermore, the median value of 0.99 seconds for small-range remained the same with and without the added delay while the mean was mostly unaffected by the removal of the delay. With both parts (a) and (b) of the figure using the same scale, the effect of the delay on runtime can be observed as the box plots are lower in the time scale, in (b), and are generally more compact.

Conclusion for RQ4 – On average, VERVE takes 4.08 seconds to classify a reported failure. This includes the use of a 200 milliseconds delay to allow elements of the web page to load and settle from a transitional effect. Without this delay, misclassifications are more likely. These results indicate that VERVE is practical, requiring developers to wait a very short amount of time for its classification results.

4.4.3 Discussion

For the classification of small-range failures, subjectiveness and mistakes in the manual classification may negatively influence the thresholds tuned by ThresholdFinder and thereafter the resulting agreement on newer subjects. This is a consequence of essentially "training" Verve to distinguish a true positive from a false positive on imperfect manual classification. Furthermore, tuning the threshold using multiple web pages simultaneously may not be ideal. In a real use case of ReDeCheck and Verve, a developer is expected to test and automatically classify layout failures from a single web page. Since the developer has the ultimate ruling over the manual classification of the reported failures, subjectivity is of little to no concern. For this reason, using custom thresholds made specifically for the page under test is expected to result in better classification. With the *Chi-Square* now identified as the best performing histogram measure for automated classification, future experiments can investigate the use of a custom threshold for each subject web page.



(a) Wider comparison. (b) Failure viewport. (c) Narrower comparison.

Figure 4.14: Three snapshots of the AccountKiller web page that capture its layout before multiple small-range failures occurs, in (a), and while the small-range failures are occurring in the range of 476–480 pixels in (b), and after the layout failures disappears again in (c), as reported by the ReDeCheck. The layout failure here is the square elements, seen in the lower portion of the screenshots, taking a three columns format for only four viewports widths.

Another related discussion point is the distribution of true positive to false positive manual classifications. This distribution may influence the thresholds and thus negatively or positively affects the agreement over newer subjects. Whether the classifications are balanced or imbalanced, the results may be either good or bad depending on if it matches the distribution of the classifications of the newer subjects. Even deeper, the visual diversity of the failures within each class, TP or FP, is another variable to consider. In the experiments, the pool of manual classifications from the initial set of subjects, indepen-

dently made by Walsh et al. [124], were used to find the thresholds. This pool contained 152 true positives and 43 false positive reports totalling 195 small-range failures. Of which, 147 true positives and 5 false positives come from a single source, the AccountKiller page. Moreover, many of these failures from the AccountKiller page were visually similar to each other. Potentially, this is one reason why ThresholdFinder was able to find thresholds that achieve an agreement as high as 98.5% using the initial set while only achieving as high as 87.8% using all of the subjects.

Many of the small-range failures arising from the AccountKiller subject are of high similarity. These failures pertain to 20 square, relatively large, elements that initially take on a five-column formation at the widest possible viewport. As the viewport width becomes narrower, the elements wrap around thus reducing the total number of columns. For reference, Figure 4.14 (a) shows these elements when the number of columns reduces to two. For the next four smaller viewports, the number of columns becomes three again as seen in part (b). Moreover, in one viewport smaller they become two columns again as seen in part (c). This anomalous layout of the 20 elements for only four viewport widths is picked up by ReDeCheck and reported as a small-range failure. Since many elements shift their relative position, this causes many reports to be generated. In turn, this large amount of reports caused by 20 elements that are visually similar and come from a single source may have biased the thresholds in the favour of this subject and specifically these elements.

The final point for discussion concerns a class of web pages with elements that feature alternating background colours or with running visual content like videos. Since Verve relies solely on a single snapshot from any given viewport and if the AOC includes this content, the captured image may result in a different colour histogram for the same AOC depending on the timing of capture. Consequently, the thresholds and classifications may vary. Since the initial conditions are inconsistent, the classification of Verve is potentially non-deterministic for this class of web pages. Logically, if it is not a temporary transitional effect, the added delay will not be of any help. One possible experiment for future work is to see if multiple snapshots of the same AOC, spanning some time delay, would aid in the automated classification of failures from this class of web pages.

One web page featured in the experiments, SB-Coming-Soon, did contain looping visual content in the form of a video. For reference, Figure 4.15 presents a snapshot of the subject at the widest viewport tested of 1400 pixels. This subject uses a video in the background that starts by showing a notebook with a hand pinning a pencil down on an empty page while the other hand holds the notebook in place. When the video is in motion, the hand begins to sketch out a rough drawing of a mobile phone. During the experiments, this was not an issue for Verve. Nevertheless, this is an impotent example since the video lies under an overlaying but partially transparent element with insignificant changes in scenery. Furthermore, this subject was not part of the initial set and hence was not part of the threshold tuning process.

4.5 Concluding Remarks

Even though responsive web design principles enable the creation of web pages that display correctly on a wide variety of devices with different viewport widths, developers may still introduce failures in the presentation of a web page even with the assistance of RWD based frameworks. Although the ReDeCheck tool automatically detects and reports responsive layout failures, a human web developer must manually classify each reported failure as being either a true positive, false positive, or a non-observable issue. This

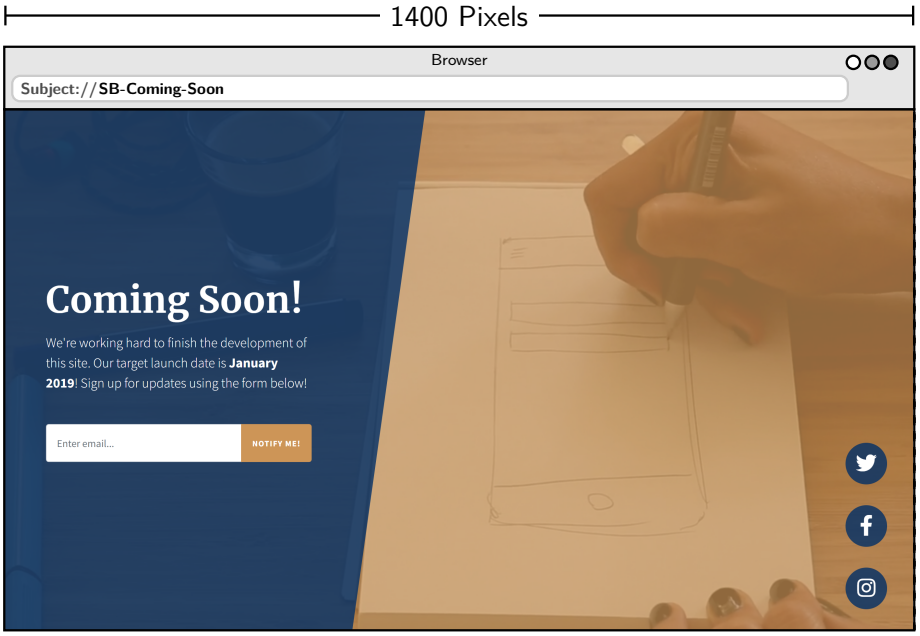


Figure 4.15: A snapshot from the SB-Coming-Soon subject which contains a looping video that spans the entire page. In motion, a human hand sketches out a mobile phone using a pencil starting with an empty page on a notebook.

process can be time-consuming, subjective, and error-prone. The previous chapter of this thesis introduced the VISER tool that is able to automatically perform this classification by manipulating the opacity of the HTML elements in a web page. While the empirical results from that previous chapter highlighted the efficiency and effectiveness of VISER, the tool was limited because it could only classify the element collision, element protrusion, and viewport protrusion layout failures reported by REDECHECK.

Since VISER does not classify either the wrapping or small-range responsive layout failures, this chapter presented VERVE, a tool that can automatically classify all five type of failures reported by REDECHECK. Along with extending VISER's opacity manipulation method to detect element wrapping failures, VERVE employs a colour histogram-based image comparison method that effectively classifies the small-range failures reported by REDECHECK. Considering 20 new pages in addition to the 25 web pages from the previous chapter's experiments, this chapter reported on the results from a comprehensive study of VERVE's efficiency and effectiveness, revealing its classification of all five types of failures frequently agreed with the manual ones produced by a human. The experiments also showed that VERVE normally took about 4 seconds to classify a failure among the 469 reported by REDECHECK. Given that a failure's classification with VERVE is less subjective and less error-prone than the manual counterpart, the results suggest that VERVE can support the testing of web pages that must responsively display at different viewport widths.

Using REDECHECK and VERVE, the developer can automatically detect layout failures in the web page and automatically filter out the more important real observable failures from the detected failures. Now comes the task of repairing these failures as quickly as possible. Without further automation, this must be done manually by investigating the code base, identifying the set of CSS properties that caused the problems, identifying the new values and CSS properties that will repair the layout, and finally making sure no new

problems arise in the patched code base. To further help the developer in this manual task, the focus of the research in the next chapter is on automating the repair of all five type of failures raised by the ReDeCheck tool.

5

Repairing Presentation Failures

The previous two chapters addressed the issue of classifying the presentation failures that were automatically detected by the responsive layout testing tool ReDeCheck. While in this chapter, I present an approach to automatically repair the failures detected in a subject web page. Fundamentally, it works to automatically repair a layout that contains a failure report by generating a patch using another layout from a different viewport where the same failure does not exist. Then it automatically verifies the success of the repair after applying the patch using the same constraints that identified the failure in the first place, namely the presentation failure detection algorithms.

The first section of the chapter begins by detailing the problem for which my automated repair approach was created to solve and introduces the tool Layout DR (Pronounced "Layout Doctor", responsive layout failure detection and repair) that detects and repairs presentation failures in responsively designed web pages. Then in Section 5.2, the detection part of the tool is covered. Essentially, this is a re-implementation of the detection algorithms of ReDeCheck into the new tool while undertaking minor improvements.

Then in Section 5.3, I will describe with details the steps involved in the repairing feature of the Layout DR tool. Briefly, the first step is to verify the reported presentation failure before attempting to repair it. Followed by the checking of the two nearest viewports that are not associated with the reported failure to be free from the same layout failure. Then I will explain how the Layout DR tool generates from these two viewports two responsive layout patches. Finally, I describe, the process of automatically accepting or rejecting a patch based on whether the failure was eliminated, or not, from the patched subject.

As part of my evaluation of the approach implemented in the Layout DR, Section 5.4 will begin by listing the research questions and methodology of the experiments of this chapter. Then the limitations are explored and the research questions are answered. The first question looks at Layout DR's capability of patching any given presentation failure and automatically verifying its repair. Then a human study is employed to judge the preference between the two repaired versions of the subject against the original unpatched web page using a subset of reported failures. Finally, the efficiency of the tool is examined.

The key contributions of this chapter are:

1. A technique to automatically create up to two repairs for each presentation failure reported by the ReDeCheck algorithms.

2. An empirical study to evaluate the effectiveness and efficiency of my automated repair technique. Demonstrating that it is always possible to automatically repair a failure reported by the ReDeCheck algorithms.
3. A human study to evaluate alternative repairs against the original subject. Demonstrating that humans generally prefer an automatically repaired web page over the original page containing the presentation failure.

5.1 Motivating the Research

Due to the dynamic nature of web content and the many viewports widths that a responsive layout must accommodate, frequent testing for presentation failures is important. The developer should make sure that the layout of the web page looks as expected in different viewport widths as the code base, content, or underlying browser changes. Tools like ReDeCheck aim to automate the process of testing a responsive layout under different viewport widths. Although beneficial for identifying presentation failures, the developer is left with the potentially long process of manually patching the layout failure of the web page. This may involve a simple change in a CSS property for a single HTML element or could require an extensive redesign of many elements and properties [128]. Therefore, it is ideal to extend automation beyond detection to suggest a solution for each reported failure that can be used at the very least to buy time for a more customized solution. To this end, the Layout DR tool is able to detect layout failures and suggest up to two patches that repair the layout issue in a responsively designed web page.

In one scenario when manually repairing a reported presentation failure, the developer must start by locating and assessing the potential problem in the webpage. This is done by first setting the browser to a viewport width where the failure reportedly manifested. Then they must identify the elements involved in the failure. Followed by an investigation of the possible causes and identification of the CSS properties that are most appropriate to repair the failure. This may take multiple attempts using different elements and CSS properties to reach an ideal layout. After repairing the failure, the developer must test the layout once again to verify that the repair eliminated the failure layout in all viewports. Furthermore, they must repair any new failures that might have been inadvertently introduced to fully repair the problem or try another approach to repair the original failure report.

The Layout DR tool alleviates the burdens associated with the manual repair process by automatically proposing up to two patches for each failure detected. These patches use a layout borrowed from another viewport width that does not contain the same failure. Moreover, the borrowed layout is scaled appropriately to fit the range of viewports where the failure originally manifested. The tool then automatically injects each patch into the live subject web page in order to reach an acceptance or rejection of the generated patch. If the patch eliminates the failure from the viewports associated with the failure it is accepted and saved for the developer as a repair. Otherwise, the patch is rejected and optionally saved for further manual analysis of the patch.

In addition to the repair feature of the tool, Layout DR integrated the detection algorithms of ReDeCheck [124] to streamline the process of automated detection and repair. A further benefit of reprogramming the algorithms into the new tool is to resolve known issues with the legacy tool. These include resolving the minor false positive reports that were generated by the legacy tool, achieving a more precise categorization of failure types, and reducing installation complexity. More importantly, the new tool is capable of detecting all five types of presentation failure found in responsively designed web pages as did the legacy tool.

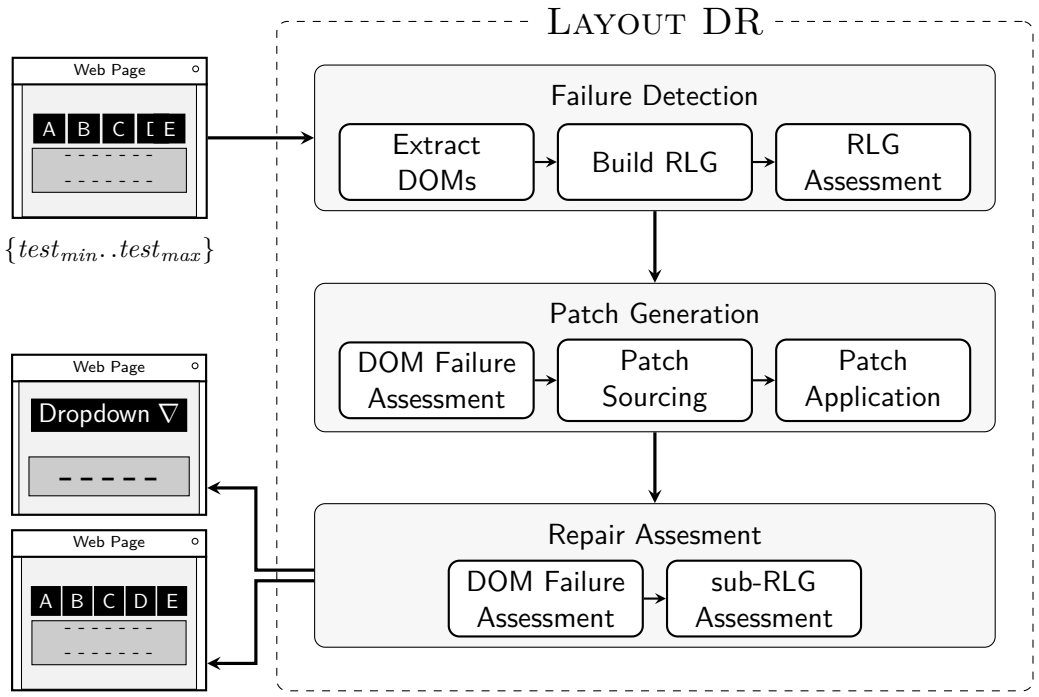


Figure 5.1: A high-level overview of Layout DR's three internal phases and tasks for the automated detection and repair of presentation failures featuring the capability of outputting up to two alternative repairs for a single presentation failure.

A high-level overview of the phases and internal components of the Layout DR tool is illustrated in Figure 5.1. As input, the tool expects a URL to the web page under test. In the figure, the page under test contains a collision failure between elements D and E over some viewports. The tool also expects as input, the range of viewport widths $\{test_{min}..test_{max}\}$ to test for responsive layout failures. The tool outputs up to two alternative repairs for each presentation failure detected as featured in this figure. The main phases of the tool are *failure detection*, *patch generation*, and *repair assessment*. In the next section, I will describe the first phase while the patch generation and repair assessment phases are described in the section to follow.

5.2 Detecting Failures

Prior to detecting the different types of failures, the Layout DR tool must first extract the positions of each HTML element of a web page from the viewport widths in the testing range, $\{test_{min}..test_{max}\}$, and build a model of the page on which the algorithms can infer failures. This involves extracting the rectangular coordinates of visible HTML elements by querying the DOM for each viewport width in the testing range. This process is part of the *Extract DOMs* component of the tool shown in Figure 5.1. The tool steps through each viewport by setting the browser viewport, querying the DOM and storing the information to be later used to build a model of the page.

In the *Build RLG* component of the tool shown in Figure 5.1, the relative position of each element is calculated using the extracted rectangular coordinates. For each of the extracted elements, a *parent* element is assigned. The parent of a *child* element is the one with the most confined rectangle containing the child elements coordinates. The children that share the same parent are referred to as *siblings*. Then the relative positions between

siblings are calculated to establish more relationships, namely the *above* or *below*, *right* or *left*, and *overlap* relationships. These relationships or relative alignments between pairs of HTML elements are the foundation for a graph-based model of the web page called the Responsive Layout Graph (RLG) [124, 127]. Over different viewport widths, the model also incorporates the range of viewports where the alignments between elements hold true. The root element in the RLG model is the `body` HTML element which is a container of other elements but does not have a parent.

With an RLG model of the web page, the detection algorithms are able to report five types of presentation failures referred to as Responsive Layout Failures (RLFs). This is a task carried out by the *RLG Assessment* component of the tool shown in Figure 5.1. By comparing the positions of elements over different viewport widths encoded into the RLG, the algorithms can infer and report failures. The five types of RLFs detected by Layout DR are *element collision*, *element protrusion*, *viewport protrusion*, *element wrapping*, and *small-range* failures. Example snapshots of the types of failures detected in the subjects used in the experiments of this chapter can be seen in Figure 5.2 (a), (d), (g), and (j). The figure shows four presentation failures as the top snapshot of each subject web page. Followed by two lower snapshots for each subject showcasing two alternative repairs after Layout DR automatically applied two alternative patches to the viewport with the failure. Although the Layout DR repair approach does not depend on the failure type, a summary of each failure type explaining how it is detected and the improvements made in the new tool are described next.

5.2.1 Element Collision

As the layout responds to smaller viewport widths, a parent element will have less space to accommodate the child elements it contains. Due to the reduced space of the parent element, siblings may start to overlap each other. As a result, one of these overlapping elements may be covering the other and distorting its visibility and accessibility. This unintended consequence of overlap between elements is referred to as an element collision failure. This type is detected and reported if over a range of viewports widths two siblings overlap but at a wider viewport do not overlap while the parent of both elements remains the same. The report of the failure made by Layout DR discloses the web page, failure type, failure range $\{fail_{min}..fail_{max}\}$ which is equal to the range of consecutive viewport widths where the overlap occurs, and the XPaths of the two elements involved in the overlap.

A real example of an element collision is presented in Figure 5.2(a). The collision in the image occurs between the blue header and the yellow information banner partly hidden behind the blue header. This failure was detected by Layout DR in viewport widths 990-991 pixels of the WillMyPhoneWork subject. Prior to the failure at the wider than the range viewport width of 992 pixels, $fail_{max}$+1, the yellow text and the banner are not in collision. This is identical to the automatically repaired version of the page seen in part (b) of the figure. Likewise, at the narrower width of 989 pixels, $fail_{min}$−1, as seen in the alternative repaired version of the subject in part (c), the two elements are not in collision.

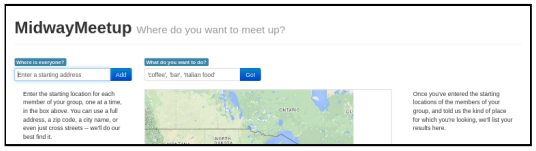
A minor improvement to the detection and reporting of an element collision failure in Layout DR is the consideration of a concurrent element protrusion failure. To suppress unnecessary reporting of a collision while the same element is involved in a protrusion (is outside the boundary of its intended parent), the failure range of the collision is checked against the failure ranges of known protrusion failure for crossover. If the range of viewports of the collision is fully within the range of a protrusion failure involving the same

(a) Presentation failure in viewport widths 990-991 pixels.

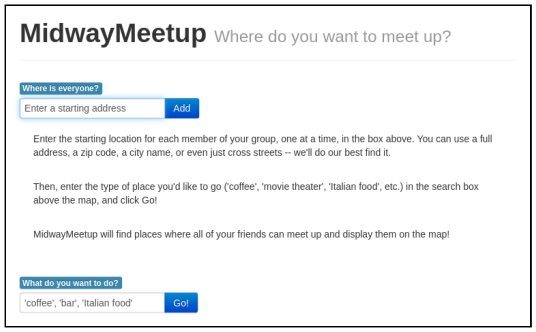
(b) Repaired using the wider source-viewport of 992 pixels.

(c) Repaired using the narrower source-viewport of 989 pixels.

