

AUTOMATIC IDENTIFICATION OF PRESENTATION  
FAILURES IN RESPONSIVE WEB PAGES

THOMAS ANDREW WALSH

SUPERVISOR: DR PHIL MCMINN

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY  
APRIL 2018

THE UNIVERSITY OF SHEFFIELD  
FACULTY OF ENGINEERING  
DEPARTMENT OF COMPUTER SCIENCE



The  
University  
Of  
Sheffield.



---

## ABSTRACT

---

With the increasing number and variety of devices being used to access the World Wide Web, providing a good browsing experience to all users, regardless of device, is a critical task. To do this, many web developers now use responsive web design (RWD) to build web pages that provide a bespoke layout tailored to the specific characteristics of the device in use, normally the viewport width. However, implementing responsive web pages is an error-prone task, as web page elements can behave in unpredictable ways as the viewport expands and contracts. This leads to presentation failures — errors in the visual appearance of the web page. As well-designed responsive web pages can have an array of benefits, identifying presentation failures quickly and accurately is an important task.

Unfortunately, current approaches to detecting presentation failures in web pages are insufficient. The huge number of different viewport widths that require support makes thorough checking of the layout on all devices infeasible. Furthermore, the current range of developer tools only provide limited support for testing responsive web pages.

This thesis tackles these problems by making the following contributions. First, it proposes the responsive layout graph (RLG), a model of the dynamic layout of modern responsive web pages. Then, it explores how the RLG can be used to automatically detect potentially unseen side-effects of small changes to the source code of a web page. Next, it investigates the detection of several common types of layout failures, leveraging implicit oracle information in place of an explicit oracle. Experiments showed both the approach for detecting potentially unseen side-effects and the approach for identifying common types of layout failure to be highly effective. The manual effort required by the user is further reduced by an approach that automatically grouped related failures together. Finally, a case study of 33 real-world responsive layout failures investigates how difficult such failures are to fix. These approaches have all been implemented into a software tool, REDECHECK, which helps web developers create better responsive web pages.



---

## PUBLICATIONS

---

Various pieces of research presented in this thesis have been previously published or are currently under review in the following peer-reviewed venues:

- [1] 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 26th International Conference on Software Testing and Analysis*. 2017.
- [2] Thomas A Walsh, Gregory M. Kapfhammer, and Phil McMinn. “RE-DECHECK: An Automatic Layout Failure Checking Tool for Responsively Designed Web Pages.” In: *Proceedings of the 26th International Conference on Software Testing and Analysis (Demonstration Papers)*. 2017.
- [3] Thomas A Walsh, Gregory M Kapfhammer, and Phil McMinn. “Automatically Alerting Developers to the Unseen Side Effects of Incremental Changes to Responsive Web Pages.” In: *Software Testing, Verification and Reliability (Under Review)* (2018).
- [4] 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.



---

## ACKNOWLEDGMENTS

---

Firstly, I would like to thank my supervisor, Dr Phil McMinn, for his continued support, without which I would never have made the achievements detailed in this thesis. I would also like to thank Dr Gregory Kapfhammer, for his invaluable guidance and fantastic enthusiasm throughout the course of my studies. Their knowledge and expertise has given me the privilege of presenting the research contained within this thesis at two prestigious software engineering conferences.

I would also like to thank all the members of the Verification and Testing research group for providing an interesting and motivating environment in which to conduct my research, as well as providing welcome relief from the stresses and strains of completing a PhD.

Finally, I would like to thank my partner Charlotte and my family for their unwavering support, encouragement, understanding and patience throughout my postgraduate degree.





---

## CONTENTS

---

1	INTRODUCTION	1
1.1	Responsive Web Design . . . . .	2
1.2	The Problem: Presentation Failures in Responsive Web Pages . . . . .	3
1.3	Aims of this Thesis . . . . .	4
1.4	Organisation and Contributions of this Thesis . . . . .	4
2	LITERATURE REVIEW	9
2.1	Web Applications . . . . .	10
2.2	The Mobile Web . . . . .	12
2.2.1	Why Provide a Mobile-Friendly Web Site? . . . . .	13
2.2.2	Making Web Pages Mobile-Friendly . . . . .	15
2.3	Web Design Methodologies . . . . .	18
2.3.1	Adaptive Web Design . . . . .	20
2.3.2	Responsive Web Design . . . . .	21
2.3.3	Comparing AWD and RWD . . . . .	23
2.3.4	Mobile-First Design . . . . .	26
2.4	Detecting Presentation Failures in Web Pages . . . . .	27
2.4.1	Errors, Faults and Failures . . . . .	27
2.4.2	Strategies of Detecting Failures . . . . .	28
2.4.3	Tools for Identifying Web Presentation Failures . . . . .	32
2.5	Mobile Presentation Failures . . . . .	49
2.5.1	Difficulties . . . . .	50
2.5.2	Current Tools and Techniques . . . . .	51
2.5.3	Applying Previous Techniques . . . . .	55
2.6	General Web Application and GUI Testing . . . . .	56
2.7	Concluding Remarks . . . . .	60
3	MODELLING RESPONSIVE LAYOUT	63
3.1	The Responsive Layout Graph . . . . .	64
3.1.1	Formal Definition . . . . .	64
3.1.2	Extracting the Responsive Layout Graph . . . . .	68
3.2	Concluding Remarks . . . . .	77
4	DETECTING POTENTIALLY UNSEEN LAYOUT SIDE EFFECTS OF SMALL CODE CHANGES	79

4.1	The Problem . . . . .	80
4.2	Usage Scenario . . . . .	82
4.3	Comparing Two RLG's . . . . .	83
4.4	Empirical Evaluation . . . . .	87
4.4.1	Experimental Design . . . . .	87
4.4.2	Empirical Results . . . . .	100
4.4.3	Discussion . . . . .	112
4.5	Concluding Remarks . . . . .	113
5	<b>DETECTING COMMON TYPES OF RESPONSIVE LAYOUT FAILURES</b>	<b>115</b>
5.1	A Categorisation of Responsive Layout Failures . . . . .	117
5.2	Detecting Responsive Layout Failures without an Explicit Oracle	121
5.3	Empirical Evaluation . . . . .	131
5.3.1	Experiment Design . . . . .	131
5.3.2	Empirical Results . . . . .	139
5.4	Refining the RLF Detection Approach . . . . .	144
5.4.1	RLG Definition Modifications . . . . .	145
5.4.2	Improving the Small-Range Detection Algorithm . . . . .	146
5.4.3	Empirical Evaluation . . . . .	147
5.5	Concluding Remarks . . . . .	151
6	<b>GROUPING RELATED FAILURES TOGETHER</b>	<b>153</b>
6.1	Grouping Individual Reports into Distinct RLFs . . . . .	153
6.1.1	Grouping Approach . . . . .	154
6.1.2	Computing Similarity . . . . .	154
6.2	Empirical Evaluation . . . . .	158
6.2.1	Experiment Design . . . . .	158
6.2.2	Empirical Results . . . . .	162
6.3	Concluding Remarks . . . . .	177
7	<b>A STUDY OF THE ROOT CAUSES OF REAL-WORLD RLFs</b>	<b>179</b>
7.1	Study Design . . . . .	179
7.2	Patching the Failures . . . . .	181
7.2.1	Element Collision Failures . . . . .	181
7.2.2	Element Protrusion Failures . . . . .	186
7.2.3	Viewport Protrusion Failures . . . . .	189
7.2.4	Small-Range Layout Failures . . . . .	194
7.2.5	Wrapping Failures . . . . .	195
7.2.6	Summary of Root Causes . . . . .	201
7.3	Study Results . . . . .	201
7.3.1	Discussion . . . . .	205

7.4	Concluding Remarks . . . . .	208
8	CONCLUSIONS AND FUTURE WORK	209
8.1	Summary of Achievements . . . . .	209
8.1.1	The Responsive Layout Graph . . . . .	209
8.1.2	Detecting Potentially Unseen Layout Side-Effects . . . . .	210
8.1.3	Detecting Common Types of Responsive Layout Failures . . . . .	211
8.1.4	Grouping Related Failures Together . . . . .	212
8.1.5	A Study of Real-World RLFs . . . . .	212
8.2	Limitations and Future Work . . . . .	213
8.2.1	Further Investigation of Web Page Mutants . . . . .	213
8.2.2	Identification of Further Types of RLF . . . . .	213
8.2.3	Automatic Classification of Identified RLFs . . . . .	214
8.2.4	Fault Localisation . . . . .	214
8.2.5	Automatic Fault Fixing . . . . .	215
8.3	Final Remarks . . . . .	215
9	APPENDIX	217
9.1	Screenshots for Part One of the Human Study . . . . .	217
9.2	Screenshots for Part Two of the Human Study . . . . .	222
	BIBLIOGRAPHY	245

---

## LIST OF FIGURES

---

Figure 1.1	An example of a responsive web page. . . . .	2
Figure 1.2	An overview of the research path taken in this thesis. . .	5
Figure 2.1	A presentation failure in a web page, where two elements are overlapping. . . . .	9
Figure 2.2	The three-layer architecture of a web application. . . . .	10
Figure 2.3	Flow of information in a three-layer architecture. . . . .	11
Figure 2.4	The BlackTie “Shield” site ( <a href="http://www.blacktie.co/demo/shield">http://www.blacktie.co/demo/shield</a> ), showing how the layout of a responsive page adjusts to different viewport widths. . . . .	19
Figure 2.5	A flexible grid and its CSS declarations. . . . .	22
Figure 2.6	A hierarchy of presentation failure detection techniques. . . . .	32
Figure 2.7	The inputs to WEBSEE. . . . .	33
Figure 2.8	An example of a Layout Graph. . . . .	38
Figure 2.9	An example of an Alignment Graph. . . . .	43
Figure 2.10	An example of CORNIPICKLE’s formal specification. . . . .	46
Figure 2.11	An example of Galen’s selection and specification syntax. . . . .	48
Figure 2.12	RESPONSIVEPX, an example of a viewport resizer. . . . .	54
Figure 2.13	Kersley’s Screenshot Tool . . . . .	55
Figure 3.1	A wireframe example web page and its RLG. . . . .	65
Figure 3.2	The two-part sampling process. . . . .	69
Figure 3.3	An example web page and its extracted layout. . . . .	71
Figure 4.1	A mock-up of a responsive web page shown at three different resolutions. . . . .	81
Figure 4.2	The main usage scenario of this chapter’s approach. . . . .	83
Figure 4.3	The RLG for the web page shown in Figure 4.1 . . . . .	85
Figure 4.4	A snippet of a report produced for the example in Figure 4.1. . . . .	87
Figure 4.5	High level structure of REDeCHECK. . . . .	90
Figure 4.6	Examples of the two mutation operators for CSS declarations. . . . .	92
Figure 4.7	Examples of the two mutation operators for media queries. . . . .	

Figure 4.8	The high-level structure of the automatic code modification approach. . . . .	93
Figure 4.9	The manual procedure for determining if a modified web page contains a layout change. . . . .	94
Figure 4.10	The number of viewport widths sampled by REDECHECK-INTERVAL and REDECHECK-COMBINED. . . . .	106
Figure 4.11	The percentage difference between the number of sampled viewport widths for REDECHECK-COMBINED and REDECHECK-INTERVAL. . . . .	107
Figure 4.12	A breakdown of the number of viewport widths sampled initially, and then during binary searches to complete the RLG, shown in dark blue and light blue, respectively. . . . .	110
Figure 5.1	An example of an element collision. . . . .	117
Figure 5.2	An example of an element protrusion. . . . .	118
Figure 5.3	An example of a viewport protrusion. . . . .	119
Figure 5.4	An example of a small-range layout. . . . .	120
Figure 5.5	An example of a wrapping failure. . . . .	120
Figure 5.6	A wireframe example of an element collision. . . . .	122
Figure 5.7	A wireframe example of an element protrusion. . . . .	123
Figure 5.8	Example of an viewport overflow failure, and its corresponding RLG fragment. . . . .	124
Figure 5.9	Example of a small-range failure, and its corresponding RLG fragment. . . . .	127
Figure 5.10	Example of a wrapping failure, and its corresponding RLG fragment. At narrow viewport widths, elements intended to align next to each other on a single row wrap to produce a second row. . . . .	129
Figure 5.11	The high-level structure of the REDECHECK tool. To the left of the dashed vertical line, the modules support regression checking, to the right, common failure checking. . . . .	133
Figure 5.12	An example report screenshot from REDECHECK. . . . .	134
Figure 5.13	A false positive small-range layout. . . . .	140
Figure 5.14	A false positive wrapping failure. . . . .	140
Figure 5.15	A non-observable element protrusion. . . . .	141
Figure 5.16	Execution times for REDECHECK . . . . .	144
Figure 6.1	An example response form for part one of the human study. . . . .	160

Figure 6.2	An example response form for part two of the human study. . . . .	161
Figure 6.3	Responses for Part One. . . . .	167
Figure 6.4	Responses for Part Two. . . . .	173
Figure 6.5	Responses for Exit Survey. . . . .	176
Figure 7.1	Fixing the Cloudconvert element collision failure. . . . .	182
Figure 7.2	Fixing the MidwayMeetup element collision failure. . . . .	183
Figure 7.3	Fixing the PepFeed element collision failure at narrow widths. . . . .	183
Figure 7.4	Fixing the PepFeed element collision failure at wide widths. . . . .	184
Figure 7.5	Fixing the StumbleUpon element collision failure. . . . .	185
Figure 7.6	Fixing the Will My Phone Work element collision failure. . . . .	186
Figure 7.7	Fixing the <i>BugMeNot</i> element protrusion failure. . . . .	186
Figure 7.8	Fixing the ConsumerReports element protrusion failure. . . . .	187
Figure 7.9	Fixing the PDFescape element protrusion failure. . . . .	188
Figure 7.10	Fixing the 3-Minute Journal viewport protrusion failure. . . . .	189
Figure 7.11	Fixing the ConsumerReports viewport protrusion failure in the banner. . . . .	190
Figure 7.12	Fixing the ConsumerReports viewport protrusion failure in “Featured Products”. . . . .	191
Figure 7.13	Fixing the ConsumerReports viewport protrusion failure in the footer. . . . .	192
Figure 7.14	Fixing the Duolingo viewport protrusion failure. . . . .	192
Figure 7.15	Fixing the Hotel WiFi Test viewport protrusion failure. . . . .	193
Figure 7.16	Fixing the PDFescape viewport protrusion failure. . . . .	194
Figure 7.17	Fixing the Accountkiller small-range layout failure. . . . .	195
Figure 7.18	Fixing the Accountkiller wrapping failure. . . . .	196
Figure 7.19	Fixing the Accountkiller wrapping failure in the footer. . . . .	196
Figure 7.20	Fixing the Airbnb wrapping failure with “Terms & Privacy”. . . . .	197
Figure 7.21	Fixing the Airbnb wrapping failure with the Instagram logo. . . . .	197
Figure 7.22	BugMeNot viewed at a viewport width of 500 pixels. . . . .	198
Figure 7.23	Fixing the CoveredCalendar viewport protrusion failure in the header. . . . .	198
Figure 7.24	Fixing the CoveredCalendar wrapping failure in the footer. . . . .	199
Figure 7.25	Fixing the Ninite wrapping failure. . . . .	200

Figure 7.26	Fixing the PepFeed wrapping failure. . . . .	200
Figure 7.27	Fixing the Usersearch wrapping failure. . . . .	201
Figure 9.1	Part One - Question One . . . . .	217
Figure 9.2	Part One - Question Two . . . . .	218
Figure 9.3	Part One - Question Three . . . . .	218
Figure 9.4	Part One - Question Four . . . . .	219
Figure 9.5	Part One - Question Five . . . . .	220
Figure 9.6	Part One - Question Six . . . . .	220
Figure 9.7	Part One - Question Seven . . . . .	221
Figure 9.8	Part One - Question Eight . . . . .	221
Figure 9.9	Part One - Question Nine . . . . .	221
Figure 9.10	Part One - Question Ten . . . . .	221
Figure 9.11	Part Two - Group One - Failure One . . . . .	222
Figure 9.12	Part Two - Group One - Failure Two . . . . .	223
Figure 9.13	Part Two - Group One - Failure Three . . . . .	223
Figure 9.14	Part Two - Group One - Failure Four . . . . .	224
Figure 9.15	Part Two - Group One - Failure Five . . . . .	224
Figure 9.16	Part Two - Group One - Failure Six . . . . .	225
Figure 9.17	Part Two - Group Two - Failure One . . . . .	226
Figure 9.18	Part Two - Group Two - Failure Two . . . . .	226
Figure 9.19	Part Two - Group Two - Failure Three . . . . .	226
Figure 9.20	Part Two - Group Three - Failure One . . . . .	227
Figure 9.21	Part Two - Group Three - Failure Two . . . . .	227
Figure 9.22	Part Two - Group Three - Failure Three . . . . .	227
Figure 9.23	Part Two - Group Four - Failure One . . . . .	228
Figure 9.24	Part Two - Group Four - Failure Two . . . . .	228
Figure 9.25	Part Two - Group Four - Failure Three . . . . .	228
Figure 9.26	Part Two - Group Five - Failure One . . . . .	229
Figure 9.27	Part Two - Group Five - Failure Two . . . . .	230
Figure 9.28	Part Two - Group Five - Failure Three . . . . .	230
Figure 9.29	Part Two - Group Six - Failure One . . . . .	231
Figure 9.30	Part Two - Group Six - Failure Two . . . . .	231
Figure 9.31	Part Two - Group Six - Failure Three . . . . .	232
Figure 9.32	Part Two - Group Seven - Failure One . . . . .	233
Figure 9.33	Part Two - Group Seven - Failure Two . . . . .	233
Figure 9.34	Part Two - Group Seven - Failure Three . . . . .	233
Figure 9.35	Part Two - Group Eight - Failure One . . . . .	234
Figure 9.36	Part Two - Group Eight - Failure Two . . . . .	234

Figure 9.37	Part Two - Group Eight - Failure Three . . . . .	235
Figure 9.38	Part Two - Group Nine - Failure One . . . . .	236
Figure 9.39	Part Two - Group Nine - Failure Two . . . . .	236
Figure 9.40	Part Two - Group Nine - Failure Three . . . . .	236
Figure 9.41	Part Two - Group Ten - Failure One . . . . .	237
Figure 9.42	Part Two - Group Ten - Failure Two . . . . .	238
Figure 9.43	Part Two - Group Ten - Failure Three . . . . .	239
Figure 9.44	Part Two - Group Ten - Failure Four . . . . .	240
Figure 9.45	Part Two - Group Ten - Failure Five . . . . .	241
Figure 9.46	Part Two - Group Ten - Failure Six . . . . .	242
Figure 9.47	Part Two - Group Ten - Failure Seven . . . . .	243

---

## LIST OF TABLES

---

Table 4.1	Responsive web pages used in the empirical study. . . . .	89
Table 4.2	Descriptions and examples of the incremental code mutation operators used. . . . .	91
Table 4.3	Results summarising how REDeCHECK, SPOTCHECK-AG and SPOTCHECK-MANUAL detect the incremental code modifications. . . . .	101
Table 4.4	The effectiveness of REDeCHECK and SPOTCHECK-AG at detecting layout changes that vary according to how “subtle” they are. . . . .	103
Table 4.5	REDeCHECK’s effectiveness at detecting layout changes at various step sizes. . . . .	105
Table 4.6	The statistical hypothesis test results. . . . .	111
Table 5.1	Web Pages Used in the Empirical Study . . . . .	132
Table 5.2	Results for the RLF Detection Approach. . . . .	138
Table 5.3	Results of the spot-checking process. . . . .	143
Table 5.4	Additional Web Pages added to the Empirical Study . . . . .	147
Table 5.5	Failure Detection Results following the modifications to the approach. . . . .	148



Table 5.6	Failure Detection Results on the additional pool of subjects. . . . .	149
Table 6.1	Failure Type Similarity . . . . .	156
Table 6.2	Grouping results for all web pages containing at least one RLF. . . . .	163
Table 6.3	Summary of the results for part one of the study. . . . .	170
Table 6.4	Summary of the results for part two of the study. . . . .	175
Table 7.1	Summary statistics for the patches implemented for the various failures detected. The labels A, D and C represent additions, deletions and changes respectively. The numbers in parentheses represent the number of individual CSS declarations that were added, deleted or modified. . . . .	204





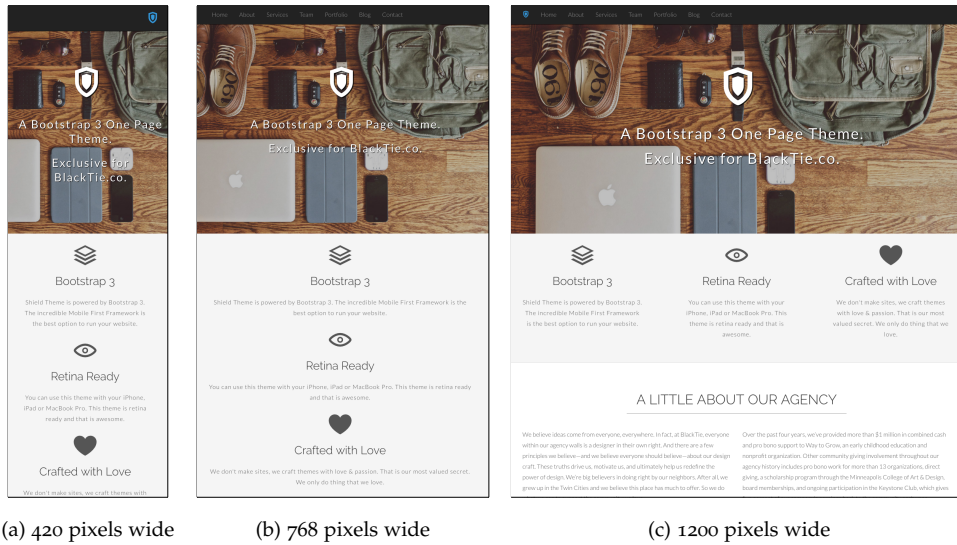
---

## INTRODUCTION

---

The World Wide Web is an integral part of our everyday lives and its influence continues to grow. Billions of people now use it to communicate with friends and family, do their shopping or run their business. As the web has developed over the last twenty years, so has the technology with which one can access it. Not so long ago, a desktop computer was the only device capable of accessing the web. Nowadays, multiple different device types are web-enabled, including “mobile devices” such as smartphones, tablets and even watches. While this ease of access is hugely convenient for users, it presents a substantial problem for web developers. Traditionally, they only had to make their web pages usable on desktop or laptop computers, which all have similar screen dimensions. However, all of these new mobile devices have screens of different sizes and resolutions. The question is, how do developers build “mobile-friendly” web pages that look good and are easy to browse on all devices?

By default, most mobile devices respond to a non-mobile-friendly web page by simply shrinking it down until it fits within the screen of the device in use. This makes many web pages completely unusable, so research has investigated ways of making web pages mobile-friendly. However, despite early promise, it soon became apparent that developers had to start developing with mobile-friendliness in mind. One popular approach was adaptive web design (AWD), which involved developing multiple different versions of the web site optimised for different types of device. However, a better approach was soon required as the number and variety of web-enabled devices continued to grow. This approach is **responsive web design**.



**Figure 1.1:** An example of a responsive web page.

## 1.1 RESPONSIVE WEB DESIGN

Responsive web design, or RWD for short, is a hugely popular design approach for building modern web pages. It was first proposed by Ethan Marcotte in 2010 [95] and has grown in popularity ever since. In 2012, it was listed as the #2 trend in web design and development and W3C have called it “a must for tablets and mobile devices” [1].

The key idea underpinning RWD is that developers need only build a single version of a web page that “responds” to the constraints of the device being used. It does this by both resizing and rearranging the content, with the aim of providing an equivalent user experience on all devices. By doing this, RWD essentially tailors a bespoke layout to every single device, meaning the layout provided should be the best it can be. This also makes responsive web sites effectively future-proof, as it simply creates a layout specific to any new device released onto the market. This is in stark contrast to AWD, which not only does not provide an optimal layout to every device but also is susceptible to new devices not working well with any of the pre-existing layouts.

Figure 1.1 presents an example of a responsive web page. At drop-down list and the main content panels are given narrow widths in a column. At the medium viewport of part (b), the navigation links have expanded to a single row and the content panels remain stacked, albeit it with wider widths. Finally, at the wide viewport width in part (c), the main content panels are made narrower

again but are displayed side-by-side rather than in a column, making best use of the increased space.

Responsive web design is based on three main “ingredients”: *grid-based layouts*, *flexible media* and *media queries*. The first two ensure all the elements on the web page are displayed at a size suitable for the device in use. Meanwhile, media queries allow developers to apply different styling rules to the web page depending on the device’s characteristics. When applied correctly, they combine to produce web pages that provide the optimal browsing experience, regardless of device.