(g) Presentation failure in viewport widths 768-1222 pixels.



(h) Repaired using the wider source-viewport of 1223 pixels.

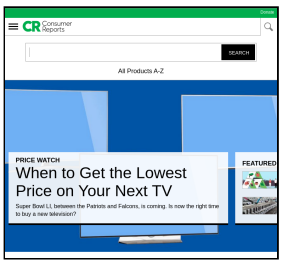


(i) Repaired using the narrower source-viewport of 767 pixels.

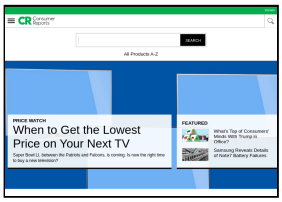
(d) Presentation failure in viewport widths 980-991 pixels.

(e) Repaired using the wider source-viewport of 992 pixels.

(f) Repaired using the narrower source-viewport of 979 pixels.



(j) Presentation failure in viewport widths 768-1028 pixels.



(k) Repaired using the wider source-viewport of 1029 pixels.

(l) Repaired using the narrower source-viewport of 767 pixels.

Figure 5.2: Four snapshots of presentation failures in (a), (d), (g), and (j) all captured at the lower bound of the failure range. Displayed below each of the failure snapshots are two alternative repair snapshots after LAYOUT DR automatically patched the subject using alternative source-viewports to create the CSS of the patch.

element, the collision report is suppressed. Beyond the benefit of reducing reports, a repair attempt benefits by resolving the origin of the problem.

5.2.2 Element Protrusion

As the width of the viewport becomes smaller and the area of a parent becomes more confined, the coordinates of a child element may exceed the boundary of the parent. Once the element protrudes from its parent, the protruding element will overlap its previous parent or with one of its *ancestors*, higher-level parents. Therefore, the overlapping elements are now siblings and share a common parent while prior to the protrusion one was in the ancestry of the other. This type of failure may lead to visual changes in the layout that are unpleasing in design and may prevent visibility and access of elements. This type of failure is detected when an overlap between siblings is observed but at the wider viewport, one of the two elements is a parent or ancestor of the other. The report includes the name of the web page under test, failure type, failure range $\{fail_{min}..fail_{max}\}$ equal to the overlap range, and the XPaths of the overlapping elements.

An example of an element protrusion from the MidwayMeetup subject can be seen in Figure 5.2(g). This failure was automatically detected by Layout DR in the viewport range 768-1222 pixels where an invisible container of two elements, a button labelled 'Add' and an input bar, protruded its own container. Although the containers are not visible, the overlap with other elements is evident in the snapshot. The overlap causes other elements to be rendered on top of the button effectively hiding the button which is no longer accessible while it is under the other elements of the page. At one viewport wider than the failure range $fail_{max}+1$, 1223 pixels, the button is both accessible and visible as seen in the repaired version of the page in part (h) of the figure. The layout of the narrower viewport $fail_{min}-1$, 767 pixels, repositions the first container above the other to allow for more space which in turn allows elements to expand horizontally. This can be seen in the second alternative repaired version of the page by Layout DR in part (i) of the figure.

Related to this failure type, two improvements were made in Layout DR to mitigate against ambiguity, account for edge cases, and to increase the stability of encoding the layout into the RLG model. The root causes of these issues are best described in two scenarios. The first is when two or more elements have the same coordinates. In this case, the ambiguity is in determining which of these elements contains the other. Adding to the complexity, they may be overlapping siblings and not containers of one another. To resolve this dilemma, Layout DR uses the XPath of the element to resolve this ambiguity emulating the approach used in the X-PERT tool of Choudhary et al. [28]. Therefore, Layout DR uses the DOM hierarchy preserved in the XPaths to discern parents from children when this scenario arises.

The second scenario where Layout DR improved over its predecessor, ReDeCheck, is when the coordinates of elements undergo minor changes between succeeding viewports. These changes could be as small as a fraction of a pixel but effectively sway the results of determining the alignments between elements. These slight variations in coordinates may also be frequent and intermittent and therefore cause an increase in the number of reported failures that are not visible. Although the predecessor tool used a tolerance value for minor protrusion of an element, it did not have a tolerance for including a candidate parent and therefore made the decision solely on the bases of the tightest container. Meanwhile, the new tool considers neighbouring coordinates within a fixed tolerance value as equals. Therefore, adding further stabilization to the extracted model across different viewports.

5.2.3 Viewport Protrusion

Similar to an element protrusion, a viewport protrusion occurs when an element exceeds the boundary of the main `body` HTML element. This may cause a portion of the element to be out of view due to the size of the viewport width which is limited by the browser window size. In a favourable outcome, this will require the user to horizontally scroll the page to view the portion of the element exceeding the viewport width. Otherwise, if the browser does not have a scroll bar displayed, the viewport protruding portion of the element will not be viewable. Since the `body` element is the root parent in the RLG model, the protruding element will not be assigned a parent. Therefore, a viewport protrusion failure is detected when an element has no parent over some viewport range while at the wider viewport it did have a parent in the RLG. The failure range for this type is equal to the range where the element did not have a parent in the RLG. Along with the failure range $\{fail_{min}..fail_{max}\}$, the failure report includes the name of the web page, failure type, the XPath of the element that temporarily had no parent, and the XPath of the `body` element.

An example of a viewport protrusion is showcased in the snapshot from the ConsumerReports subject in Figure 5.2(j). In the snapshot of the web page, an element with a white background containing images and text does not fit in the viewport width of 768 pixels and therefore is largely not visible. The protrusion of the viewport was detected by Layout DR in the viewport range of 768-1028 pixels. At the wider viewport width of 1029, $fail_{max}$+1, the element and its content are fully visible within the viewport size. The successfully patched web page seen in part (k), is based on and therefore identical to, the layout of the wider viewport. The alternative repair seen in part (l) showcases the layout of the narrower viewport size of 767 pixels, $fail_{min}-1$. In this layout, the developers hid some elements to allow more space for the remaining elements.

For the detection of viewport protrusion failures, two improvements were made in the Layout DR tool. The first is to the modifying the height of the `body` element in the RLG model. Because some web pages limit the height of the `body` element while other elements go beyond the defined height, the page would not be accurately modelled in the RLG. Ideally, a pseudo node in the RLG should represent the viewport width and should have infinite height. Not to deviate from the essence of the RLG model, Layout DR used an infinite height for the `body` element to resolve this issue. The second improvement is to add a secondary condition to the algorithm when checking that the viewport protruding element has a parent at the wider viewport. This secondary condition makes sure that the element must also have an ancestry connecting back to the root of the RLG model, the `body` element. In edge cases, elements may have a parent that is not connected to the root of the RLG and in smaller viewports does not have a parent. This evolution in Layout DR properly handles these cases.

5.2.4 Element Wrapping

At larger viewport widths, related elements may be laid out in row formation as intended by design. For example, the row may present the main menu of links for the user to access different parts of the page. An element wrapping failure occurs when an element breaks from row formation and wraps into a new row due to a decrease in viewport space. This may lead to an unpleasing layout in the web page. Detecting this failure mainly relies on the ability to determine rows of elements. To achieve this, a minimum of three HTML elements is predetermined to constitute a row. Then, a row is formed only if the elements have a *right* or *left* alignment with each other but do not have an *above* or *below* alignments.

Once detected, the web page name, failure type, failure range $\{fail_{min}..fail_{max}\}$, the XPath of the wrapped element, and the XPaths of the other row elements are reported.

An example of a wrapping failure from the MantisBT subject is shown in the snapshot of Figure 5.2(d). In this snapshot, the *Hosting* link has wrapped below other menu items to form its own row. This holds true for the entire viewport width range of 980-991 pixels. At one viewport wider than the failure range, $fail_{max}$+1 or 992 pixels, all menu items are properly aligned in a row. This is identical to the repaired version of the page seen in part (e) of the figure. At one viewport narrower than the range, $fail_{min}-1$ or viewport 979, the menu changes to a drop-down list. This effect can be seen in the snapshot of the repaired version of the page in part (f) of the figure.

For this failure, the new tool uses a reversed form of the original detection algorithm process. While the original approach determines rows of elements first and then seeks to find a wrapped element, the new approach in Layout DR searches for the wrapped element and then the existence of the row. Furthermore, the original algorithm involved sorting and splicing of alignment ranges that were not necessary using the RLG implementation choices used in Layout DR. While the implementation of the predecessor tool used a sibling edge between children of the same parent, the Layout DR tool infers this relationship using a common parent as an identifier of siblings. Furthermore, the new tool uses an edge to model each type of alignment between siblings while its predecessor used constraint labels (*i.e.* , above and right) on sibling edges to achieve the purpose. More importantly, no false positives were encountered during the experiments of this chapter as was the case in the predecessor tool during the experiments of the previous chapter.

Layout DR detects a wrapping failure of an element by first iterating over all of its ranges to find above/below edges. The element that does have another sibling above it will surely have an above/below edge between the two nodes. Meaning, that the element below is a candidate wrapped element and the element above is a candidate row member. If this candidate wrapped element was part of a row – that includes the above element – before the above/below edge was observed at the wider viewport, then it meets the criteria of a wrapping failure. Namely, the element is part of a row and later, at a smaller viewport, wrapped below the other row elements. Therefore, the algorithm checks the wider viewport for siblings in row formation with the wrapped element as a member. Then checks again that the row is intact after the element has wrapped. If the other elements of the row maintain formation after the wrapping occurs then the failure is reported. Here, the failure range is equal to the range when the row is above the wrapped element.

5.2.5 Small-Range

As the viewport width changes, a responsive web page uses media rules to change the layout of the web page in response to the available viewport width. Inaccurately setting one or more rules may cause unintended layouts over a few viewports widths. Hence, a small-range failure occurs when elements take on a layout only true for a small number of viewport widths. The failure is detected by first identifying the alignments with ranges spanning five viewports or less for any of the above, bellow, right, left, or overlap alignments between any two elements. The Layout DR tool uses the five viewports threshold by default to duplicate what the predecessor tool did. Then, the number of alignments between the two elements is calculated within this range. The calculation is repeated to know the number of alignments at the narrower viewport and again at the wider viewport. If the set difference in the number of alignments is two or greater within the range and at the narrower and wider viewports, the failure is reported. The report includes the name of the web page, failure type, failure range $\{fail_{min}..fail_{max}\}$, and the XPaths of the two

elements involved.

An example of a small-range failure from the WillMyPhoneWork subject is presented in Figure 5.2(a). This failure is reported twice by LAYOUT DR, once as an element collision and again as a small-range failure. The detection algorithm for small-range and that of the element collision failures are independent of each other but in this case, are both satisfied. Since the range of viewports where the overlap between the elements occurs in only two viewports, 990 and 991 pixels, the small-range algorithm creates its own report of the failure. Not surprisingly, both reports with two different failure types detected by LAYOUT DR were successfully repaired as seen in parts (b) and (c) of the figure. When I compared both failures along with their repairs, I found them to be indistinguishable.

For this type of failure, LAYOUT DR used the latest version of the detection algorithm [128]. This is because the older version of the algorithm was prone to reporting a large number of false positive failures, up to thousands of reports. After all, it was too sensitive to minor changes in alignments. On the other hand, the latest version is less sensitive because it looks for multiple simultaneous alignment changes. These changes are observed in the number of alignments for a pair of sibling elements over a maximum of five consecutive viewports against the immediately wider and against the immediately narrower than the range viewports. During the experiments of this chapter, the LAYOUT DR tool reported a low number of small-range failures and did not have the same problem of creating false positives reports as did the older version of the algorithms.

5.3 Repairing Failures

The main feature of LAYOUT DR is the automated repair of presentation failures detected in responsively designed web pages. The previous section looked at the failure detection feature integrated into the tool while in this section the approach implemented in LAYOUT DR to repair the presentation failure is explained. This section starts with a summary of the approach followed by the detailed process of repair.

5.3.1 Summary of Approach

To repair a responsive layout that contains a presentation failure, the LAYOUT DR tool creates a patch using a layout from a viewport outside of the set of consecutive viewports where the failure was reported $\{fail_{min}..fail_{max}\}$. This viewport used as a source for harnessing the CSS of the patch that makes up the layout of the repair is referred to as the *source-viewport*. Once the patch is applied, the layout derived from the source-viewport is imposed on the viewports in the failure range $\{fail_{min}..fail_{max}\}$ by rules added to the patch. The current implementation of LAYOUT DR uses up to two source-viewports to generate up to two alternative repairs. More specifically, the two bordering viewports on either side of the failure range are used by the tool as source-viewports. I refer to the smaller bordering viewport $fail_{min}-1$ as the *narrower source-viewport* and the larger bordering viewport $fail_{max}+1$ as the *wider source-viewport*.

Prior to harnessing the source-viewport, the repair process starts by first assessing the findings of the RLG based detection algorithms at three sample viewports spanning the reported failure range. These are $fail_{min}$, $fail_{max}$, and the middle point of the range $fail_{mid}= floor((fail_{min}+fail_{max})/2)$. This is similar to the process presented in the previous two chapters where the DOM is used to classify a given failure report at three viewports from the failure range. This process of corroborating the failures detected using the RLG

model is important in order to identify false positives before a repair is attempted. More importantly, the corroboration of the report allows the tool to confidently assess the success of a repair once the patch is applied. Since the failure was detected using the RLG model and corroboration using the DOM prior to patching the web page, LAYOUT DR only accepts a patch if the failure is not observed in the DOM nor detected in the RLG of the patched subject.

There are two requirements that must be met before LAYOUT DR uses a source-viewport to create a patch. The first is that it must be free from the reported presentation failure and the second is that the source-viewport was included in the testing range $\{test_{min}..test_{max}\}$ used to find the failure in the first place. Both of these requirements are automatically checked by the LAYOUT DR tool prior to generating a patch from any source-viewport. For the source-viewport that meets the requirements, the tool sets the browser's viewport to the source-viewport and captures all of the CSS making up the layout for usage in the patch. Using the failure range, media rules are added to the patch to restrict the application of the layout to only the viewports of the failure range. Finally, the layout of the patch is scaled appropriately for each viewport in the failure range. The patch is then injected into the web page to repair the failure at hand. Then the success of the patch at repairing the failure is automatically determined by LAYOUT DR. It is deemed successful if the same failure no longer exists in the subject after applying the patch. Outputting, for each failure report, up to two alternative patches that successfully repaired the presentation failure.

5.3.2 DOM Failure Assessment

With a failure report ready from the *Failure Detection* phase of the tool, the second phase, *Repair Generation* begins with the *DOM Failure Assessment* component as shown in Figure 5.1. The role of this component is to confirm the existence of the failure that was detected using the RLG model. For this, the coordinates of the elements associated with the failure are extracted from the DOM and examined using the relative positions of the elements. Therefore, attempting to corroborate what the RLG-based failure report indicated. Looking ahead into the next phase of *Repair Assessment*, this component is repeated in order to reject or accept a given patch depending on whether the failure persists or not in the patched web page. Therefore, the tool must assess for the existence of the failure using the DOM prior to using it to assess the repair in order to avoid a false indication of successful repair if the failure never existed. This may be the case if the report was made in error due to a flaw in the code or as a result of a limitation in the RLG model. In general, the DOM failure assessment step raises the quality assurance of the reports generated using the RLG model and the later automated repair assessment.

To assess a reported failure using the DOM, the *DOM Failure Assessment* component of LAYOUT DR, see Figure 5.1, revisits the viewports where the failure was detected to check the relative position of the elements. More precisely, three sample viewports spanning the reported failure range are used to evaluate the entire range $\{fail_{min}..fail_{max}\}$. These are the $fail_{min}$, $fail_{mid}$= $floor((fail_{min}+fail_{max})/2)$, and $fail_{max}$ viewports of the reported failure range. To do this, the LAYOUT DR tool sets the browser to these viewports and extracts the bounding-box coordinates of the elements associated with the failure from the DOM. Next, the relative positions of the elements are checked for failure. Depending on the failure type, this may be rectangles not containing one another, rectangles overlapping, or the rectangles are not in the correct relative direction like being below instead of to the right of the other.

Each type of presentation failure is assessed using the elements mentioned in the re-

port based on the semantics of the RLF type. For an element collision, the semantics of the failure indicates that two elements are overlapping when they should not be, hence the word collision. Therefore, if the DOM coordinates indicate an overlap between the two elements then the failure is confirmed. For an element protrusion or a viewport protrusion failure, they are confirmed if the DOM readings conclude that the coordinates of the protruding element go beyond the coordinates of the parent element. For an element wrapping failure, the wrapped element is checked to make sure it is below all other elements. Finally, a small-range failure is confirmed if, based on the report, the alignments between the two elements holds true at the viewport being investigated.

One requirement for a layout of the source-viewport to be used to generate a patch is that it must be free from the reported failure. This is another task for the *DOM Failure Assessment* component where the tool now seeks a negative outcome when checking for failure at the source-viewport. Since the detection algorithms primarily infer the proper layout of elements from the wider viewport, the failure should never be observed in the wider source-viewport. This does not follow for the narrower source-viewport. The reported failure range is specific to the failure type and hence does not indicate the end of the failure at the $fail_{min}-1$ viewport. For example, when an element protrudes out of its non-root container in the RLG over a range of viewports it is of type element protrusion. If at a smaller viewport the protrusion is great enough to also leave the confines of the root container of the RLG, then it can no longer be reported as a protrusion failure. Instead, it would be a new viewport protrusion failure with its own range. Therefore, the narrower source-viewport must undergo a DOM-based failure assessment to confirm it is free from the same failure. Although expected to be free from the failure, the wider source-viewport is also checked. Assuming it is true on faith and finding out it was not will cause a repair to be rejected during its assessment because the patch copied the CSS from a layout containing the same failure. This may have a negative impact on the approach for the wrong reasons.

There is also a second requirement for a source-viewport that must be met before Layout DR accepts the layout for usage in a patch. Only if the source-viewport is a member of the testing range $\{test_{min}..test_{max}\}$ used to build the RLG will it be used to generate a patch. This is always true for the wider source-viewport since it is used in the RLG to infer the proper layout of elements. Unfortunately, this not the case for the narrower source-viewport which may be smaller than the smallest viewport in the presentation testing range. For example, if the RLG is built using the viewport widths 320-1400 pixels and a failure is reported within the range 320-450 pixels, the narrower source-viewport of 319 will not be used by the tool. In this case, the narrower source-viewport would be categorized as "not applicable" in the experiments of this chapter.

The Repair() procedure described in Algorithm 7 takes care of assessing the failure and checks the viability of the source-viewport for usage in repair. As input, the HTML elements, type, and range of the failure are passed to the procedure to repair the failure. The algorithm starts by calculating the minimum, middle, and maximum points of the failure range in lines 2 to 4 and puts them into a set in line 5. Then the isFailing() procedure of line 6 is used to assess whether the failure manifests in these three viewports or not. Then the narrower viewport and wider viewport are calculated in lines 8 and 9. Followed by another call to the isFailing() procedure, line 10, but this time it is to confirm that the layout of the wider source-viewport is free from the failure. If it is, a call to generate the patch is made in line 11. Then the isCoveredByRLG() procedure, in line 12, makes sure that the narrower source-viewport is part of the viewports covered during the detection phase. Finally, the isFailing() procedure, line 14, is used on the narrower source-viewport to confirm the failure does not exist there before an attempt to

Algorithm 7 Repair Presentation Failures

INPUT: Failure HTML *elements*, viewport *range*, and RLF *type*.

```
1: procedure REPAIR(elements, range, type)
2:     Fmin ← MIN(range)                                          ▷ fail_min
3:     Fmid ← FLOOR((MIN(range) + MAX(range))/2)                  ▷ fail_mid
4:     Fmax ← MAX(range)                                          ▷ fail_max
5:     F ← {Fmin, Fmid, Fmax}                  ▷ viewports covering failure range
6:     confirmedFailure ← ISFAILING(F, elements, type)
7:     if confirmedFailure = True then
8:         ns ← Fmin − 1                                          ▷ narrower source
9:         ws ← Fmax + 1                                          ▷ wider source
10:        if ISFAILING(ws, elements, type)) = False then
11:            PATCH(elements, range, type, ws)
12:        covered ← ISCOVEREDBYRLG(ns)
13:        if covered = True then
14:            if ISFAILING(ns, elements, type)) = False then
15:                PATCH(elements, range, type, ns)
16:    return
```

patch the failure is made in line 15 using the narrower source-viewport.

5.3.3 Patch Sourcing and Application

Once LAYOUT DR confirms the existence of the failure within the reported range and verifies that the layout of the source-viewport is free from the failure being repaired, the heart of the repair process begins. Its purpose is to collect the CSS from the source-viewport layout, rescale it, and apply it to all viewports in the failure range. First, the *Patch Sourcing* component of the tool sets the browser viewport width to the source-viewport in order to access the final CSS that is *applied* to all elements in that viewport. Here, applied refers to the final CSS values making up a layout after the browser resolved all rules and property settings of the web page's external CSS files, internal `style` HTML elements, inline `style` attribute, and any browser-specific defaults. Starting at the root element of the DOM tree, the `html` element, the DOM is traversed to capture all available CSS properties of each element in the tree. To do this, the `getComputedStyle()` JavaScript method is utilized by LAYOUT DR. This method returns the CSS settings and the computed pixel values associated with the element after the browser resolves the CSS. These values of all CSS properties for each element in the DOM are what the tool needs to reproduce the layout in any other viewport.

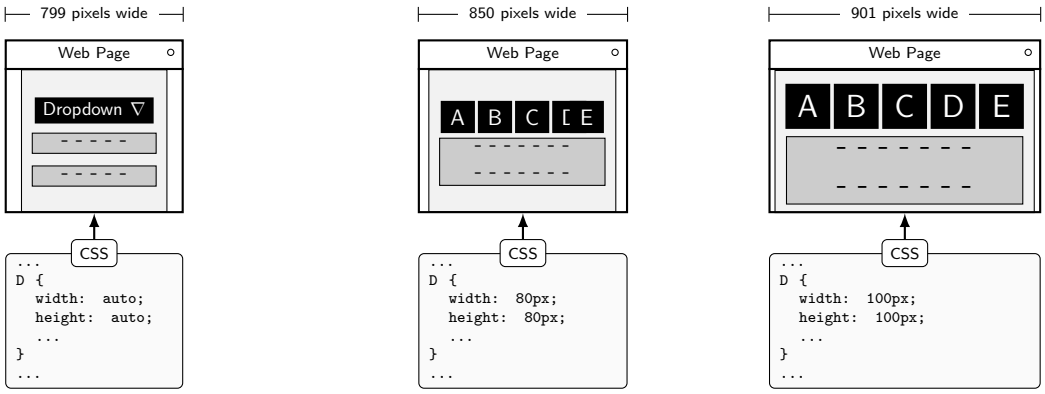
Now that the tool has the CSS properties needed to build the patch, it must first create a CSS selector for each element so that the properties can be reapplied to the same element they were extracted from. For this, LAYOUT DR uses the XPath of each element to generate a unique CSS selector which will encompass all the properties of an individual element in the patch. Since the CSS properties in the patch will be competing with other CSS of the original web page, the `!important` flag is added to all properties in the patch to override any competing declarations. With the selectors set to target each element, the tool must now target the viewports for which the patch should take effect. Otherwise, the patch will be applied to all viewports. To restrict the patch to the failure range, the selectors and their properties are encapsulated within a media rule spanning the failure range.

Without further improvements to the patch, the web page will cease to be responsive where the patch is applied. This is because the patch contains absolute values specific to a single viewport. These pixel specific values from the source-viewport are not ideal for displaying in other viewports of a responsively designed web page. Making the repaired version of the web page too wide for the viewport using the wider source-viewport patch. In which case, only some portion of the total width of the layout will be displayed within the viewport of the page. To illustrate this, the first web page wireframe from the left-hand side of Figure 5.3(d) is an example of this effect where the layout borrowed from part (c) with a viewport of 901 pixels is too big for the smaller viewport width of 850 pixels where the failure used to be before applying the borrowed layout. Conversely, the narrower source-viewport patch will not use all the available viewport width and will have empty space displayed. This is due to the failure viewport width being bigger than the narrower source-viewport where the layout is borrowed from thus allowing for additional empty space to exist within the viewport. This effect is illustrated in the layout of the first wireframe of Figure 5.3(e) that is borrowed from part (a).

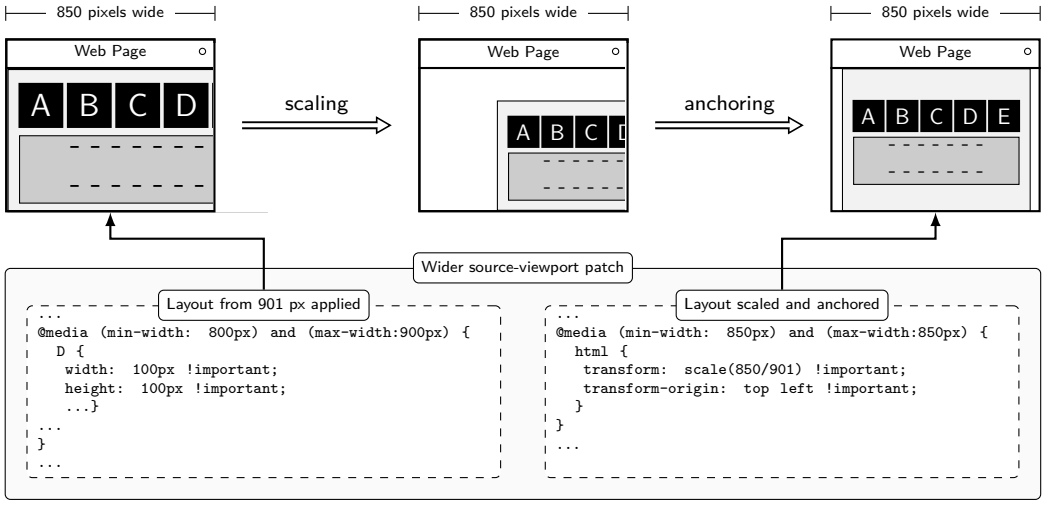
To equalize the size of the layout to match the size of the viewport width and nothing more or less, Layout DR uses the `scale()` CSS method with the `transform` CSS property for this purpose. The `transform` property can modify the coordinates of the target element to rotate, scale, skew, or translate from its original coordinates. As for the `scale()` method, it is used to scale the coordinates of the target element to be smaller or larger than the original coordinates resulting in a 'zoom'. The effect of the `scale()` method is illustrated in the second wireframe of Figure 5.3(d) and Figure 5.3(e) using the layouts borrowed from the wider source-viewport and narrower source-viewport respectively. The result is a scaled-down version of the layout that shrinks to the centre. The application of this property and method on an element exceeds the element itself and affects all descendant elements in the DOM tree. Therefore, the tool applies this property to the root element of the DOM, the `html` element, to scale all elements of the layout appropriately.

Since presentation failures usually manifest in more than a single viewport, the layout of the patch should adjust appropriately to each viewport in the reported failure range. The Layout DR tool considers this when creating the patch by calculating a different scale value for each viewport. The value is calculated based on the ratio of the browser's current viewport over the source-viewport, making the patch responsive to different viewports. For example, the presentation failure of Figure 5.3 has a failure range of 800-900 pixels. The failure illustrated in part (b) is from the viewport 850 pixels and has two different ratios depending on the source-viewport of the patch. For the wider source-viewport shown in part (d), the value is 850/901 which decreases the size of the layout imposed by the patch. While for the narrower source-viewport shown in part (e), the ratio is 850/799 which increases the size of the layout. The full CSS of the patch would include multiple media rules, one for each viewport in the range of the failure.

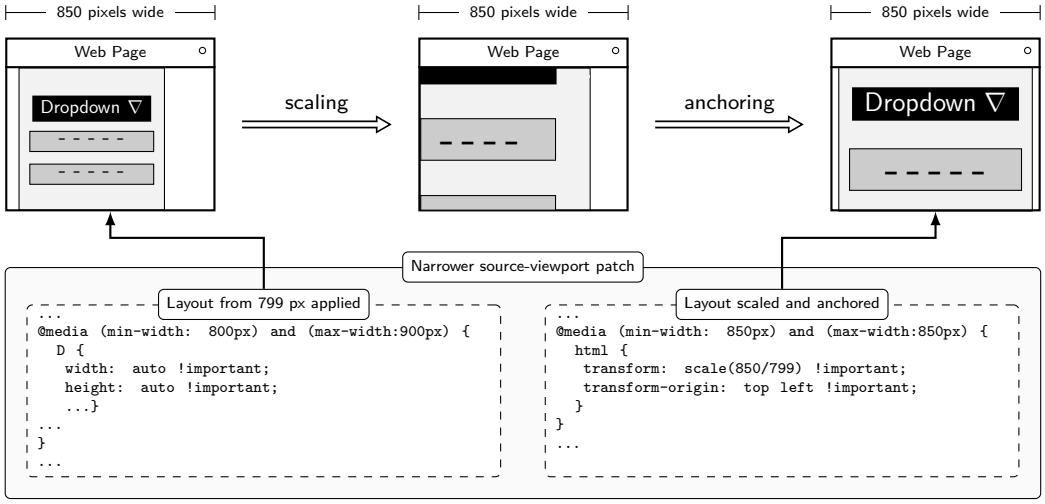
Although Layout DR scaled the patch using the `transform` property, the scaled page transformation is anchored to the centre of the original coordinates. The result is a web page with empty space to the top, right, bottom, or left of the page. This is the case in the layout of the second wireframe seen in Figure 5.3(d) that used the wider source-viewport. Even worse than the empty space surrounding the layout, is the portion of the page that will not be visible without scrolling horizontally breaking one of the responsive design principles. For the narrower source-viewport viewport, the scale effect seen in part (e) is even worse. Since no scrolling will allow a user in the negative direction of coordinates of the page, a large portion of the layout from the left and the top will be missing. To resolve this issue, the `transform-origin` CSS property is also added to the



(a) Narrower source-viewport. (b) Viewport with failure. (c) Wider source-viewport.



(d) The failure patched using the wider source-viewport and showing the scaling and anchoring effect.



(e) The failure patched using the narrower source-viewport and showing the scaling and anchoring effect.

Figure 5.3: A wireframe example of the steps involved in producing two repairs for the presentation failure illustrated in (b) where the elements labelled D and E are in collision from viewport 800px to 900px. The first repair in (d) uses the layout of the wider source-viewport of 901px while the repair in (e) uses the layout of the narrower source-viewport of 799px. The CSS snippets of different viewports from the original page are shown (a), (b), and (c) while the CSS snippets of the two patches are in (d) and (e).

patch to position the scaled page appropriately at the *top left* of the original coordinates. This property identifies the position around which a transformation is applied using the `transform` property. Therefore, anchoring the scaled layout to the proper location. This can be seen in the final wireframe of Figures 5.3(d) and (e) where the repair is completed.

With the patch now ready, LAYOUT DR can proceed to apply the patch to the web page in order to repair the failure. To this end, the *Patch Application* component of the tool creates a `style` element that contains the patch and injects it into the DOM tree. The browser then automatically applies the CSS of the patch and refreshes the rendered web page. This injection method results in a patch that persists when the viewport width is changed but does not persist through a page reload or if the page navigates to another URL. Once LAYOUT DR assesses the repair, whether the patch is accepted or rejected, the tool ejects the patch from the page by removing the `style` element from the DOM allowing another patch to be processed.

5.3.4 Repair Assessment

With the web page now patched, LAYOUT DR automatically checks to see if the repair was successful. To reach a negative verdict faster, the tool starts by trying to reject the patch as part of verifying its success. The patch is rejected if after checking the DOM at a single viewport from the failure range, the tool finds that the failure remains in the patched subject. To do this, the *DOM Failure Assessment* component of the tool is employed again to check for the failure using the DOM. It first sets the viewport width to the minimum viewport of the reported failure range, $fail_{min}$. The minimum is used because it has the most confined display space and hence has a higher potential of retaining the failure than larger viewports of the failure range. To reject the failure, LAYOUT DR repeats the DOM-based failure assessment process explained in Section 5.3.2. If the failure persisted with the patch applied, it is rejected and saved as a failed attempt. Otherwise, to fully confirm the success of the repair, the tool needs to build a new RLG using the patched subject and confirm that the failure is not detected using the RLG detection algorithms.

To check the RLG of the patched web page, the tool uses information gathered from a subset of viewports in the patched subject that are associated with the reported failure range. Therefore, skipping some viewports for time efficiency. This approach to efficiency by not going through all viewports was used in the legacy tool to detect failures faster while LAYOUT DR uses it during repair assessment. I refer to this model that uses a subset of the viewport that were used to build the full RLG as the sub-RLG. They include $fail_{min}$, $fail_{mid}$, and $fail_{max}$ from the failure range of the original report. Furthermore, since the detection algorithms depend on the viewport immediately wider than the failure range to infer the failure, $fail_{max}+1$ is added to the sub-RLG. The narrower viewport, $fail_{min}-1$, is included when repairing a small-range failure because it is required by the detection algorithm of this type of failure. If the original failure from the original RLG is not reported by the detection algorithms in the sub-RLG, the patch is accepted, saved, and reported as a successful repair by LAYOUT DR.

The patch sourcing, patch application, and repair assessment processes of LAYOUT DR are described in the PATCH() procedure of Algorithm 8. As input, the algorithm expects the HTML elements, type, and range of the failure along with the intended patch source-viewport. The algorithm first calculates the $fail_{min}$, $fail_{mid}$, and $fail_{max}$ of the failure in lines 2 to 4 and creates a set from these viewports in line 5. The wider source-viewport and narrower source-viewport are calculated next in lines 6 and 7. Then the GETCSS() procedure of line 8 sets the viewport width to the source viewport and extracts the applied CSS of all DOM elements. Then, the ADDMEDIARULES() procedure of line 9 adds

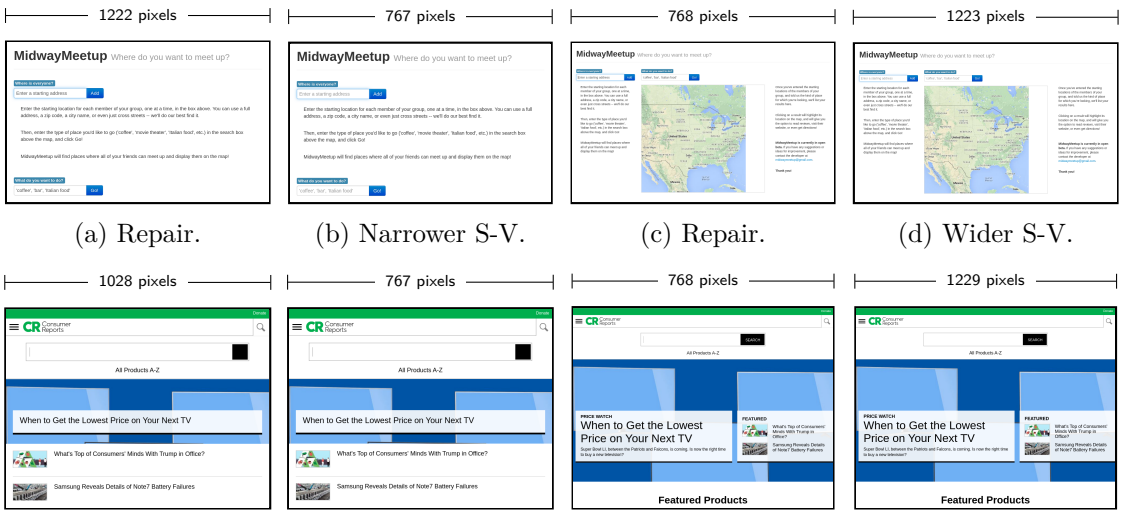
Algorithm 8 Patch Presentation Failures

INPUT: Failure *elements*, *range*, *type*, and patch *sourceViewport*.

```
 1: procedure PATCH(elements, range, type, sourceViewport)
 2:     Fmin ← MIN(range)                                          ▷ fail_min
 3:     Fmid ← FLOOR((MIN(range) + MAX(range))/2)                  ▷ fail_mid
 4:     Fmax ← MAX(range)                                          ▷ fail_max
 5:     F ← {Fmin, Fmid, Fmax}                  ▷ viewports covering failure range
 6:     nv ← Fmin − 1                                              ▷ narrower viewport
 7:     wv ← Fmax + 1                                              ▷ wider viewport
 8:     css ← GETCSS(sourceViewport)
 9:     patch ← ADDMEDIARULES(sourceViewport, css)
10:     patchHandle ← INJECT(patch)
11:     if ISFAILINGAT(Fmin, elements, type)) = True then
12:         SAVE(patch, 'Failed')
13:     else
14:         V ← F ∪ wv                              ▷ viewports covered by sub-RLG
15:         if type = SmallRange then
16:             V ← V ∪ nv                                         ▷ add narrower viewport
17:         subRLG ← CAPTURERLG(V)
18:         failures ← DETECTFAILURES(subRLG)
19:         if {elements, type} ∈ failures then
20:             SAVE(patch, 'Failed')
21:         else
22:             SAVE(patch, 'Successful')
23:     EJECT(patchHandle)
24:     return
```

media rules to the patch in order to cover the failure range and scales the layout to the size of the viewports in the failure range. The patch is then injected into the page in line 10. If the patch is rejected using the ISFAILING() procedure of line 11 which used the lower bound viewport of the failure, the patch is saved in line 12 and marked as a failed attempt. Otherwise, the wider source-viewport and narrower source-viewport are added to the set of failures as needed for the sub-RLG and noted in lines 14 to 16. The CAPTURERLG() procedure of line 17 uses this set of viewports to create the sub-RLG and the DETECTFAILURES() procedure of line 18 detects the failures in the sub-RLG. If the same failure is found in the sub-RLG then the patch failed to repair the web page. Otherwise, LAYOUT DR has automatically generated a successful patch that repaired the presentation failure. In either case, the patch is saved and marked appropriately. Finally, the patch is ejected from the page in line 23.

The resulting layout of a repair is indistinguishable from the source-viewport layout where the patch was created from. To showcase the similarity between the repaired layout and the source-viewport layout, Figure 5.4 displays adjacent snapshots of both layouts. The repair snapshots, in parts (a), (c), (e), and (g) of the figure, are from the MidwayMeetup and ConsumerReports subjects presented earlier in Figure 5.2. These two failures were chosen from the earlier example of four because they had longer failure ranges than the other two. The failure from MidwayMeetup had a range of 768-1222 pixels and the failure from ConsumerReports had a range of 768-1028 pixels. Despite the length of the ranges being over 450 viewports, the scaled layouts are still identical to the source-viewport layout. Therefore, the relative layout of elements in the webpage is preserved as the automated repair assessment of LAYOUT DR indicated. It is worthy to notice that the snapshots of the repaired subjects were captured at the most extreme distant point in



(a) Repair. (b) Narrower S-V. (c) Repair. (d) Wider S-V.

(e) Repair. (f) Narrower S-V. (g) Repair. (h) Wider S-V.

Figure 5.4: Snapshots of four successful repairs compared with snapshots of their source-viewport, denoted as S-V, which is the layout that was used to build the patch. The repairs are of the presentation failures from the MidwayMeetup and the ConsumerReports showcased in Figure 5.2. Here the repair is compared to the source-viewport side by side.

the range away from the source-viewport.

5.4 Empirical Evaluation

This section evaluates the effectiveness and efficiency of the repair approach implemented in LAYOUT DR and presented in this chapter. I carried out experiments over 31 responsively designed web pages to answer the following research questions:

Research Question One – *Can* LAYOUT DR *repair any presentation failure detected in the subjects using the narrower or wider source-viewport?* To answer this question, I ran LAYOUT DR's to detect and patch any presentation failure found. Using LAYOUT DR's automated DOM and RLG repair assessment (see Section 5.3.4), the results are analysed.

Research Question Two – (a) *How many of the automated repairs are free from other presentation failures?* (b) *Do the narrower or wider source-viewport patches introduce new presentation failures in the web page?* To answer part (a) of this question, I manually investigate a subset of presentation failures repaired by LAYOUT DR to see if they copy other failures detected by the tool. To answer part (b), I ran LAYOUT DR setting it to create a patch from the largest viewport in the testing range $\{test_{min}..test_{max}\}$ and apply to all other viewports $\{test_{min}..test_{min}-1\}$ to reach a conclusion about the wider source-viewport. In a second run, I set it to create a patch from the smallest viewport and apply it to $\{test_{max}+1..test_{max}\}$ to decide if the narrower source-viewport introduces any new presentation failures that are detectable by LAYOUT DR.

Research Question Three – *Would a human prefer the repaired web page based on the narrower or wider source-viewport over the unpatched web page that contains the presentation failure?* To answer this question, I presented a subset of the detected failures

along with their associated repairs to participants in a human study to identify human preferences. Presenting them with the original web page manifesting the failure and both the automatically repaired versions by LAYOUT DR.

Research Question Four – *How long does* LAYOUT DR *take to detect and repair presentation failures?* To answer this question, I recorded the time the tool takes to collect data from the web page to build the RLG and detect failures, and the time it takes to repair the failures detected in each subject.

The design of the experiments set forth to answer these research questions are explained next.

5.4.1 Design of Experiments

In this section, I will identify the subject web pages used in the experiments and their details, the runtime environment used to build LAYOUT DR and used to run the tool during the experiments, the methodology followed to answer each of the research questions, and finally disclose any known threats to the validity of the results and any mitigating steps taken to reduce these threats.

Subject Web Pages

In the experiments of this chapter, a total of 31 web pages were used to evaluate LAYOUT DR converged from two sources. First, I enrolled the 21 subjects used by Walsh et al. [124] to evaluate REDECHECK, the predecessor tool that the detection algorithms of LAYOUT DR were adopted from. Then, I added 10 new subjects not used in previous experiments. The details of all the subjects used in the experiments of this chapter are listed in Table 5.1.

The re-employed 21 web pages were previously selected in a random fashion by Walsh et al. using the `randomusefulwebsites.com` website which has recently changed to `discuvver.com`. I downloaded these subjects and used them without modification from the repository cited in their paper `github.com/redecheck/example-webpages`. Although the repository contains 26 web pages, five were excluded from the experiments for two reasons. First, I did not use the *StumbleUpon* subject because the downloaded web page was not loading successfully. This is due to online resources that are requested by the page but are no longer available online. This is because the original web page does not exist and the resources were not saved offline. The other four subjects, *ZeroDollarMovies*, *RunPee*, *RainyMood*, and *Mailinator*, were excluded because they did not report any presentation failures during preliminary experimentation.