## 1.2 THE PROBLEM: PRESENTATION FAILURES IN RESPONSIVE WEB PAGES

Unfortunately, due to the innovative nature of RWD, the code required to implement a responsive web page is often quite complex. Many web pages have a large amount of content and an equally substantial number of styling rules. Developers have to be careful that the correct style rules are applied to the correct elements at the correct viewport widths. If they aren’t, the web page’s layout can often behave peculiarly as the viewport expands and contracts, leading to unwanted aesthetic issues. These are called **presentation failures** [89, 90] and they can have serious detrimental effects on a web page.

Identifying these presentation failures and fixing them as quickly as possible is a critical task for developers. Well designed error-free responsive web pages have been shown to have positive psychological influences on end users. For instance, two thirds of surveyed smartphone users said they would be more likely to make a purchase on a mobile-friendly site [67] and visible failures are likely to cause users to stop purchasing from a particular site [86]. Furthermore, studies have observed increased perceived usability [66, 84, 107] and improved loyalty [40].

Unfortunately, detecting presentation failures in web pages is currently a difficult task. They often occur at unpredictable and sometimes small numbers of viewport widths, meaning the web page may not be examined at one of the faulty widths. Furthermore, as many modern web pages contain large amounts of content, smaller aesthetic issues can be harder to spot as developers may instead focus on larger issues.

The myriad of different devices and screen sizes currently poses a significant problem, as testing a web page on all possible devices would be extremely labour-intensive. Therefore, developers and testers frequently employ “spot-checking”, which involves testing the layout of a web page at a small subset of viewport widths, often corresponding to popular devices. However, this is generally insufficient and therefore presentation failures continue to make their way into production web pages. The current range of techniques and tools available to developers of responsive web sites also offer only limited support. Like spot-checking, they are both labour-intensive and error-prone as they require a user to manually inspect the web page under test to observe any presentation failures that may be present.

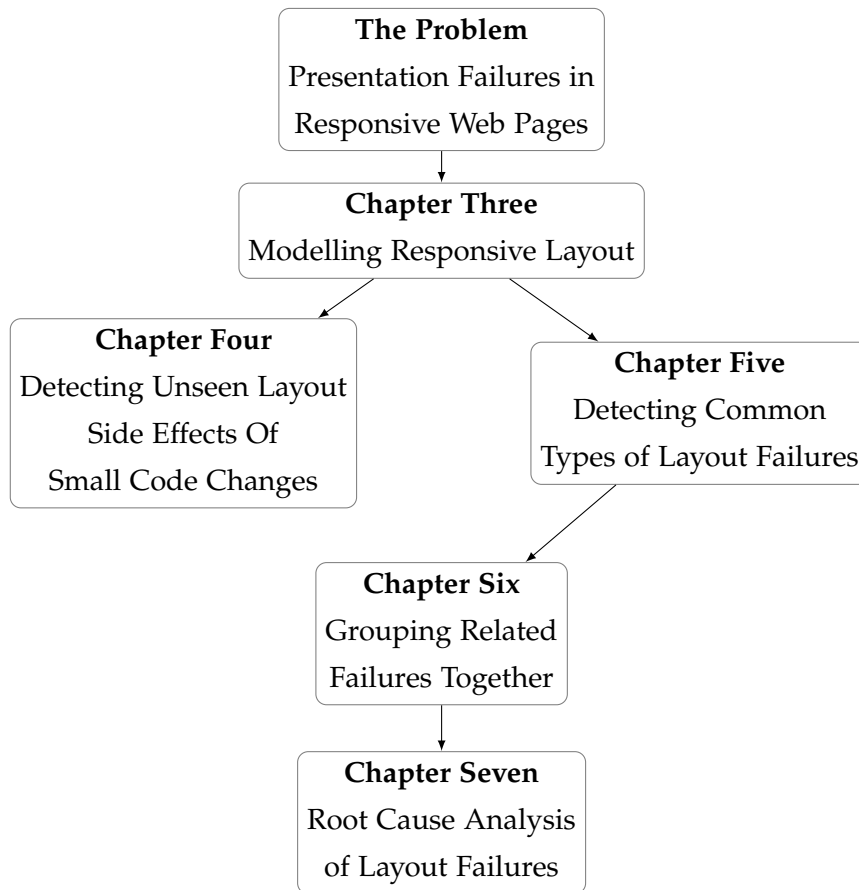
### 1.3 AIMS OF THIS THESIS

This thesis is concerned with automatically identifying presentation failures in responsive web pages with minimal effort from the developer. This goal is comprised of two main aims:

1. Develop a model for representing the dynamic visual nature of modern responsive web pages.
2. Explore the use of this model as a means of identifying a variety of presentation failures in real-world web pages.

### 1.4 ORGANISATION AND CONTRIBUTIONS OF THIS THESIS

In response to the importance of the problem of presentation failures and the shortcomings of current testing techniques, this thesis is concerned with developing automated methods for identifying a variety of presentation issues in responsive web pages. This section describes the path of research taken throughout this thesis. Figure 1.2 shows this path visually. The thesis begins with the identification of the core problem — presentation failures in responsive web pages— and a discussion of current options for tackling this problem, which constitutes the literature review in Chapter 2. As Chapter 2 finds that accurate automatic detection of presentation failures in mobile web applications is an open, unsolved problem, Chapter 3 introduces and defines the responsive layout graph (RLG), a model of a web page’s dynamic layout across a wide range



**Figure 1.2:** An overview of the research path taken in this thesis.

of viewport widths. As developers can often introduce unintended side-effects into the layout of a web page, Chapter 4 presents an approach that uses the RLG as a means of detecting potentially unseen side effects of code changes to a web page. Chapter 5 then tackles the problem of detecting common types of mobile presentation failures, such as overlapping or protruding elements, when an oracle describing the correct layout of the web page is unavailable. Chapter 6 presents a technique to group related failures together. Finally, Chapter 7 investigates the root causes of the failures detected in Chapter 5.

**Chapter 2 : “Literature Review”** This chapter reviews the literature relevant to the problem of detecting presentation failures in mobile web applications. It begins by introducing the mobile web and the challenges it presents to developers of web sites. Then, it presents the various approaches to developing so called “mobile-friendly” web applications. Next, the chapter introduces the concept of presentation failures and explains the nature of testing the appearance of web applications, before reviewing previous approaches addressing presentation is-

sues in web pages. Finally, the chapter concludes with a brief review of web and GUI testing in general.

**Chapter 3 : “Modelling Responsive Layout”** This chapter presents a model of a web page’s dynamic responsive layout, called the **responsive layout graph (RLG)**. Building on concepts introduced by Choudhary et al. in their alignment graph [123], it models both the changing visibility and relative alignments of web elements. This is the first model of responsive layout and was published in the paper “*Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages*” [149]. The chapter then presents a series of algorithms that can be used to extract the RLG of a given web page.

The key contributions made in this chapter are:

1. The formal definition of a model of a web page’s responsive layout behaviour, called the **responsive layout graph (RLG)**, which describes both the visibility and relative alignment of the elements on the web page across a specified range of viewport widths.
2. A series of algorithms to automatically extract the RLG of a web page.

**Chapter 4 : “Detecting Potentially Unseen Layout Side Effects of Small Code Changes”** When making small incremental changes to the layout of responsive web pages, developers can easily introduce unintended layout changes that detrimentally impact the quality of the web site. Therefore, this chapter presents a technique that compares the RLGs of two consecutive versions of a web page to report a list of potentially unseen layout side-effects to the developer.

The chapter then evaluates the approach using 15 responsive web pages, which were randomly modified by a suite of 8 mutation operators. The experiments show the proposed technique is highly accurate at detecting these layout changes and outperforms both manual and automated baseline approaches. They also show the approach is especially advantageous when detecting “subtle” changes visible at few viewport widths. Finally, the last experiment investigates the effectiveness and efficiency trade-offs of the approach and suggests a set of configuration parameters that should perform well on a wide variety of subject web pages. An initial, smaller version of this experimental study was published along with the RLG definition in “*Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages*” [149], while the full study is currently under review as part of the journal paper “*Automatically Alerting Developers to the Unseen Side Effects of Incremental Changes to Responsive Web Pages*” [148].



The key contributions of this chapter are:

1. An algorithm for comparing two RLG models which outputs a list of model differences representing potentially unintended layout issues, which along with the RLG extraction algorithms described in Chapter 3 has been implemented into a prototype software tool called REDECHECK.
2. A code mutation framework for HTML and CSS, targeting rules affecting web page layout that can be used to create modified versions of a subject web page.
3. A thorough empirical study evaluating the ability of the proposed approach to detect potential layout issues following modifications to a responsive web page and its efficiency.

**Chapter 5 : “Detecting Common Types of Responsive Layout Failures”** This chapter first introduces a categorisation of five different types of layout failures in responsive web pages. It then presents a collection of four algorithms that query the RLG of the web page under test to detect these failures without the need for an explicit oracle. The chapter then evaluates the approach on a corpus of 26 randomly selected responsive web pages, which shows that not only are layout failures prevalent in real-world web pages, but also that the proposed approach is highly effective at detecting them. This approach - the first to identify specific types of responsive layout failure without an explicit oracle - was published in the paper “ *Automated Layout Failure Detection for Responsive Web Pages without an Explicit Oracle*” [147]. The chapter concludes by describing several small modifications to the RLF identification technique to improve its accuracy and precision.

The key contributions of this chapter are:

1. A categorisation of five common types of *responsive layout failure (RLF)* discoverable without the need for explicit oracles.
2. A collection of four algorithms that automatically analyse the RLG of a web page in order to detect the five types of RLF. These algorithms have been implemented as a module of REDECHECK.
3. An empirical evaluation of 26 randomly selected production web pages, showing that the RLF types identified are prevalent in live sites and the algorithms are capable of detecting them and reporting them to the developer.

4. Modifications to the RLG model and failure detection algorithms that reduce the quantity of misleading false positive results reported by the approach.

**Chapter 6 : “Grouping Related Failures Together”** As the task of grouping related responsive layout failures together is currently manual, it can be labourious when many layout failures are identified. This chapter therefore begins by describing an automated technique for grouping them together, further removing the burden of effort from the user. While other approaches have grouped issues in web pages together, this is the first approach that does so with a focus on responsive layout failures. It then presents with an empirical evaluation showing that the grouping technique is effective and that human web users generally agree with both the failures reported and groupings produced by the approach.

The key contributions of this chapter are:

1. An automated approach for grouping related RLFs together.
2. An empirical evaluation on a large collection of real-world responsive web pages, showing the effectiveness of the grouping approach, and that humans generally agree with the reported failures and the clusterings produced.

**Chapter 7 : “A Study of Root Causes of Real-World RLFs”**

This chapter presents potential fixes for each of the RLFs detected by the approach in Chapter 5. These are obtained by identifying the “root cause” of each failure and then manually modifying the source code of the web page to fix them. It then presents an evaluation showing the vast majority of failures can be fixed with very few lines of code and discusses some common mistakes made by the developers of responsive web pages. This investigative case study forms the key contribution of the chapter. To the best of my knowledge, it is the first case study of responsive web pages, as previous studies have focussed on cross-browser incompatibilities [123] and more general presentation failures [62].

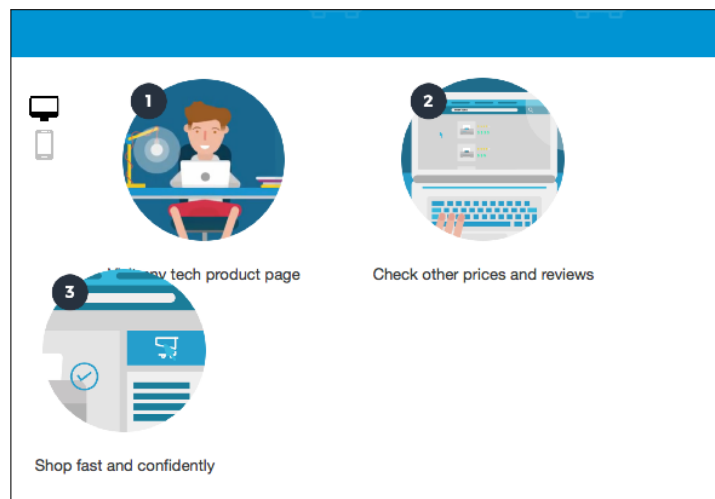
**Chapter 8 : “Conclusions and Future Work”** This final chapter summarises the work presented in this thesis. It then identifies several potential directions for future research, including approaches for improving the REDECHECK responsive web testing tool presented in this thesis.

---

## LITERATURE REVIEW

---

This chapter begins with an introduction to web applications in general. It then focusses in more detail on the emergence of “mobile-friendly” web applications which provide a good browsing experience to users on devices of all sizes. It then goes on to discuss the array of techniques proposed to make desktop-only web pages more mobile-friendly. Next, it describes various design approaches for creating web pages which perform well on a wide variety of devices, before introducing the concept of presentation failures, which are errors in the appearance of an application. The web page in Figure 2.1 presents an example of such a failure. It also covers how developers can detect such failures in web applications.

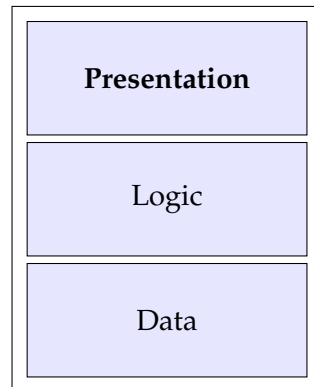


**Figure 2.1:** A presentation failure in a web page, where two elements are overlapping.

Next, it reviews the current body of literature aiming to automatically identify presentation issues in web applications. This review of previous work establishes a gap in the field concerning the detection of presentation failures in mobile-friendly web pages, the core problem addressed in this thesis. Finally,

the chapter concludes with a discussion of research addressing other types of web application testing, graphical user interface (GUI) testing and mobile application testing.

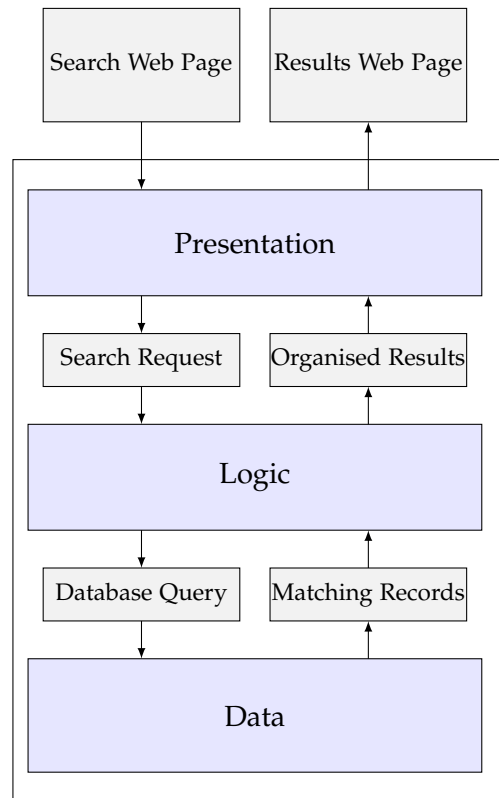
## 2.1 WEB APPLICATIONS



**Figure 2.2:** The three-layer architecture of a web application.

Modern software applications are generally separated into a number of distinct tiers. This structure is called the *n-tier architecture*, in which each tier handles a specific part of the application and is physically separate from the others [23]. Modern web applications typically consist of three tiers and Figure 2.2 presents this high-level architecture [48]. The bottom tier, the data tier, stores and manages access to the data used by the application. This usually takes the form of a *database management system (DBMS)* such as MySQL, MongoDB or PostgreSQL. Next is the application logic tier, responsible for all the major functionality of the application. This tier uses an application server and is normally written in a language such as Java, C#, Ruby or Python. Finally, the presentation tier is the top-most tier and handles the rendering of the application in the user's web browser. In a web application, this takes the form of web pages written in *hypertext markup language (HTML)*, *cascading style sheets (CSS)* and *JavaScript* — the three core components of modern web pages. HTML describes the structure of a web page, CSS applies visual styles to that content, and JavaScript allows developers to make web pages interactive [142, 143]. To explain how the individual layers interact with one another to form a cohesive application, this section now presents a simple example.

Let's consider a user is using an online shopping application and wants to search for a specific category of products. They input their search query into a



**Figure 2.3:** Flow of information in a three-layer architecture.

field on a web page presented to the user by the presentation tier. The input is then passed to the application logic tier, which converts the input into a search query and passes it down to the database tier. This tier then executes the query and returns the list of matching items to the application tier, which formats it. The presentation tier then creates a web page showing these formatted results. Finally, the application stack returns this web page to the user's web browser which renders it for them to see. This example, shown by Figure 2.3, illustrates how it is impossible for the presentation tier to communicate directly with the data tier and that the presentation tier is the only one the user can actually see and interact with. Instead, all traffic must pass through the logic tier of the application. This is in contrast to the widely used *model-view-controller* (MVC) architecture, in which the communication flow is triangular, with all three tiers communicating with each other [135]. However, the two approaches are not mutually exclusive. For instance, many web applications use the three-tier architecture for the overall architecture while using MVC in the presentation tier. As this thesis addresses the problem of presentation issues in web applications, the remainder of this literature review primarily focusses on topics related to the presentation tier of the web application stack. Next, it introduces the mobile

web, the problems associated with building mobile web applications and the various approaches proposed for doing so.

## 2.2 THE MOBILE WEB

The mobile web has existed in a limited form since the early 21st century. Originally, a “mobile device” was “a handheld device that is made for portability” with the first mass-market types being traditional mobile phones and personal data assistants [136]. Its recognized emergence only took place in 2007 with the launch of the iPhone and other multitouch smartphones. These are phones with displays which can sense input from multiple points of contact simultaneously [137]. It began to grow at an exponential rate with the introduction of tablet computers such as the iPad in 2010. Both smartphones and tablets provide a far superior web browsing experience to users than any previous generation of mobile device. Because of this, developers face an important problem. How can they make their web pages easily browsable on these next generation devices?

Initially, when developing the presentation tier of their web applications, the vast majority of developers simply “shrank down” the desktop version of a web page to fit the screens of mobile devices. Despite the significant improvements provided by smartphones and other mobile devices for users browsing the web, the end result was still a very poor user experience. It took considerable effort to complete straightforward tasks like navigating around a web site or browsing its content. This was due mainly to limitations brought about by the substantial reduction in screen size. These limitations frequently resulted in severe issues. Firstly, when a mobile browser rendered a desktop site, most if not all of the text was far too small to be easily readable. This forced the user to constantly zoom in and out to make the text appear at a reasonable size and read it comfortably. The reduction in screen size also often resulted in content overflowing the device’s viewport window. This meant users had to scroll to access all of the page. Clickable links were also regularly drawn too small and too close together, as evidenced in a study which found 44% of respondents found navigation difficult on mobile devices [73]. These behavioural traits may cause no issue on a desktop when the user had access to a larger screen and a mouse for navigation. However, browsing on a mobile device requires a user to use their fingertip. This is a much less precise form of computer interaction and can often lead to problems. For example, the user could accidentally select the

wrong link, which is hugely frustrating. It could also be potentially dangerous, if a user inadvertently selects a link providing functionality such as a “Buy with One-Click” feature.

Web sites which provide a good browsing experience to users accessing them from a mobile device are referred to as *mobile-friendly*. The issues discussed above are just three of many prevalent problems when trying to view a non mobile-friendly web site on a mobile device. To highlight the frequency of these issues, a survey found 46% of respondents had experienced problems when viewing a website that was not mobile-friendly [73]. With all of these issues, it is very surprising to observe the lack of “mobile-readiness” of many high profile websites. For instance, the work of McCorkindale and Morgoch [98] found the majority of Fortune 500 companies do not have a mobile-friendly site. While this may have changed since the study’s publication in 2013, it is still a fairly alarming statistic. These organisations are likely to have a huge number of users, many of whom may do the bulk of their browsing on mobile devices. This begs the question, if so many highly valued multinational companies are providing their customers with a desktop web site only, what reason is there for smaller organisations to invest significant time, effort and money into implementing mobile-friendly web sites? The next section presents and discusses a collection of studies highlighting various reasons.

### 2.2.1 *Why Provide a Mobile-Friendly Web Site?*

The motivating factors for web developers to implement mobile-friendly layouts in their web sites are varied. Some are purely business-related, while others focus on facets of human psychology. The first and most likely the strongest motivation is the colossal rise in the amount of web traffic attributed to mobile devices. Statistics for the mobile share of organic search engine visits in the United States show mobile usage has almost doubled in the last 4 years, from 27% in the third quarter of 2013 to 53% in the first quarter of 2017 [164]. This suggests if developers choose not to implement a mobile-friendly layout, they are in effect alienating more than half of their potential customer base.

Providing a mobile-friendly web site is also vital for search engine optimisation. This process involves making a web site appear as high as possible on the list of ranked results supplied to a user by a search engine. Thus, obtaining a high ranking can help to increase the flow of users to a site. An update

by Google, dubbed “Mobile-geddon” by various media sources [27, 28, 39, 68, 134], now means that when the search originates from a mobile device, the algorithm assigns lower rankings to web sites which are *not* mobile-friendly. This effectively gives well designed mobile-friendly sites a rankings boost. This in turn improves the browsing experience for the user, as they are more likely to navigate to an easily usable web page. Google’s mobile-friendly testing tool is responsible for determining these new rankings [35]. It checks several important usability criteria to determine if a site is mobile-friendly. These include whether a user has to zoom to read text or scroll horizontally to view all parts of the page and whether navigational links are far enough apart to be easily selectable using a finger tip. Despite considerable publicity regarding the update, many important websites, such as Nintendo, MI5, the European Union and even the British Monarchy, “failed” this test when Google introduced the update. They therefore ran the risk of suffering a drop in ranking and a possible reduction in visitors and business [134]. This likely resulted in a trend of businesses investing in mobile-friendly web sites in order to restore their ranking and mitigate the potential losses. For instance, less than a year after the update, companies including Next, Dyson and the Daily Mail had upgraded their web sites to ensure they were mobile-friendly [134].

Once the user has navigated to a particular web page, its mobile-friendliness continues to be an important factor in its success. For instance, a 2012 study of 1,100 adult smartphone users found that 67% of respondents stated they would be more likely to purchase a product or service if its web site was mobile-friendly and easy to use [67]. Furthermore, Li et al. [86] found that the presence of visible failures in the appearance of a web site was likely to cause users to stop purchasing products from that particular web site. Obviously, this demonstrates the importance of providing a good user experience, as not doing so can have severely detrimental economic impacts on the organisation responsible for the site and its ongoing strategic and financial goals.

There are more psychological effects which should motivate organisations to invest in a mobile-friendly web site. For instance, Hartmann et al. [66] researched the attractiveness of user interfaces on the web. They found mobile-friendly layouts caused users to perceive a higher degree of usability. Similarly, a study by Lee et al. [84] determined the layout of a web page to be the key determinant in its usability. Wu et al. [156] found the layout to be one of the main factors in the web page’s visual quality, and Michailidou et al. [107] found a strong correlation between the visual appearance of a web page and the level



of usability perceived by end users. These studies emphasised the importance of providing an easily usable layout to users, regardless of the device they are using. Mbipom et al. [97] also showed pleasing aesthetics to be more accessible for visually impaired users.

Polished aesthetics in a web page can also lead to higher credibility for the web page and the organisation to whom it belongs [122]. A web site is often the first interaction a user has with a company and this impression can be made in the first few seconds. This can in turn lead to increased company profitability, especially for e-commerce sites. Good web page layouts can also engender user loyalty [40], with almost 75% of people saying they would be more likely to return to a web site if it was mobile-friendly. Furthermore, more than half of people surveyed by Google in 2012 stated that they would not recommend a *non* mobile-friendly site to another web user [67].

In summary, it is clear that implementing a mobile-friendly layout for a web site can have substantial commercial benefits. The next section therefore reviews the various approaches to creating such layouts that have been proposed over the last few years.

### 2.2.2 *Making Web Pages Mobile-Friendly*

Considerable research has aimed to improve the user experience of mobile web browsing by transforming the appearance and behaviour of webpages to make them mobile-friendly. Before touchscreen smartphones, traditional mobile phones and PDAs suffered from significant problems. When a user tried to access the web, the reduced screen size hampered their ability to access and consume content [8]. In fact, Jones et al. [76] observed a drop in effectiveness of up to 50% when users performed two web-based tasks. This shows how early mobile devices struggled to provide even a remotely satisfactory user experience. Addressing the problem clearly required alternative techniques.