For the 10 subjects newly accrued for the experiments of this chapter, I set out to collect web pages related to the topic of software. Initially, I used the `google.com` search engine with the keywords *open-source software* to find subjects to use in the experiment. To access posts on the internet that had multiple links to open-source software web pages, I prefixed the keyword *top* to reach more candidate web pages. Although I did find five web pages containing presentation failures using this approach, the lists of "top open-source software" were limited and largely repetitive using this approach. Therefore, I migrated to searching `github.com/search` for candidates using the keywords *HTML*, *JavaScript*, *Python*, and *Java* alternatively. After sorting the results using "Most Stars", I combed the results of the first 20 pages for a web page linked to the repositories.

As part of the criteria for accepting a subject web page into the empirical study is that it must be responsively designed, use English for display, and contain a presentation

Table 5.1: The details of the web pages used in the experiments of this chapter.

Web Site Name	URL	Number of HTML Elements	Number of CSS Declarations
3MinuteJournal	3minutejournal.com	80	5499
AccountKiller	accountkiller.com/en	344	4691
AirBnb	airbnb.com	1470	9890
Ardour	ardour.org	222	3774
Bottender	bottender.js.org	243	2202
Bower	bower.io	370	844
BugMeNot	bugmenot.com	42	658
CloudConvert	cloudconvert.com	908	6731
ConsumerReports	consumerreports.org	1042	8005
CoveredCalendar	coveredcalendar.com	148	8414
DaysOld	daysold.com	66	2930
Dictation	dictation.io	195	8271
Django	djangoproject.com	242	4732
DjangoRest	django-rest-framework.org	610	3787
Duolingo	duolingo.com	856	4260
ElasticSearch	elastic.co/elasticsearch	1243	21467
Honey	joinhoney.com/install	461	7903
HotelWiFiTest	hotelwifitest.com	359	6746
MantisBT	mantisbt.org	247	7731
MarkText	marktext.app	560	1890
MidwayMeetup	midwaymeetup.com	86	4147
Ninite	ninite.com	642	4213
OrchardCore	orchardcore.net	234	6352
PDFescape	pdfescape.com	179	1954
PepFeed	pepfeed.com	343	7276
Pocket	getpocket.com	664	6607
Selenium	selenium.dev	286	4980
TopDocumentary	topdocumentaryfilms.com	411	1501
UserSearch	usersearch.org	866	3900
WhatShouldIReadNext	whatshouldireadnext.com/search	112	2314
WillMyPhoneWork	willmyphonework.net	782	6576
Total		14313	170245

failure detectable by Layout DR. Moreover, the web page must be successfully copied offline using the GNU Wget tool to mitigate against unexpected changes in the content between experiments. I verified this by manually ensuring that the content loaded using the downloaded copy resembles that of the online web page. Loading of advertisement banners was not part of the criteria and an empty element was acceptable. Another criterion is that the candidate must not have any continuous visual effects (e.g. moving elements across the page in a loop). In other words, the state of the web page, specifically the element relative positions, must be stable. The detection algorithms do not support this feature in web pages and enhancing the algorithms to support it in Layout DR is out of scope for the experiments of this chapter. Finally, I did not use any links that lead to personal social media web pages or that lead to corporate sponsors. To make it easier for others to reproduce the findings of my experiments, the newly acquired subjects used in this study are available at `github.com/ResponsiveRepair/webpages`.

Runtime Environment

To evaluate Layout DR, the experiments were done on a laptop with 16GB RAM, 1 TB SSD, Intel Core i7-4720HQ, and running Ubuntu version 20.04.2 64-bit operating system. The GNU Wget version 1.20.3 was used to download all candidate web pages for the experiments. The laptop had Node version 14.15.4 with NPM version 6.14.10 installed during the experiments. Layout DR used Puppeteer version 4.0.1 to control the browser. Puppeteer is a Node library that provides an API to control the browser. The Chromium browser packages with Puppeteer was used in the experiments. Moreover, the browser was configured to run in headless mode with a fixed viewport height of 1000 pixels.

Methodology

This section presents the methodology followed to answer the research questions proposed in this chapter.

RQ1 Methodology – To answer RQ1, I ran LAYOUT DR on all 31 subject web pages using the testing range 320-1400 pixels. Then, using the automated DOM and RLG repair assessment of the tool (see Section 5.3.4 for full details), the results are disclosed and analysed for both the narrower source-viewport and wider source-viewport based repairs.

Since many of the reports raised by LAYOUT DR are only true at the DOM level and do not equate to a visible problem on the web page that would justify the repair, I manually categorized the severity of all 398 reported failures after examining the minimum viewport of the failure range. The intention being, to add insight into the evaluation of LAYOUT DR repairs based on the severity of the layout problem. To do this, I categorized each report as either *dismissible*, *disputable*, or a *definite* failure by manually analysing the web page and the snapshots made by LAYOUT DR. Here, the *dismissible* category refers to the DOM only failures presented in the first two chapters of this thesis as "non-observable issues" while the *disputable* label refers to the other visible but contentious issues that may not require a repair. All other failures that showed a visible problem in the layout that are most likely to warrant a repair, are labelled as *definite* failure reports. All of the manual categorizations are available at `responsiverepair.github.io/manual/rlf-classifications.html` with highlighted borders around the failing elements.

RQ2 (a) Methodology – To answer RQ2 (a), I investigated the set of *definite* failures for possible copying of other presentation failures that manifest the source-viewport used in the repair. First, I identified the other failures, in the same set, that coincided with a source-viewport used in a repair. Since the severity of the failure may change depending on the viewport chosen from the failure range for examination, it does not necessarily follow that the coincidence of source-viewport with failure ranges that a repair "copied" a definite failure. This was observed in the findings of the first two chapters of this thesis, where the classification of the failure changed depending on the viewport used. Therefore, I manually investigated the web page at the source-viewport to see if the cross-over failures are also *definite* at that viewport. If it is, then the repair is not *failure-free* for that specific source-viewport. Otherwise, it is a *failure-free* repair. I did this for both the narrower and wider source-viewport that were used in a repair made by LAYOUT DR.

RQ2 (b) Methodology – To answer RQ2 (b), I ran the LAYOUT DR tool using a specialized mode for this research question. Without the need for a failure report, the specialized mode uses a viewport from the testing range used in the experiments in this chapter, 320 to 1400 pixels, to "repair" (patch) all other viewports. First, to investigate the wider source-viewport, the tool uses the 1400 pixels viewport as a source-viewport and applies it in a patch to the range of 320 to 1399 pixels. Followed by a second run that uses the 320 pixels viewport as source-viewport to patch the range of 321 to 1400 in order to investigate the narrower source-viewport. For each run, the detection phase of the tool is run to report any failures after applying the patch to each of the 31 subjects. If the patch does not introduce a failure, there should be no presentation failure reported by the tool. This is because the relative position of all elements remains consistent throughout the testing range 320 to 1400 pixels during both runs.

RQ3 Methodology – To answer RQ3, I employed a human study to record and evaluate the preference of participants when presented with a choice between the web page with the failure against one of two alternative repairs made by LAYOUT DR.

For the human study, I utilized the services of Mechanical Turk at `mtruk.com` to reach participants willing to answer the questionnaire for compensation. Sensibly, the experiments that require a distinction to be made should focus on the reports labelled as *definite* failures. This is to facilitate proper judgement of the repairs against only the failures that are apparent in the webpage and therefore require a repair. Moreover, to showcase both the narrower and wider source-viewport based repairs, I used all 20 definite failures for which LAYOUT DR produced two alternative repairs. To limit the time each participant spends in the study, only a total of 10 randomly selected and randomly ordered failures are presented to each participant. The questionnaire was hosted on an external web page that I hosted and linked to the services of Mechanical Turk. You can see the web page as seen by the participants in Figure 5.5.

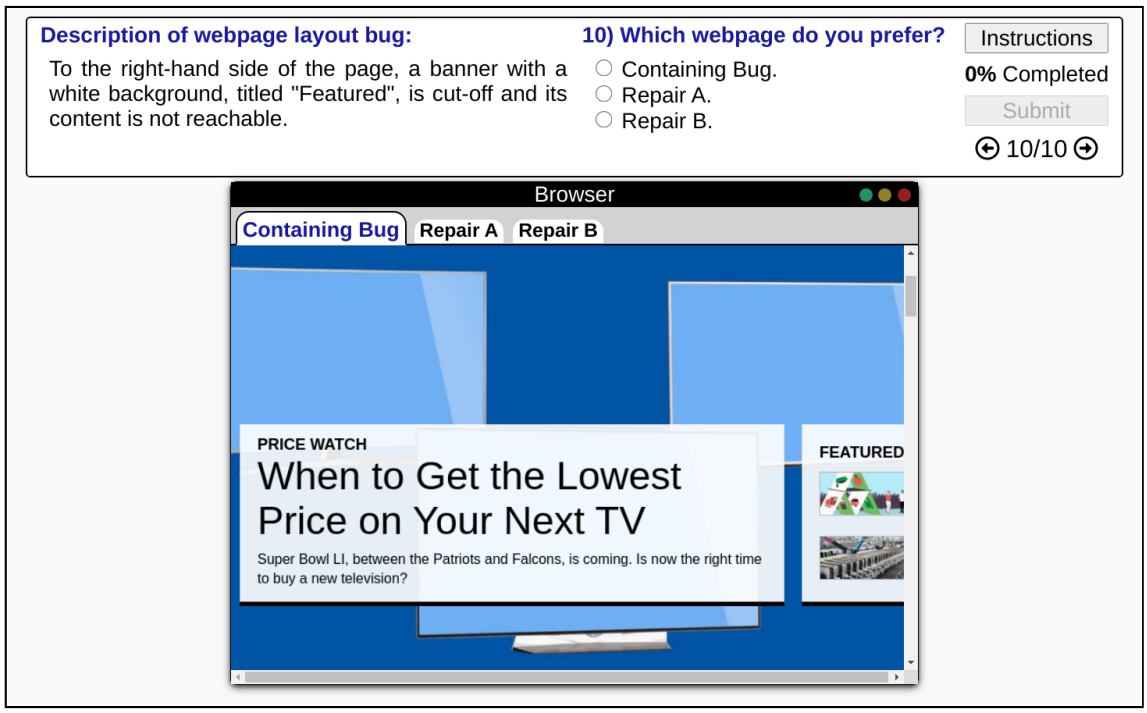


Figure 5.5: The user interface of the webpage used as a medium for the human study showcasing a presentation failure from the ConsumerReports subject.

As clear in Figure 5.5, each section presents the participant with a mock browser with three tabs. These tabs are labelled as *Containing Bug*, *Repair A*, and *Repair B*. The *Containing Bug* tab is the default initial tab presented to the participant to showcase a snapshot of the web page that contains the failure. While the *Repair A* and *Repair B* tabs present the participant, once they are clicked, with either the narrower source-viewport repair or the wider source-viewport repair depending on the result of a random assignment for each participant. All the snapshots presented to the participant captured the entire web page and were never explicitly scaled for the study. Nevertheless, due to the height limitation caused by presenting a mock browser in a real browser and to leave enough space for a question and answer at any one time, the participant is presented with 500 pixels in height of the snapshot. This amounts to half the size of the browser height used in the experiments of this chapter. Meanwhile, the viewport width of the mock browser matches that of the browser during the experiments. Importantly, the mock browser allows the user to scroll freely to investigate other parts of the web page but the scroll position is initially set to the location of the presentation failure within the web page.

Above the mock browser, the participant is presented with a short text-based descrip-

tion of the presentation failure. The description was mostly a single sentence pointing out the position of the failure within the mock browser and briefly describing the layout failure. In the same area above the mock browser, I asked the participant *Which web page do you prefer?*. Using three radio buttons, the participant is forced to choose between *Containing Bug*, *Repair A*, or *Repair B* before they can proceed to the next section. Furthermore, the same area provides the participant with a button labelled “Instructions" that will lead them to the instructions that were presented to them at the start of the study. In the instructions page, the participant is instructed to read the description of the bug, identify the bug in the web page, and compare the three tabs before choosing a preference. Finally, above the mock browser, the participant has another button labelled “Submit" which is only enabled after all questions are answered.

I released the questionnaire to participants on Mechanical Turk over two batches. An initial batch of 25 jobs was released for participants and a second batch of 75 was later released after making sure there were no errors in the initial batch. From which, there were a total of 101 responses with 98 unique participant identifiers submitted. Each participant was paid $1 US dollar for their participation.

As part of controlling the quality of the data from the questionnaire, I added code in the webpage to monitor the number of clicks on each of the three tabs *Containing Bug*, *Repair A*, and *Repair B*. Furthermore, I stored the loading event of each `img` element containing the snapshot of each tab. To control against participants that did not follow instructions before voting, I filtered the results to use only the votes of participants that clicked on each of the tabs at least once (the default display of the *Containing Bug* tab is automatically counted as one-click) and that the loading event for all three tabs was fired.

RQ4 Methodology – To answer RQ4, I ran Layout DR using the entire set of 31 subjects to extract the DOM data and build the RLG, detect failures using the RLG, generate patches, and assess the success or failure of a repair. To obtain a reliable estimate of Layout DR's running time, I repeated this process 10 times for each subject and recorded the time taken for Layout DR to run in each instance. This is to minimize the chance of interference from other underlying operating system operations on the recorded time. To allow HTML elements of a web page to load and their transitional effects to settle into their final position, Layout DR's runtime includes a 400 millisecond added delay. This delay time is repeated whenever Layout DR resizes the viewport width, scrolls the web page, applies a patch, or removes the patch. The chosen value for the delay was determined in preliminary experimentation where I incremented the value until the output of all web pages were deemed deterministic.

Threats to Validity

The process of identifying the subset of reports that have a real or *definite* presentation failures may pose a threat to validity. Since the detection algorithms rely on the DOM, manually inspecting the web page for the potential failure is a necessary process in order to find the failures that are significant enough to require a repair. It is critical to identify this set of real failures in order to study the effect of the patch on altering the visible problem in the layout. Although the source-viewport repair technique was applied to all reported failures to measure its ability to repair any DOM issue, I only used the real failures that are suitable for a human to study what is an acceptance or unacceptable repair. For independent analysis, I made the snapshots and the manual categorization available at `responsiverepair.github.io/manual/rlf-classifications.html`.

Regarding the participants of the human study, multiple threats were considered and

mitigated. First, since I used the crowdsourcing service `MTurk.com` to conduct the study, this minimizes any intentional selection bias because the service provides a large pool of anonymous participants. On the other hand, using any crowdsourcing service could create a threat that arises if participants care more about meeting a higher throughput of completed questionnaires instead of giving the questionnaire the time it deserves to answer the questionnaire genuinely. To mitigate against this, criteria provided by Amazon Mechanical Turk were set to limit participation to people who have completed at least 500 other questionnaires with 90% approved for payment record. Furthermore, I discarded any unauthentic votes that occurred without clicking on all options and did not wait for all images to load.

Another threat surrounding the medium of the human study is the unknown size of the browser being used to complete the questionnaire. To ensure that a participant is using a device with a browser big enough size to embody the questions of the questionnaire and the associated images as originally designed, I set clear instructions that only participants using a large screen of laptops or desktops are allowed to participate. Moreover, I set rules within the web page to display an error message for viewports less than 1400 pixels in width and 768 pixels in height thereby preventing them from proceeding to the questionnaire. Finally, to prevent any inconsistently in displaying the presentation failure and its repair between different users who may be using different browsers, I used snapshots of the subject web page instead of the live page ensuring all participants compare the same layouts. Although the participant cannot interact with the subject webpage, the images showcase the entire layout of the web page without any explicit scaling. Moreover, automated scrolling is used to reach and thus display the area where the presentation failure (or its repair) is located but the participant is free to scroll through the entire page if they desire to.

As with many experiments, one threat to validity is the generalizability of the results to other subjects. Although the number of web pages on the internet is extremely large, the 31 subjects used in the experiments of this chapter were randomly gathered and have varying properties. From these properties, the most fundamental are the number of HTML elements that ranged from 42 to 1,470 and the number of CSS declarations that ranged from 658 to 21,467 declarations. Furthermore, the motivation or objective of the published web pages also varied. They included a browser automation tools (Selenium), an audio editing software (Ardour), and an older versions of an educational platform (Duolingo), and a lodging service (AirBnb). Although the sample is relatively small, the inherent diversity in random selection could mitigate against this threat.

One validity of threat to the experiments of this chapter is the proper re-implementation of ReDeCheck's failure detection algorithms. Along with the using the algorithms from Walsh et al. [124], I used the latest version of the tool for fine-grained guidance. Nevertheless, some differences are to be expected because I made improvements where I found bugs with the previous tool and when I reached edge cases using the newer subjects. Further discrepancies may arise due to the tools using two different browsers. While ReDeCheck relied on Firefox, the Layout DR tool uses the Chromium browser to detect and repair presentation failures. To reduce this threat, I used automated tests while developing Layout DR as well as manually inspected all the reported presentation failures.

Finally, there is the threat of properly implementing the repair technique into the Layout DR tool. To reduce the threat of bugs in the tool, I used unit tests while developing the tool and manually inspected the results and thereafter made improvements where needed. Furthermore, I made the tool publicly available for others to use and replicate the finding of this chapter. All of the subjects, snapshots, and the tool are available in repositories at `layoutdr.github.io`.

Table 5.2: The results of Layout DR attempting to repair all 398 presentation failures detected using patches created from the narrower and wider source-viewport. The data points are derived from Layout DR's fully automated assessment process based on the DOM and an RLG of the patched subject.

		Narrower Source-Viewport									Wider Source-Viewport								
		Repaired			Not Applicable			Failed			Repaired			Not Applicable			Failed		
Subject	**Failures**	**Definite**	**Disputable**	**Dismissible**	**Definite**	**Disputable**	**Dismissible**	**Definite**	**Disputable**	**Dismissible**	**Definite**	**Disputable**	**Dismissible**	**Definite**	**Disputable**	**Dismissible**	**Definite**	**Disputable**	**Dismissible**
3MinuteJournal	13	2	-	1	2	-	8	-	-	-	4	-	9	-	-	-	-	-	-
AccountKiller	44	-	-	38	-	2	4	-	-	-	-	2	42	-	-	-	-	-	-
AirBnb	9	-	-	4	-	1	4	-	-	-	-	1	8	-	-	-	-	-	-
Ardour	19	-	-	17	2	-	-	-	-	-	2	-	17	-	-	-	-	-	-
Bottender	9	-	-	-	5	-	4	-	-	-	5	-	4	-	-	-	-	-	-
Bower	3	1	-	-	-	-	2	-	-	-	1	-	2	-	-	-	-	-	-
BugMeNot	7	-	-	-	1	2	4	-	-	-	1	2	4	-	-	-	-	-	-
CloudConvert	1	-	-	1	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-
ConsumerReports	22	2	1	2	5	-	12	-	-	-	7	1	14	-	-	-	-	-	-
CoveredCalendar	7	-	-	6	-	-	1	-	-	-	-	-	7	-	-	-	-	-	-
DaysOld	1	-	-	-	-	-	1	-	-	-	-	-	1	-	-	-	-	-	-
Dictation	2	-	-	-	-	-	2	-	-	-	-	-	2	-	-	-	-	-	-
Django	4	-	-	-	1	-	3	-	-	-	1	-	3	-	-	-	-	-	-
DjangoRest	2	1	-	1	-	-	-	-	-	-	1	-	1	-	-	-	-	-	-
Duolingo	12	1	-	5	-	-	6	-	-	-	1	-	11	-	-	-	-	-	-
ElasticSearch	20	1	2	9	1	4	3	-	-	-	2	6	12	-	-	-	-	-	-
Honey	14	1	-	3	-	-	8	-	1	1	1	1	12	-	-	-	-	-	-
HotelWiFiTest	1	1	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-
MantisBT	15	2	-	6	1	-	6	-	-	-	3	-	12	-	-	-	-	-	-
MarkText	94	3	1	35	12	8	31	-	-	4	15	9	70	-	-	-	-	-	-
MidwayMeetup	13	1	1	2	-	2	7	-	-	-	1	3	9	-	-	-	-	-	-
Ninite	1	-	-	-	-	-	1	-	-	-	-	-	1	-	-	-	-	-	-
OrchardCore	18	-	2	5	5	-	6	-	-	-	5	2	11	-	-	-	-	-	-
PDFescape	6	-	1	2	-	-	3	-	-	-	-	1	5	-	-	-	-	-	-
PepFeed	6	1	-	4	-	-	1	-	-	-	1	-	5	-	-	-	-	-	-
Pocket	8	-	-	6	-	-	2	-	-	-	-	-	8	-	-	-	-	-	-
Selenium	15	1	-	-	-	-	14	-	-	-	1	-	14	-	-	-	-	-	-
TopDocumentary	12	-	-	11	-	-	-	-	-	1	-	-	12	-	-	-	-	-	-
UserSearch	4	-	-	-	-	1	3	-	-	-	-	1	3	-	-	-	-	-	-
WhatShouldIReadNext	6	-	-	2	-	1	3	-	-	-	-	1	5	-	-	-	-	-	-
WillMyPhoneWork	10	2	-	3	-	-	5	-	-	-	2	-	8	-	-	-	-	-	-
Total Failures	398	20	8	163	35	21	144	-	1	6	55	30	313	-	-	-	-	-	-
		191			200			7			398			-			-		

5.4.2 Results of Experiments

Answer to RQ1 – Table 5.2 gives the results of running Layout DR to detect presentation failures, generate up to two patches for each failure, and to automatically assess the repair over 31 web pages. The first numerical column discloses the number of presentation failures detected in each subject totalling 398 reports. The table then separates the repairs over multiple columns based on where the patch was sourced from. This can be either the narrower source-viewport corresponding to $fail_{min}-1$ or the wider source-viewport corresponding to $fail_{max}+1$ of the reported failure range. Each repair is then further divided based on the results of the automated assessment feature of the tool as either *Repaired*, *Not Applicable*, or *Failed*. Finally, for added clarity, I describe each failure report as either a *definite*, *disputable*, or as a *dismissible* report depending on the severity of its impact on the visible layout. It is important to note that each failure is repaired in isolation from other failures and hence the numbers in the table do not measure the compounding effect of multiple repairs. Nevertheless, a *repair* listed in the table means the removal of the failure based on the DOM and the sub-RLG of a patched web page.

The results showed that for over half of the presentation failures, 200 out of 398, the narrower source-viewport based patches were not applicable. Of which, 112 reports had 320 pixels as the lower bound of the failure range. Since the testing range of the experiments was 320 to 1400 pixels, the tool cannot use 319 pixels as a narrower source-viewport for repair. For the other 88 non-applicable repairs, the DOM structure at the narrower

source-viewport still contained the failure. In this case, LAYOUT DR automatically knows not to apply the narrower source-viewport repair because it will not repair the failure.

This left 198 failure reports where the narrower source-viewport repair is deemed by the tool as applicable. From which, 191 failures were successfully repaired by the tool bringing the results to a near full success rate. Only 7 failed the tool's automated assessment of repair, *i.e.* the elimination of the failure report from the DOM and the RLG of the patched subject. I investigated these failures from the Honey, MarkText, and the TopDocumentary subjects and found that the DOM positions of elements were only fractions of a pixel off. In other words, the repairs 7 were actually successful apart from being fractions of a pixel away from its expected repaired position. Nevertheless, this makes the narrower source-viewport at least 96.46% successful at repairing failures when it is applicable.

The wider source-viewport based repairs were more straight forward with it being always an applicable source-viewport and never failed. All 398 failure reports were successfully repaired by LAYOUT DR using it as a source-viewport. These can be divided into 313 dismissible, 30 disputable, and 55 definite failure reports repaired.

Conclusion for RQ1 – Using the automated assessment of repairs method that utilizes the DOM and RLG, LAYOUT DR demonstrated a 100% success rate using the wider source-viewport repairs and conservatively had a 96.46% success rate using the narrower source-viewport for repairs after discounting the 50.25% that were non-applicable.

Answer to RQ2 (a) – After manually inspecting each repair of the 55 definite failures in order to determine if the repair is *failure-free*, the results are described in Table 5.3. In the table, a check-mark symbol (✓) indicates a repair that is failure-free while a times symbol (×) contrarily indicates a repair that has copied another presentation failure into the repair. meanwhile, a dash symbol (-) indicates no repair was attempted by the tool. The table showcases the findings of the narrower source-viewport based repairs in the column labelled “Narrower" and that of the wider source-viewport in the column titled “Wider". Moreover, the “Both" column indicates where both repairs were found to be failure-free and the “Either" column indicates when at least one repair was failure-free.

Overall, 40 out of 55 repairs had at least one that is failure-free, as in, the repair did not copy any other failure from the set of definite failures. On a per source-viewport bases, the narrower had 18 out of 20 that are failure-free while the wider had 38 out of 55 repairs. The current version of LAYOUT DR does not actively avoid source-viewports where a failure is detected, but there is good reason for future versions to do so. This is not as simple as checking for cross-over of other failure ranges. Since a report may be disputable or dismissible, the existence of the failure at the source-viewport does not mean it cannot be used in a repair. Furthermore, even a definite failure at some viewport in its failure range may be usable as a source-viewport. This is because the classification or severity of the failure may change throughout its range as seen in the finding of the first chapter of this thesis.

Answer to RQ2 (b) – The Table 5.4 lists all the failures introduced by LAYOUT DR after running the tool twice to simulate a repair over all viewports other than the source-viewports of 320 and 1400 pixels. Since there were no failures introduced by LAYOUT DR when using the narrower source-viewport of 320 pixels, the table contains only the two failures reported from the 31 subjects when using the wider source-viewport repair of 1400 pixels. Looking at the report from the PepFeed subject first, it was an element protrusion

Table 5.3: Listing all 55 *definite* presentation failures detected by LAYOUT DR, to identify the repairs that are free from other failures detected by the tool, with ✓, and the ones that copied another failure with ×, while the - indicates no repair made by the tool. The column labelled "Both" showcases where the tool provided both a narrower and wider source-viewport failure-free repair while the "Either" column indicates at least one is failure-free.

						Failure-Free Repair			
ID	**Web Site Name**	**Type**	**Failure Range**			**Narrower**	**Wider**	**Both**	**Either**
8	3MinuteJournal	Element Protrusion	992	to	1199	✓	✓	✓	✓
9	3MinuteJournal	Element Protrusion	347	to	583	-	✓	-	✓
11	3MinuteJournal	Viewport Protrusion	320	to	568	-	✓	-	✓
12	3MinuteJournal	Viewport Protrusion	992	to	1137	✓	✓	✓	✓
11	Ardour	Viewport Protrusion	320	to	658	-	✓	-	✓
12	Ardour	Element Protrusion	659	to	678	-	✓	-	✓
1	Bottender	Viewport Protrusion	320	to	349	-	×	-	×
2	Bottender	Viewport Protrusion	320	to	426	-	×	-	×
4	Bottender	Viewport Protrusion	320	to	718	-	✓	-	✓
7	Bottender	Element Protrusion	566	to	608	-	×	-	×
8	Bottender	Viewport Protrusion	320	to	470	-	×	-	×
1	Bower	Viewport Protrusion	681	to	697	✓	✓	✓	✓
2	BugMeNot	Element Protrusion	324	to	671	-	✓	-	✓
1	ConsumerReports	Viewport Protrusion	320	to	372	-	✓	-	✓
5	ConsumerReports	Viewport Protrusion	320	to	373	-	✓	-	✓
6	ConsumerReports	Viewport Protrusion	320	to	372	-	✓	-	✓
8	ConsumerReports	Viewport Protrusion	1025	to	1101	✓	✓	✓	✓
9	ConsumerReports	Viewport Protrusion	320	to	372	-	✓	-	✓
10	ConsumerReports	Viewport Protrusion	320	to	373	-	✓	-	✓
21	ConsumerReports	Element Protrusion	768	to	1028	✓	✓	✓	✓
4	Django	Element Wrapping	768	to	778	-	✓	-	✓
2	DjangoRest	Viewport Protrusion	767	to	767	✓	✓	✓	✓
2	Duolingo	Viewport Protrusion	981	to	1098	✓	✓	✓	✓
3	ElasticSearch	Element Collision	768	to	768	✓	✓	✓	✓
15	ElasticSearch	Element Protrusion	320	to	554	-	✓	-	✓
13	Honey	Element Collision	539	to	544	✓	✓	✓	✓
1	HotelWiFiTest	Viewport Protrusion	415	to	766	✓	✓	✓	✓
11	MantisBT	Element Wrapping	768	to	991	✓	✓	✓	✓
12	MantisBT	Element Wrapping	320	to	378	-	✓	-	✓
14	MantisBT	Element Wrapping	980	to	991	×	✓	×	✓
16	MarkText	Element Protrusion	882	to	997	-	×	-	×
32	MarkText	Viewport Protrusion	769	to	1246	✓	×	×	✓
33	MarkText	Element Protrusion	522	to	552	×	×	×	×
34	MarkText	Element Protrusion	320	to	515	-	×	-	×
35	MarkText	Element Collision	320	to	419	-	×	-	×
39	MarkText	Viewport Protrusion	769	to	1198	✓	×	×	✓
48	MarkText	Element Protrusion	985	to	1090	-	×	-	×
49	MarkText	Element Collision	882	to	1089	-	×	-	×
50	MarkText	Element Protrusion	882	to	908	-	×	-	×
82	MarkText	Viewport Protrusion	320	to	428	-	×	-	×
86	MarkText	Element Protrusion	1039	to	1356	-	✓	-	✓
87	MarkText	Element Protrusion	320	to	575	-	✓	-	✓
89	MarkText	Element Protrusion	852	to	860	-	×	-	×
90	MarkText	Element Protrusion	874	to	882	-	×	-	×
91	MarkText	Element Protrusion	896	to	904	-	×	-	×
10	MidwayMeetup	Element Protrusion	768	to	1222	✓	✓	✓	✓
5	OrchardCore	Element Protrusion	768	to	1199	-	✓	-	✓
8	OrchardCore	Element Protrusion	768	to	1199	-	✓	-	✓
11	OrchardCore	Element Protrusion	768	to	1199	-	✓	-	✓
15	OrchardCore	Element Protrusion	768	to	1199	-	✓	-	✓
17	OrchardCore	Element Protrusion	768	to	1199	-	✓	-	✓
1	PepFeed	Viewport Protrusion	415	to	768	✓	✓	✓	✓
15	Selenium	Element Wrapping	901	to	933	✓	✓	✓	✓
3	WillMyPhoneWork	Element Collision	990	to	991	✓	✓	✓	✓
4	WillMyPhoneWork	Small-Range	990	to	991	✓	✓	✓	✓
			Total ✓/(× + ✓)			18/20	38/55	16/20	40/55

where an element protrudes a fraction of a pixel the entire failure range. Hence the report is not a real failure nor a real DOM issue to be concerned about. This is similar to the case of the 7 failed repair attempts disclosed in the answer to the first research question. Only there, all 7 issues were related to the narrower source-viewport repair and here the issue is associated with the wider source-viewport repair.

The second failure introduced by the repair was is in the Duolingo subject. Contrary to what was seen previously, the report is of an element protruding its parent by more than one pixel. My investigation revealed that the height of the element increased in the viewport width 1399 by 380 pixels when compared to its height from the viewport 1400 pixels wide. This results in a protrusion from the bottom of the expected parent in the RLG model by a total of 380 pixels. Nevertheless, the protrusion is not visible on the page and would be categorized as a dismissible report. In other words there is nothing visibly incorrect with the repair and only a DOM level issue. The complexity of the original CSS with the imposed CSS from the repair makes it difficult to pinpoint the cause behind the increase in height for this element in this specific subject. Although the repair appears to have worked in the rendered page, it still introduced a DOM level issue which may be a cause for concern. In the future, I plan to do controlled experiments to see the effect of consecutive iterations over the detection and repair phases.

Table 5.4: The full list of presentation failures introduced by the technique after using the source-viewport of 1400 to “repair" the entire range of 320 to 1399 pixels and again using the source-viewport of 320 to repair the range 321 to 1400 pixels.

ID	Web Site Name	Type	Failure Range	Source-Viewport
1	Duolingo	Element-Protrusion	320 to 1399	1400
1	PepFeed	Element-Protrusion	1080 to 1399	1400

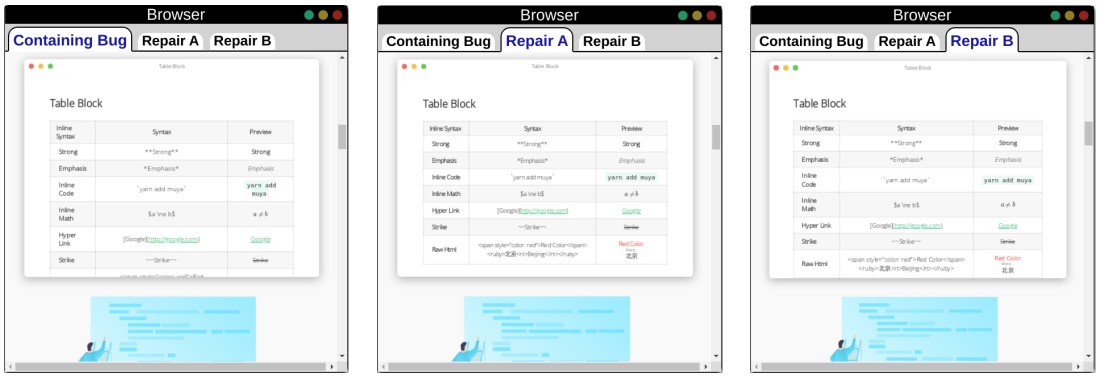
Conclusion for RQ2 (a) and (b) – For 72.72% of the definite failure reports, there was at least one failure-free repair made by Layout DR. Furthermore, the repair technique followed by the tool did not introduce any real presentation failures regardless of the source-viewport chosen for the repair but did introduce one DOM level issue.

Answer to RQ3 – Table 5.5 shows the results from participants voting for their preferred web page. The votes for the original webpage with the presentation failure are under the column labelled “Failure" while the votes for a repaired web page either go under the “Narrower" or “Wider" columns depending on the source-viewport of the repair. The last column labelled “Prefer Repair" gives the total percentage of votes that went to either of the repairs made by Layout DR. Overall, 91.87% of votes preferred a Layout DR repair across all 20 failures when comparing any of the two repaired versions against the original web page. That is, 678 votes preferred a Layout DR repair from a total of 738 votes.

Among the set of failures used in the study, the report with ID 33 from the MarkText subject had the lowest number of votes that preferred a repair at 82.23% of votes. The failure and the repaired versions presented to the participants can be seen in Figure 5.6. As seen in part (a), this report caught the bottom row of the table that starts with "Raw Html" protruding the table and missing. Meanwhile, the row and its content are clearly visible in both of the repairs generated by the tool as seen in parts (b) and (c). Furthermore, since the narrower source-viewport repair is only scaled to one viewport

Table 5.5: The results from the human preference study where each participant is forced to choose between either the original web page with the presentation failure, the narrower source-viewport repaired web page, or the wider source-viewport repaired web page.

						Scale Value		Votes				
ID	Web Site Name	Type	Failure Range			Narrower	Wider	Original	Narrower	Wider	Total	Prefer Repair
8	3-MinuteJournal	element protrusion	992	to	1199	1.001	0.827	3	7	25	35	91.43 %
12	3-MinuteJournal	viewport protrusion	992	to	1137	1.001	0.872	2	22	12	36	94.44 %
1	Bower	viewport protrusion	681	to	697	1.001	0.976	3	27	2	32	90.62 %
8	ConsumerReports	viewport protrusion	1025	to	1101	1.001	0.930	5	33	2	40	87.50 %
21	ConsumerReports	element protrusion	768	to	1028	1.001	0.746	2	9	20	31	93.55 %
2	DjangoRest	viewport protrusion	767	to	767	1.001	0.999	3	6	29	38	92.11 %
2	Duolingo	viewport protrusion	981	to	1098	1.001	0.893	4	8	25	37	89.19 %
3	ElasticSearch	element collision	768	to	768	1.001	0.999	6	9	21	36	83.33 %
13	Honey	element collision	539	to	544	1.002	0.989	3	11	27	41	92.68 %
1	HotelWifiTest	viewport protrusion	415	to	766	1.002	0.541	1	22	10	33	96.97 %
11	MantisBT	element wrapping	768	to	991	1.001	0.774	4	10	26	40	90.00 %
14	MantisBT	element wrapping	980	to	991	1.001	0.988	4	5	40	49	91.84 %
32	MarkText	viewport protrusion	769	to	1246	1.001	0.617	2	20	12	34	94.12 %
33	MarkText	element protrusion	522	to	552	1.002	0.944	6	20	8	34	82.35 %
39	MarkText	viewport protrusion	769	to	1198	1.001	0.641	0	12	21	33	100.00 %
10	MidwayMeetup	element protrusion	768	to	1222	1.001	0.628	1	15	24	40	97.50 %
1	PepFeed	viewport protrusion	415	to	768	1.002	0.540	3	20	10	33	90.91 %
15	Selenium	element wrapping	901	to	933	1.001	0.965	2	7	27	36	94.44 %
3	WillMyPhoneWork	element collision	990	to	991	1.001	0.998	5	14	22	41	87.80 %
4	WillMyPhoneWork	small-range	990	to	991	1.001	0.998	1	15	23	39	97.44 %
							Total	60	292	386	738	91.87 %

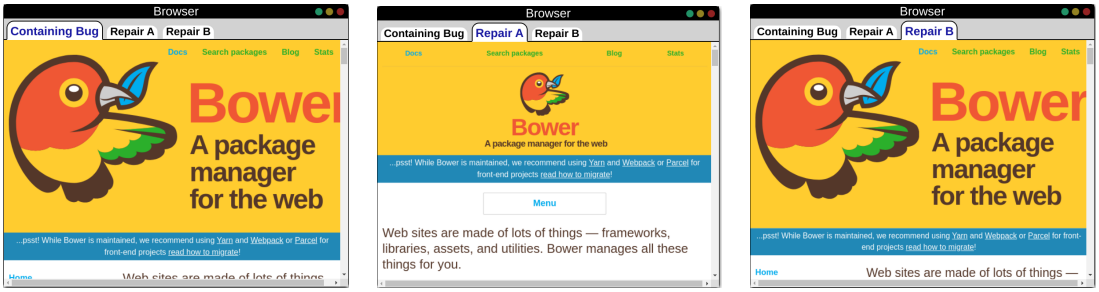


(a) Presentation Fialure. (b) Narrower source-viewport. (c) Wider source-viewport.

Figure 5.6: Three snapshots of the MarkText subject from the human study.

wider, there is no reason to attribute this to the problem of over-scaling. Hence, it is safer to assume a rushed decision or that an improper comparison was made rather than assuming the participant preferred an entire row of content missing.

The lowest number of votes, between the alternative repairs proposed by LAYOUT DR, was 2 votes for the repair from the Bower subject and 2 votes for the repair from the ConsumerReports subject both corresponding to the wider source-viewport. For these two cases, over 90% of the votes went to the narrower source-viewport viewports making it a clear winner. The reason for this low marginal of votes for the wider source-viewport can be inferred by examining the presentation failure seen in Figure 5.7(a) from the Bower subject. Here, the letter “r" from the word Bower is cut-off to the right-hand edge of the page. By looking at the repair with the low votes, seen in part (c), I put forward that the letter is still unconformably close to the edge of the page while in the narrower repair it is a safe distance away. More evidence for this is seen in the case from the ConsumerReports subject in Figures 5.8 (a) and (c). Here the link “Privacy Policy", in part (a), is missing to the left edge of the page. Meanwhile, in the repaired version seen in part (c), the link is still too close to the edge for comfort.



(a) Presentation Fialure. (b) Narrower source-viewport. (c) Wider source-viewport.

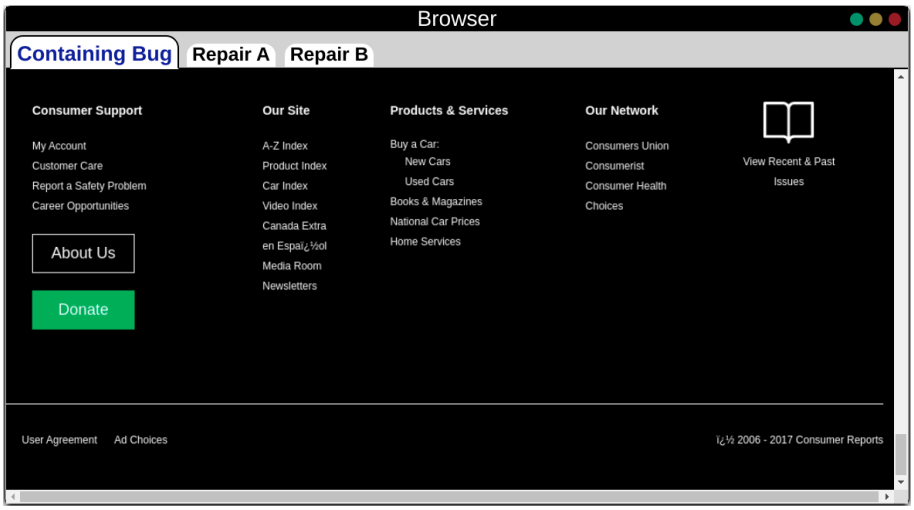
Figure 5.7: Three snapshots of the Bower subject from the human study.

This trend of the narrower repair winning more votes because the wider repair is aesthetically imperfect, even though it did repair the issue, can also be seen in the previous example from the MarkText subject. In this example, the wider repair, seen in Figure 5.6(c), has the row and its content showing but the aesthetics are not perfected. More specifically, the lower white padding of the table is missing and therefore not as aesthetically pleasing as the narrower repair. These findings suggest that either an offset should be considered when using the wider source-viewport based repairs or the detection algorithms should be improved to report better ranges that truly represent the problem aesthetically.

Overall, there were a total of 386 votes that preferred the wider source-viewport repair and 292 total votes that preferred the narrower source-viewport repair which is only a 56.93% preference. On a per failure report bases, there were 13 out of 20 failure reports where the majority of votes went to prefer the wider source-viewport repair. That is, the participants preferred the usage of the wider source-viewport repair for 65% of the failures used in the study. It is important to note that all the snapshots — of the failure and both repair options — presented to the participant were from the minimum viewport of the failure range, $fail_{min}$, for each failure. This means that the narrower source-viewport repairs were scaled one viewport up for all the failure reports in the study. On the other hand, the wider source-viewport had a more significant scaling effect for longer failure ranges as seen by the scale value presented in Table 5.5.