One of the first approaches for making a web page easier to browse on a mobile device was that of Buyukotten et al. in 2001 [24]. It used and evaluated five different text summarisation techniques to convert web content into “text units” which easily fit into the screen constraints of mobile devices. A user then had the ability to hide, partially display, summarise or fully open any of these text units to access the web page’s content. However, this approach was unable to

handle images and other media and so rapidly became unsuitable as the web continued to develop.

As desktop web pages can often contain unnecessary content, several techniques attempted to filter out these pieces of content to provide a better browsing experience. For instance, Baudisch et al. [18] developed an approach which allowed users to collapse unwanted areas of the web page and then zoom in to more interesting ones. Lee et al. [83] developed a more advanced approach that split a web page into “blocks”. It then filtered out blocks deemed to be irrelevant, before finally reordering the remaining blocks to place those of most interest to the user as high up on the page as possible. The approach of Xiao et al. [158] filtered out “noise” from the web page, split the remaining content into smaller, easier to manage blocks and then structured the blocks in an easily browsable way. Yin and Lee [160] developed an approach that used a ranking algorithm to identify the most relevant and important content on a web page. Their approach then refactored the layout to minimise vertical scrolling and completely eradicate horizontal scrolling. It also aimed to keep the new version of the web page as similar as possible to the original. More recently, Ahmadi et al. proposed a more advanced technique which split webpages into “subpages” of related content [6]. The user could then browse through them using a series of navigation menus. The technique utilised both the structure and visual appearance of the page, but split up the web page by topic and content, rather than appearance. It began with visual analysis of the page to extract the core sections of the page, such as the header and the main content area. Complex DOM-based heuristics then split these sections into smaller subpages. Next, it generated a navigation system and assigned each subpage a meaningful title to reflect its content. These titles allowed the user to easily find their way between the different subpages. The approach removed unnecessary elements and other clutter to better display the information on the limited screen space available. This resulted in an overall better design and therefore a better experience for the user browsing the web page.

More approaches have attempted to split up a web page to increase browsability. However, these approaches differ from those above as they did not filter any unwanted or unneeded content. Chen et al. [30] used a page splitting algorithm to divide the web page into subpages, which would be easier to view on a mobile device. The main aim was to remove the need for a user to scroll horizontally to access the page’s content. Xiao et al. [157] employed a similar technique, using “slicing” to divide a large desktop web page into smaller in-

dividual pages that would each fit onto the smaller displays of mobile devices. Xie et al. [159] also used slicing to divide a web page into blocks. However, their approach analysed the content of each block to decide how best to present the block to the user. For instance, the approach might summarise some blocks to save space, but then when a user clicks on the summary link, it would render the full content in an adapted layout suitable for the device.

Baluja et al. [16] used a technique called segmentation, which splits a large desktop web page into smaller, more manageable areas to create a more mobile-friendly browser experience. There are usually 9 areas, corresponding to the phone keys 1-9. When browsing the web page, the user could then select which section they wished to view. The approach would then render the chosen area, removing the need for panning and scrolling. However, there is a significant risk of splitting important content into different segments. To mitigate this, the authors developed an approach which utilised machine learning to perform an intelligent segmentation of a web page. Rather than simply splitting the page into a 3x3 grid of 9 equal segments, their approach used both the DOM and visual appearance of the site to recursively select two vertical and two horizontal “cuts”, again producing 9 segments. They also implemented extra fine-tuning heuristics. For instance, one ensured no segments were too small to contain any meaningful content. It even reduced the number of generated segments if the layout of the page was simplistic enough to warrant it.

While the approach functioned well when it was first proposed, the mobile web has advanced considerably in the last ten years. Therefore, it is unlikely the approach would function to a similarly high standard nowadays. Furthermore, with the worldwide adoption of touchscreen smartphones, the number of phones which actually possess 1-9 keys is very small. This further limits the application of any segmentation technique.

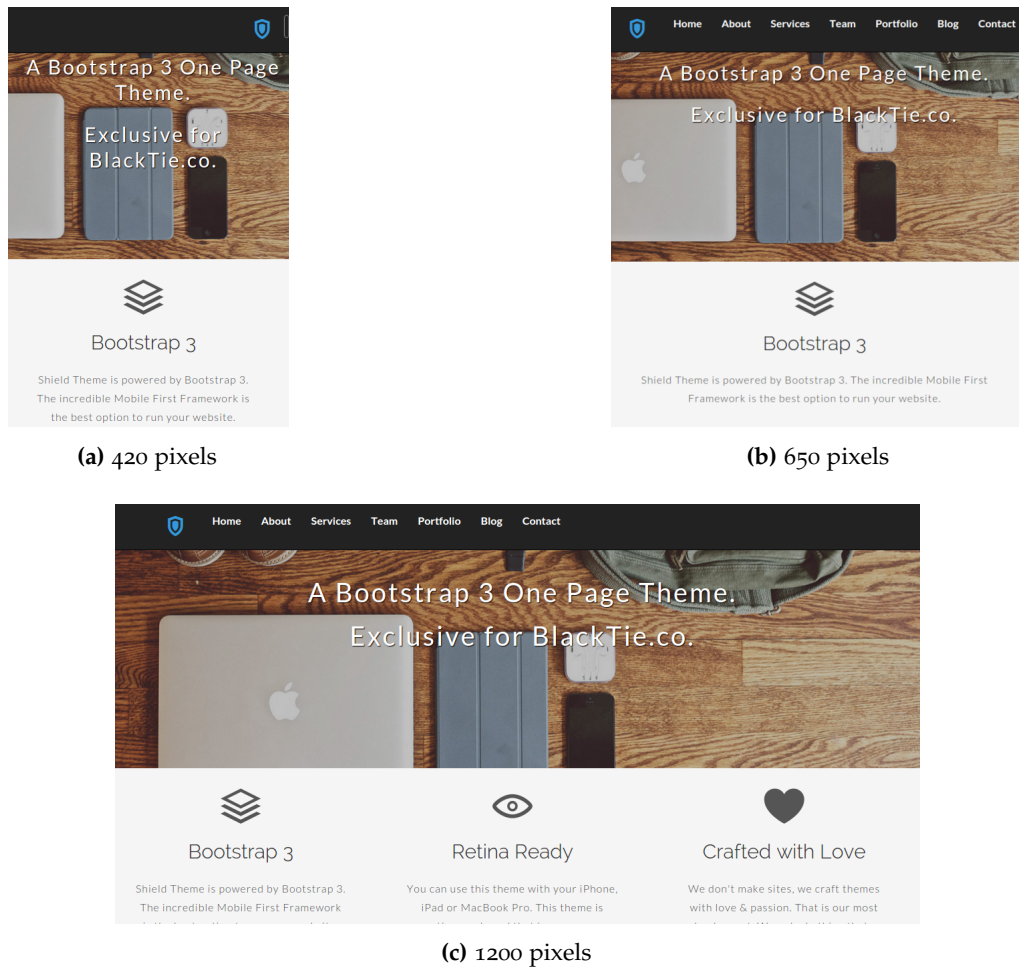
Some research aimed to improve the visualisation of native applications rather than web pages on mobile devices. Again, the hardware limitations require a different approach to a desktop computer. For example, Chittaro observed “technical limitations such as small screen size make it impossible to simply port visualization applications from desktop computers to mobile devices” [31]. He also highlighted the negative effects on forcing the user to zoom and scroll around the page, calling it “cognitively complex” and noting it can be difficult for users as well as highly frustrating.

Each type of data (text, images, etc) can present a different set of challenges to the developer. Therefore, most research focussed on improving the visualisation of only a single data type. Oquist [116] introduced *rapid serial visual presentation (RSVP)* for displaying text on a small screen. This split up text into small chunks and displayed them one after the other in the same location. This approach allowed users to read content just as efficiently as they would do on a larger screen. The RSVP approach was also applied to images by Liu et al [87]. Their approach attempted to extract potential points of interest for the user, such as faces, with mixed results. Unfortunately, due to the many different browsers available on both mobile and desktop devices, realistically applying these techniques is problematic. Modern web sites are inherently different from native applications and any additional computational effort on either the server-side or client-side could detract from the user experience further.

### 2.3 WEB DESIGN METHODOLOGIES

Despite the promising results of the various web page adaptation approaches discussed in the previous section, they are poorly suited for the highly complex and interactive web pages now commonplace on the web. Making a web page usable on devices with differing screen sizes — especially small ones — requires a different approach. Designing and building web pages from the ground up with these devices in mind will likely be much better than building a desktop-only web page and then trying to “collapse” it down to mobile devices. The two most popular approaches to doing this are *adaptive web design (AWD)* [57] and *responsive web design (RWD)* [95]. This section compares and contrasts the two methodologies. With the help of an example, it discusses the specific details, benefits and shortcomings of each. It also introduces the concept of “mobile-first” web page design.

Figure 2.4 presents an example of a mobile-friendly web page, “Shield”, a template available open-source under the Creative Commons Attribution 3.0 license. The three screenshots illustrate the dynamic layout of the web page at three different viewport widths. At 420 pixels, as shown by part (a), the main content panels are in a single column while the navigation links are in a drop-down list. At the wider viewport width of 768 pixels in part (b), the single column layout is still used for the main content, but the navigation links are in a single row in a navigation bar. Finally, at the widest viewport width shown by part (c), 1200 pixels, the main content panels have also switched into a single



**Figure 2.4:** The BlackTie “Shield” site (<http://www.blacktie.co/demo/shield>), showing how the layout of a responsive page adjusts to different viewport widths.

horizontal row. Continuing to display them in a stacked fashion would be an unnecessary and poor design choice. To describe the foundations of both adaptive and responsive web design and demonstrate the implementation of both, the following sections describe how developers could use each methodology to create this example web page. The key idea underpinning both approaches is any user, regardless of the device or web browser with which they choose to view a web page, should be able to have a smooth and enjoyable browsing experience.

### 2.3.1 Adaptive Web Design

Adaptive web design [57] is based on a process called *progressive enhancement*, first proposed by Aaron Gustafson. This originally described a methodology for designing and developing web sites which would function well on a wide variety of web browsers (and different versions of those browsers). The latest browsers received the fully-fledged, most advanced version of the site. Meanwhile, less advanced browsers would receive a simplified version that discarded the new features not supported by the browser. To better illustrate progressive enhancement, Gustafson showed how the three main web programming languages — HTML, CSS and JavaScript — could support different versions of browsers. The full version of the site served to the latest browsers would contain the latest complex CSS styles and novel JavaScript features. Slightly older browsers would get a version with some of the newer JavaScript and CSS functionality removed. Finally, the very oldest browsers would receive an even simpler version, with just the HTML content and minimal CSS styling. This meant each user received a version of the site suitable for their hardware and software environment.

In 2011, Gustafson [57] then applied a similar approach to the problem domain of supporting a wide range of different devices. Each specific device would receive the best version of the site, based on the characteristics of that device. Rather than targeting different browsers, adaptive web design advocates the creation of versions of a web page that target specific device types. At the simplest level, the versions could just target the three main device types — smartphone, tablet and computer. To provide better support to more devices a more complex setup could target device types such as small-phone, large-phone, small-tablet, large-tablet, laptop and widescreen. When a user browses the web page the server simply detects the characteristics of their device and presents the most appropriate version.

Suppose a developer wished to implement the three layouts shown by Figure 2.4 using AWD. They would likely define the main content panel widths using *static* width declarations, as shown below:

<pre>div {   width: 400px; }</pre>	<pre>div {   width: 630px; }</pre>	<pre>div {   width: 380px; }</pre>
a) Smartphone version	b) Tablet version	c) Desktop version

### 2.3.2 *Responsive Web Design*

To address the shortcomings of adaptive web design, Ethan Marcotte first proposed responsive web design (RWD) in 2010 [95]. Despite being introduced at a similar time as AWD, RWD has generally been the more widely adopted of the two. It was the #2 trend in web design and development in 2012 [54] and has enjoyed increased popularity since. W3C have described it as “a must for tablets and mobile devices” [1].

The key concept underpinning RWD is the creation of a single version of a web page which “responds to” the viewport constraints of the device in use. By resizing and rearranging the page’s content, a responsive web page provides an equivalent user experience regardless of device. According to Marcotte, a fully responsive design consists of three main “ingredients”: grid-based layouts, flexible media and media queries. Each one is responsible for and contributes to a different component of the overall design.

#### **Grid-Based Layouts**

To provide a straightforward method for scaling web pages and their contents, RWD advocates a change in how a developer declares the widths of elements. The previously described AWD example declared them as static values. In contrast, RWD declares widths as a proportion of the element’s container. To make this concept more understandable and easier to implement, Marcotte visualised the web page as a grid consisting of 12 columns. Developers then declare each element’s width to take up a particular number of columns. Figure 2.5 illustrates an example of a flexible grid. It labels each element with both the number of columns it takes up and the relevant CSS width declaration required to implement it.

Using this guide, it is easy to use a grid-based layout to implement the various layouts shown in Figure 2.4. For parts (a) and (b), the main content panels take up the entirety of the viewport width. Therefore, they take up all twelve columns in the grid and have the declaration `width: 100%`. Meanwhile, for part (c), each panel takes up only a third of the overall horizontal space available. In this case, each one fills only four columns of the grid with the style rule `width: 33.3%`.

Provided the sizing of all elements follows this concept, the web site should scale correctly and be usable on a much wider range of devices. No element will ever be wider than the browser and elements will make the best use of the

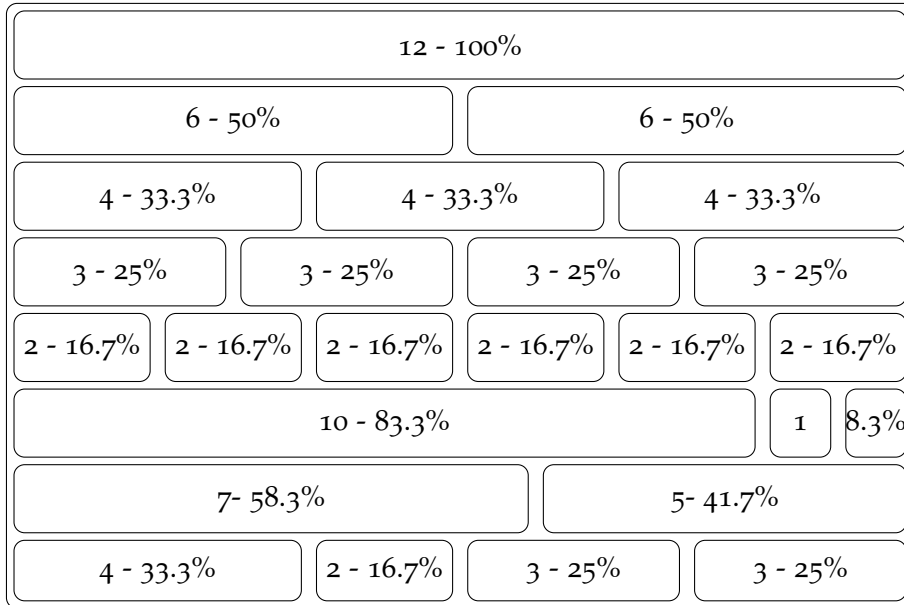


Figure 2.5: A flexible grid and its CSS declarations.

available space. This is especially true when on smartphones as there is limited space. Grid-based layouts go some way to solving the problems associated with AWD, as each device gets a bespoke layout. Also, the developer no longer has to worry about whether one of their pre-defined layouts will work well on a new device.

### Flexible Media

Modern web sites generally contain large amounts of media such as images and videos. It is critical that they also scale and respond in a similar fashion to the elements in the grid-based layout. Marcotte refers to this concept as flexible media. By applying rules such as `max-width: 100%` to the relevant elements, flexible media forces the image or video to resize itself to fit inside its container without overflowing. For example, imagine an image with a size of 600x600 and a container with a width of 400 pixels. Making the image flexible would result in the page rendering it at a size of 400x400 to perfectly fit its container. Without flexible media, it would expand to its full size, beyond the bounds of the container. Obviously, this could potentially negatively impact the appearance of the page. When developers implement flexible media in partnership with grid-based layouts, all elements on a web page should scale correctly in response to changes in the viewing constraints.



## Media Queries

The third and final ingredient of responsive web design is media queries, a module of CSS3. These allow for the querying of the real-world physical attributes of the device and browser in use. Developers can then use this information to activate sets of CSS style rules if and only if a specific set of conditions are true. The current specification supports the querying of a wide variety of attributes, such as device orientation, aspect ratio and resolution. However, by far the most commonly used query is the browser window width, using the `min-width` and `max-width` commands. For instance, the two fluid width declarations previously described must be inside media queries to ensure they are only applied at the correct ranges of viewport widths, as shown below:

```
@media(max-width: 1199px) {      @media(min-width: 1200px) {
    div { width: 100%; }          div { width: 33.3%; }
}                                  }
```

a) Smartphone/tablet version

b) Desktop version

The media query in part a) ensures that while the viewport width is narrow, each of the three content panels span the entire width of their container. The code in part b) only switches the layout to the three-column style when the viewport is wide enough i.e., 1200 pixels or wider. This value of 1200 pixels is an example of a *breakpoint*. In the context of RWD, a breakpoint is a threshold where different CSS rules apply to elements on either side of the boundary. By using breakpoints based on the viewport widths of popular device types, developers can easily and effectively support a wide range of viewport widths.

### 2.3.3 Comparing AWD and RWD

There are several advantages to using adaptive web design rather than responsive design. Firstly, adaptive web sites are easier and less complex, as the developer only has to design for a pre-defined group of devices. Secondly, the server decides which layout to display, rather than the user's browser. Therefore, the server sends only the required resources to the user's device. This often results in faster load times for adaptive web sites in comparison to their responsive counterparts [65]. Finally, suppose an organisation already has a desktop web site. It is easier for a development team to create alternative versions for smartphones and tablets, rather than scrapping the original version and building a

responsive web site from scratch. This means the organisation can start providing a mobile-friendly design to their users in a shorter period of time.

While AWD has its benefits, it also suffers from two key shortcomings. The first is that, for a particular device, there is no guarantee any of the predesigned layouts will suit the characteristics of the device particularly well. For instance, imagine a device with a viewport width of 580 pixels. If the server chose to render the design from part (a), there would be wasted space on either side of the rendered content, or the content would be "blown up" to correctly fit the screen. Neither of these outcomes is particularly desirable. Meanwhile, if it attempted to display part (b)'s layout, the end result could be even more problematic. There would not be enough room to display the content as intended and the page would force the user to pan and scroll to access any hidden parts of the page. The severity of this issue is exacerbated by the number of devices and, by extension, the number of users it could affect.

The second key problem with adaptive web design is flexibility. If the developer wished to better accommodate the 540 pixel viewport width, they have two main options. They could modify one of the existing designs or create a new one. Their current set of layout presets is not flexible enough to handle the wide range of possible devices [64]. Furthermore, overarching design changes are problematic for adaptive web sites. Developers must update each individual version separately to ensure a consistent look and feel across different devices. This would obviously be a costly and potentially error-prone task.

The main advantage of responsive design is a well designed and correctly implemented responsive web page will provide an optimal browsing experience to users across all devices. As each device essentially receives a bespoke layout, the web page sizes and arranges all the elements and content in a manner that makes the best use of the space available. Another key benefit is the ease with which a responsive web page accommodates new devices. With AWD, a developer might worry whether any pre-defined layouts will work well on a new device. In contrast, a responsive web page simply responds automatically to the new device. This makes it highly unlikely the developer would need to do anything. When it comes to maintenance, RWD again has the upper hand over AWD. The developer only needs to modify and update a single version of the web page, rather than implementing changes in multiple versions of the page to ensure consistency.

Unfortunately, there are a couple of significant issues with responsive web design. Firstly, as RWD differs quite drastically from more traditional web design, converting a desktop-only web page to a responsive one can be highly problematic. It is often easier to start from the beginning and implement the responsive design from the foundations. The fact that this approach obviously carries significant costs compounds the initial problem.

Another problem is that responsive web pages can be more complex to build than adaptive ones. This is because the developer is working with many devices in mind, rather than just the small selection of devices considered using AWD. They must also be mindful of how the styles of elements change and how they interact with each other as the viewport expands and contracts. Failure to do this correctly can lead to programming errors, which can cause aesthetic defects on the web page. These errors can further increase the amount of time and effort required to implement a fully responsive web page.

Finally, probably the most pressing issue with developing responsive web sites is the lack of understanding of RWD from web developers. Many do not know how it works, which can result in severe issues during development. For instance, a recent study on HTML and CSS errors [118] found the most serious errors arose when the developer experiments with a problem they did not possess the knowledge to solve properly. This is often the case when developers attempt to build responsive sites. This is further demonstrated by analysing the StackOverflow programming help forum. When checked, there were over 1 million questions labelled with tags related to responsive web design [162]. This indicates many web developers are still struggling to adapt to building responsive web sites.

### **Helping People Build Responsive Web Sites**

Many front-end frameworks for responsive design have been released to the public. These are pre-prepared software solutions using CSS and other web design technologies to aid developers in creating robust, standards-compliant web sites. Most, if not all of them, contain at least a grid-based layout system to help lay out web page elements. Many also provide modules for Javascript, web typography and icons, among other things. Some though are far more lightweight and provide little more than a responsive grid and a font. The CSS provided by the framework can be fully customised by the developers to create fully responsive and unique web sites. By far the most popular frameworks are Twitter's Bootstrap [19] and Zurb's Foundation [161]. Their attractive aesthetics

and high levels of usability have helped responsive design grow over the last few years.

Finally, to further help people create responsive web sites, there are now a variety of online web site building tools available [49]. These help people with no programming knowledge create attractive responsive web sites by allowing them to “drag and drop” the various components into position on the page. Among the most popular are Squarespace [26], Weebly [154] and Duda One [46], which all allow for the creation of great responsive web sites. The latter even allows a developer to upload their current desktop-only web site as a starting point and convert it into a mobile-friendly one.

#### 2.3.4 *Mobile-First Design*

In 2013, when Bootstrap version 3.0 was launched, one of the main announcements was it was a “mobile-first” framework. Traditional web page design uses a top-down approach, where the developer designs first for desktop devices and then modifies the web site to fit and function on smaller mobile devices. Mobile-first design advocates a bottom-up approach. Here, the developer develops for mobile devices first. They then scale up the layout through tablets, laptops and finally desktops. As an example, the code snippet from Section 2.3.2 would simply be as follows:

```
div { width: 100%; }
@media(min-width: 1200px) {
  div { width: 33.3%; }
}
```

Here, the default width assigned to the div element is the one for mobile devices i.e., mobile-first. Only when the viewport width is large enough is a different width applied.

There are both practical and design benefits to this approach. In terms of design, beginning on the small screen allows the developer to design for ease of usability. This approach mitigates the risk of trying to fit too much content onto a small screen, a common problem when trying to scale down a desktop web site to display on a mobile device. Mobile-first design also prevents the developer overwhelming the less powerful hardware of mobile devices. This often results in slow loading speeds for a site. This is problematic, as the majority

of web users expect mobile sites to load either as fast, or even faster than their desktop counterparts. Therefore, if an increase in loading time occurs, it can have a knock-on commercial impact. For instance, a recent study found 74% users would leave a site if it took more than 5 seconds to load. In practice, this could result in a substantial reduction in web traffic [36].

## 2.4 DETECTING PRESENTATION FAILURES IN WEB PAGES

This section introduces the general concepts of errors, faults and failures. It then describes presentation failures, which are issues with a web application's appearance. Next, it discusses the array of tools and techniques for detecting presentation failures. Finally, it identifies mobile web presentation failures as a currently unaddressed problem domain.

### 2.4.1 *Errors, Faults and Failures*

When a developer makes a mistake during the implementation of an application, it is called an *error* [22]. The cause of the error could be a lack of understanding of the application's requirements or perhaps a simple typing error. For instance, in a web page, a developer might set an element's width to be 500px when they actually intended it to be 50px.

If this error causes the program's actual behaviour to differ from its intended behaviour, then the error has manifested as a *fault* [22]. Faults are also sometimes referred to as *bugs* or *defects*. In any task, but particularly in software development, one can classify faults as one of two types; *faults of commission* or *faults of omission*. Faults of commission arise when a developer does something incorrectly, such as using the wrong method or data type. Faults of omission are mistakes where the developer did not do something they definitely should have done. In general, faults of omission are by far the more common of the two categories. In fact, Robert Glass [53] published research showing around three out of every four software defects were due to faults of omission. It is important to note not all errors end up manifesting as faults. They can be located in areas of the code not executed or not have any effect on the application's observable behaviour.

If one observes the effect of the fault during the execution of the application and the application requirements are violated, then the fault is exposed as a *failure* [22]. Continuing with the example introduced above, if the incorrect width declaration causes the browser to incorrectly render the faulty element, then it can be considered a failure. Given they appear in the presentation tier of the application stack, this thesis refers to these visual failures as *presentation failures*. However, just as not all errors lead to faults, not all faults cause failures. In fact, faults can remain hidden in software for long periods of time, until eventually the right conditions allow them to be observed.

#### 2.4.2 *Strategies of Detecting Failures*

During the development of an application, failures are normally detected by testers. However, once the application is released, end users can also observe and report failures. In either case, any failures reports go to the developers, who are responsible for locating and fixing the underlying faults.

##### **Test Cases and Suites**

For the task of detecting failures in a piece of software, several options are available. Perhaps the most common approach is to create and execute *test cases* on the software under test. Test cases usually execute a small part of the program and compare the actual observed behaviour to a pre-programmed expected output. If the two behaviours match, then the test case has passed. If not, the test case has failed and highlights the presence of a failure in the application. When it comes to web application testing, Selenium [130] is a popular choice of tool for creating and running such test cases. To test a wide variety of functionality, many test cases are frequently collated in a *test suite*. Individual test cases or whole test suites can then be automatically executed. This removes the effort burden from the human and ensures a consistent testing procedure. To evaluate the quality of a test suite the notion of *coverage* is often used. Any code that is executed by at least one test case is said to have been covered by the test suite. Intuitively, the higher level of code coverage achieved, the better the test suite is generally considered to be.

##### **Manual Testing**

Sometimes, programming a test case detailing the code to be executed to evaluate the software is infeasible. In these scenarios, a widely used alternative is

*manual testing*. Here, a human user interacts with the software under test, by clicking buttons or entering text, for example. However, rather than an automatic comparison between the observed and expected behaviours, the human decides whether the test case has passed or failed. Despite its labour intensive nature, manual testing can sometimes be a better choice than automated testing. A good example of this is when whether the software passes a test is not a binary decision. Here, the human tester uses their domain-specific knowledge to make an informed choice.

A *spot check* is “a quick examination of a few members of a group instead of the whole group” [44]. Due to the labour requirements of testing a piece of software with every conceivable input, developers and testers will often employ spot-checking. This can maximise the testing benefits while at the same time, minimising the costs. For example, when trying to detect presentation failures in a web application, one might perform spot-checking on the most common browsing environments. This could be particular operating systems, web browsers or perhaps specific devices.

### **Mutation Analysis**

Mutation analysis allows developers and testers to evaluate the quality of an existing suite of test cases. Mutation testing involves modifying the program or software in some small way. Each modified version is called a *mutant*. The current test suite is then evaluated by executing the test suite against the newly created mutant and observing the results. Ideally, the test suite will observe differing behaviour between the original version of the software and the mutant. This is commonly known as *killing the mutant*. Mutation testing normally involves generating a large number of mutants and then assigning a score to the test suite based on the percentage of mutants it was able to kill. The developers can then use the results of a phase of mutation testing to modify the test suite. They could modify existing tests, remove unnecessary ones or add new ones with the aim of killing more mutants.

The creation of mutants is performed using well defined *mutation operators*. These attempt to replicate common programming errors, the specifics of which will differ between languages and domains. The number of different possible mutants is potentially infinite. Furthermore, Ammann and Offutt said “the notion of a mutation operator is extremely general” [11]. They also stressed the key to effective mutation testing was the design of the mutation operators. Poorly designed operators often leads to ineffective test suites. This problem

can then have a knock-on impact on the quality of the final software, as there is a much higher chance of defects remaining undetected.

Unfortunately, implementing mutation analysis is problematic due to two factors. Firstly, mutation operators are very difficult to apply by hand. Second, automating the process of mutant generation can be very complicated. Because of this, mutation analysis is generally an expensive test criterion, despite its increased effectiveness. It is therefore not as commonly used. Despite this, mutation analysis has found common use in research endeavours as a “gold standard” during empirical evaluation of new techniques and as a means of obtaining test subjects for a variety of empirical studies [74].

Despite the benefits of using mutation analysis and the huge variety of problem domains in which it has been employed [117], little work has investigated mutation analysis of web applications. Praphamontrimong and Offutt [119] implemented a series of mutation operators for HTML and JavaServer Pages (JSP) code. These targeted various functional aspects of a web page, which were then used as a means of evaluating the effectiveness of a suite of web application tests. However, the mutation operators presented in this work did not target the layout of web pages or in fact any aesthetic properties. As part of the evaluation of their presentation failure detection technique, Mahajan et al. used mutation of HTML attributes and CSS attributes with the potential to alter the visual rendering of an element, to automatically introduce presentation failures into the subject web pages [89, 90].

### **The Oracle Problem**

In software engineering, testers require a way of determining a particular test has passed or failed. This mechanism is the *oracle*. First proposed by Howden and Miller in 1978 [109], the oracle determines whether given a specific test case input, the observed behaviour of the system matches the expected behaviour. Actually obtaining this description of correct behaviour is known as the *oracle problem* [11]. It can be very difficult to solve, and as such oracles can take many forms. These include documentation, specifications, models describing the system behaviour, or in some cases a human.

Automatic generation of test oracles is one of the main problems when trying to detect failures. Achieving it would remove a bottleneck in the current pipeline and make failure detection quicker, more accurate and more consistent. Barr et al.’s 2015 survey evidences the importance of oracle automation, analysing a repository of 694 publications [17]. Their conclusion was despite the wide vari-



ety of promising techniques proposed over the past 35 years, when no existing technique is perfectly suitable the human will be responsible for using their domain-specific knowledge as the final source of oracle guidance.

Testing of the presentation tier of an application must check two main components. The first is the *functional* aspect of the user interface. This can be tested using test cases that interact with the application and verify it responds in the expected way. For instance, an automated test case could check “given I have entered valid log in details, when I click the login button, I should see confirmation I have logged in”. This chapter discusses some previous approaches using this concept later on. However, when it comes to testing the actual aesthetic presentation of the application — “does the login button look right within the login form?” — things are far more difficult. Encoding an automated oracle that can make an accurate and informed decision is hugely problematic. This is due to the inherently subjective nature of judging the aesthetic properties of an application’s interface. Presentation testing therefore generally relies on a human oracle.

A previous study of web applications found that failures in the appearance of a web page are the second largest category of defects in live web sites, with only “logic and control flow faults” being more prevalent [56]. Detecting them accurately and reliably is therefore clearly a vital task. However, despite the importance of the task, a systematic mapping study of the field conducted by Garousi et al. [52] found the main focus of the majority of publications was testing the functionality of applications. Minimal research tested the GUI components compared to more popular areas such as search-based software engineering/testing (SBSE/SBST) [63, 100] and mutation testing [74, 117].

### **Back-to-back Testing**

When detecting presentation failures in modern web applications, numerous researchers have chosen to use a version of the web site under test as the oracle [7, 32, 33, 105, 123]. This is due to the inherent difficulties of generating automated oracles. This process, called *differential testing* or *back-to-back testing* [99], involves executing two different versions of a program and observing any differences in behaviour between them [141]. Others used graphical mockups of a web page’s intended appearance [89–91]. Given the lack of test cases generated by these approaches, calling them testing tools is potentially misleading. Instead, this thesis refers to them henceforth as *checking tools*. Other approaches proposed have been more similar to traditional test-case based techniques. One used formal

specifications describing the intended appearance of a web page and compared them to its actual appearance [61, 62]. Another is a framework for defining the expected appearance of a web page using human-readable declarations [133]. The next section describes these techniques in more detail.

### 2.4.3 Tools for Identifying Web Presentation Failures

This section reviews the previous work addressing presentation failures in web pages. While some target generic presentation failures, others target specific types of failures. These include *internationalisation presentation failures* and *cross-browser incompatibilities*. Also, while many approaches utilise back-to-back testing, as mentioned previously, other techniques make use of very different kinds of oracles, such as *specification failures*. Figure 2.6 illustrates the topology of approaches and techniques. Grey rounded rectangles represent the different categories of presentation failures, while blue rectangles represent the individual techniques and tools. After introducing each type of presentation failure, this section discusses each specific technique in detail. It presents technical details, benefits, shortcomings and any relevant empirical results.

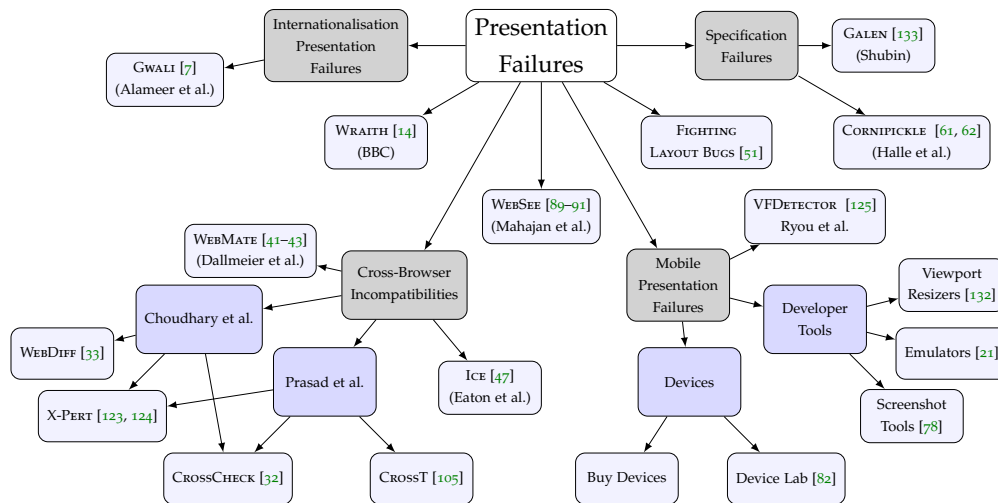
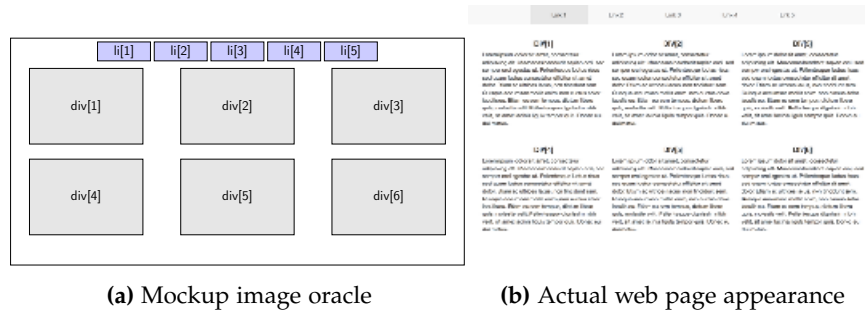


Figure 2.6: A hierarchy of presentation failure detection techniques.

Majahan et al. first proposed an automated method for detecting presentation failures in 2014 [89]. It used sophisticated image comparisons to detect and then localise presentation failures. They defined these as visual differences between the original graphic design illustrating how the web page should look (i.e., the oracle) and the actual appearance of the produced web page when rendered in a web browser under the same environmental conditions. The web page under

test and the oracle image are the two inputs to the approach. Figure 2.7 shows examples of each.



**Figure 2.7:** The inputs to WEBSEE.

The approach began by obtaining a visual representation of the web page under test. It used a screenshot of the web browser rendering the page, which it compared to the oracle image. In order to facilitate a fair comparison, the browser must replicate the environment of the oracle image as faithfully as possible. Therefore, the approach carefully controlled the image dimensions, browser window zoom and the size of the browser window itself. Their technique utilised the ImageMagick library [38] to compare the two images on a pixel-by-pixel basis. It compared both the colour and saturation of each pixel pair and reported the  $x$  and  $y$  coordinates of any pixels that are not identical.

Evaluation of the approach required a set of web pages containing presentation failures. Therefore, the authors implemented an automated fault-seeding algorithm. This mutated the source code of the web page under test to introduce a random presentation failure. The technique targeted the visual attributes of both the HTML and CSS of the web page, such as font-size. It was then used to generate between 41 and 53 random mutated versions of the four subject web pages. The approach was shown to be capable of detecting 100% of the presentation failures tested in the experiment, indicating the approach’s potential.

Their approach also aimed to identify the faulty element responsible for each failure to aid the developer in the debugging process. To do this, their technique generated a model of the web page under test. In this case they used an *R-Tree* [58], a tree data structure widely used to store multi-dimensional data. The tree stored the individual elements and the bounding boxes assigned to them when the browser rendered them. For each faulty pixel reported by the previous phase, the approach traversed the R-Tree to obtain the set of elements containing that particular pixel. As CSS allows elements to overlap, it is quite possible that a single pixel will be inside multiple elements. The approach col-

lated these to form the set of potentially faulty elements. This set was then reported to the user to help with the debugging of the presentation failure. Results showed this additional localisation step to be relatively effective, with the returned element set containing the actual faulty element in 77% of the cases studied. However, as this set was often quite large, an extensive amount of manual work on the part of the developer was still required. This reduced the practical benefits of the approach. The author also noted other shortcomings such as poor handling of JavaScript and other features of dynamic web sites.

Follow-up research addressed these issues [90]. Firstly, Majahan et al. declared a pixel-perfect match impractical for real world testing scenarios. The oracle image and the web page itself may have been created in different environments. Also, small differences between the two may be “close enough” that they are not considered to be presentation failures. Because of this, they used a new computer vision-based algorithm, namely, *perceptual image differencing (PID)*. This approach introduced the notion of “similarity” into the comparison, rather than a simple binary match (i.e., pixels are either identical or different). In its attempt to model a human’s idea of similarity as closely as possible, PID uses spatial, luminance and colour sensitivity when comparing a pair of pixels. This allowed the approach to *not* report subtle differences caused by factors such as platform differences and coding decisions made during development as failures.

Another key advancement over the previous work is the introduction of *dynamic regions*. These are areas of the web page that the browser renders differently each time, with no user modification of the web page source code. Common examples of these are social media widgets. These display only the most recent posts, and so are subject to frequent change. Automatic advertisements are another example, as they may be different every time a user loads the page. When running this new approach, the user could specify a collection of dynamic regions to help filter out unnecessary reports.

As with their original work [89], the comparison produces a set of pixel coordinates that the PID algorithm considered to be “perceptually different”. Then, the approach removed any pixels which fall within any of the specified dynamic regions. The technique handled them in a special way, as they were highly likely to be different in the reference image compared to the oracle image. However, rather than simply reporting the set of difference pixels, the approach executed an additional step in this first “phase”. It grouped the dif-

ference pixels into clusters representing the difference presentation failures detected.

The fault localisation approach was also drastically improved. The foundation of the approach was again the R-tree. This time though, the approach utilised domain specific heuristics devised through manual analysis of false negative results to add other potential faulty elements to the result set. For instance, given an element  $e$  is in the faulty element set but its neighbours (eg., parent, children and sibling elements) are not, then this step added in any missing elements that could potentially be faulty.

Finally, the approach ranked the set of potentially faulty elements in order of likelihood to be directly responsible for the failure. This allowed the developer to immediately focus their attention on the likely source of the issue. This in turn reduced the amount of time required for the overall debugging process. The approach used the following four heuristics:

- Contained Elements: if an element's parent and all of its siblings are in the faulty set, the parent is more likely to be the cause of the fault.
- Overlapped Elements: if an element has at least one faulty child, but not all, then the fault is more likely to originate in the child.
- Cascading: any element that was simply displaced, with no change to its individual appearance, is "more likely to have been moved by the faulty element than to actually be the faulty element" [90].
- Pixels Ratio: an element with a higher proportion of faulty pixels is more likely to be the faulty element.

When evaluating whether the dynamic regions contain presentation failures or not, the approach used one of two different functions, depending on the type of the dynamic region in question. For instance, if it was a dynamic text region, then the function checked the CSS properties applied to the elements in the region matched those defined in the oracle image. The approach then repeated the previous steps of detection, localisation and ranking to identify and report presentation failures in the dynamic regions of the page.

The evaluation of this improved technique used a very similar experimental setup to the previous work. Eight web pages were randomly mutated to introduce presentation failures with a wide variety of visual impacts. The new PID comparison algorithm was again capable of detecting 100% of the failures introduced. The results also showed the iterative developments had a profound

impact on the quality of the localisation. For instance, the localisation accuracy of 77% achieved in [89] was improved to 93% using the new technique. Further analysis of the results showed the element ranking heuristics were also effective, with the faulty element ranked in the top five elements in 45% of cases, and in the top ten in 70%. This meant the developer would likely have the ability to identify and remedy the fault quickly. Finally, the study compared the approach's performance to that of humans performing the detection and localisation tasks manually. The human participants achieved just 76% detection and 36% localisation accuracy, respectively. This is substantially lower than the results for the proposed approach. The humans also required substantially more time to perform the tasks than the automated approach.

The authors have released the approach as an open-source tool, *WEBSEE*, available for download and use by the public [91]. Promising early feedback indicates the tool has good potential for real-world use. The tool was then extended to the problem domain of debugging presentation failures to make it easier for developers to fix issues by highlighting the most likely "root causes" [88].

Developed by a software team at the BBC, *WRAITH* is a screenshot comparison tool for responsive web pages [14]. It works by comparing the visual appearance of two versions of a web page. It then reports any differences to the user in a gallery, which may potentially be presentation failures. However, the novel component of the tool is the way in which it obtains the screenshots. Normally, the images are manually saved and stored so a user can load them at any time. In contrast, *Wraith* generates the images "live". It automatically renders both the oracle version and test version of the web page in a browser. It then extracts images of each version and then performs the comparison step. By doing this, any dynamic content which may change with no user intervention will show identical content in both versions. If the tool saved one image at an earlier time and then loaded it, the image comparison process would likely report a high number of false positives.

A potential usage scenario for the tool is for testing CSS changes in a development/sandbox environment against the current live version of a web page. In this scenario, the developer will easily be able to observe the impacts of the changes. However, one key downside to the approach is that manual inspection of the difference images produced is still required. The human must also spend time determining the root cause of any failures detected. Due to *WRAITH*'s industrial origins, no official empirical study has been conducted to determine the effectiveness of the tool. However, the developers of the tool have evaluated

its performance and recently trialled the tool as part of their regular development and testing cycle. They reported a significant reduction in the testing effort required and fewer bugs remaining undetected.

Other automated screenshot comparison tools have also been developed. For example, a developer at Google presented instructions on how to use Puppeteer (an alternative to Selenium) to gather screenshots of two different versions of a web page at various viewport widths and automatically compare them. However, this approach suffers from the same problem, in that if the image comparison reports a difference, a human must still manually inspect the result [45].

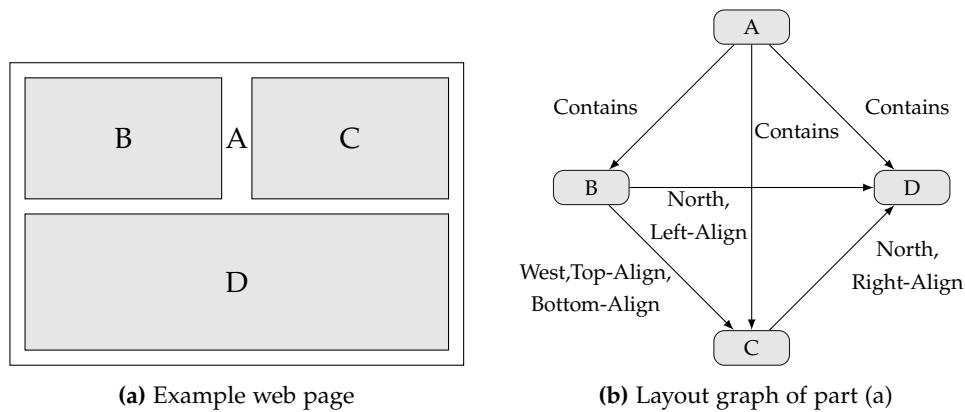
Selay et al. [128, 129] also proposed an approach using image comparison to find presentation failures in web applications. Their approach was similar to both WEBSEE and WRAITH in that it compared two screenshots; one from a staging environment and one from the live version. However, the approaches diverge significantly when one considers the image comparison itself. The authors said comparing on a pixel-by-pixel basis was too labourious and wanted to streamline the process while still detecting as many issues as possible. Their approach sampled pixels from the whole set of possible pixels and compared these to detect issues, substantially reducing the time required to detect issues.

FIGHTING LAYOUT BUGS is a library for detecting layout bugs in web pages, first introduced in 2009 [51]. It consists of a suite of detectors, each targeting a specific category of layout bug. Examples are images with invalid URLs or pieces of text that are nearly overlapping. Unfortunately, the current iteration of the tool is insufficient for testing most modern web pages. The limited detectors are only capable of revealing a small subset of layout issues.

### **Internationalisation Presentation Failures**

Alameer et al. defined a subcategory of presentation failure called *internationalisation presentation failures (IPFs)*. These are differences in the appearance of a web page when a browser renders it in two different languages. This is in contrast to Mahajan et al., whose approach compared an oracle screenshot and the page's actual appearance. Translations between certain pairs of languages can often result in strings of vastly different lengths. These strings can introduce layout issues as various elements on the page expand, contract or even move to handle the different text.

Rather than using computer vision techniques, the approach instead employed models based on the visual rendering of the web page to detect the issues. It paid specific attention to the “visual relationships and relative positioning” of



**Figure 2.8:** An example of a Layout Graph.

HTML elements. This model, the *layout graph (LG)*, is a complete graph where each node representing an HTML element is connected to every other node. The edges connecting them describe their visual relationship. To help illustrate, Figure 2.8 presents a simple example. The web page shown by part (a) corresponds to the LG in part (b). As the largest element, A, is the container for the other three, the respective edges have the “Contains” attribute. Meanwhile, as B and C are on the same row, the edge between them has the “West” attribute, while the attributes “Top-Align” and “Bottom-Align” model the fact they are equal in height.

The approach built the layout graph by first analysing the Document Object Model (DOM) to identify all of the elements present in the web page. It then used properties such as each element’s coordinates to determine the relationships between pairs of elements.

The approach detected IPFs by building layout graphs for both versions of the web page and then comparing them. Differences in the graphs could potentially represent IPFs the developer should be aware of. Finally, to make the output reports more beneficial and easier to use, the approach analysed the set of potentially faulty elements identified in the previous step and ranked them in descending order of how likely they were to be the offending element.

The authors implemented the whole approach as a prototype tool, *GWALI*, and evaluated it on a corpus of 54 web applications selected from a wide variety of sources. They compared it against three other tools; *WEBSEE* (previously described in this section), *X-PERT* (introduced in Section 2.4.3) and *FIGHTING LAYOUT BUGS*. When detecting IPFs, *GWALI* demonstrated identical 100% recall to *WEBSEE* and *X-PERT*. It also achieved substantially higher precision



(91% against 55%), showing the potential of the approach. The evaluation then showed the localisation techniques applied by GWALI were also effective. They consistently ranked the actual faulty element in the top five elements reported to the user, providing further empirical evidence of the tool's usefulness.

### **Cross-Browser Inconsistencies**

There has been a reasonable body of work in the literature addressing the problem of *cross browser incompatibilities*, or XBIs. These are differences in the functionality or the appearance of a web page when viewed in different web browsers. While this literature review focusses primarily on visual presentation failures, this section also describes the components targeting functional XBIs to give a better overall view of the approaches.

The main motivation for this work was the severe lack of standardisation in terms of client-side behaviour between different web browsers. These led to many inconsistencies in the way a web page both looked and appeared across different browsing environments. XBIs can vary drastically in both nature and severity. On one end of the spectrum, they can be small cosmetic changes such as subtle changes in fonts. On the other, they can be critical functional issues like navigation links not redirecting the browser anywhere. Researchers stressed trying to detect XBIs manually required a considerable amount of effort and was also error-prone. This led to the development of the various automated techniques.

The first major piece of research addressing the problem of XBIs was WEBDIFF, an approach proposed by Choudhary et al. It aimed to detect both *layout* and *functional* inconsistencies (although the former was the main focus) and report them to the user [33]. The amount of manual effort required in order to obtain any meaningful benefits plagued previous commercial and research tools targeting the problem. Therefore, the authors designed WEBDIFF from the outset to be a fully automated technique for XBI detection.

The technique used the previously introduced concept of back-to-back testing, where the same application is run under different hardware/software environments with any differences in behaviour likely representing problems. First, the approach rendered the web page under test in two different browsers. It ensured the browser characteristics such as the viewport size were the same in both, to allow a fair comparison. Then, it captured full page screenshots of the web page in each browser. The DOM was also queried in each browser to obtain a number of properties related to each element on the web page. These

included the unique XPath of each element, its coordinates and whether it is clickable or not.