The scaling effect of the wider source-viewport repairs was the most extreme for three failures where the scaling of the repairs was about 39% to 46% smaller than the original layout. These reports were from HotelWiFiTest, PepFeed, and MarkText with report ID 32. For all three, the participants preferred the narrower source-viewport for good reason. After examining the web pages, I found the elements and especially the text to be uncomfortably small. This may support the intuition that scaling should be limited to sustain readability or comfort level. Although such a scaling limit is expected to vary from one layout to another, further studies are needed to investigate this effect on multiple layouts. Along with developing a scaling limit, I plan to study the simultaneous usage of both the narrower and wider source-viewport to repair a single failure in an effort to mitigate the limitation of scaling.

Conclusion for RQ3 – The results of the study showed that participants prefer a LAYOUT DR repair over the original web page with the failure 91.87% of the time. Furthermore, the participants preferred the wider source-viewport repair over the narrower source-viewport repair for 65% of the failures in the study.



(a) Presentation Failure.

(b) Narrower source-viewport.

(c) Wider source-viewport.

Figure 5.8: Three snapshots of the ConsumerReports subject from the human study.

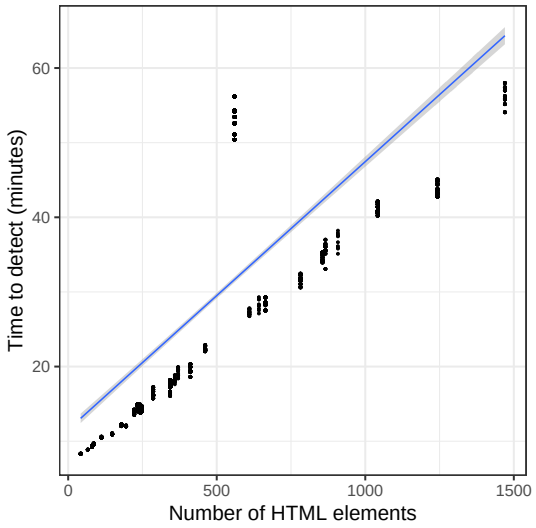
Answer to RQ4 – For any of the 31 subjects, LAYOUT DR took anywhere between 8.29 and 57.99 minutes in runtime during the detection phase with an average of 29.40 minutes. This included the time it takes to iterate over all viewports in the testing range, extract information from the DOM, build the RLG, and run the presentation failure detection algorithms on the RLG. From the experiments, the shortest runtime belonged to the BugMeNot subject which has 42 HTML elements. Given the testing range of 320 - 1400 pixels and a time delay of 400 milliseconds that is repeated for every viewport visited, there is a total of 7.20 minutes of delay per subject. Meaning, that the shortest runtime only really took about a minute to complete. Nevertheless, the longest runtime of experiments took about 50 minutes, discounting the delay. This runtime belonged to the AirBnb subject which has 1470 elements.

Since all the HTML elements of a web page are traversed in order to build the RLG, the runtime is expected to be correlated to the number of elements in the web page. For the detection phase and using spearman's method, this correlation results in the coefficient ρ of 0.86. For illustration, Figure 5.9(a) plots the runtime of the detection phase against the number of elements in the web page. As expected, most of the data points follow the trajectory of the drawn linear regression line with one exception, the points above the line. All of which belong to the MarkText subject with only 560 elements suggesting that other variables may have a big influence on the runtime as well.

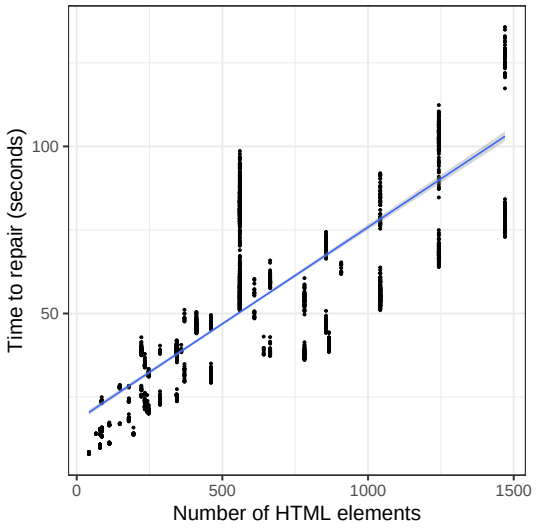
To repair a reported presentation failure with up to two alternative patches, the tool took anywhere between 7.98 and 135.69 seconds of runtime averaging 46.74 seconds per report. This is the total runtime of the second and third phases of the tool combined that are dedicated to repairing the web page. It includes the time it takes to assess the DOM for the reported failure, any change of viewports required, creating up to two patches, applying the patches, and automatically accepting or rejecting the patches based on their ability to repair the failure. Furthermore, it includes the time it takes to capture a full snapshot of the page for manual assessment of both the failure and the repair. Once again, the shortest runtime to repair any given failure belonged to a report from the BugMeNot subject while the AirBnb subject again recorded the longest runtime during the process of repairing a reported failure.

Just like the detection phase, a correlation between the runtime of the repair processes and the number of HTML elements is also expected. This is because the creation of a patch and its assessment, using the RLG, both involve iterating over all the elements of the page. Using spearman's method of correlation, the coefficient ρ was found to be 0.85 this time which is just as strong of a correlation as found with the runtime of the detection phase. Figure 5.9 (b) plots the relationship between the runtime of repair against the number of elements in the page. Clearly, a positive correlation exists as seen by the blue line in the plot.

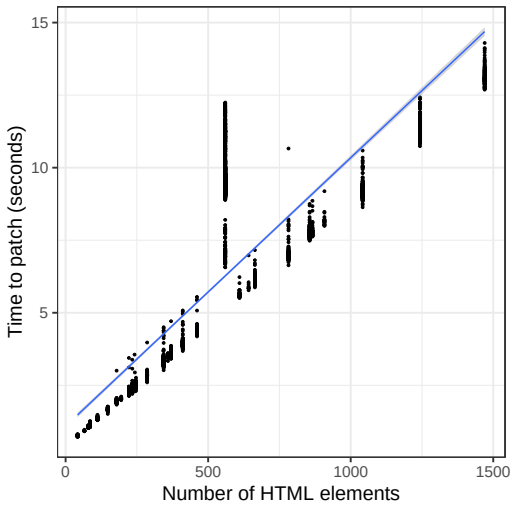
To get a better understanding of the runtime devoted to the repair process, I broke it down into the two key parts of repair. These are the creation of the patch and its assessment which includes time to take a snapshot of the repaired web page. These runtimes are showcased in parts (c) and (d) of Figure 5.9 respectively. For any given patch, the time it took the tool to create it ranged from 0.72 seconds up to 14.29 seconds of runtime with an average of 5.68 seconds per patch. The assessment portion takes slightly longer with times ranging from 3.34 seconds up to 36.69 seconds with an average of 13.25 seconds per patch assessment. The correlation of runtime and number of elements was higher for the duration of creating the patch with a coefficient ρ of 0.92 and was lower during the assessment of the patch with a ρ equal to 0.84.



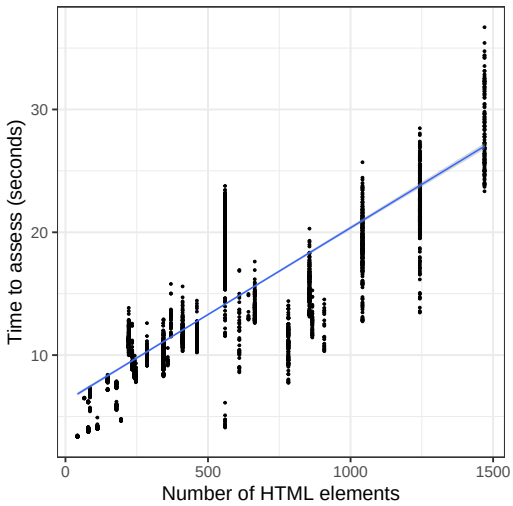
(a) Detect failures in a web page, $R^2 = 0.49$.



(b) Repair each failure, $R^2 = 0.61$.



(c) Create a single patch, $R^2 = 0.72$.



(d) Assess a single patch, $R^2 = 0.68$.

Figure 5.9: The figures compare different aspects of Layout DR runtimes to the number of HTML elements found in the web page. The runtime to detect all presentation failures for each web page is shown in (a) and the runtime of completing all possible repairs for each of the detected failures is in (b). The parts (c) and (d) are a breakdown of a portion of the runtime shown in (b). They show the time dedicated to creating a single patch as seen in part (c) and the time dedicated to assessing the success of a repair in (d) using any given patch.

Conclusion for RQ4 – The runtime of the detection phase of the tool ranged from eight minutes up to an hour with an average of about half an hour to scan the page and report presentation failures, if any are found. To repair any of the failures detected, as in creating up to two working patches, the tool took on average 48 seconds to complete but varied from eight seconds up to two minutes and a half per reported failure. Noteworthy, the runtime was found to be positively correlated to the number of elements in the web page.

5.4.3 Discussion

Although the approach followed by Layout DR has demonstrated that it is capable of repairing any given failure report, it may not be a necessary action in the first place. This is the case for the 313 dismissible reports and may include 30 more disputable reports. Nevertheless, virtually no burden is added beyond what is already required post detection of presentation failures. The patches generated do not have to be used, nor examined, except after manually confirming the failure reports. If the manual analysis is not ideal, automated visual classification can be introduced into the tool in order to reduce the number of reports into a subset that actually require a repair based on their classification. This would require integrating my contributions of the first two chapters of this thesis into the tool. Regardless of the solution, the evidence suggests that Layout DR is able to repair any DOM issue thrown at it by the detection algorithms.

Even though the DOM issue behind the failure report can be repaired using Layout DR, this does not equate to a full repair of the problem observed in the layout visually. Repairing the DOM issue may only partially move the element towards its intended position relative to visual cues. For example, if an element protrudes from one parent followed by a protrusion of another parent. Fixing the protrusion of the second parent only half solves the real issue that spans multiple parents. Another perspective on this is that the repair is limited to how good the layout of the source-viewport is. Therefore, both of the detection and repair approaches implemented in the current version fo Layout DR are not ideal for a web page that is still under development since it may contain more failures than a page that is more matured. A better fit is with live web pages since they are likely to have fewer failures and can benefit from a quick “hot fix". To improve on this, detection needs to account for the correct DOM structure inferred from all observed viewports. Currently, the detection algorithms using the RLG mainly rely on a single viewport to assert the expected DOM structure. Although small-range failures are inferred using two viewports, it is still not enough nor does it simplify automated repairs using a source-viewport. To this end, I plan to improve the detection of Layout DR as part of future work.

Choosing a source-viewport is an item for discussion on its own. The Layout DR tool uses the two bordering viewports of the reported failure range for possible usage in a patch. Theoretically, the ideal candidate layout is the one that closely resembles the layout of the viewport with the failure but does not contain the failure. These are most likely to be found in the viewports nearest to the failure range from either side as done Layout DR. Moreover, choosing the nearest viewports minimizes the scaling effect that needs to occur to fit the layout appropriately in each viewport that requires the patch. These two bordering viewports provide two alternative sources of CSS for creating two different patches. In the experiments of this chapter, the layout of the wider source-viewport best resembled the layout of the viewports where the presentation failure occurred. While the layout of the narrower source-viewport looked different due to new responsive layout rules being applied in the original page. This allows Layout DR to

prescribe two, most likely different, layouts for the user to choose from as the final repair. One alternative approach is to use both layouts simultaneously on the same failure range. This would further reduce the scaling that needs to occur but I leave this secondary study for future work.

One critical feature of the approach implemented in Layout DR is to use the CSS `scale()` method to scale the layout taken from the source-viewport. An undesired side effect of this method is that it may cause some text to be blurry for some scaled-down values. This was the case for some of the repairs from our experiments including the two failures in the human study from the 3MinuteJournal subject. Although the blurriness was not that significant, in future work I plan to look at both readability of font-size and blurriness caused by this method. A fallback for this method is to change the computed pixels values of all CSS properties directly. In my initial experiments, I found this fallback approach to be very successful. Another advantage is that it did not require any anchoring to reposition the elements in the appropriate position within the page. One complication is that some properties with pixel values should not be scaled. For example, the pixel values that indicate the cropping coordinates of a large image.

5.5 Concluding Remarks

To assist a web developer in finding possible presentation failures in their responsively designed web page ensuring its proper layout on a large number of possible devices, Layout DR re-implemented the automated detection algorithms of the legacy tool ReDeCheck. These algorithms are able to detect five types of presentation failures associated with responsive web pages. New to Layout DR, is the ability to automatically repair all five types of failures reported by the detection algorithms. By taking a layout from a viewport width not associated with the same failure, Layout DR is able to automatically scale it down and apply it to all other viewports where the failure was detected. For many presentation failures, the tool is able to propose up to two successful repairs sourced from different viewports. Moreover, the tool automatically assesses all candidate repairs for successful elimination of the presentation failure by passing a DOM-based assessment and passing another check of the detection algorithms that were used to find the failure in the first place.

The experiments of this chapter showed that Layout DR is able to produce at least one patch that successfully and automatically repairs any DOM issues detected using the algorithms for responsive layout failures. For about half of the reported failures, the tool was also able to produce an alternative working patch that can be used to repair the issue. Furthermore, when I presented 20 real presentation failures with two alternative repairs generated by the tool, 91% of participants in the human study preferred a Layout DR repair over the original page with the failure. The experiments also found no evidence of the tool introducing any real presentation failures using the source-viewport technique. Nevertheless, only 72.72% of the patches used to repair real failures were found to be free from other failures detected by the tool. Finally, the runtime of the tool was found to be positively correlated to the number of HTML elements in the page. The tool had an average runtime of half an hour for the duration of the detection phase and about 48 seconds on average to generate up to two working patches per reported failure. All of the findings of this chapter encourage more research towards automatically repairing the layout failures. The specific topics and direction for future research is the focus of the next chapter along with a summary of the research that composes this thesis.

6

Conclusions and Future Research

In this thesis, I aimed to contribute new automated means that support the developer in testing a responsively designed web page for presentation failures and in resolving them. More precisely, given a set of failures reported for the web page, I wanted to automate the manual process of visually investigating the reported presentation failures in order to filter out the ones worthy of attention and repair. Furthermore, I wanted to also automate the process of repairing the reported failures by creating patches that the developer can use to quickly repair the layout. In the previous three chapters, I proposed and evaluated multiple techniques to achieve these aims. In this chapter, I will begin by summarizing the research that I undertook for this thesis in Section 6.1 followed by a description of the known limitations and opportunities for future research in Section 6.2 before making my final remarks in Section 6.3.

The main contributions that I conceived and evaluated for this thesis were:

1. A technique to automatically classify reports of three types of presentation failures known to have DOM-based structural issues that are non-observable in the layout when visually investigated.
2. Automate the classification of two more type of presentation failures and reassess my approach to automated classification on newer web pages for all five type of failures.
3. A technique to automatically suggest up to two working repairs for each reported presentation failure for all five types.

6.1 Summary of Research

In this section, I will summarize the research that I undertook for this thesis. In Chapter 1, I introduced the research problem and aims of this thesis. In Chapter 2, I reviewed the existing research in the literature. Next, I will summarize the problem from the initial point of research and continue following the path of research that I took in order to complete the contributions of this thesis.

6.1.1 Initial Point of Research

Testing responsively designed web pages for presentation failures can be challenging without the assistance of automated tools. Since there is a wide range of screen sizes that the layout must conform to, the developer must manually check that the web page is presented appropriately for all screen sizes. With help from ReDeCheck, the layout of the web page can be tested for consistency across different viewport widths and checked for the signature of five type of layout failures. Thus, the tool can help the developer find presentation failures in the layout associated with smaller smartphone screens up to larger screen sizes connected to workstations. Unfortunately, the tool relies solely on DOM-based information to detect the failures, thus it may generate reports that are either non-observable in the layout, to the developer, or simply a false alarm. To benefit from ReDeCheck, the developer must manually investigate each report to identify the real failures worthy of repair from other trivial problems or false reports. This is a simple task to complete once or twice for a handful of reports but if the tool over reports anything other than a real failure, it can be a time consuming and error-prone task. Furthermore, as the developer continuously introduces major or minor changes to the layout, more testing is required and thus more filtering of the ReDeCheck reports is needed to prioritize important issues.

6.1.2 Classifying Non-Observable Failures (Chapter 3)

Even though the ReDeCheck tool helps front-end developers check their responsively designed web pages for presentation failures, the reports generated by the tool contain many structural issues that do not manifest into a visually observable defect in the presentation of the layout. These non-observable issues in the layout may not be a high priority for the developer to repair when compared with the failures where there is both a structural problem and a visible defect in the presentation of the layout. From the 117 presentation failures reported by ReDeCheck from the 20 subjects used in the experiments, a total of 83 reports were non-observable issues after the failures were manually classified by Walsh et al. [124]. I proposed and empirically evaluated an automated technique to classify these failure reports relieving the developer from this burden.

The automatic classification technique implemented into the tool Viser, analyses a region of the web page where a failure is located for an observable defect. This region, referred to as an *area of concern*, is calculated using the coordinates of the elements associated with the detected failure and included in the report. Using multiple snapshots of the region, the tool uses specialized algorithms that take into consideration the type of failure reported and alternative possible layout scenarios to evaluate the report on a graphical level. To achieve this, the tool changes the `opacity` CSS property of these elements to reveal the underlying graphical layer in the area of concern and compares them for colour-based pixel differences. If there is a difference between the layers where one layer overwrites another or is written out of position, the tool deems the report as an observable true positive failure. Otherwise, the failure is classified as a non-observable issue. Prior to graphically analysing the area of concern, the tool checks the structure of the web page to corroborate the reported failure using the DOM. Thereby, filtering false positive reports before any graphical analysis is carried out.

Using the baseline manual classification, independently made by Walsh et al. [124], the results of the empirical study showed that Viser agreed with the baseline classifications 86.3% of the time. With it taking less than a second to classify any of the 117 presentation failures, it is fit to relieve the developer from having to manually classify the failures. The

study also found that the minimum or narrowest viewport from the range of viewports where the presentation failure was detected has the highest probability of uncovering an observable true positive failure and matching a manual classification. By automating the classification of element collision, element protrusion, and viewport protrusion failures, I believe that Viser is an essential companion for the automated testing of responsively designed web pages.

6.1.3 Classifying Observable Failures (Chapter 4)

Although Viser showed encouraging results for automating the classification of three type of responsive layout failures reported by ReDeCheck, its primary goal was to classify non-observable issues and thus is not able to classify element wrapping and small-range failures. To put this limitation into perspective, there were 209 reports of wrapping and small-range failures from the 326 total failures detected in the 25 subjects used to evaluate ReDeCheck and Viser. That is, Viser cannot automate the classification of 64% of the reported failures. Therefore, I proposed and empirically evaluated two automated techniques to classify these two new failures on the 25 subject web pages.

To completely automate the classification of all five failure types, I implemented two new techniques into Viser and rebranded the tool as Verve. The first technique extends the usage of the `opacity` property and the algorithms already implemented in Viser in order to classify element wrapping failures. The second technique required developing a new approach to classy small-range failures because this type of failure presents itself differently than the other four types. This difference begins as early as the detection phase of small-range failures which requires the extraction of layouts from at least three viewports, instead of two like all other failure types, to detect the problem. The newly developed technique, implemented in Verve, uses snapshots from three viewports that were originally used to detect the failure in order to classify it. One of these viewports comes from the failure range and the other two are the narrower and wider bordering viewports on either side of this range. In comparison, the other four types only required one viewport from the reported failure range in order to classify the failure. Using multiple areas of concern captured in the snapshots, Verve compares the distances between the colour histogram of these images. If the distance is above a predetermined threshold, the failure is considered a true positive, otherwise, it is a false positive report.

Using the pool of 25 subjects previously used to evaluate Viser and re-utilizing the same baseline manual classification independently made by Walsh et al. [124], the results of the empirical study showed that Verve agreed with the baseline classifications 78.6% of the time for element wrapping failures. Furthermore, when classifying wrapping failure using the minimum, middle, or maximum viewports from the reported failure range, the agreement was consistent regardless of the viewport chosen for the classification. For the results of small-range classification, with the threshold tuned to the best possible value, the analysis showed that the tool is capable of achieving up to a 98.5% agreement for small-range failures by configuring Verve to use the *Intersection* histogram measure and the horizontal-plus-vertical referencing approach for dissecting the snapshots into smaller areas of concern.

6.1.4 Reassessing Automated Classifications (Chapter 4)

Even though Verve is now able to classify all failure types produced by ReDeCheck, its performance on newer web pages is unknown. Since the threshold values used to

classify small-range failures were tuned using the original subjects, testing Verve on newer subjects is especially important for this failure type. For this purpose, I gathered 20 additional web pages made up from popular responsive example web pages and from real web pages used by the creator of ReDeCheck to test newer algorithms of the tool. Prior to the experiments, I manually classified all the failures reported from this additional set of pages and used it as the baseline to measure Verve's performance.

From the results of reassessing Verve on the 20 additional subjects, the tool showed good agreement levels with the baseline manual classifications. More precisely, using the minimum viewport of each reported failure range for classification, the change in agreement went from 93.5% to 100% for element collision, 84.6% to 92.2% for element protrusion, 83% to 80% for viewport protrusion, and 78.6% to 64.7% for wrapping failures. For small-range failures, there were only false positives failure reports from this set. Therefore, I manually injected synthetic failures into the additional subjects to overcome this threat. Then, I separately investigated the original reports from the unmodified additional subject and the new failures reported after injecting the synthetic failures. Since there were multiple choices for histogram measures and for dissecting the area of concern, no consistent recommendation could be made without combining all failures including the 25 initial subjects. Using all 45 subjects, Verve achieved 87.5% for small-range failures while configured to use the *Chi-Square* measure combined with the horizontal-plus-vertical referencing approach. This is a decrease compared to the 98.5% agreement over the 25 initial subjects. Impressively, the recommended threshold to use is the original threshold determined prior to introducing the additional set of subjects.

In a test of the runtime required for Verve to automatically classify any of the 469 failures reported from the 45 subject web pages, the tool took 4 seconds on average to complete for any of the five type of failures produced by ReDeCheck.

6.1.5 Repairing Presentation Failures (Chapter 5)

With Verve assisting the developer classify and thus filtering out the real presentation failures, the developer must now manually repair these higher priority failures. To further aid the developer in this matter, I implemented a tool, named Layout DR, that is able to automatically create up to two working patches that repair the layout. These quick fixes, generated by Layout DR, can be applied to the web page to buy the developer valuable time needed to investigate the root cause of the problem and create a customized repair.

The idea behind Layout DR stems from ReDeCheck's successful ability to detect presentation failures in the layout of certain viewports widths. Moreover, it is able to determine and report the consecutive viewports where the failure manifests, this is known as the failure range. For any given failure, Layout DR benefits from this information by inferring that the narrower and wider viewports bordering the failure range are free from the reported failure. Thus, the layouts from these two viewports can be used as alternative sources for a solution to repair the layouts that display the failure. In other words, Layout DR can borrow code from the narrower source-viewport or the wider source-viewport and apply it over the viewports reported with the failure in order to mend the page.

To repair a failure, Layout DR retrieves the entire code used to render the layout of the bordering source-viewport and adds extra code in order to properly fit the borrowed layout to the viewport widths in the range of the failure. To avoid overwriting layouts of viewports outside the failure range, the tool also adds rules that restrict where the

borrowed code is applied using the failure range. With the code injected into the page, Layout DR automatically verifies that the patch successful eradicated the failure from the previously associated viewports. For this, the tool uses the original detection algorithms used to find the failure in the first instance to ensure that the same failure is not reported in the patched web page.

During the empirical evaluation of Layout DR, I employed 21 of the subject web page used to evaluate ReDeCheck and later Verve plus 10 newly accrued subjects for evaluating Layout DR. One of the major findings of the evaluation is that Layout DR can repair 100% of the reported failures using the wider source-viewport. Similarly, the narrower source-viewport based repairs had a 96% success rate when the tool determined the viewport to be valid for use in a repair. Nevertheless, it was only valid for 50% of all the reported failures. Thus, Layout DR tool is expected to always produce at least one working patch and up to two patches for half the reported failures. I further investigated whether the tool introduces new failures when a patch is applied and found that the technique does not introduce true positive failures that can be detected using automated means. This is true whether the tool is using the narrower source-viewport or the wider source-viewport.

For a deeper evaluation of Layout DR that goes beyond the tool's ability to produce working patches for any failure regardless of whether it should be repaired or not, I manually narrowed down the 398 reports to 55 *definite* failures that are observable with minimal subjectivity. Using this set of definite failures, I manually investigated each repair to find out if Layout DR copied any other definite failure into a working patch. I found out that 72% of the 55 definite failures had at least one failure-free repair option. Furthermore, I presented the 20 definite failures that had both repair options to participants in a human study to determine whether they preferred a repaired version of the page over the original with the failure. The study showed that a Layout DR repair was preferred over the web page with the failure 91% of the time. Furthermore, the participants preferred the wider source-viewport based repair 65% of the time.

In a test of the runtime required for Layout DR to automatically create up to two working repairs for any of the 398 failures reported from the 31 subject web pages, the tool took less than one minute on average to complete for any of the five type of failures.

6.2 Future Research

The techniques developed, tools implemented, and the experiments conducted as part of this thesis had some limitations and open-opportunities for future research. In this section, I will briefly describe these limitations and suggest ideas to overcome them or bring general improvement to the results.

6.2.1 Improving Detection of Failures

Because I have re-implemented the detection algorithms of ReDeCheck into Layout DR in order to resolve known issues with the legacy tool, I gained valuable knowledge about the approach and its limitations. Based on this experience, I will discuss, next, the limitations and opportunities for an improved approach to detect presentation failures in responsively designed web pages.

HTML structure – One of the main features of the ReDeCheck approach is a graph-based model called the Responsive Layout Graph (RLG) that decouples the HTML docu-

ment structure from the rendered layout. This decoupling reduces the complexity involved in understanding the HTML and CSS rules and how different browsers, and versions, finally render the page. To achieve this, the RLG assumes that the tightest element encompassing another element is its intended container. Though, this simplification of the problem does come at a cost when dealing with elements that are designed to be out of normal flow in the page as the user scrolls the page, essentially *floating* around. An example of this type of element can be seen in Figure 4.10 from Chapter 4. As seen in the experiments, when the viewport width is changed while the page is being tested for presentation failures, the position of the out of flow element is also changed. In consequence, the out of flow element may leave the falsely assigned container in the RLG model and be incorrectly reported as a failure. To overcome this problem, I plan, as part of future research, to introduce a condition in the RLG that assigns out of flow elements only to the root container in the RLG. This may also require the inclusion of additional rules that take into consideration the descendants of an out of flow element.

It is worthy to also note that the RLG is not fully decoupled from the HTML structure because the `body` element is hard-coded to be the root container of all other elements in the RLG. This is because the `body` element is the main display element based on the DOM structure. Moreover, the RLG assumes that the width of the `body` element never exceeds the viewport width which is a problem when testing a responsively designed web page. When detecting viewport protrusion failures, this may lead to under-estimating or over-estimating the number of failures if the `body` element is wider or narrower than the viewport width, respectively. A simple solution to this problem is to introduce a pseudo-element as the root container in the RLG that more accurately represents the viewport width and let us call it the `page` element. Based on RWD design principles, this `page` element should have infinite height and be only as wide as the viewport width being extracted. I believe this will further improve the detection of responsive layout failures and I plan to test this approach in the future.

Pixel tolerance – One technical issue that arises when reading the coordinates of an HTML element, using the DOM, is that it may include decimal points. If the readings are kept as raw values and one element is overlapping another by a fraction on any given axes, the tool making the readings must decide whether this is an overlap or not. Regardless of the choice made, this may or may not reflect how a specific browser renders the two elements. This seemingly trivial problem only grows in complexity as the tool attempts to determine the tightest container of an element, siblings within the same container, ancestors and descendants, and other relative positions like above-of and right-of. To overcome this problem, the tools must provide some leeway when making these decisions. Although this tolerance is necessary using the approach originated by ReDeCheck, changing the tolerance value can change the output of the tool significantly. Even though every tool must decide how to handle these fractional values, the problem for ReDeCheck largely exists because at each viewport, the tool extracts the layout and makes these decisions solely on the information from a single viewport. This can be due to the fact that the ReDeCheck extended existing techniques without considering more improvements. One possible solution to this problem is to infer the relative positions using readings from multiple, if not all, viewports. This can be achieved through basic heuristics or a more advanced artificial intelligence approach. Either way, the solution is expected to increase the stability and quality of the output.

Limited usability – one of the main reasons preventing the wider adoption of automated means to detect presentation failures, as implemented in the ReDeCheck tool, is the dynamic nature of modern web pages. These modern pages sometimes rely on JavaScript code that runs in the browser to automatically and routinely update certain elements in

the page. If these routines change the relative size, relative position, tag type, or the number of elements that make up the layout, the underlying model used to infer the failures cannot properly test the layout of the page. In other words, the tool is only built to test layouts that are static in relative positions, relative sizes, and the overall HTML structure. One simple way for the developer to support automated testing is to introduce a unique identifier for each element, manually or automatically, and any JavaScript-based change must update the identifier as it makes a change to the element. These identifiers can be used by the testing tool to infer the state of the layout and thus make more accurate assumptions and thereafter reporting. Further research would need to compare such manual effort to an alternative and automatic way to detect the state of the page for usage during the testing. This will not only increase the number of pages that can be tested for responsive layout failures but should generalize to aid in testing other type of presentation failures like cross-browser issues.

6.2.2 Improving Classification of Failures

Although Verve's automated classification of responsive layout failures is expected to improve the process of testing responsively designed web pages, there are at least five opportunities for improvement and more research. Next, I will discuss the details of these improvements and the current limitations motivating them.

Horizontal scrolling – the empirical study of Chapter 3, and later Chapter 4, revealed that there were a few misclassifications made by the tools Viser and Verve. These were mostly specific to the viewport protrusion failure type. A limitation of the current algorithm that classifies this type of failure is that it does not take into consideration the fact that horizontal scrolling defeats the purpose of a responsive design. Currently, the algorithm looks for differences in the snapshots to conclude if the failure is observable or not in the layout but it does not account for differences that require a horizontal scroll. In other words, the algorithm should check on a graphical level if the content is written beyond the viewport width size and therefore require a horizontal scroll to be viewed. This improvement should not be limited to viewport protrusion failures, it should be applied to all failure types in order to preserve the sanity of the responsive design. To be verified by future experiments, the newly added condition is expected to raise the agreement between manual and automated classifications.

Degree of Change – another limitation revealed by the experiments carried out in Chapter 3 and Chapter 4 is that not all changes caused by the failure are equally problematic to the presentation of the layout. Unlike a human that is manually classifying a presentation failure, the current technique implemented in Verve does not make an exemption for "minor" changes in the layout. Mimicking these subjective exemptions in Verve to dismiss minor defects in the layout is anticipated to further improve the agreement between manual and automated classification. Based on my investigation of these subjectively issued exemptions, three main heuristics may help the algorithm in prescribing these exemptions. The first is to measure the number of pixels changed in the layout and establish a threshold for a valid exemption. Moreover, it may be more helpful to account for the number of changed pixels along the x and y axis of the web page. The second heuristic should account for the general size of elements involved in the failure. Finally, one heuristic should investigate the degree of change in colours and define a threshold of a human visible colour change. Although intuitively valid, these improvements need to be validated experimentally in the future.

Alternative CSS property – the approach implemented in Viser and thereafter Verve uses the `opacity` CSS property to make an element transparent in order to "see" the layer

behind it. Although the tool does not gradually change the opacity and instead sets it to complete transparency, during the experiments I ran into a case where an element was only partially transparent when a snapshot was taken immediately after modifying the opacity value. This issue of asynchronicity was resolved by introducing an added delay in order to wait for an element to be completely transparent before a snapshot is taken. The alternative `visibility` property is expected to require the same delay or more. Since descendant elements can override this property, multiple delays may be needed. Future work would need to investigate the trade-offs of the `visibility` property instead.

Classifying other failures and possible detection benefits – Even though VERVE demonstrated the benefits of automatically classifying the REDECHECK reported failures, it is limited as a tool to the failure types reported by REDECHECK. As part of future work, I plan to adapt VERVE to handle failures reported using other DOM-based tools. These can include cross-browser testing tools, internationalization testing tools, and front-end state exploration and testing tools like VFDETECTOR [99]. Another future direction is to explore the viability of using graphical information from different layers, perhaps using the `opacity` CSS property, to detect presentation failures. Empowered with knowledge about the DOM structure and the customized CSS rules of the page, this approach may be capable of detecting failures without requiring information from multiple viewports, browsers, or other front-end states. Nevertheless, using more information, say from other viewports, is expected to increase the chance for success.

Running visual content – One possible limitation of VERVE's classification approach, especially for small-range failures, is that it expects the page to remain visually consistent. Since snapshots are used to analyse the failures, a drastic change in the colours (*e.g.*, a video playing in the page) may increase the chance of making a mistake while the tool is analysing the failure. To overcome this limitation, I plan to experiment with using multiple snapshots of the same area of concern. By differencing multiple images of the same area of concern, the tool can infer a change in colours. The alternative, or in combination, is to check for known elements like the `video` html element, JavaScript events, or CSS changes that may affect the outcome of the classification. Either way, VERVE should notify the user, as part of its classification, of the certainty or confidence level in the output.

6.2.3 Improving Repair of Failures

The LAYOUT DR tool has been shown to successfully automate the repair of responsive layout failures but there are at least five opportunities for improvement and more research. Next, I will discuss the details of these improvements and the current limitations motivating them.

Multi-failure pages – Although LAYOUT DR experimentally proved that it can automatically repair one viewport by borrowing or extending the layout from another viewport, known as the *source-viewport*, it will also bring with it any other failure already in the source-viewport. Therefore, the current version of LAYOUT DR does not actively avoid source-viewports where a failure is detected. Although this is a good feature to have, the task is not as simple as checking for cross-over of other failure ranges. Since the severity of the failure changes depending on the viewport, the presence of a failure at the source-viewport does not mean the viewport cannot be used in a repair. One possible solution is to integrate VERVE into LAYOUT DR in order to answer if the source-viewport can be used for repair while known to have failures. If the failures are only non-observable issues, LAYOUT DR can proceed with the repair. The current approach implemented in LAYOUT DR is more suited for published web pages that are expected to have fewer presentations failures and would also benefit from a quick fix.

Partial repairs – The approach implemented in Layout DR technically does repairs a presentation failure but the web page may need further repairs before the visual problem is fully repaired. This should not be confused as a problem with the technique itself, rather it is a limitation of using the failure range reported by ReDeCheck. Since ReDeCheck indicates the starting and ending viewports of a failure using only DOM-based information, the range may not reflect the start and end of the problem graphically. As seen in Figure 5.6(c) of Chapter 5, the source-viewport may not always provide an aesthetically pleasing point for repair. If a newer tool with superior detection capability is more accurate in determining the real failure range, the problem will cease to exist. Until then, more research would need to investigate if a simple offset, added to the failure range, is sufficient to overcome this problem.

Partial repair is also possible when related elements are involved in consecutive failures, in terms of viewports that is. For example, if an HTML element is involved in an element protrusion failure and then a descendant element is involved in another element protrusion failure, repairing the protruding descendant element will only bring the layout into a state where the ancestor element is still protruding. Therefore, one failure is repaired but it may seem like it is partly repaired due to another visually related failure. In other words, the repair can only be as good as the source-viewport. In the future, I plan to experiment with added conditions that better examine the source-viewport and perhaps reach further out for a viewport that is better suited for repairing the failure. Again, the integration of Verve is expected to largely resolve this problem.

Scaling limits – The most vulnerable feature of Layout DR for criticism involves the chance for extreme scaling. During the evaluation of the tool, there was no research question that investigated the limitation of scaling a layout, up or down. Intuitively, there exists some limit on how much a layout can be scaled for readability and aesthetic appeal. This limit will depend on multiple factors but the most important is the original readability and aesthetic appeal of the layout prior to scaling. Arguably, scaling is the key feature that allows tools to automatically repair a web page along with the repositioning of content in the page. Moreover, it is not expected that an automated repair approach would introduce new content nor remove any without context. To mitigate the problem of extreme scaling, an easy solution for Layout DR is to simultaneously apply both repairs from the narrower and wider source-viewport, when possible, to reduce the scaling effect by half. Moreover, I plan to study the limits of scaling a layout in general and experiment with the repositioning of elements in the page as part of the patch. Finally, I also plan on using a method to prevent font scaling as done by CBRepair [1] or like IFix^{++} [65] by leaving it as a last resort.

Fault localizing – One feature making Layout DR undesirable is that it copies over an entire layout, by traversing the entire DOM, in order to repair a problem limited to a portion of another layout. Although an ideal version would attempt to localize the failure in the CSS and HTML to limit the patch to the problematic properties and elements, there is another alternative that is inspired by the current success of Layout DR. Instead of fault localization, or in addition to it, one avenue for future research is to reduce the code of a successful patch to a smaller sized working patch. Since the DOM tree, rooted at the `html` element, is used to apply the patch, it would be interesting to learn if iteratively pruning the tree would yield an acceptable repair. Moreover, I expect that a top-down pruning approach would be the best approach to take and also speculate that limiting the pruning to the direct children of the `body` element would provide a good trade-off between performance and an acceptable repair. Additional heuristics like which subtree contains the failing elements, as indicated in the detection phase, may provide good guidance when pruning. Furthermore, this should improve the performance of the task. Another level of

systematic reduction that will come at a higher performance cost is at the CSS properties level for each node in the tree. Finally, I would also like to see the results of bringing a human, ideally the developer, into the decision loop while the tool is pruning the tree.

A less invasive approach to automate the repair process is to use basic heuristics in order to resolve the failure. For example, the repair can start by targeting the elements reported as failing, from the detection phase, for scaling and repositioning. Alternatively, it can target some ancestors of the reported elements or only a single common ancestor instead. To better guide the repair, the type of failure reported can add another layer of insight. Finally, collecting information about the size and relative position of the elements from one or both bordering viewports, of the failure range, will further aid in the repair. During experiments that I did not include in this thesis, I probed this idea and found that it can achieve good results but pails in comparison to the approach that I finally relied on which borrows the full layout from the bordering viewport. More specifically, I found that using these heuristics was less methodological, required my personal domain knowledge, and grew in complexity in order to reach good results. Nevertheless, I plan as part of future work to revitalize this approach, improve it, and compare it to the successful approach currently implemented in Layout DR.

Repurposed repairs – Just as the ReDeCheck reported failures are limited to reflect only the browser being used to test the web page, the Layout DR repairs only reflect the requirements of a single browser. More so, it is not a good idea to apply the repair on any other browser since it was not designed or tested for that purpose. In the real world, it is not uncommon that a developer may use certain features from one browser that are not available in another. Furthermore, there may be additional differences in how each browser interprets CSS and HTML standards. Since the patch currently uses absolute pixel values and uses all the CSS properties available by the browser, any Layout DR generated patch will require additional code that is able to detect if the browser being used by the end-user requires the patch. This is one reason why I plan to investigate the benefits of repurposing the source-viewport based repairs.

Another reason to repurpose the Layout DR repair strategy is that the developer may not prefer to use the resulting automated patch. Nevertheless, the technique of using a source-viewport for repair has potential that goes beyond the developer's needs. In the future, I see a potential to repurpose the type of repairs created by Layout DR for usage by the browsers to present a different layout to the end-user. In one use case, an end-user that is using a smaller screen size can ask the browser to correct the layout, due to any given problem, to an alternative layout designed for larger screen sizes. This can extend, in a semi-automated fashion, to the user clicking on the element with the problem so the browser can try to find a better source-viewport. As part of future research, a human study would need to investigate the technical and practical viability of using a layout correcting browser.

6.3 Final Remarks

In this thesis, I have proposed and evaluated techniques that support the developer, of a responsively designed web page, in testing the layout of the page as it changes to conform to different screen sizes. Specifically, the main contributions were (1) a technique that can automatically classify non-observable issues found in three type of presentation failures, (2a) I extended this technique and created a new one in order to classify two more failure types, (2b) I reassessed the performance of my automated classification techniques for all five failure types, and (3) automated the repair of all five failure types. These are in

addition to a technical contribution, the re-implementation of the ReDeCheck algorithms with minor improvements into a newer tool named Layout DR. This is to make installing the tool easier, update known issues with the legacy tool, and to bridge the gap between the detection of presentation failure and their repair.

In this chapter, I have identified known limitations and many avenues for further research that I expect to improve the automated detection, classification, and repair of presentation failures. Although my prototype tools and many of those presented in the literature that I reviewed for this thesis were largely successful within their research-based scope, the real-world requires more advancements. I expect that a more universal tool that incorporates multiple layout testing techniques to be more beneficial in the real-world. With the Layout DR tool, the developer can be presented with both the problem and up to two possible solutions. Furthermore, I expect the tool to be even more beneficial once I complete the integration of Verve's helpful automated classifications into future versions of Layout DR. Critically, for developers to adopt Layout DR, future research should prioritize advancement in the detection of presentation failures that aim to raise the accuracy of the detection phase and increase the number of subjects that can be tested. In consequence, this should reduce the need to visually classify the reported failures and is expected to reflect positively on the results of automated repair.

Although I have Identified some opportunities for further research, the findings from this thesis concluded that (1) my automated classification approach for the three failure types associated with non-observable issues, implemented in Viser and Verve, matches the human-based manual classification with high agreement, (2a) this high agreement was also achieved by additional approaches, implemented in Verve, to classifying the other two failures types (2b) furthermore, the high agreement does extend well into newer subjects for all five failure type, finally (3) my automated repair technique, implemented in Layout DR, is able to successfully patch any of the five failure types and output up to two working patches for the developer to use. Ultimately, I see Verve and Layout DR playing an important role in helping web developers surface, classify, and repair responsive layout failures. My own contributions implemented in Verve and Layout DR should aid the developer in prioritizing the defects reported and help efficiently and automatically deploy fixes for a defective web page.