After completing this data collection phase, the approach employed a two-part analysis to identify XBIs. The first of these was *structural analysis*, which compared the collections of DOM nodes extracted from each of the browsers. The aim was to “match” nodes across different browsers. The approach did this by computing a measure of similarity between the nodes, using information such as the tag name, XPath and properties of the elements. It reported any unmatched nodes at this stage as XBIs. The technique then moved onto the next stage, *visual analysis*. Here, it compared the matched nodes in terms of their actual visual appearance as rendered in the different browsers. Properties such as the element’s position, size, visibility and appearance were all considered when determining whether an XBI was evident. Finally, it collated any identified inconsistencies into a report which was output to the user. They could then use it to debug and fix any issues found.

The evaluation of the approach used a pool of nine web pages and three popular browsers, Mozilla Firefox, Google Chrome and Microsoft Internet Explorer. Overall, WebDiff detected 121 different XBIs, split between positional, size and general appearance differences. The results also showed a reasonably low false-positive rate of 17%. The authors deemed this to be an encouraging result, as given 10 reported XBIs to evaluate and debug, on average more than 8 of them would be actual XBIs. However, due to the huge number of potential root causes for each XBI, the debugging and fault-patching processes were still manually labour-intensive.

Mesbah and Prasad developed CrossT [105], another fully automated solution for detecting XBIs. The main difference between this work and that of WEBDIFF is that CrossT could identify trace-level behavioural inconsistencies as well as the more visual XBIs identified on individual pages.

The approach focussed more on what they referred to as “functional consistency” issues, which occur when a web site behaves in a different way across different browsers and platforms. The approach gave the actual look and feel of the web site a lower priority. It began by exploring the application to build a *navigation model*, using CRAWLJAX (a tool for exploring modern web applications [104]) to interact with the web page (e.g., by clicking navigation buttons). It also analysed the DOM after each interaction to identify the different states of the application. For ease of understanding, the approach split the model into

two tiers. In the top level, the *state graph*, each state represented a screen observed by the user. The connections between various states represented the user actions required to get from one screen to another. The model also recorded the visual appearance of the application in each state using an abstracted version of the DOM. This second tier is the *screen model*.

As with *WEBDIFF*, the approach detected XBIs using a pairwise comparison of navigation models, done in two phases. Firstly, the approach compared the state models to detect any trace-level XBIs. This comparison also paired matching screens from the two navigation models. The second part then compared these matched screens via their DOM representation to identify screen-level XBIs. This process took the two DOM trees and filtered out any known compatibility issues in the internal DOM between browsers. They did not represent actual XBIs and would therefore be false positives. The approach considered any remaining differences to be real XBIs. The report output by the tool presented these XBIs to the user in the form of snapshots of the issue in the page and the highlighting of the offending parts of the screen model.

The technique was empirically evaluated on 5 web sites, using difference combinations of Chrome, Firefox and Internet Explorer as the reference and test browsers. Again, this was a fairly small scale empirical study in terms of the number of subjects. However, the authors claimed 5 web sites was enough to obtain reliable and generalisable results. To form baselines to compare against *CROSST*, the authors manually inspected the five subject web pages to identify any trace-level and screen-level differences. The authors also stated that determining whether reported screen-level XBIs are actual XBIs or not is significantly more difficult than for trace-level XBIs, due to the subjective nature of presentation failures.

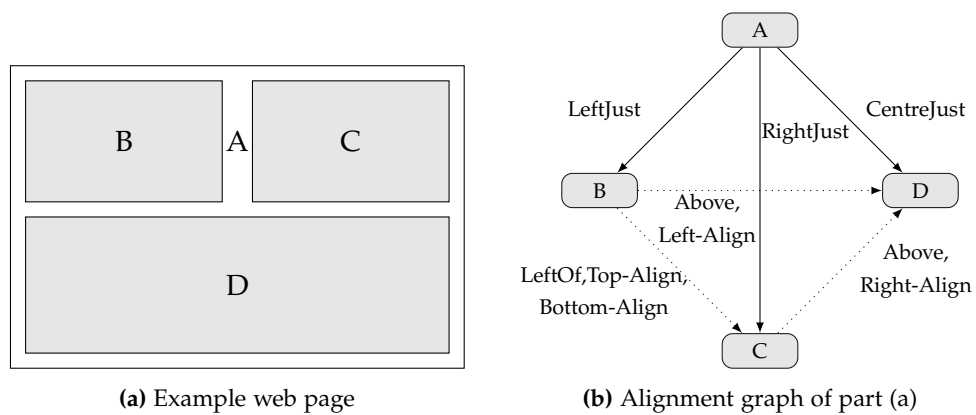
When detecting trace-level XBIs, the approach was highly effective. It detected 44 true positives with just 2 misleading false positives. For screen-level XBIs, the false positive rate was higher, but still significantly lower than that of the baseline comparison technique. This indicated the superiority of the proposed approach. However, while *CROSST* detected a large amount of XBIs, it could not detect them all.

*CROSSCHECK* [32] was the result of a collaboration between the authors of *WEBDIFF* and *CROSST*. *CROSSCHECK* took advantage of the benefits of both while mitigating the shortcomings. *WEBDIFF* focussed mainly on appearance or “screen-level” differences and *CROSST* targeted more functional “trace-level” differ-

ences. Therefore, the combined approach tried to provide a more rounded technique for a wide variety of XBIs.

The extracted models and the approach for detecting trace-level functional XBIs were very similar to those used previously. However, the visual comparison component of the technique underwent a major change. The DOM differencing employed by `CrossT` was rather simplistic for the problem domain. Furthermore, the comparison heuristic used by `WebDiff` was susceptible to missing many important visual differences. To address this, the new approach employed machine learning to provide a more comprehensive XBI detection technique. It compared each pair of matched screen-element pairs in terms of their visual appearance and reported them as XBIs if the algorithm found them to be different. More specifically, rather than the binary techniques used in `WebDiff` and `CrossT`, `CrossCheck` used a decision tree classifier. Although this was less complex than other types of classifier, the results showed it to be very effective for this problem domain. The classifier considered five features: *size difference ratio (SDR)*, *displacement*, *area*, *DOM text difference* and  $\chi^2$  *image distance*, which used colours histograms to compare the appearance of the matched elements. These features were specifically chosen as they frequently represent the manifestation of actual XBIs, according to the authors' experience during their previous work. As with all machine learning approaches, the classifier required training before it could detect XBIs. To do this, the authors used 10 web pages not included in the study, that all contained known XBIs. The collections of screen-element pairs from each web page were then manually labelled as either true (represented an XBI) or false (did not), and provided to the classifier as training data. In total, the authors provided 2,137 labelled examples. 178 of these were true examples, which should be more than sufficient for the classifier to learn what does and does not constitute a visual XBI. Finally, the reports produced by the tool were also improved in this new version. It grouped related cross-browser differences (CBDs) together into clusters representing individual XBIs. This reduced the manual effort required to fix the detected XBIs.

The evaluation of `CrossCheck` used a pool of seven web applications. The results found it capable of detecting a wide variety of cross-browser differences and clustering those CBDs into the relevant XBIs. When compared to `CrossT` and `WebDiff`, `CrossCheck` was significantly more effective. It detected 314 true CBDs compared to 49 for `CrossT` and 119 for `WebDiff`. It was also considerably more precise, demonstrating a 64% false positive rate, compared to 98% and 79% for the other two rival approaches.



**Figure 2.9:** An example of an Alignment Graph.

The approach was further enhanced with X-PERT [123]. To begin, the authors conducted a study of real-world XBIs. They determined several key categories of XBI; namely, *structural*, *behaviour* and *content* errors. This study also found structural XBIs to be by far the most common category of XBI. Again, the approach for detecting behaviour and content XBIs was very similar to that of CROSSCHECK. However, for detecting structural XBIs they developed a novel approach called an *alignment graph*.

The alignment graph (AG) captured alignment relationships between pairs of elements on a web page. Web page elements formed the nodes of an AG, while relationships between pairs of nodes formed directed edges. The AG is inherently similar to the layout graph. Therefore, this section revisits the simple example from Figure 2.8 to demonstrate how the alignment graph works, as shown by Figure 2.9. The main difference is that the AG is a tree-based structure, whereas the LG is a complete graph. Therefore, only elements in specific types of relationships have edges between them. These edges are of one of two types. A *parent-child* relationship exists when one web page element, the parent, is the direct container of another, the child. This is in contrast to the LG, which used an attribute to model containment. A *sibling* relationship exists between two elements that share the same parent. Using the example, the edges between A and the other three elements are instances of parent-child relationships. The figure shows these as solid lines. Sibling edges exist between each combination of B, C and D and the figure uses dotted lines to show these.

The algorithm for constructing the AG extracts relationships from the DOM of the page. It uses the co-ordinates of the rectangles of each element to determine which elements to connect with relationship edges. It then further assigns a set

of attributes to each relationship. These further describe the nature of the alignment between pairs of elements relative to one another. Child elements may be left-justified (“LJ”), right-justified (“RJ”), or centre-justified (“CJ”) within their parents. They may also be potentially top-justified (“TJ”), bottom-justified (“BJ”) or middle-justified (“MJ”). Sibling relationships may represent the first node being to the left of (“L”) or right of (“R”) the second. In this case, the elements may be also aligned on their top edges (“TE”) or bottom edges (“BE”). Additionally, an element may be above (“A”) or below (“B”) its sibling. In this case they may also align on their left edge (“LE”) or right edge (“RE”). In this regard, the main difference between the two representations is simply the names assigned to the attributes, e.g., “North” in the LG and “Above” in the AG.

More formally, Choudhary et al. [124] defined the alignment graph as a 5-tuple  $\mathcal{AG} = (\mathcal{E}, \mathcal{R}, \mathcal{T}, \mathcal{Q}, \mathcal{F})$ .  $\mathcal{E}$  is set of HTML elements on the web page, which form nodes in the graph.  $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$  is the set of relative alignment relationships that exist between HTML elements on the page. That is,  $\forall (e_1, e_2), e_1 \in \mathcal{E}$  and  $e_2 \in \mathcal{E}$ ,  $e_1$  is a parent of  $e_2$  or  $e_1$  and  $e_2$  are siblings. Each relationship forms an edge in the graph, with a single edge being sufficient to represent a sibling relationship.  $\mathcal{T} = \{pc, s\}$  is the set of relationship types (i.e., parent-child and sibling), while  $\mathcal{Q} = \{LJ, RJ, CJ, \dots\}$  is the set of relative alignment attributes for each relationship. Finally,  $\mathcal{F} : \mathcal{R} \rightarrow \mathcal{T} \times 2^{\mathcal{Q}}$  is a function that maps each edge to its relationship type and to a set of relative alignment attributes.

The approach began by generating the alignment graphs of the two web pages. Then, it compared them and reported any discrepancies to the user as structural inconsistencies. Results showed the overall technique to be a significant improvement over CROSSCHECK. X-PERT demonstrated both substantially higher precision and recall. On average, X-PERT achieved 76% precision in comparison to CROSSCHECK’s 18%. It also demonstrated a 12% average increase in recall, up from 83% to 95%. X-PERT also reported a significant reduction in the number of duplicate reports of XBIs, down from 52 to just 1. This further illustrated the effectiveness of the approach. According to the authors, one of the main contributors to this increase in performance was the alignment graph. It is therefore clear that the alignment graph presents a very useful approach for modelling the layout of modern web pages.

Semenenko et al. focussed solely on visual XBIs in their tool, BROWSERBITE [126, 131], in a similar fashion to Choudhary et al. with WEBDIFF. However, their approach was very different. Rather than primarily using the DOM, BROWSERBITE is based entirely around image-comparison. The approach was split into three

key stages; *screenshot capture*, *screenshot comparison* and *classification*. The first stage captured a full-page screenshot of the web page in each browser. Then, the approach splits each image into *regions of interest (ROIs)*, based on factors such as borders in the web page. Next, the approach attempted to match each ROI from the oracle image to an ROI in the comparison image. This is similar to previously described approaches that attempted to match individual web page elements. BROWSERBITE reported any unmatched ROIs as potential incompatibilities. Next, the approach took each pair of matched ROIs and computed the similarity between them. If this similarity fell below a specified threshold, the approach again reported the ROI pair as a potential XBI.

The original version of BROWSERBITE only performed the first two stages. However, after performing an initial empirical evaluation, the authors found that in order to detect a high proportion of actual XBIs, they had to configure BROWSERBITE in such a way it reported many false positives. Therefore, they added the classification module to filter out these false positives and only report true positives to the user. It did this by employing machine learning. Firstly, the authors used crowdsourcing to classify 1200 pairs of matched ROIs as either true XBIs or “non-issues”. Then, they trained a neural network using this labelled data which analysed a variety of features (e.g., the colour histogram of the images, the position on the web page and the similarity score) to decide whether a given mismatched pair represented an XBI or not. The subsequent empirical evaluation found this approach capable of achieving high levels of both recall and precision.

WEBMATE was proposed by Dallmeier et al. and designed to thoroughly and automatically test complex web applications [41]. It functioned by exploring the different states of the application, building a usage model as it did so. This model represented how specific user interactions affected the application state. As with previous approaches, it used an abstraction of the DOM to represent the appearance of the application in each state. The authors also implemented several refinements to handle key issues such as state abstraction and interactive elements. This aimed to make the produced model more usable. It can perform cross-browser testing by comparing a pair of usage models from two different browsers and is capable of detecting both functional and aesthetic XBIs. It reported them to the user in the form of a textual description of the issue and a pair of screenshots highlighting it in both the reference browser and the test browser.

Eaton and Memon proposed ICE [47], one of the first automated techniques for detecting XBIs in web pages. The approach used an inductive model that represented how web applications should perform in each configuration (such as different browsers). This model was based on a collection of previously classified “good” and “faulty” web pages, which were manually provided by the user. The model then reported a list of potentially faulty HTML tags that the developer could use to debug and fix their site. Finally, the user could manually update the model by adding more instances of good and bad pages to make it more effective in the future. The evaluation of the approach showed the proposed approach was effective. It also emphasised the scale of the XBI problem, as ICE found serious XBIs in many popular web applications.

### Specification Failures

Halle et al. [61, 62] took a rather different approach to detecting what they called “layout-based bugs”. They began with a detailed case-study of 35 web pages containing more than 90 layout-based bugs. They then devised a categorisation of layout and behavioural bugs. Examples included misaligned elements, overlapping elements and elements overflowing out of their container. Next, they defined a declarative language a developer/tester could use to express the intended layout of the web page. A key focus was it should be highly readable and easily understandable for humans. This language, CORNIPICKLE, worked by first selecting elements on the web page. Then, it verified the actual layout and appearance of those elements in the browser matched the specified constraints defined by the user. The authors then showed how CORNIPICKLE was capable of expressing layout constraints to detect all of the layout bugs identified in the case study. Figure 2.10 presents a simple example constraint. It defines a group of navigation links (HTML li elements) should always be in a row. It does this by checking they all have the same top and bottom coordinates.

```

For each x in (li)
  For each y in (li)
    When x is not y
      x's top equals y's top
    And
      x's bottom equals y's bottom

```

**Figure 2.10:** An example of CORNIPICKLE’s formal specification.



They then implemented the approach in a prototype tool of the same name. This tool verified the web page satisfies all of the defined constraints in real-time as the user interacted with it. It reports any violated constraints to the user in the form of highlighted web page elements and error messages. This allows the user to test the layout at different viewport widths and thus the tool provides a small level of support for the testing of responsive web design. However, the approach suffered from one major shortcoming. Its effectiveness was entirely dependent on the quality of the produced layout specification. A poorly written specification would not produce good testing results. Furthermore, due to the logical foundations of the language, using it to declare the intricacies of modern responsive web pages is likely to be both error-prone and labour-intensive. This further reduced the usefulness of the approach.

As previously discussed, automated unit testing is a common practice among software developers. For the problem domain of web page presentation failures, this type of testing has recently become possible. The GALEN framework [133] gives developers and testers the ability to define the layout of their web site across multiple resolutions using a simple specification-based syntax.

Figure 2.11 presents a simple example. It shows the two main components of GALEN's specification syntax. The first, *selection*, involves defining the location of individual elements on the web page. This allows GALEN to test their characteristics and layout. GALEN can identify elements in three ways; element id, CSS selectors or XPath. The second component is *specification*. This involves detailing the specific features which should apply to each element defined in the selection phase. This code example presents a width specification for the menu element. Additionally, it shows how GALEN's syntax can define different specifications for different categories of device. A user can write separate declarations for mobile and desktop devices, highlighting Galen's focus on testing responsive web design.

Given the similarity between the two approaches, one can compare CORNIPICKLE and GALEN. Halle et al. stated that readability and ease of understanding were high priorities when developing CORNIPICKLE. However, given its logical foundations, there is potentially a steep learning curve for users who perhaps do not have sufficient background knowledge. In contrast, as GALEN is a unit test framework, it is likely to be easy to use for users with vastly different levels of experience.

```

==== SELECTORS ====
@objects
  header      id      page-header
  menu        xpath   //div[@id='header']/ul
  header-logo css    #header img

==== SPECIFICATIONS ====
@on mobile
  menu:
    width: 100% of header/width

@on desktop
  menu:
    width: 250px

```

**Figure 2.11:** An example of Galen’s selection and specification syntax.

Once a developer has defined their specification for the responsive web page, GALEN provides the ability to run the web page against the specification at any resolution. This allows for testing on the myriad of devices currently in the market. The framework also generates an informative and easy-to-understand HTML report detailing any failed tests in the specification. Developers and testers can then use this to fix any defects in the page.

Unfortunately, like all unit test frameworks, GALEN’s effectiveness is dependent solely on the quality of the specifications provided. These must be manually generated, which exacerbates the problem. Given the vast complexity of many modern responsive web sites, this could prove to be very labour-intensive and possibly error-prone. Also, the design and layout of the web site can often change dramatically with each iteration. In this case, one may have to repeat the entire process. This is because refactoring the original specifications to match the latest version of the site may actually require more effort than beginning from scratch.

## 2.5 MOBILE PRESENTATION FAILURES

Just as cross-browser incompatibilities can arise from differences in how certain browsers render a web page, presentation failures can occur on certain sizes of device. For instance, a browser might render a web page perfectly on a desktop computer. However, the page may contain presentation failures on devices with smaller viewports, such as smartphones or tablets. These often stem from the incorrect implementation of adaptive or responsive web design. This highlights the problematic and error-prone process of creating mobile-friendly web pages.

Despite the prevalence of mobile devices, there is currently almost no previous work in the literature addressing the problem of presentation failures in mobile web applications. Therefore, after describing this previous work, this section discusses the various problems specific to this problem domain. Then, it revisits the techniques previously described and explains why they are ill-suited to this problem. Finally, it presents a variety of tools and techniques used by human testers to identify mobile presentation failures.

Ryou and Ryu [125] addressed what they called *visibility faults (VFs)*. They defined these as issues where a specific element on a web page demonstrates differing visibility across different viewport widths. As such, the approach made an attempt to address responsive design. Like many other approaches, the foundation was a model of the web page. In this case, the authors proposed the *UI State Graph*, representing the states of the web page and the transitions between them. Each state of the graph detailed the visibility of so-called *event objects*, which are DOM elements that have at least one event handler (used to trigger some behaviour) assigned to them. The visibility of each element can be *absent*, *fully-covered*, *partially-covered*, *off-screen* or *normal*. The transitions of the graph denoted the switch from one application state to another by executing the event handler of a particular event object.

To detect visibility faults, the approach built the UI state graphs of the web page at a set of different viewport widths. It then used the assumption that if a particular event object exists in two different UI state graphs, then its visibility should be the same in both. The authors defined two subcategories of visibility fault. The first, *inconsistency faults*, occur when the visibility of a particular event object is different in a pair of graphs. The second, more severe subcategory, *covered errors*, occur when the object in question has the visibility “fully-covered” for one of the two graphs. In these cases, the user cannot click the object and fire its

event handler, reducing the usability of the web page substantially. Finally, after detecting any visibility faults present, the approach generated “replay descriptions”. These are sets of instructions on how to recreate the fault for debugging and fixing purposes.

The approach was implemented into a tool, VFDDETECTOR and then evaluated on 35 real-world web pages. Results showed the approach could detect visibility faults with very high precision (only 9 false positives out of 492 total reports), suggesting the approach has great potential for addressing this clearly important problem. Unfortunately, the approach has a couple of drawbacks. Firstly, while it does target responsive design, it essentially runs multiple times at different resolutions and so is therefore prone to missing presentation failures occurring in between the tested viewport widths. Secondly, visibility faults are only one subcategory of presentation failure and therefore a more comprehensive approach is required to thoroughly check modern responsive web pages.

### 2.5.1 *Difficulties*

There are currently several issues hampering effective and efficient checking and testing of mobile-friendly web sites. The Quality Control Team at Segue Technologies presented what were, in their view, the main three challenges [163]. The first is the *testing environment*, which contains such a plethora of different devices, operating systems and web browsers. This results in a huge number of different configurations to test a web site on. For instance, in 2015, OpenSignal reported over 24,000 unique devices running the Android mobile operating system [2].

The second key challenge is the *misleading design* of responsive web sites. Testers may not always know exactly what they should be testing on each device. They may not know how to verify whether the web site satisfies the defined requirements or not. For instance, they may not know the expected layout behaviour under a particular configuration. This could be attributable to a lack of graphical mockups or written specification. In this scenario, a tester has to make a subjective decision as to whether presentation failures are present. This is in itself a threat to the quality of the testing, as different testers may have differing opinions on the same web page. This can be problematic as a web page could easily go live on the web when it contains presentation failures.

The third and probably most important challenge is the *lack of an automation framework*. This means testing on a wide variety of devices is a purely manual process. Testing on a device by device basis is both time-consuming and error-prone. In many cases, this leads to insufficient testing, and in some cases, no proper testing at all. Most of the available technologies and tools discussed in this section use automation for parts of the testing process. However, they require considerable manual effort in the rest of the process, which limits their usefulness in the real world.

These issues are well recognised among web development and testing professionals, as they have been evident for several years. Furthermore, as more devices come into circulation, the testing environment becomes even more fragmented. This is only likely to exacerbate these problems. This suggests that developing techniques for responsive design testing addressing all three challenges is a worthwhile objective.

### 2.5.2 *Current Tools and Techniques*

If a developer or tester wishes to replicate the browsing environment of end users as faithfully as possible, the best option is performing the testing directly on real devices. The tester can evaluate non-aesthetic characteristics such as web page response times and the touch sensitivity of the device. This is impossible without testing on the physical device. Furthermore, by inspecting the minutiae of the responsive design on the exact same hardware configuration as the end users they can potentially detect issues other testing approaches simply would not.

Recent advances now allow the automation of interactions with devices, such as clicks and swipes. However, the burden of actually inspecting the appearance of the web page and verifying its correctness still falls upon the developer/tester. Suppose a developer has to test a fairly simple responsive web page. Performing manual inspection on even a small number of devices is likely to be fairly labour-intensive. Doing so on a much more complex web page on a larger subset of devices would likely be infeasible for a single tester. Even a small testing team with a limited budget may struggle. This perhaps goes some way to explaining why responsive web pages often do not undergo thorough testing.

In an ideal world, to ensure a responsive web page was fully device and platform agnostic, a developer would test it on all possible hardware and software configurations. In the real world however, this is completely infeasible, in terms of both effort and cost. Nowadays, even mid-level devices can cost £500, with flagship devices often demanding prices approaching £1000. Therefore, testing requires a degree of prioritisation to ensure the number of devices and platforms tested is manageable.

Despite recent device proliferation, a large proportion of users are concentrated in a small, select group of devices. This allows people to more easily test web pages on the browsing environments of the large majority of their user base. They can also substantially reduce the overall testing workload. For instance, Khalid et al. investigated the devices used to download the most popular free Android applications. They found by streamlining testing to focus on just the most popular 20% of devices, it could cover 80% of the total user base [80]. Montague and Hogan echoed this advice. They advocated focussing on the devices producing the bulk of the web traffic while also trying to keep the devices selected as diverse as possible [69]. Obviously, extrapolating these findings from Android applications to responsive web pages is potentially risky. Responsive web pages are accessible from any web enabled device rather than just those running one OS. However, the findings show intelligent testing strategies based on usage data can reduce testing effort significantly. What's more, with services such as Google Analytics [34] this usage data is readily available for anyone to use.

One potential problem with prioritising test devices based on popularity is these devices often have similar characteristics. This means the testing may overlook some device-specific issues occurring on less popular devices. An alternative is to test on devices that all have different characteristics. However, this can potentially lead to testing occurring only on old or unpopular devices, which makes the testing not worthwhile. Therefore, Vilkomir et al. proposed an approach that combined the two [140]. They used combinatorial approaches which struck a balance between selecting popular devices to detect issues affecting lots of users and selecting additional devices with a diverse array of characteristics. They performed a pilot study comparing their approach to that of simply randomly selecting devices. The results showed that their proposed approach detected more issues than random device selection in the majority of cases, and in only one case did random device selection perform better.

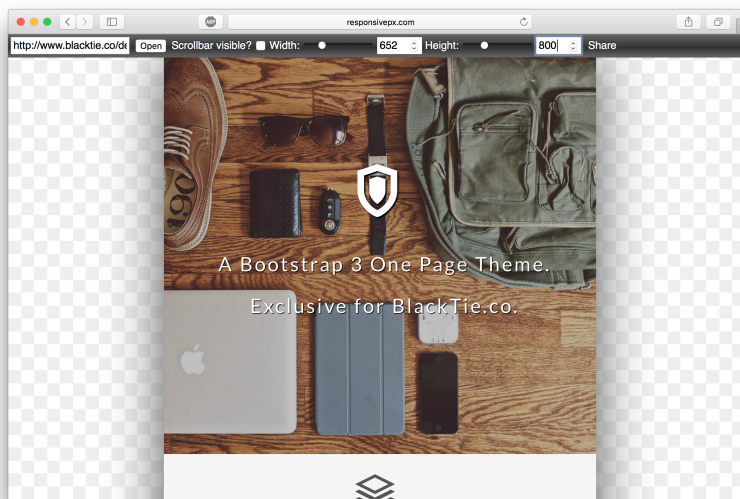
Once a developer has selected a set of test devices, two options are available. Firstly, they could purchase all of the devices and perform all the testing “in-house”. While this is probably the most convenient option, it is potentially a very expensive solution even with very few devices. The alternative is to use a “device lab”, where one can rent a huge array of different devices. This originally involved the testers travelling to the device lab itself. However, the recent advent of “device labs in a box” means the testing can again happen in-house, but without the long term financial commitment of purchasing a set of devices [82]. For smaller development companies with lower budgets and fewer web sites to test, this can be a more sensible choice. They can reap the majority of the benefits of testing on real devices at a substantially lower cost.

If obtaining real devices for testing is not possible, using *emulators* is a viable alternative. Emulators allow the developer or tester’s host machine to mimic the behaviour of a particular system or device. In this scenario, they mimic the different devices the user wishes to test their web page on. There are several options available to a developer. The iOS Simulator allows the user to test their software on a variety of Apple devices, such as the iPhone and iPad. It also has the ability to change the software and hardware combinations used. Unfortunately, it is only usable on Apple machines, so developers and testers running different operating systems must use alternative solutions. Similarly, the Android Emulator, included with the Android Software Development Kit (SDK) [72], allows for testing on a range of Android devices. A very comprehensive emulator, it allows for the simulation of incoming calls, location specification, different network speeds and device rotation, among many other features and interactions.

Third-party services such as BrowserStack [21] provide access to hundreds of real devices in the cloud. Developers can then pay a subscription fee to use them for testing purposes. Importantly, BrowserStack recognises the fragmentation of the mobile device landscape. They use insights gained by analysing the usage data of 36,000 customers and global market trends to produce three different “tiers” of device testing. They advocate testing on steadily wider varieties of popular devices to achieve higher levels of market coverage. This is a similar approach to Khalid et al. [80], but using actual web-specific usage data rather than app download data. Even the lowest tier achieved a respectable level of coverage (40% of viewports and 60% of device sizes). This further indicates the potential benefits of prioritising the test devices used.

Most emulators allow for in-depth interaction with the web page under test and the emulated device. They are therefore well suited for testers who want to perform a more thorough quality assurance process. They can emulate a huge variety of devices without the costs associated with actually buying the physical devices. Finally, they can also be useful for recreating reported bugs or issues on a particular device or software version. This makes debugging much easier, especially if the device in question is unavailable to the developer or tester.

Viewport resizers allow a user to resize the effective viewport of the browser to a certain dimension. The user can then inspect the web page's appearance for issues. Some only allow for resizing to the sizes associated with common devices (e.g., Window Resizer [155]). However, others such as RESPONSIVEPX [132] allow for the fine tuning of the viewport size on a pixel-by-pixel basis, as shown by Figure 2.12. This can be useful for targetting specific devices and for refining media queries to improve the overall responsive behaviour. They can be a good option if specific devices are not required; instead a user can test at common viewport widths instead. Unfortunately, the manual effort required is large. The user must regularly resize the browser itself and perform the manual inspection of the layout. Despite this, most modern browsers now have viewport resizing functionality built-in. For instance, Firefox has "Responsive Design View" [3] and Chrome has "Device Mode" [15]. This indicates developers and testers do find them useful in practice.



**Figure 2.12:** RESPONSIVEPX, an example of a viewport resizer.

Screenshot tools are essentially simplified viewport resizers. Given a URL, they render the web page at a number of common resolutions. These often represent



different categories of device. They therefore allow a developer to obtain a general impression of how their site appears on different types of device. However, they present the web page at only around 5 resolutions. This makes them incapable of providing the levels of coverage required to adequately test a responsive web page. A common example is the online tool developed by Matthew Kersley [78]. This renders a web page at screen widths of 240px, 320px, 480px, 768px and 1024px, as shown by Figure 2.13. It does not however, give the developer the option to change the viewport widths tested.

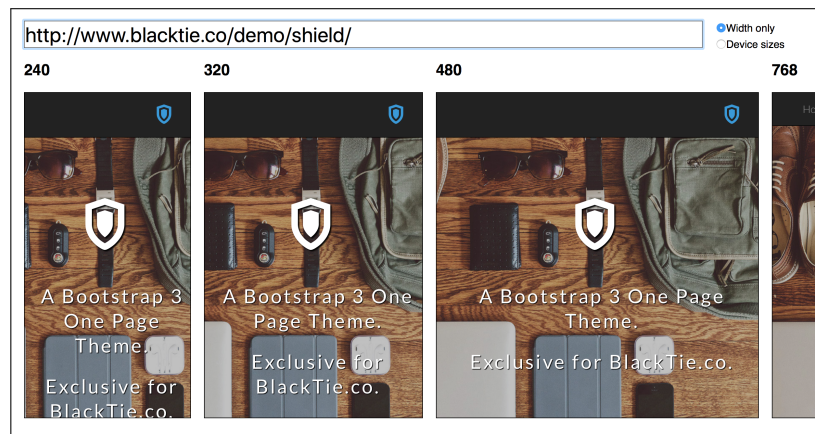


Figure 2.13: Kersley's Screenshot Tool

### 2.5.3 Applying Previous Techniques

The techniques of Mahajan et al., are effective for detecting general presentation failures between a design mockup of a web page and its actual appearance. However, they would unfortunately be very difficult to apply to a mobile testing scenario, as they operate at a single viewport width only. They would require design mockups at a very high number of widths, making the approach massively labour-intensive.

The approaches for detecting internationalisation presentation failures and cross-browser incompatibilities are also unsuitable. One could modify an approach to render the web page under test in the same browser at two different viewport resolutions, rather than in different languages or different browsers. However, the approach would likely report a large amount of issues. These would however be deliberate on the part of the developer, as the appearance of the web page is intentionally different on devices of different sizes.

CORNIPICKLE allows developers to define the intended layout of a web page. Unfortunately, the types of presentation failures it targets are generally static in nature and it does not allow a developer to specify mobile-specific rules. This makes it unsuitable for detecting failures manifesting at different viewport widths. GALEN on the other hand, is specifically designed for detecting presentation failures in mobile web pages. It allows a user to describe the intended appearance of the web page under test when viewed on devices of different sizes. However, the main issue with GALEN is its labour-intensive nature. A developer would have to spend a long time writing the initial specification for the web page to cover all aspects of its appearance. Not only this, but any changes to the web page would likely result in lots of failures being erroneously reported. The developer would then have to spend more time modifying the specification to match the new appearance. This currently limits its real-world usefulness.

In conclusion, the current literature presents a variety of techniques addressing the problem of detecting presentation failures in web applications. However, there is a dearth of applicable techniques for the important problem domain of mobile web applications. Furthermore, the array of developer tools are insufficient for accurate and reliable detection of mobile presentation failures. For this reason, it is the focus of this thesis.

## 2.6 GENERAL WEB APPLICATION AND GUI TESTING

Before concluding the literature review, this section looks at general web application testing research. It also looks at previous testing approaches developed for other types of graphical user interface (GUI) application.

Automated web testing has, in general, been drastically under-researched. Garousi et al's survey [52] evidenced this, as despite covering a decade-long period it obtained a pool of just 79 publications to survey. However, this number was the result of applying exclusion criteria. There were initially 147 publications in the survey. In the five years since, more research has been conducted, but many key problems in automated web testing remain open.

Because presentation failures are one of the largest categories of failure in web sites [56], much research has been conducted to try to test and analyse web sites. Wang et al.'s technique tried to improve the appearance of web applications [151], but presentation failures still frequently occur even with such

approaches. Therefore, several papers attempted to address the problem [59, 60, 106, 152, 153]. However, these were not tailored to identify failures in a web page's responsive layout. More recent research such as that of Dallmeier [41], Milani Fard [108] and Wang [150] do not even focus on the page's appearance, let alone addressing the specific problems associated with responsive design.

In 2003, Memon et al. proposed an approach called *GUI ripping*. This generated a representation of both the appearance and behaviour of an application. The approach then used this model to automatically create complex test cases which could test the application [102]. Several models made up the representation of the GUI. The first of these was the GUI forest, which was a representation of the windows making up the GUI. However, the GUI forest alone was not enough to generate test cases. Therefore, the approach next determined the various event types compatible with each component of the GUI. It then collated these different events in an event flow graph (EFG). Finally, it generated an integration tree (IT) to model the interactions between the different GUI components. This allowed for a fuller understanding of the overall behaviour of the application.

The approach traversed the generated EFG and IT models to simulate a real user. To create test cases it enumerated the different events encountered. To ensure a satisfactory level of testing, it used coverage criteria to guide the test generation. Once generated, one could execute the tests against modified versions of the application. Memon's DART tool [101] demonstrated this regression testing, which allowed developers and testers to conduct nightly build testing. Results showed it to be highly useful for detecting bugs.

The test case generation was substantially improved in 2007 through the use of *usage profiles*. These are sequences of events executed on a system by a real-world end user [20]. The motivation and thought process behind the technique was as follows. By generating test cases with a high occurrence probability in the final application, any faults detected are likely to be ones that an end user would likely experience. The technique involved modifying the event flow graph from [101, 102] and labelling each edge with a probability table showing the prior probability and several conditional probabilities of the edge's event occurring. This resulted in a probabilistic event flow graph (PEFG). This reduced the overall test suite size required to reach a certain coverage level drastically. The generated test cases also detected many more faults per test case than other approaches.

Faria et al [55] also researched the possibilities of reverse engineering GUI models for testing purposes. Like Memon, their technique began with the automatic exploration of the application to generate a model of the GUI. However, their approach differed considerably from that point on. The model was manually completed and validated, to ensure it missed no functionality. This also prevented the model from inheriting any bugs in the system. The model was then used to create an abstract test suite to test the GUI of the complete application. Unfortunately, the complexity of modern applications caused significant problems. A disappointingly large amount of the model had to be created manually (around 50%). This made use of the technique fairly labour intensive.

More recently, Moreira et al. [112, 113] have proposed *pattern-based GUI testing (PBGT)*, which aimed to provide a more platform or implementation-independent approach to testing. They provided a language with which developers and testers could build models of the application under test based on common UI test patterns, such as user input boxes and login forms. Once complete, the approach used this model to automatically generate test cases that exercise the application under test. Initially, work was focussed on web applications, but more recently the authors have adapted the approach to work on mobile applications [114].

Research has also investigated the automatic testing of the GUIs of mobile applications. For instance, Amalfitano et al. [9] extended the GUI ripping technique and applied it to Android applications. One key difference, however, is the approach did not generate a model during the exploration of the GUI. Instead, it generated test cases as it discovered new events and interactions. Amalfitano et al. also investigated GUI failures caused by changing the orientation of the mobile device, finding them to be prevalent in a wide variety of popular Android applications [10]. Hu et al. used dynamic analysis and log file analysis to detect GUI issues [71]. Meanwhile, Mirzaei et al. used symbolic execution — a means of analysing a program to determine what inputs cause each part of a program to execute — to perform their Android testing [110].

Moran et al. [79] implemented a technique similar to that of Mahajan et al. [90] for detecting differences between the mockup image of a mobile applications and its actual appearance, which they referred to as *GUI design violations (DVs)*. The authors identified a taxonomy of design violations, with the main categories being *text* (eg. font colour and style), *resource* (eg. incorrect image) and *layout* (eg. incorrect size or position). The approach matched the GUI components on each screen and then compared them using PID. Then, using the set

of difference pixels, the approach checked whether the differences constituted a design violation. Finally, it generated violation reports for the developers and designers of the web page containing written descriptions of the issue, annotated screenshots and links to the responsible lines of source code. To evaluate the approach, the authors also developed a “synthetic design violation injection tool” that modified the code of an application to introduce DVs. Results showed the approach could detect DVs with extremely high precision and recall. Finally, an industrial study of real-world developers using the tool implementation of the approach showed it to have a noticeable positive impact on the workflow, resulting in applications with better user interfaces.

Fazzini and Orso employed a similar approach to Choudhary et al. when testing mobile applications [50]. Rather than searching for *cross-browser* incompatibilities, they targetted *cross-platform* incompatibilities (CPIs). Their approach crawled a mobile application on a reference platform (device and OS combination) to build a model of the UI. The approach then converted this model into a test case, before executing this test case on a series of test platforms and generated UI models for each. It finally compared the collection of models to detect any CPIs in the application. The comparison checked for both functional consistency by matching states across the models, as well as visual consistency by comparing the actual appearance of elements in the application. Results showed the approach capable of finding 96 CPIs across 5 applications and 140 different platforms, with minimal misleading false positives.

Meniar et al. [103] proposed a vastly different approach. They utilised the fact almost every technology ecosystem has a set of guidelines dictating how the user interface of software applications should look and behave. For instance, Apple has guidelines for iOS [13], just as Google does for Android [12]. Following these guidelines should in theory guarantee a reasonably usable, consistent and aesthetically pleasing user interface. Meniar et al. proposed the first automated approach for verifying an application follows these guidelines. The tool, CORNIDROID, checks the appearance of an application in real-time and reports any violated guidelines to the developer of the application.

Joorabchi and Mesbah proposed an approach for reverse engineering models of iOS applications [77]. It generated a model of the application, with the individual UI “states” acting as the nodes of the model. Transitions between them corresponded to interactions with the application. Although not yet implemented, the authors plan to develop an automated test case generation technique using the reverse-engineered models.

Researchers have also used image comparison for the purpose of general GUI testing. One example is SIKULI [29], a unit-testing framework for GUI applications. Unlike testing tools such as Selenium [130], Sikuli used image comparison rather than specific hard-coded “selectors” to identify the various components to interact with on the page. It also used image comparison to detect and verify the expected behaviour. As it was platform independent, users could apply the approach not only to websites but to native applications too. However, as each step only tested a small part of the web page, performing complete testing on the overall application was infeasible.

While these techniques performed well in their own problem domain, very few of them focussed on the application’s actual appearance. Instead, the majority of them focus on the functionality of the GUI. Coupled with the environment-specific nature of the techniques (web applications are very different to desktop and mobile applications, for example), this makes them unsuitable for addressing presentation failures in mobile-friendly web applications.

## 2.7 CONCLUDING REMARKS

Researchers have proposed many techniques to tackle the problem of detecting presentation failures in web applications. The foundations of these approaches have often differed drastically. Some depended on computer vision and image comparison techniques, others on graph-based representations of the application under test. A couple were dependent on logic-based formal specifications. They have also targetted a variety of subcategories of presentational issues. These include cross-browser incompatibilities and internationalisation presentation failures.

There has been little to no research conducted attempting to address the issue of mobile presentation failures, especially when it comes to detecting failures automatically. This is despite the prevalence of “mobile-friendly” web sites. Furthermore, all of the approaches targetting the orthogonal issues mentioned above are unsuitable for the problem domain of mobile presentation failures. Some are far too labour-intensive while others are not equipped to handle the nature of mobile web applications. Approaches for detecting general presentation failures that use the rendering of the web page such as WEBSEE [89–91] and WRAITH [14] require a large number of oracles. They also make no consideration of a web page’s responsive design. XBI detection approaches such as

WEBDIFF [33], CROSS T [105], CROSSCHECK [32] and X-PERT [123, 124] are completely unsuitable for mobile presentation failures, because a web page's layout may intentionally change at different screen sizes. Finally, specification-based approaches such as CORNIPICKLE [61, 62] and GALEN [133] allow for automatic detection of presentation issues in web pages. However, their performance is entirely dependent on the quality of the specification. Creating and maintaining a high quality specification is a highly labourious task.

There is currently no suitable automated approach specifically tailored for the problem of mobile presentation failures. Because of this, humans generally check for presentation failures manually. The array of tools available to support this process are useful, but all suffer from shortcomings. Therefore the remainder of this thesis addresses this gap in the literature, developing automated approaches for identifying presentation failures in mobile-friendly web pages.





---

## MODELLING RESPONSIVE LAYOUT

---

The previous chapter outlined the various existing approaches to detecting presentation failures in web pages. In particular, it discussed their shortcomings for the problem domain of mobile-friendly responsive web pages. For instance, some only operated at a single viewport width rather than considering a range of viewport widths, as is required by this scenario. Others required a substantial amount of manual effort to use.

To address these issues, this chapter presents the Responsive Layout Graph (RLG), which models the dynamic responsive layout of a web page across a series of viewport widths. The RLG takes into account both the changing visibility and relative alignment of elements, the two main aspects of responsive web design. The RLG can then form part of a variety of testing techniques, as demonstrated in later chapters.

To begin, the chapter presents formal definitions of the various components of the RLG and explains them with the use of a worked example, before defining and describing the algorithms used to automatically extract the RLG for a given web page.

The key contributions of this chapter are:

1. The formal definition of a model of a web page's responsive layout, called the **responsive layout graph**, which describes both the visibility and relative alignment of the elements on the web page across many viewport widths.
2. A series of algorithms to automatically extract the RLG of a web page.

## 3.1 THE RESPONSIVE LAYOUT GRAPH

## 3.1.1 Formal Definition

Given a web page  $\mathcal{W}$  containing elements  $\mathcal{E}$ , two types of layout constraint can be defined to represent the responsive layout of  $\mathcal{W}$ :

**VISIBILITY CONSTRAINT:** A visibility constraint  $vc$ , for some node  $e \in \mathcal{E}$ , is a pair  $(x_1, x_2)$  representing an inclusive range of viewport widths for which  $n$  is present in the DOM of the web page and has properties making it visible when the web page is rendered in a browser.

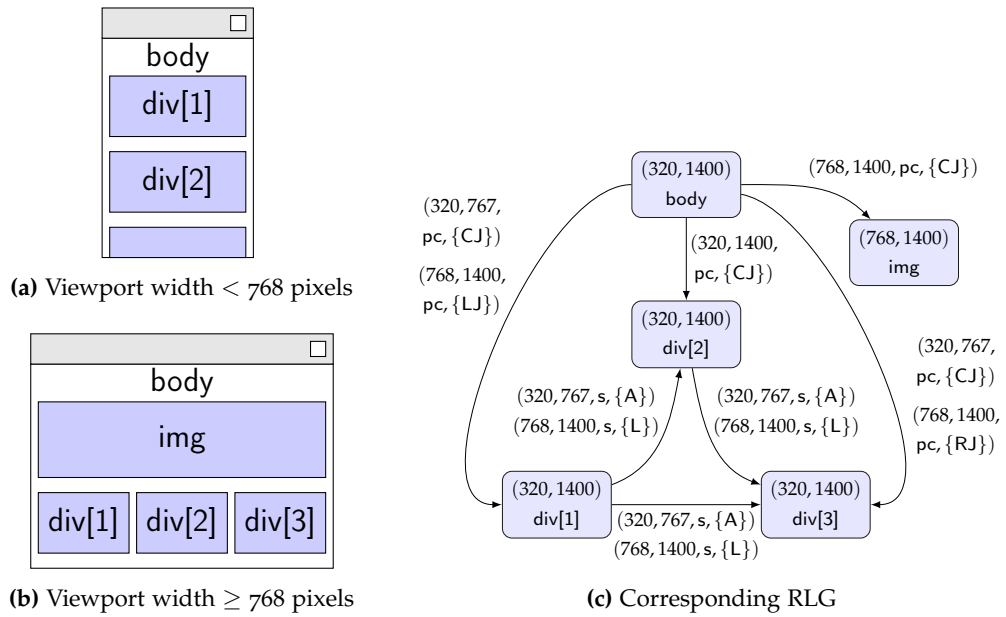
The second definition, the *alignment constraint* draws on the notion of relative layout first proposed by Choudhary et al. [123] and described in Chapter 2, in particular the concepts of *parent-child* and *sibling* relationships between web page elements, and the set of alignment attributes  $P$ , which describe the specifics of a particular relationship.

**ALIGNMENT CONSTRAINT:** Given a pair of RLG nodes  $e_1, e_2 \in \mathcal{E}$ , an alignment constraint  $ac$  is a 4-tuple  $(x_1, x_2, t, P)$  representing that between the inclusive viewport widths of  $x_1$  and  $x_2$ ,  $e_1$  and  $e_2$  are laid out in a relationship of type  $t$ , with alignment attributes  $P$ , where  $P \in 2^{\mathcal{Q}}$ .

Alignment constraints represent the dynamic nature of responsive web pages, so multiple alignment constraints describing different layout behaviours can exist between a single pair of nodes, detailing the changes in their relative layout caused by the changing viewport.

Please note, in both of the above definitions,  $x_1$  represents the lower bound of the constraint and so the condition  $x_1 \leq x_2$  holds for all valid constraints.

Due to the inherent hierarchical nature of web pages, in which the `<body>` element acts as the “root” presentational element of the web page and therefore contains all other web page elements, the RLG is also hierarchical in nature. Nodes represent individual web page elements and are arranged in a tree structure, connected by edges. Visibility constraints are then added to their respective nodes, while alignment constraints are mapped onto the edges connecting the nodes whose layout they describe.



**Figure 3.1:** A wireframe example web page and its RLG.

To explain further, Figure 3.1 presents a simple wireframe responsive web page at two different viewport widths, along with the RLG representing its layout. Each web page element in the wireframes is represented by a node in the graph, in which the labels uniquely identify each element, as the full XPath expressions have been omitted to save space and maintain readability. The web page contains five main elements: the body, three content panels, `div[1]`–`div[3]`, and an image banner, `img`. At narrow viewport widths, the web page stacks the content panels one on top of the other, requiring the user to scroll to access all the content, while not displaying the `img` at all due to the space constraints. At wider viewports, it switches the panels to a side-by-side layout to make better use of the available space and the banner image is rendered above them, spanning the entire width of the page.

Figure 3.1 displays the visibility constraints above the node labels for each element. Generally,  $w_{min}$  and  $w_{max}$  represent the smallest and largest viewport width sampled by the RLG, which in this example are 320 pixels and 1400 pixels, respectively. Therefore, elements visible at every viewport width, such as `div[1]`, have the visibility constraint (320,1400). Meanwhile, the `img` element only comes into view at wider viewport widths and thus has the visibility constraint (768,1400).

The RLG does have various similarities with previous approaches for modelling the layout of web pages, such as the alignment graph [123] and layout

graph [7], including the concepts of element containment and relative layout of neighbouring elements. However, there is one key difference that sets it apart. The RLG models the entire range of viewport widths at which the web page might be viewed, while the other two only consider the layout at a single static viewport width. Theoretically, one could use either of these models multiple times in order to test at a variety of different viewport widths. However, this approach would be prone to missing potential presentation failures occurring inbetween a pair of test widths. For example, one could use the AG/LG at viewport widths of 320 pixels and 400 pixels. However, if a presentation failure only occurs between 350 pixels and 370 pixels, neither approach would be able to detect it. In contrast, by modelling the dynamic layout across a full range of viewport widths, the RLG can be used to detect presentation failures occurring at any viewport width that it models.

The directed edges connecting the RLG nodes are each labelled with *at least* one alignment constraint, describing the relative layout of the two nodes being connected by the edge. For instance, there are two alignment constraints on the edge between  $\text{div}[1]$  and  $\text{div}[2]$ ,  $(320, 767, s, \{A\})$  and  $(768, 1400, s, \{L\})$ . The former models the layout between the two elements at narrow viewport widths (320 pixels – 767 pixels), where  $\text{div}[1]$  is above (A)  $\text{div}[2]$ , while the second represents the shift in layout for wider viewports, where  $\text{div}[1]$  is now to the left of (L)  $\text{div}[2]$ .