Bibliography

[1] Abdulmajeed Alameer, Paul T. Chiou, and William G. J. Halfond. "Efficiently Repairing Internationalization Presentation Failures by Solving Layout Constraints". In: *Proceedings of the 12th International Conference on Software Testing, Validation and Verification*. 2019, pp. 172–182.

[2] Abdulmajeed Alameer and William G.J. Halfond. "An Empirical Study of Internationalization Failures in the Web". In: *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*. 2016.

[3] Abdulmajeed Alameer, Sonal Mahajan, and William G. J. Halfond. "Detecting and Localizing Internationalization Presentation Failures in Web Applications". In: *Proceedings of the 9th International Conference on Software Testing, Verification and Validation*. 2016.

[4] Pablo Fernández Alcantarilla, Adrien Bartoli, and Andrew J Davison. "KAZE features". In: *European conference on computer vision*. Springer. 2012, pp. 214–227.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd ed. Cambridge University Press, 2016.

[6] "Automatic detection of potential layout faults following changes to responsive web pages". In: *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015* (2016), pp. 709–714. DOI: 10.1109/ASE.2015.31.

[7] Md Aquib Azmain and Kishan Kumar Ganguly. "Automated Repair of Asymmetric Web Pages during Resolution of Mobile Friendly Problems." In: *ENASE*. 2021, pp. 461–468.

[8] *BDD Testing and Collaboration Tools for Teams | Cucumber*. URL: https://cucumber.io/ (visited on 04/01/2022).

[9] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The oracle problem in software testing: A survey". In: *IEEE transactions on software engineering* 41.5 (2014), pp. 507–525.

[10] T. Berners-Lee and Fischetti M. *Weaving the web, the Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. HarperBusiness, 2000. ISBN: 0-06-251587-X.

[11] *Bing - Mobile Friendliness Test Tool*. URL: https://www.bing.com/webmaster/tools/mobile-friendliness (visited on 04/01/2022).

[12] Barry W Boehm. "Software engineering: R&D trends and defense needs". In: *Research directions in software technology* (1979).

[13] *Bootstrap*. URL: https://getbootstrap.com/ (visited on 03/23/2022).

[14] *Box Model*. URL: https://www.w3.org/TR/CSS2/box.html (visited on 04/12/2022).

[15] Gary Rost Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. First. O'Reilly Media, Inc., 2008.

[16] Gary Bradski and Adrian Kaehler. *Learning OpenCV 3*. O'Reilly, 2016.

[17] Browsera. *Browsera - Automated Cross Browser Web Application Testing Service*. URL: http://www.browsera.com/ (visited on 05/06/2018).

[18] Browserbite. *Browserbite - Automatic cross browser testing*. URL: http://browserbite.com/ (visited on 05/06/2018).

[19] Paul Butcher and Jacquelyn Carter. *Debug it!: find, repair, and prevent bugs in your code*. Pragmatic Bookshelf, 2009.

[20] *CSS Reset - YUI Library*. URL: https://clarle.github.io/yui3/yui/docs/cssreset/ (visited on 04/17/2022).

[21] *CSS Tools: Reset CSS*. URL: https://meyerweb.com/eric/tools/css/reset/ (visited on 04/17/2022).

[22] *CSS3 Media Queries: Simple Gotchas and Easy Fixes*. URL: https://www.crimsondesigns.com/blog/css3-media-queries-simple-gotchas-easy-fixes/ (visited on 05/01/2020).

[23] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. *VIPS: a Vision-based Page Segmentation Algorithm*. Tech. rep. MSR-TR-2003-79. 2003, p. 28. URL: https://www.microsoft.com/en-us/research/publication/vips-a-vision-based-page-segmentation-algorithm/.

[24] Raymond Camden and Brian Rinaldi. *Working with Static Sites: Bringing the Power of Simplicity to Modern Sites*. O'Reilly Media, 2017.

[25] John Canny. "A computational approach to edge detection". In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), pp. 679–698.

[26] SR Choudhary, Husayn Versee, and Alessandro Orso. "WEBDIFF: Automated identification of cross-browser issues in web applications". In: *Software Maintenance (ICSM)*. 2010, pp. 1 –10. ISBN: 9781424486281. DOI: 10.1109/ICSM.2010.5609723. URL: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5609723.

[27] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. "CrossCheck: Combining Crawling and Differencing To Better Detect Cross-browser Incompatibilities in Web Applications". In: *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*. 2012.

[28] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. "X-PERT: Accurate identification of cross-browser issues in web applications". In: *Proceedings - International Conference on Software Engineering*. 2013, pp. 702–711. ISBN: 9781467330763. DOI: 10.1109/ICSE.2013.6606616.

[29] Association for Computing Machinery. *Inventor of World Wide Web Receives ACM A.M. Turing Award*. URL: https://awards.acm.org/about/2016-turing (visited on 05/06/2018).

[30] Crossbrowsertesting. *Cross Browser Testing Tool: 1500+ Real Browsers & Devices*. URL: https://crossbrowsertesting.com/ (visited on 05/06/2018).

[31] Dianne Cyr, Milena Head, and Alex Ivanov. "Design Aesthetics Leading to M-loyalty in Mobile Commerce". In: *Information & Management* 43.8 (2006).

[32] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. "Hints on test data selection: Help for the practising programmer". In: *Computer* 11.4 (1978), pp. 34–41.

[33] *Desktop vs Mobile vs Tablet Market Share Worldwide*. 2022. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet#monthly-201202-202202 (visited on 03/25/2022).

[34] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. Vol. 96. 34. 1996, pp. 226–231.

[35] *Fast and reliable end-to-end testing for modern web apps | Playwright*. URL: https://playwright.dev/ (visited on 04/01/2022).

[36] Masha Fisch. *Mobile-friendly sites turn visitors into customers*. 2012. URL: http://googlemobileads.blogspot.com/2012/09/mobile-friendly-sites-turn-visitors.html (visited on 03/25/2022).

[37] Fred Glover. "Tabu search—part I". In: *ORSA Journal on computing* 1.3 (1989), pp. 190–206.

[38] *Google - PageSpeed Insights*. URL: https://developers.google.com/speed/pagespeed/insights/ (visited on 04/01/2022).

[39] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. "Automated program repair". In: *Communications of the ACM* 62.12 (2019), pp. 56–65.

[40] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. "Deepfix: Fixing common c language errors by deep learning". In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[41] *HTML Tidy Project Page*. URL: http://tidy.sourceforge.net/ (visited on 04/14/2022).

[42] Sylvain Hallé, Nicolas Bergeron, Francis Guerin, and Gabriel Le Breton. "Testing Web Applications Through Layout Constraints". In: *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*. 2015.

[43] Maggie Hamill and Katerina Goseva-Popstojanova. "Common trends in software fault and failure data". In: *IEEE Transactions on Software Engineering* 35.4 (2009), pp. 484–496.

[44] Mark Harman. "Automated patching techniques: the fix is in: technical perspective". In: *Communications of the ACM* 53.5 (2010), pp. 108–108.

[45] Meimei He, Guoquan Wu, Hongyin Tang, Wei Chen, Jun Wei, Hua Zhong, and Tao Huang. "X-check: A novel cross-browser testing service based on record/replay". In: *2016 IEEE International Conference on Web Services (ICWS)*. IEEE. 2016, pp. 123–130.

[46] Robert Hof. *Google Research: No Mobile Site = Lost Customers*. 2012. URL: https://www.forbes.com/sites/roberthof/2012/09/25/google-research-no-mobile-site-lost-customers/?sh=4cb69c8259d1 (visited on 03/25/2022).

[47] *JQuery*. URL: https://jquery.com/ (visited on 03/10/2022).

[48] Stéphane Jacquet, Xavier Chamberland-Thibeault, and Sylvain Hallé. "Automated Repair of Layout Bugs in Web Pages with Linear Programming". In: *International Conference on Web Engineering*. Springer. 2021, pp. 423–439.

[49] *JavaScript End to End Testing Framework | cypress.io testing tools*. URL: https://www.cypress.io/ (visited on 04/01/2022).

[50] Joseph Kempka, Phil McMinn, and Dirk Sudholt. "Design and analysis of different alternating variable searches for search-based software testing". In: *Theoretical Computer Science* 605 (2015), pp. 1–20.

[51] James Kennedy and Russell Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95-international conference on neural networks*. Vol. 4. IEEE. 1995, pp. 1942–1948.

[52] Bogdan Korel. "Automated software test data generation". In: *IEEE Transactions on software engineering* 16.8 (1990), pp. 870–879.

[53] Thanh Le-Cong, Xuan Bach D Le, Quyet Thang Huynh, and Phi Le Nguyen. "Usability and Aesthetics: Better Together for Automated Repair of Web Pages". In: *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2021, pp. 173–183.

[54] Vladimir I Levenshtein et al. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[55] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. "Scalable statistical bug isolation". In: *Acm Sigplan Notices* 40.6 (2005), pp. 15–26.

[56] Fan Long, Peter Amidon, and Martin Rinard. "Automatic inference of code transforms for patch generation". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 727–739.

[57] Fan Long and Martin Rinard. "Automatic patch generation by learning correct code". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 298–312.

[58] Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. "Are faults localizable?" In: (2012), pp. 74–77.

[59] Rastislav Lukac. *Perceptual digital imaging: methods and applications*. CRC Press, 2017.

[60] S. Mahajan, B. Li, P. Behnamghader, and W. G. J. Halfond. "Using Visual Symptoms for Debugging Presentation Failures in Web Applications". In: *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*. 2016.

[61] Sonai Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G. J. Halfond. "Automated Repair of Mobile Friendly Problems in Web Pages". In: *Proceedings of the 40th International Conference on Software Engineering*. 2018.

[62] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. "Automated Repair of Layout Cross Browser Issues Using Search-Based Techniques". In: *Proceedings of the International Conference on Software Testing and Analysis*. 2017, pp. 249–260.

[63] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. "XFix: Automated Tool for Repair of Layout Cross Browser Issues". In: *Proceedings of the International Conference on Software Testing and Analysis*. 2017, pp. 368–371.

[64] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. "Automated Repair of Internationalization Failures Using Style Similarity Clustering and Search-Based Techniques". In: *Proceedings of the 11th International Conference on Software Testing, Validation and Verification*. 2018.

[65] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. "Effective automated repair of internationalization presentation failures in web applications using style similarity clustering and search-based techniques". In: *Software Testing, Verification and Reliability* 31.1-2 (2021), e1746.

[66] Sonal Mahajan and William G. J. Halfond. "Finding HTML Presentation Failures using Image Comparison Techniques". In: *Proceedings of the 29th International Conference on Automated Software Engineering*. 2014.

[67] Sonal Mahajan and William G. J. Halfond. "Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques". In: *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*. 2015.

[68] Sonal Mahajan and William G. J. Halfond. "WebSee: A Tool for Debugging HTML Presentation Failures". In: *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*. 2015.

[69] Ethan Marcotte. *Responsive Web Design*. URL: `https://alistapart.com/article/responsive-web-design/` (visited on 02/10/2022).

[70] Ethan Marcotte. *Responsive Web Design*. A Book Apart, 2011.

[71] Wolfgang Mayer and Markus Stumptner. "Modeling programs with unstructured control flow for debugging". In: *Australian Joint Conference on Artificial Intelligence*. Springer. 2002, pp. 107–118.

[72] William M. McKeeman. "Differential Testing for Software". In: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998), pp. 100–107.

[73] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. "Angelix: Scalable multiline program patch synthesis via symbolic analysis". In: *Proceedings of the 38th international conference on software engineering*. 2016, pp. 691–701.

[74] Ali Mesbah and Mukul R Prasad. "Automated Cross-browser Compatibility Testing". In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011.

[75] E Fagan Michael. "Advances in software inspections". In: *IEEE Transactions in Software Engineering* 12.7 (1986).

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

[77] *Mobile-Friendly Test - Google Search Console*. URL: `https://search.google.com/test/mobile-friendly` (visited on 04/01/2022).

[78] Mobiletest. *MobileTest.me - Test your mobile sites and responsive web designs*. URL: `http://mobiletest.me/` (visited on 05/06/2018).

[79] Mongotest. *MogoTest*. URL: `http://mogotest.com/` (visited on 05/06/2018).

[80] Martin Monperrus. *The Living Review on Automated Program Repair*. Tech. rep. hal-01956501v3. 2021. URL: `https://hal.archives-ouvertes.fr/hal-01956501v3`.

[81] Fernando Monteiro. *Learning Single-page Web Application Developement)*. PACKT Publishing, 2014.

[82] Mozilla. *Introduction to the DOM - Web APIs | MDN*. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Document{\_}Object{\_}Model/Introduction` (visited on 05/08/2018).

[83] *Multi-screen overview - Google AdSense Help*. URL: `https://support.google.com/adsense/answer/6051803` (visited on 04/01/2022).

[84] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. "Auto-locating and fix-propagating for HTML validation errors to PHP server-side code". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 13–22.

[85] *Normalize.css: Make browsers render all elements more consistently*. URL: `https://necolas.github.io/normalize.css/` (visited on 04/17/2022).

[86] *OR-Tools - Google Developers*. URL: `https://developers.google.com/optimization` (visited on 04/12/2022).

[87] *OpenCV: Open-Source Computer Vision Library*. URL: `https://opencv.org` (visited on 11/06/2019).

[88] *PageGen Repository*. URL: `https://github.com/sylvainhalle/pagegen` (visited on 04/14/2022).

[89] Pavel Panchekha and Emina Torlak. "Automated reasoning for web page layout". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2016, pp. 181–194.

[90] Clarissa Peterson. *Learning Responsive Web Design: A Beginner's Guide*. 1st ed. O'Reilly Media, 2014.

[91] *Puppeteer: Headless Chrome Node.js API*. URL: https://github.com/puppeteer/puppeteer (visited on 04/01/2022).

[92] *Quran - 21:25 (Translated by Abdullah Yusuf Ali)*. URL: https://quran.com/21/25 (visited on 04/14/2022).

[93] *Responsinator*. URL: https://www.responsinator.com/.

[94] *Responsive Design Checker*. URL: http://responsivedesignchecker.com (visited on 05/06/2018).

[95] Brian Rinaldi. *Static Site Generators: Modern Tools for Static Website Development)*. O'Reilly Media, 2015.

[96] David Robins and Jason Holmes. "Aesthetics and Credibility in Web Site Design". In: *Information Processing & Management* 44.1 (2008).

[97] Richard Romero and Adam Berger. "Automatic partitioning of web pages using clustering". In: *International Conference on Mobile Human-Computer Interaction*. Springer. 2004, pp. 388–393.

[98] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. "The Earth Mover ' s Distance as a Metric for Image Retrieval". In: *International Journal of Computer Vision* 40.2 (2000), pp. 1–20. ISSN: 0920-5691. DOI: 10.1023/A:1026543900054. arXiv: 0005074v1 [arXiv:astro-ph]. URL: http://www.springerlink.com/index/W5515K817681125H.pdf.

[99] Yeonhee Ryou and Sukyoung Ryu. "Automatic Detection of Visibility Faults by Layout Changes in HTML5 Web Pages". In: *Proceedings of the 11th International Conference on Software Testing, Validation and Verification*. 2018.

[100] Tõnis Saar, Marlon Dumas, Marti Kaljuve, and Nataliia Semenenko. "Cross-browser testing in browserbite". In: *International Conference on Web Engineering*. Springer. 2014, pp. 503–506.

[101] Tõnis Saar, Marlon Dumas, Marti Kaljuve, and Nataliia Semenenko. "Browserbite: cross-browser testing via image processing". In: *Software: Practice and Experience* 46.11 (2016), pp. 1459–1477. ISSN: 00380644. DOI: 10.1002/spe.2387. arXiv: 1008.1900. URL: http://doi.wiley.com/10.1002/spe.2387.

[102] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. "Automated repair of HTML generation errors in PHP applications using string constraint solving". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 277–287.

[103] *Sass: Syntactically Awesome Style Sheets*. URL: https://sass-lang.com/ (visited on 04/01/2022).

[104] *Screen Resolution Stats Worldwide*. 2022. URL: https://gs.statcounter.com/screen-resolution-stats#monthly-201202-202202 (visited on 03/25/2022).

[105] *Selenium: Web Browser Automation*. URL: http://www.seleniumhq.org/ (visited on 07/11/2018).

[106] *Selenium*. URL: https://www.selenium.dev/ (visited on 04/01/2022).

[107] Nataliia Semenenko, Marlon Dumas, and Tonis Saar. "Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features". In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 528–531. ISBN: 978-0-7695-4981-1. DOI: 10.1109/ICSM.2013.88. URL: http://ieeexplore.ieee.org/document/6676949/.

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

[109] Dennis Sheppard. *Beginning Progressive Web App Development: Creating a Native App Experience on the Web*. Apress, 2017.

[110] *Skeleton: Responsive CSS Boilerplate*. URL: http://getskeleton.com/ (visited on 03/23/2022).

[111] Ian Sommerville. *Software Engineering*. 9th ed. Addison-Wesley, 2011.

[112] Higor A de Souza, Marcos L Chaim, and Fabio Kon. "Spectrum-based software fault localization: A survey of techniques, advances, and challenges". In: *arXiv preprint arXiv:1607.04347* (2016).

[113] *StaticGen*. URL: https://www.staticgen.com/about (visited on 11/06/2018).

[114] Michael Tamm. *Fighting Layout Bugs*. 2009. URL: https://code.google.com/archive/p/fighting-layout-bugs/ (visited on 05/01/2018).

[115] Haruto Tanno and Yuu Adachi. "Support for finding presentation failures by using computer vision techniques". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2018, pp. 356–363.

[116] *The W3C CSS Validation Service*. URL: https://jigsaw.w3.org/css-validator/ (visited on 04/14/2022).

[117] *The W3C Markup Validation Service*. URL: https://validator.w3.org/ (visited on 04/14/2022).

[118] *The most advanced responsive front-end framework in the world - Foundation*. URL: https://get.foundation (visited on 03/23/2022).

[119] David Thomas and Andrew Hunt. *The Pragmatic Programmer: your journey to mastery*. 2nd ed. Addison-Wesley Professional, 2019.

[120] *TypeScript: JavaScript With Syntax For Types*. URL: https://www.typescriptlang.org/ (visited on 04/01/2022).

[121] W3C. *Web Design and Applications Standards*. URL: https://www.w3.org/standards/webdesign/ (visited on 05/07/2018).

[122] W3C. *World Wide Web Consortium (W3C)*. URL: https://www.w3.org/Consortium/ (visited on 05/07/2018).

[123] Kathleen Wahlbin. *Responsive web design*. A Book Apart, 2013. ISBN: 9780984442577.

[124] Thomas A Walsh, Gregory M. Kapfhammer, and Phil McMinn. "Automated Layout Failure Detection for Responsive Web Pages without an Explicit Oracle". In: *Proceedings of the International Conference on Software Testing and Analysis*. 2017.

[125] Thomas A Walsh, Gregory M. Kapfhammer, and Phil McMinn. "ReDeCheck: An Automatic Layout Failure Checking Tool for Responsively Designed Web Pages". In: *Proceedings of the International Conference on Software Testing and Analysis – Demonstration Papers*. 2017.

[126] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. "Automatically Identifying Potential Regressions in the Layout of Responsive Web Pages". In: *Software Testing, Verification and Reliability* 30.6 (2020).

[127] Thomas A Walsh, Phil McMinn, and Gregory M Kapfhammer. "Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages". In: *Proceedings of the 30th International Conference on Automated Software Engineering*. 2015.

[128] Thomas Walsh. "Automated Identification of Presentation Failures in Responsive Web Pages". PhD thesis. The University of Sheffield, Apr. 2018.

[129] Wenhua Wang, Sreedevi Sampath, Yu Lei, Raghu Kacker, Richard Kuhn, and James Lawrence. "Using combinatorial testing to build navigation graphs for dynamic web applications". In: *Software Testing, Verification and Reliability* 26.4 (2016).

[130] Mark Weiser. "Program slicing". In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357.

[131] James Q Wilson and George L Kelling. "Broken windows". In: *Atlantic monthly* 249.3 (1982), pp. 29–38.

[132] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. "A survey on software fault localization". In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[133] W Eric Wong and Yu Qi. "BP neural network-based effective fault localization". In: *International Journal of Software Engineering and Knowledge Engineering* 19.04 (2009), pp. 573–597.

[134] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[135] Shaopeng Xu, Chenyu Zhou, Zhiwei Gu, Guoquan Wu, Wei Chen, and Jun Wei. "X-Diag: Automated Debugging Cross-Browser Issues in Web Applications". In: *2018 IEEE International Conference on Web Services (ICWS)*. IEEE. 2018, pp. 66–73.

[136] Hector Yee, Sumanita Pattanaik, and Donald P. Greenberg. "Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments". In: *ACM Transactions on Graphics* 20.1 (2001), pp. 39–65. ISSN: 07300301. DOI: 10.1145/383745.383748. URL: http://portal.acm.org/citation.cfm?doid=383745.383748.

[137] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. "Multiple fault localization of software programs: A systematic literature review". In: *Information and Software Technology* 124 (2020), p. 106312.

[138] Frank Zammetti. *Practical Webix*. Apress, 2018, pp. 1–5. ISBN: 9781484233832.