An RLG can be formally defined as a 4-tuple  $\mathcal{RLG} = (\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ .  $\mathcal{E}$  represents the complete set of nodes in the graph, one for each web page element visible on the page at some viewport width.  $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$  is the set of edges between elements for which at least one alignment constraint exists. Next,  $\mathcal{F}_{VC}: \mathcal{E} \rightarrow 2^{\mathcal{V}C}$  is a function that maps an element to a set of visibility constraints where  $\forall e \in \mathcal{E}$ ,  $|\mathcal{F}_{VC}(e)| \geq 1$ . In other words, each element is mapped to at least one visibility constraint and is therefore visible on the web page for at least one viewport width. Similarly,  $\mathcal{F}_{AC}: \mathcal{R} \rightarrow 2^{\mathcal{A}C}$  maps the relationship edges to their sets of alignment constraints, such that  $\forall r \in \mathcal{R}$  and  $\forall ac_a = (x_{a1}, x_{a2}, t_a, \mathcal{P}_a) \in \mathcal{F}_{AC}(r)$  and  $\forall ac_b = (x_{b1}, x_{b2}, t_b, \mathcal{P}_b) \in \mathcal{F}_{AC}(r)$ , if  $ac_a \neq ac_b$ , then  $x_{a1} \geq x_{b2} \vee x_{a2} \leq x_{b1}$ . Put simply, given that the alignment constraints mapped to an edge never have overlapping bounds, for a particular viewport width, there is at most one alignment constraint which holds true.

### Modelling Other Characteristics

The RLG models the two main characteristics of responsive design: visibility and alignment. There are obviously several other characteristics that the RLG could model. This section discusses some of these and explains why the RLG does not model them.

One obvious example is the *width* of elements, as this is one of the main characteristics that changes in a responsive web page. An initial version of the RLG modelled the dynamic widths of elements through *width constraints* [149]. However, they are no longer present in the formal definition for two main reasons. Firstly, after conducting initial experiments using the approach presented in Chapter 4, it was apparent that in scenarios when the only change in the RLG was a width constraint, there was no observable presentation failure in the web page. Secondly, a large change in the width of an element will almost certainly cause a significant change in the relative alignment of nearby elements. Therefore, the RLG can detect any presentation failures using only visibility and alignment constraints.

Another alternative was to model the height of elements as well. While browsing a responsive web page, it is immediately apparent the height of elements may fluidly change as the viewport expands or contracts. However, this is almost always due to the browser re-organising content automatically in response to the new viewport width. The advent of mobile-friendly web sites means a web page does not generally have a fixed height. Instead, the web page adapts its height to the device in use. There is then a reasonable assumption that the user will scroll vertically to access any content further down the page. This assumption has led to many developers no longer implementing web “sites”, instead developing single-page, vertical-scrolling web pages, which have recently become a more mainstream trend [37].

Other models of the GUI of applications have represented information such as colour and font size among many others. However, the RLG does not. This is for two simple reasons. The first is that these characteristics frequently do not change as the viewport expands and contracts i.e, they are not responsive characteristics. Secondly, the RLG is an abstraction of the actual visual appearance of the web page under test. As the subsequent sections will describe, the RLG is obtained using the DOM of the web page, in particular the coordinates of each element. By doing this, the complex visual appearance of modern web pages is converted into a simple collection of rectangles in two-dimensional space.

**Algorithm 1** RLG Extraction

---

```

1: procedure EXTRACTRESPONSIVELAYOUTGRAPH( $\mathcal{W}, w_{min}, w_{max}, w_{step}$ )
2:    $\mathcal{RLG} \leftarrow$  INITIALISERLG()
3:    $\mathcal{S} \leftarrow$  GETSAMPLEWIDTHS( $\mathcal{W}, w_{min}, w_{max}, w_{step}$ )
4:    $\mathcal{D} \leftarrow$  COLLECTDOMS( $\mathcal{W}, \mathcal{S}$ )
5:    $\mathcal{L} \leftarrow$  EXTRACTLAYOUTSFROMDOMS( $\mathcal{D}$ )
6:    $\mathcal{VC} \leftarrow$  EXTRACTVISIBILITYCONSTRAINTS( $\mathcal{L}$ )
7:   for  $(e, vc) \in \mathcal{VC}$  do
8:      $\mathcal{E} \leftarrow \mathcal{E} \cup \{e\}$ 
9:     MAPVISIBILITYCONSTRAINT( $vc, e$ )
10:  end for
11:   $\mathcal{AC} \leftarrow$  EXTRACTALIGNMENTCONSTRAINTS( $\mathcal{L}$ )
12:  for  $(r, ac) \in \mathcal{AC}$  do
13:     $\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$ 
14:    MAPALIGNMENTCONSTRAINT( $ac, r$ )
15:  end for
16:  return  $\mathcal{RLG}$ 
17: end procedure

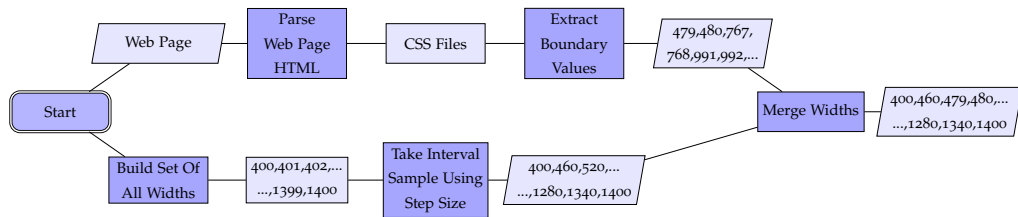
```

---

Furthermore, the model focusses solely on **layout**, meaning there is little to be gained from modelling other visual characteristics such as colour.

### 3.1.2 Extracting the Responsive Layout Graph

Given a web page under test  $\mathcal{W}$ , Algorithm 1 can derive the set of HTML elements  $\mathcal{E}$ , along with the respective sets of both visibility and alignment constraints  $\mathcal{VC}$  and  $\mathcal{AC}$ , to form a complete RLG. As well as the web page under test, the algorithm also takes as input three parameters,  $w_{min}$ ,  $w_{max}$  and  $w_{step}$ , which represent the minimum viewport width, maximum viewport width and sampling step size, respectively. The first step is to determine the set of widths  $\mathcal{S}$  at which to sample the layout of  $\mathcal{W}$  (line 3). The algorithm then obtains the DOM of the web page at each width (line 4) and stores it in the set  $\mathcal{D}$ , before extracting the web page layouts from those DOMs and storing them in the set  $\mathcal{L}$  (line 5). Next, the algorithm analyses the layouts to extract the visibility constraints  $\mathcal{VC}$  (line 6), with the relevant nodes and constraints added to the RLG (lines 7-10). This process then repeats for the extraction of alignment constraints  $\mathcal{AC}$  (line 11). Once the algorithm has added the edges and alignment constraints



**Figure 3.2:** The two-part sampling process.

In this diagram, inputs and outputs are shown as parallelograms, automatic processes are depicted as dark-grey rectangles and entities derived from inputs are light-grey rectangles. Rectangles and parallelograms which have numerical values inside them (e.g., “400,460,520, . . . ,1280,1340,1400”) contain viewport widths at which the automated technique will sample the DOM of the web page under test.

to the model (lines 12-15), it returns the complete RLG. The following sections now describe each individual step in thorough detail.

### Sampling the Web Page

Choosing the viewport widths at which to sample a web page is a difficult task. This is because each web page is different and may exhibit different responsive layout behaviour. Manually selecting the sample widths would not only be labourious, but also prone to missing parts of the layout behaviour. One simple approach would be to sample on either side of each breakpoint defined in the page’s CSS. Unfortunately, as elements respond to the changing viewport constraints, layout changes not programmed in the CSS can occur. Breakpoint boundary sampling would fail to observe these, so the sampling technique implemented in the method `GETSAMPLEWIDTHS` uses a combination of sampling approaches. Figure 3.2 illustrates this approach which aims to obtain the best possible sample of the layout.

First, the function takes the web page  $\mathcal{W}$  and inspects its HTML code to get the set of CSS files used to style the page. It then takes each CSS file and finds the media queries contained within it. Next, it extracts the boundary values of each query and adds them to the sample width set  $\mathcal{S}$ . For instance, the media query `@media (min-width: 992px)` would add the widths 991 pixels and 992 pixels to  $\mathcal{S}$ , while `@media (max-width: 767px)` would add 767 pixels and 768 pixels. Then, to sample the layout changes not defined by explicit breakpoints, the function also performs a systematic sampling of the web page. This covers the entire sample range ( $w_{min}$  to  $w_{max}$ ), using a step size  $w_{step}$ , as shown by the bottom half of Figure 3.2. The sample widths chosen from both techniques are then merged to form the final sample set. This is then used throughout the RLG extraction process.

The function extracts the DOM of the web page at each of these widths. This produces a sequence of pairs  $\mathcal{D}$  of the form  $(w, d_w)$ , where  $w$  is the viewport width and  $d_w$  is the DOM of the webpage as captured at  $w$ . The set below shows a general example.

$$\mathcal{D} = \langle (w_{min}, d_1), (w_{min} + w_{step}, d_2), (w_{min} + (w_{step} \times 2), d_3), \dots, (w_{n-1}, d_{n-1}), (w_{max}, d_n) \rangle$$

Given the media query `@media(min-width:768px)`, a  $w_{step}$  value of 60, and  $w_{min}$  and  $w_{max}$  values of 320 and 1400, the set  $\mathcal{D}$  would be:

$$\mathcal{D} = \langle (320, d_1), (380, d_2), (440, d_3), (500, d_4), (560, d_5), (620, d_6), \dots, (767, d_x), (768, d_{x+1}), \dots, (800, d_{x+2}), (860, d_{x+3}), (1340, d_{n-1}), (1400, d_n) \rangle$$

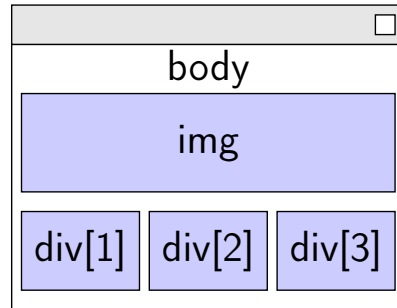
### Obtaining the Page Layout

At each sample width, the approach extracts and analyses the DOM to extract the page layout, using the bounding boxes of each element, which have coordinates  $(x_1, y_1, x_2, y_2)$ . To illustrate this process, this section revisits the wireframe example in Figure 3.1.

Consider the layout shown by part (a-ii) of Figure 3.1, which Figure 3.3(a) repeats for ease of reference. The function begins by using the coordinates of each element to convert the DOM of the web page into an R-tree [58]. This is essentially an abstraction where the size and location of each element is a rectangle in 2-dimensional space. It then analyses the R-tree to organise the elements into a tree structure depending on which elements are contained within others. After this, it connects the elements with either parent-child or sibling edges. In the example, parent-child edges exist between body and each of `img`, `div[1]`, `div[2]` and `div[3]`. Meanwhile, sibling edges exist for each combination of the four smaller elements as they all share the same parent.

Next, the function inspects the coordinates of each pair of connected nodes to extract any alignment attributes. Child elements may be left (LJ), right (RJ) or centre-justified (CJ) within their parents and may also be top (TJ), bottom (BJ) or middle-justified (MJ). Sibling attributes fall into two categories, positioning and alignment. Positioning labels describe the location of one element in relation to the other, while alignment labels describe how the borders of each element align with each other. If an element is either left-of (L) or right-of (R) its sibling, it may also align on its top (TE) and/or bottom edge (BE). Similarly, if an element is above (A) or below (B) its sibling, it may also align on its left (LE) and/or right edges (RE).





(a) Example web page

Relationship	Type	Attributes
body $\rightarrow$ img	pc	CJ
body $\rightarrow$ div[1]	pc	LJ
body $\rightarrow$ div[2]	pc	CJ
body $\rightarrow$ div[3]	pc	RJ
div[1] $\rightarrow$ div[2]	s	L, TE, BE
div[2] $\rightarrow$ div[3]	s	L, TE, BE
img $\rightarrow$ div[1]	s	A, LE

(b) The extracted layout

**Figure 3.3:** An example web page and its extracted layout.

In terms of parent-child alignments, `img` spans the entire width of `body`, resulting in the centre-justified (CJ) attribute. Likewise, `div[1]`, `div[2]` and `div[3]` are left, centre and right justified within `body`, respectively, hence the LJ, CJ and RJ attributes. For the sibling edges, `div[1]` is to the left of `div[2]` (determined by observing `div[1]`'s  $y_2$  coordinate is greater than `div[2]`'s  $y_1$  coordinate), hence the L attribute applied to the relevant edge. The `img` element is *above* the other elements, as `img`'s  $x_2$  coordinate is smaller than the  $x_1$  coordinates of the other elements. It also aligns with `div[1]` on its left edge, resulting in the attribute set {A, LE}. The remaining attributes can be calculated through simple comparisons of the element coordinates.

It is important to note sibling relationships between web page elements are symmetrical in nature. For instance, two edges representing *X is above Y* and *Y is below X* are intuitively identical. Therefore, for simplicity and efficiency, the RLG only models one of the two possible edges to define the relationship. It then checks for the existence of *symmetric* edges, during the process of extracting alignment constraints defined in Section 3.1.2.

The function `EXTRACTLAYOUTSFROMDOMs` obtains the layout from the DOM at each viewport width, resulting in a sequence of web page layouts,  $\mathcal{L}$ . Algorithm 1 then proceeds to extract both visibility and alignment constraints.

### Extracting Visibility Constraints

Algorithm 2 shows the process for extracting visibility constraints. It takes the sequence of web page layouts as input and returns  $\mathcal{VC}$ , the complete set of visibility constraints. The algorithm inspects the elements visible in each layout to determine the visibility constraints.

**Algorithm 2** Visibility Constraint Extraction

---

```

1: procedure EXTRACTVISIBILITYCONSTRAINTS( $\mathcal{L}$ )
2:    $\mathcal{VC} \leftarrow \{\}$ 
3:    $(w_p, l_p) \leftarrow \mathcal{L}.pop()$ 
4:   for  $e \in l_p$  do
5:      $\mathcal{VC} \leftarrow \mathcal{VC} \cup \{(e, (w_{min}, -))\}$ 
6:   end for
7:   while  $|\mathcal{L}| \neq 0$  do
8:      $(w_c, l_c) \leftarrow \mathcal{L}.pop()$ 
9:      $U_p \leftarrow \{\}, U_c \leftarrow \{\}$ 
10:    for  $e_p \in l_p$  do
11:      if  $e_p \in l_c$  then
12:         $l_c \leftarrow l_c \setminus \{e_p\}$ 
13:      else
14:         $U_p \leftarrow U_p \cup \{e_p\}$ 
15:      end if
16:    end for
17:     $U_c \leftarrow l_c$ 
18:    for  $e_{new} \in U_c$  do
19:       $(w_{invis}, w_{vis}) \leftarrow \text{BINARYSEARCHFORLAYOUTCHANGE}(e_c, w_p, w_c, \text{appear})$ 
20:       $\text{ADDNEWCONSTRAINT}(e_{new}, (w_{vis}, \perp), \mathcal{VC})$ 
21:    end for
22:    for  $e_{dis} \in U_p$  do
23:       $(w_{vis}, w_{invis}) \leftarrow \text{BINARYSEARCHFORLAYOUTCHANGE}(e_{dis}, w_p, w_c, \text{disappear})$ 
24:       $\text{UPDATECONSTRAINT}(e_{dis}, w_{vis}, \mathcal{VC})$ 
25:    end for
26:     $(w_p, l_p) \leftarrow (w_c, l_c)$ 
27:  end while
28:  for  $e \in l_p$  do
29:     $\text{UPDATECONSTRAINT}(e, w_p, \mathcal{VC})$ 
30:  end for
31:  return  $\mathcal{VC}$ 
32: end procedure

```

---

The algorithm begins with the first layout, corresponding to  $w_{min}$ . Iterating through the elements found, it initialises visibility constraints for each one and adds them to the set  $\mathcal{VC}$ . Each entry in  $\mathcal{VC}$  is an element-constraint pair  $(e, (w_a, w_d))$ , where  $e$  and  $(w_a, w_d)$  represent the element and the visibility constraint, respectively. When first added to  $\mathcal{VC}$ , these constraints have the value  $(w_{min}, \perp)$ . The  $x_1$  value of  $w_{min}$  represents the elements being visible at the first viewport width sampled. Meanwhile, the  $x_2$  value is currently unknown, as denoted by  $\perp$ . This is because the algorithm has not yet inspected other layouts to determine if and when the element disappears from the web page. The algorithm then updates  $\mathcal{VC}$  as it processes each subsequent layout in  $\mathcal{L}$ , as described next.

The algorithm then takes  $l_c$ , the next layout in  $\mathcal{L}$ , and compares it to the previous one,  $l_p$ . This involves trying to match each element in  $l_c$  to an element in  $l_p$ ,

---

**Algorithm 3** Binary Search for Layout Change

---

```

1: procedure BINARYSEARCHFORLAYOUTCHANGE( $k, l, u, b$ )
2:   if ( $u - l > 1$ ) then
3:      $mid \leftarrow (l + u) / 2$ 
4:      $d_{mid} \leftarrow \text{EXTRACTDOM}(mid)$ 
5:      $l_{mid} \leftarrow \text{GENERATELAYOUT}(d_{mid})$ 
6:     if ( $k \in l_{mid} \wedge b = \text{appear}$ )  $\vee$  ( $k \notin l_{mid} \wedge b = \text{disappear}$ ) then
7:       BINARYSEARCHFORLAYOUTCHANGE( $k, l, mid, b$ )
8:     else
9:       BINARYSEARCHFORLAYOUTCHANGE( $k, mid, u, b$ )
10:    end if
11:  else
12:    return ( $l, u$ )
13:  end if
14: end procedure

```

---

using the unique XPath of the elements (lines 9-17). Once the algorithm has attempted to match all the elements, two sets  $U_p$  and  $U_c$  remain. These represent the unmatched web page elements from  $l_p$  and  $l_c$ , respectively. If both  $U_p$  and  $U_c$  are empty, then the algorithm has matched all the elements and it stops processing the layout  $l_c$ . If  $U_c$  is *not* empty, then each element  $e_{new} \in U_c$  — an element present in  $l_c$  but not in  $l_p$  — has become visible on the web page between the widths of  $w_p$  and  $w_c$ . To determine the exact width at which  $e_{new}$  becomes visible on the page, the algorithm calls the BINARYSEARCHFORLAYOUTCHANGE function (line 19). This function, illustrated by Algorithm 3, begins with the search bounds of  $w_p$  and  $w_c$  and searches for widths  $w_{invis}$ ,  $w_{vis}$  such that  $e_{new}$  is present at  $w_{vis}$  but not at  $w_{invis}$ . This intuitively makes  $w_{vis}$  the *appearance point* of  $e_{new}$ . After finding the appearance point, Algorithm 2 adds a visibility constraint for  $e_{new}$  to  $\mathcal{VC}$  with an  $x_1$  value of  $w_{vis}$  (line 20). For each disappearing element  $e_{dis} \in U_c$ , a similar search is performed. However, this time the search is for widths  $w_{vis}$ ,  $w_{invis}$ , where  $w_{vis}$  represents the final viewport width at which  $e_{dis}$  is visible. As seen by the calls on lines 19 and 23, Algorithm 2 passes a parameter with a value of either *appear* or *disappear* to set the target of the binary search. The incomplete visibility constraint for  $e_{dis}$  is then updated with an  $x_2$  value of  $e_{dis}$  (line 24).

Algorithm 2 repeats this process until it has inspected all layouts. Finally, it iterates through each element in the final layout and completes its visibility

constraint with an  $x_2$  value of  $w_{max}$  (lines 28-30). This models the element's visibility at the widest viewport represented by the RLG.

### Extracting Alignment Constraints

Algorithm 4 shows how the approach extracts alignment constraints. It again analyses the set of layouts  $\mathcal{L}$ , inspecting the relative layout of neighbouring elements. This is in contrast to Algorithm 2, which only investigated which elements are visible at each viewport width.

As with Algorithm 2, the algorithm begins by taking the first layout. It then creates partial alignment constraints of the form  $(w_{min}, \perp, t, a)$  for each observed layout relationship and adds them to the set  $\mathcal{AC}$ . Next, it iterates through the remaining layouts, attempting to find a match in  $l_c$  for each relationship in  $l_p$ . In this case, two relationships match if they are between the same two elements and have the same type and attributes. As with Algorithm 2,  $U_p$  and  $U_c$  represent the unmatched relationships from  $l_p$  and  $l_c$ , respectively. If either set is non-empty, the algorithm uses two heuristics to determine the cause of each unmatched relationship.

**SIMILAR EDGES:** The disappearance of one relationship can often correspond with the appearance of another at the same viewport width. A common example is when a pair of elements change their relative alignment. For instance, consider the relationship between `div[1]` and `div[2]` from Figure 3.1. The disappearance of the *above* relationship intuitively coincides with the appearance of the *left of* one. Therefore, as the elements change from a single-column to a single-row layout, the attribute sets change from {A} to {L}.

In an attempt to leverage this, Algorithm 4 searches for relationships where the elements and relationship type are the same, but the attribute sets differ. Algorithm 5 shows this process in the `PAIRUNMATCHEDRELATIONSHIPS` function. The algorithm then removes any matching relationships from their respective sets and adds them to the set of paired edges,  $P$ . Next, the algorithm does not find the appearance point of  $r_c$  and the disappearance point of  $r_p$  through two separate searches. Instead, it finds only the disappearance point of  $r_p$ ,  $w_{last}$ . It then assumes  $w_{last} + 1$  is the appearance point of  $r_c$ , removing the need for a second binary search. The algorithm then updates the constraint for  $r_p$  and creates the new constraint for  $r_c$  with their respective values.

**APPEARING/DISAPPEARING NODES:** When an element appears on a web page, intuitively its alignment relationships also appear at the same time. Algorithm 4

---

**Algorithm 4** Alignment Constraint Extraction
 

---

```

1: procedure EXTRACTALIGNMENTCONSTRAINTS( $\mathcal{L}$ )
2:    $\mathcal{AC} \leftarrow \{\}$ 
3:    $(w_p, l_p) \leftarrow \mathcal{L}.pop()$ 
4:   for  $r \in l_p$  do
5:     ADDNEWCONSTRAINT( $r, (w_p, \perp), \mathcal{AC}$ )
6:   end for
7:   while  $|\mathcal{L}| \neq 0$  do
8:      $(w_c, l_c) \leftarrow \mathcal{L}.pop()$ 
9:      $U_p, U_c \leftarrow \text{MATCHEDGES}(l_p, l_c)$ 
10:    if  $|U_p| + |U_c| \neq 0$  then
11:       $P \leftarrow \text{PAIRUNMATCHEDRELATIONSHIPS}(U_p, U_c)$ 
12:      for  $(r_p, r_c) \in \text{paired}$  do
13:         $(w_{last}, w_{first}) \leftarrow \text{BINARYSEARCHFORLAYOUTCHANGE}(r_p, w_p, w_c, \text{disappear})$ 
14:        UPDATECONSTRAINT( $r_p, w_{last}, \mathcal{AC}$ )
15:        ADDNEWCONSTRAINT( $r_c, (w_{first}, \perp), \mathcal{AC}$ )
16:      end for
17:      CHECKFORNODEBASEDCHANGES( $U_p, U_c, \mathcal{V}$ )
18:      for  $r_p \in U_p$  do
19:         $(w_{vis}, w_{invis}) \leftarrow \text{BINARYSEARCHFORLAYOUTCHANGE}(r_p, w_p, w_c, \text{disappear})$ 
20:        UPDATECONSTRAINT( $r_p, w_{dis}, \mathcal{AC}$ )
21:      end for
22:      for  $r_c \in U_c$  do
23:         $(w_{invis}, w_{vis}) \leftarrow \text{BINARYSEARCHFORLAYOUTCHANGE}(r_c, w_p, w_c, \text{appear})$ 
24:        ADDNEWCONSTRAINT( $r_c, (w_{vis}, \perp), \mathcal{AC}$ )
25:      end for
26:    end if
27:     $l_p \leftarrow l_c$ 
28:  end while
29:  for  $r \in l_p$  do
30:    UPDATECONSTRAINT( $r, w_p, \mathcal{AC}$ )
31:  end for
32:  return  $\mathcal{AC}$ 
33: end procedure

```

---

uses this insight to remove the need to search for the appearance or disappearance points of relationships. This reduces the effort required to extract the RLG, as the algorithm reuses previously extracted layout information rather than searching for it again. The CHECKNODEVISIBILITY function shows the approach taken.

Using Figure 3.1 as an example once again, the alignment relationship between the body and img elements will intuitively appear at the same viewport width as the img element: 768 pixels. The algorithm analyses every relationship in  $U_p$  and checks the visibility constraints of the two nodes. If either one disappears between  $w_p$  and  $w_c$ , then the function assumes the disappearance point of the node to also be the disappearance point of the relationship. This allows the completion of the relevant alignment constraint without a costly binary search.

---

**Algorithm 5** Utility Functions
 

---

```

1: procedure MATCHEDGES( $L_p, L_c$ )
2:    $U_p \leftarrow \{\}, U_c \leftarrow \{\}$ 
3:   for  $r_p \in l_p$  do
4:     if  $r_p \in l_c$  then
5:        $l_c \leftarrow l_c \setminus \{r_p\}$ 
6:     else
7:        $U_p \leftarrow U_p \cup \{r_p\}$ 
8:     end if
9:   end for
10:   $U_c \leftarrow l_c$ 
11:  return  $U_p, U_c$ 
12: end procedure
13: procedure PAIRUNMATCHEDGES( $U_p, U_c$ )
14:   $P \leftarrow \{\}$ 
15:  for  $(n_{1p}, n_{2p}, t_p, a_p) \in U_p$  do
16:    for  $(n_{1c}, n_{2c}, t_c, a_c) \in U_c$  do
17:      if  $n_{1p} = n_{1c} \wedge n_{2p} = n_{2c}$  then
18:        if  $(t_p \neq t_c) \vee (t_p = t_c \wedge a_p = a_c)$  then
19:           $P \leftarrow P \cup \{((n_{1p}, n_{2p}, t_p, a_p), (n_{1c}, n_{2c}, t_c, a_c))\}$ 
20:        end if
21:      end if
22:    end for
23:  end for
24: end procedure
25: procedure CHECKNODEVISIBILITY( $w_p, w_c$ )
26:  for  $r_p = (n_1, n_2, t, a) \in U_p$  do
27:     $(x_1, x_2, t, a) \leftarrow \mathcal{F}_{AC}(r_p)$  where  $x_2 = -$ 
28:     $(w_{d1}, w_{d1}) \leftarrow \mathcal{F}_{VC}(n_1), (w_{d2}, w_{d2}) \leftarrow \mathcal{F}_{VC}(n_2)$ 
29:    if  $(w_p \leq w_{d1} \leq w_c)$  then
30:       $(x_1, x_2, t, a) \leftarrow (x_1, w_{d1}, t, a)$ 
31:    else if  $(w_p \leq w_{d2} \leq w_c)$  then
32:       $(x_1, x_2, t, a) \leftarrow (x_1, w_{d2}, t, a)$ 
33:    end if
34:  end for
35:  for  $r_c = (n_1, n_2, t, a) \in U_c$  do
36:     $(x_1, x_2, t, a) \leftarrow \mathcal{F}_{AC}(r_c)$  where  $x_2 = -$ 
37:     $(w_{d1}, w_{d1}) \leftarrow \mathcal{F}_{VC}(n_1), (w_{d2}, w_{d2}) \leftarrow \mathcal{F}_{VC}(n_2)$ 
38:    if  $(w_p \leq w_{d1} \leq w_c)$  then
39:       $\mathcal{R} \leftarrow \mathcal{R} \cup \{(n_1, n_2)\}$ 
40:       $\mathcal{A} \leftarrow \mathcal{A} \cup \{(n_1, n_2), (w_{d1}, -, t, a)\}$ 
41:    else if  $(w_p \leq w_{d2} \leq w_c)$  then
42:       $\mathcal{R} \leftarrow \mathcal{R} \cup \{(n_1, n_2)\}$ 
43:       $\mathcal{A} \leftarrow \mathcal{A} \cup \{(n_1, n_2), (w_{d2}, -, t, a)\}$ 
44:    end if
45:  end for
46: end procedure

```

---

The function then performs a similar analysis of  $U_c$ . This time, it checks if the appearance of a relationship is due to an element's appearance rather than its disappearance. If so, it creates the relevant alignment constraint with the same  $x_1$  value as that of the appearing element.

If any relationships still remain unmatched, Algorithm 4 performs binary searches to find the exact viewport widths at which the relationships appear or disappear. It then completes or creates the relevant alignment constraints (lines 20, 24). As with Algorithm 2, it repeats the entire process until it has processed all the layouts. It then finally takes any relationships still visible and updates their alignment constraints with an  $x_2$  value of  $w_{max}$ .

### 3.2 CONCLUDING REMARKS

This chapter introduced the responsive layout graph, a model of a web page's responsive layout which represents the dynamic reorganisation of content across a wide range of viewport sizes. It introduced and described the concept of visibility constraints and alignment constraints, which are used to model the main aspects of responsive design. It then presented an example of a simple RLG to help illustrate the foundational concepts. Finally, it presented a series of algorithms that extract the RLG of a given web page.





# 4

---

## DETECTING POTENTIALLY UNSEEN LAYOUT SIDE EFFECTS OF SMALL CODE CHANGES

---

The previous chapter introduced the responsive layout graph (RLG) as a way to model the dynamic layout of a web page. This chapter takes the RLG and uses it to detect and highlight potentially unseen layout side-effects unintentionally introduced by developers.

The chapter begins by presenting a usage scenario in which a developer makes a modification to a web page. They want to ensure they have not introduced any layout issues, without having to verify the web page's layout manually. Thus, this chapter proposes an approach that obtains the RLGs for both the original and modified web pages. It then compares the two models using a pairwise matching algorithm to identify any issues. The approach reports any model differences to the developer as they may represent unintentional layout issues requiring analysis and attention.

This chapter then presents REDECHECK (Responsive Design Checker, pronounced "Ready Check"). This is an implementation of the proposed approach, built using the Java programming language. It then describes an automated code mutation technique that creates modified versions of web pages. This aims to replicate the sorts of changes a developer might make during real-world development. This technique uses eight operators that randomly mutated different HTML and CSS constructs commonly used to define the layout of a responsive web page.

Using a pool of 15 responsive web pages as subjects, this chapter then evaluates the proposed approach in three experiments. The first investigates whether REDECHECK could detect changes introduced through the code modification technique. It then investigates how REDECHECK's performance compared to both manual and automated techniques. The second experiment investigates

whether the “subtlety” of the layout change caused by a code modification affects the detection capability of ReDeCHECK and the automated baseline. Finally, the third experiment investigates the effectiveness and efficiency of ReDeCHECK under various configurations.

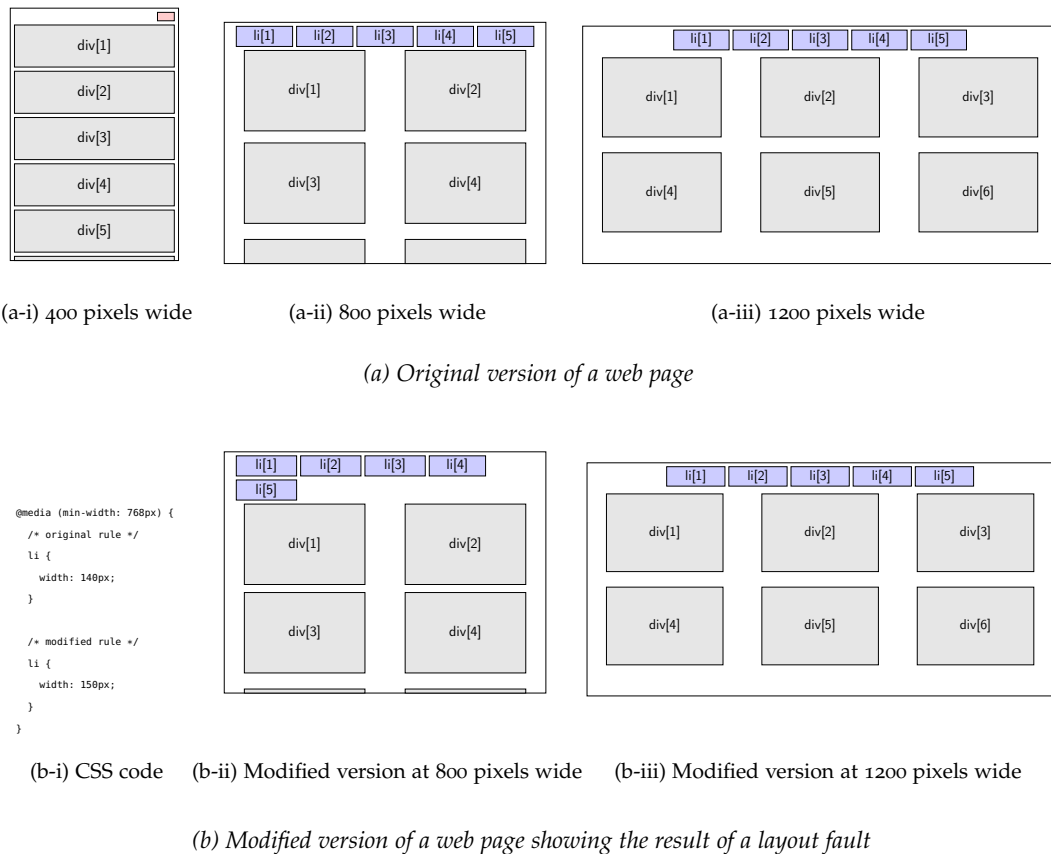
The key contributions of this chapter are:

1. An algorithm for comparing two RLG models. This outputs a list of model differences representing potentially unintended layout issues. This change detection approach has been implemented into a software tool called ReDeCHECK.
2. A code mutation technique for HTML and CSS that seeds changes into code affecting web page layout.
3. A thorough empirical study evaluating the effectiveness and efficiency of the proposed approach for detecting potential layout issues.

#### 4.1 THE PROBLEM

Due to the intricate nature of modern responsive web pages, the HTML and CSS required is often quite complex. Hundreds or even thousands of web page elements often interact with thousands of CSS declarations to create a fully responsive layout. Because of this, elements on a web page can respond in peculiar ways as the viewport conditions change. During development, small changes to the source code intended for one range of viewport widths (e.g. those associated with desktop devices) can introduce unintended side-effects to the responsive layout at other viewport widths. As these side-effects are often serious aesthetic issues, and given that well-designed responsive web pages can have wide-ranging positive impacts upon users, detecting them and mitigating their impact is vital.

Detecting these unseen side-effects in an accurate and timely manner is a very difficult task for developers. They can manifest at very unpredictable ranges of viewport widths, sometimes a single viewport width. Manual checking may therefore fail to inspect the web page at one of the “faulty” viewport widths. Secondly, modern web pages often contain a lot of information and developers may only look for “obvious” layout issues. If they do so, they can easily overlook side-effects with a low visual impact. Finally, the current range of developer tools for developing and checking responsive web pages provide only



**Figure 4.1:** A mock-up of a responsive web page shown at three different resolutions. (a) The original version of the web page, with li elements making up a list of menu items and div elements making up content panels. (b) The result of a change to the CSS code (b-i) that increases the width of the menu items, and is intended to only influence the 1200 pixel viewport width (b-iii). However, the CSS modification unintentionally causes a layout issue at the 800 pixel viewport width (b-ii) such that the menu items are now too wide to fit on one line and the last element (i.e., li[5]) incorrectly wraps to the next line.

limited support. It is therefore clear a reliable, automated approach for identifying unseen layout side-effects would be highly beneficial to web developers.

Figure 4.1 presents a simple example which demonstrates how a small change to the HTML or CSS of a web page can cause unwanted, detrimental side-effects to the layout at unpredictable viewport widths. This chapter henceforth refers to these programming errors as *layout faults* and their visual manifestations in the web page as *layout failures*. Part (a) of the figure shows the layout of the “original” version of the web page at narrow, medium and wide viewport widths. Suppose a developer wishes to add icons to the navigation links. Part (b-i) then presents the code change implemented to increase the width of the five li elements, as they require more horizontal space. Given the main focus of this design change is the desktop layout, the developer confirms the presence of the widened links at a viewport width of 1200 pixels, as shown by part (b-iii) of

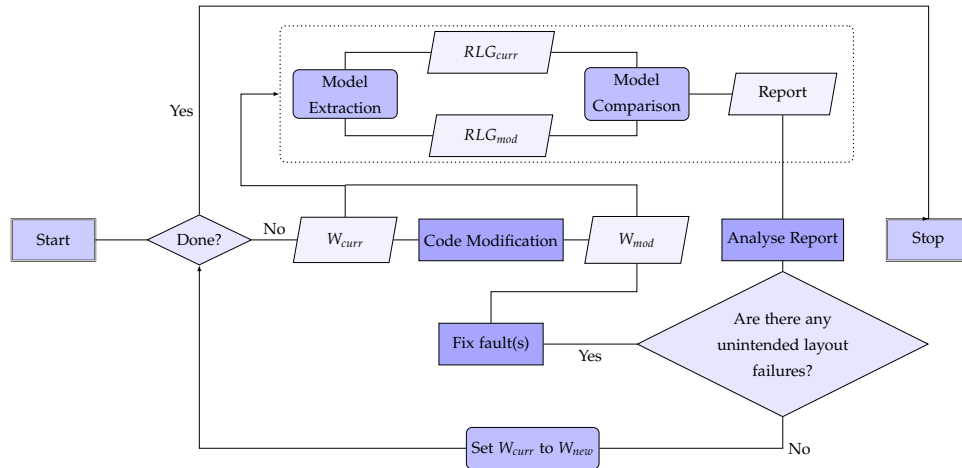
the figure. They then assume their code change had no unintended effects. Depending on the viewport widths at which the developer checks the web page, they may not become aware of the unintended side-effect their modification introduced. As the viewport narrows, there is no longer enough space for the web page to display the five links side-by-side and so `li[5]` wraps onto a second line. The functionality of the web page has not been detrimentally impacted by this failure, as `li[5]` can still be clicked by a user. However, the inconsistent layout could negatively impact a user's perception of the page's quality or their ability to navigate it. Other layout failures could be more serious, such as two elements overlapping and obscuring some of the web page's content or making a navigation element unclickable. These failures would impact both the functionality and the aesthetics of the web page and therefore have potentially costly repercussions.

This motivating example emphasises the problem developers face when incrementally tweaking the responsive design of web pages. Without the use of this chapter's automated approach, following each and every code modification they must manually check thoroughly for any unintended side-effects. Furthermore, as they make more and more changes after introducing a layout failure, developers may struggle to recall the specific modification responsible for its introduction. This makes diagnosing and fixing the problem a much more involved task.

## 4.2 USAGE SCENARIO

Figure 4.2 shows an envisaged usage scenario which uses the RLG introduced in Chapter 3 to automatically highlight potentially unseen side-effects in the layout of a web page following code modifications. To illustrate the various steps and components, this section links them to the motivating example in Figure 4.1.

The scenario begins with the current version of the web page,  $W_{curr}$ , shown in part (a) of Figure 4.1. Then, the developer implements a code modification, such as the one shown by part (b-i) of the figure, resulting in the modified version of the web page,  $W_{new}$ . Parts (b-ii) and (b-iii) of Figure 4.1 show this new version. Ordinarily, the developer would manually check  $W_{new}$  to make sure no unintended layout behaviour occurs. In this scenario, both versions of



**Figure 4.2:** The main usage scenario of this chapter's approach.

The approach aims to automatically alert developers to the unseen side effects of changes to a responsive web page. In this diagram, the proposed approach is contained within the dotted rectangle, inputs and outputs are shown as parallelograms, the decision is depicted as a diamond and automatic and manual processes are shown as rectangles with and without rounded corners, respectively.

the web page are instead input into the proposed approach, represented by the dotted rectangle in Figure 4.2.

The *model extraction* step of the approach takes  $W_{curr}$  and  $W_{new}$  and extracts the respective RLG models,  $RLG_{curr}$  and  $RLG_{new}$ . Next, the *model comparator* compares the two models and creates a report detailing the differences between them. The developer then analyses this report to determine whether any of the reported differences are unintentional side-effects rather than intended layout changes. They no longer have to resize their web browser and inspect the web page at each viewport width. This substantially reduces the manual effort required, but unfortunately does not completely eradicate it. If the developer decides an issue requires attention, they implement a fix. Then, they rerun the approach to determine whether the layout issue still persists. Once there are no side-effects requiring attention, the developer updates  $W_{curr}$  to  $W_{new}$ . This is because any future code modifications require comparison against the latest version of the web page. This process repeats until the developer has implemented all of their code changes and there are no unintended layout side-effects.

### 4.3 COMPARING TWO RLG'S

To alert developers to unseen layout side-effects, the approach requires a comparison method. This section therefore presents an algorithm that takes as input two RLGs and outputs a list of differences between them. A change in the lay-

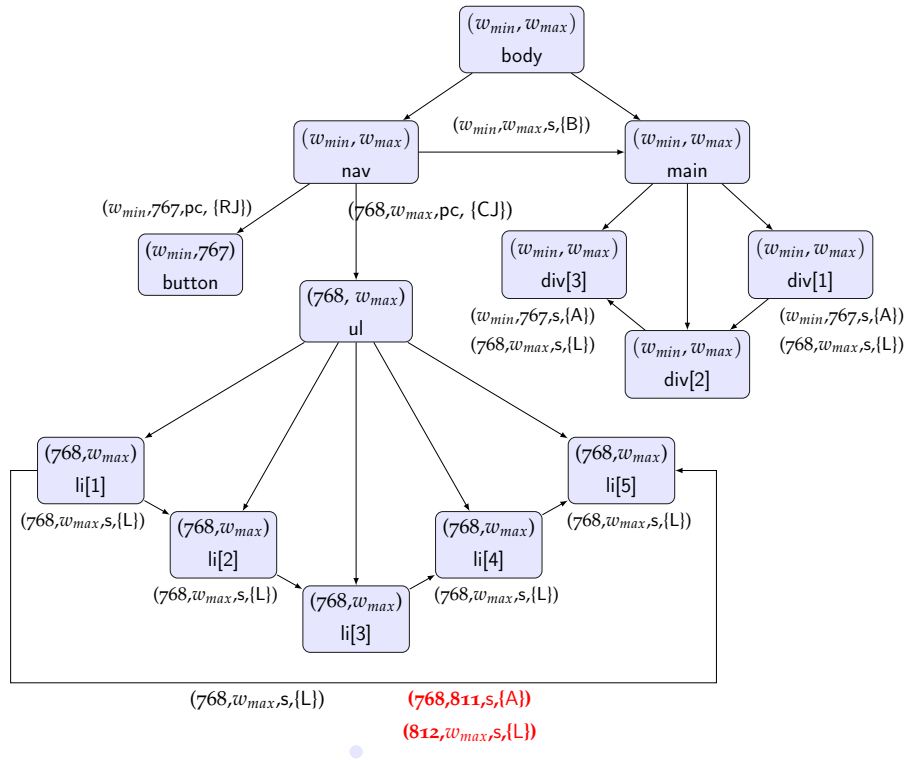
out of a web page will generally cause a change in the RLG extracted from it, so reported differences could be unseen side-effects of a code change. Algorithm 6 presents this comparison approach.

Figure 4.3 presents a snippet of the RLG extracted for the motivating example in Figure 4.1. The models extracted,  $RLG_{curr}$  and  $RLG_{new}$  for parts (a) and (b) of the figure are almost identical. The only difference is the alignment constraints on the edge between  $li[1]$  and  $li[5]$ . For this edge,  $RLG_{curr}$  contains only one constraint (shown in black), while  $RLG_{new}$  has two (shown in red). The first corresponds to  $li[5]$  first being below  $li[1]$  as it wraps onto a new row. Then, the second constraint models  $li[5]$  reverting back to its intended location to the right of  $li[1]$ .

Algorithm 6 begins by attempting to match each node in  $\mathcal{RLG}_1$  to a node in  $\mathcal{RLG}_2$  using the unique XPath of each element (line 2) and reports any unmatched nodes to the developer (lines 3-8). Then, it compares the visibility constraints mapped to each pair of matched nodes,  $e_1$  from  $\mathcal{RLG}_1$  and  $e_2$  from  $\mathcal{RLG}_2$  (lines 9-13). Any differences are reported to the developer as *visibility differences*. For instance, if  $e_1$  has the visibility constraint (768,1400) while  $e_2$  has the constraint (766,1400), the approach reports this potential issue to the developer.

The comparison technique then moves on to matching the alignment constraints on the edges of  $\mathcal{RLG}_1$  and  $\mathcal{RLG}_2$  (lines 14-34). Given an alignment constraint  $a_1$  from  $\mathcal{RLG}_1$ , a full match exists **if and only if** a constraint  $a_2$  from  $\mathcal{RLG}_2$  exists on an edge connecting the same two nodes, between the same lower and upper viewport bounds, with the same relationship type and set of alignment attributes. This is determined by the `CONTAINS PERFECTMATCH` function. The approach then reports any discrepancies as *alignment differences*, which can be one of three distinct types. If a constraint has the same relationship type and alignment attributes, but differing lower or upper bounds (i.e.,  $x_1$  or  $x_2$ ), then it is a *bounds difference*. If the bounds and relationship type match, but the attribute sets are different, then it is an *attributes difference*. Finally, if the matching algorithm fails to find a match for either the bounds or the attributes, then the constraint is completely unmatched. In this case, the algorithm reports the constraint in question as a *compound difference*.

To increase the usability of the model comparison report, the algorithm also inspects the model differences to determine the viewport widths at which each layout change is observable. For an attributes difference or compound differ-



**Figure 4.3:** The RLG for the web page shown in Figure 4.1

The RLG covers a range of viewport widths between  $w_{min}$  and  $w_{max}$ . To ensure that this RLG is easy to understand, it omits some elements and constraints; the constraints in red represent the unseen side effects between the two RLGs by the code change, in which the final `li` element is forced to wrap onto a new row. Also, for space reasons, attribute labels are shortened.

ence, it is simply the range of viewport widths covered by the constraint or constraints. For a bounds difference, it is the symmetric difference of the two sets of bounds. In other words, the set of viewport widths for which only one of the two constraints holds.

Revisiting the motivating example in Figure 4.1, this section now presents a worked example of the comparison algorithm. The RLG example in Figure 4.3 presents detected model differences in red. Every node and their visibility constraints are correctly matched by the algorithm. The algorithm also matches all alignment constraints except the ones mapped to the edge between `li[1]` and `li[5]`. In  $\mathcal{RLG}_1$ , this edge contains only one alignment constraint,  $(768, w_{max}, s, \{L\})$ , while the corresponding edge in  $\mathcal{RLG}_2$  has two constraints,  $(768, 811, s, \{A\})$  and  $(812, w_{max}, s, \{L\})$ . The algorithm reports a bounds difference for the constraints  $(768, w_{max}, s, \{L\})$  from  $\mathcal{RLG}_1$  and  $(812, w_{max}, s, \{L\})$  from  $\mathcal{RLG}_2$  as they have identical attribute sets. It also reports a compound difference for the constraint  $(768, 811, s, \{A\})$  from  $\mathcal{RLG}_2$ . Given the number of constraints differs between the two models, it will always fail to match at least one constraint.

---

**Algorithm 6** RLG Comparison
 

---

```

1: procedure COMPARERLGs( $\mathcal{RLG}_1, \mathcal{RLG}_2$ )
2:    $M \leftarrow \text{MATCHNODES}(\mathcal{E}_1, \mathcal{E}_2)$ 
3:   for all  $u_1 \in \mathcal{E}_1$  do
4:      $\text{REPORTDIFFERENCE}(\text{unmatched-node}, u_1)$ 
5:   end for
6:   for all  $u_2 \in \mathcal{E}_2$  do
7:      $\text{REPORTDIFFERENCE}(\text{unmatched-node}, u_2)$ 
8:   end for
9:   for all  $(e_1, e_2) \in M$  do
10:    if  $\mathcal{F}_{VC_1}(e_1) \neq \mathcal{F}_{VC_2}(e_2)$  then
11:       $\text{REPORTDIFFERENCE}(\text{visibility}, e_1, e_2)$ 
12:    end if
13:  end for
14:  for all  $r \in \mathcal{R}_1$  do
15:    for all  $ac \in \mathcal{F}_{AC_1}(r)$  do
16:       $\text{perfectMatch} \leftarrow \text{CONTAINS PERFECTMATCH}(\mathcal{F}_{VC_2}(r), ac)$ 
17:       $\text{attributesMatch} \leftarrow \text{CONTAINS ATTRIBUTESMATCH}(\mathcal{F}_{VC_2}(r), ac)$ 
18:       $\text{boundsMatch} \leftarrow \text{CONTAINS BOUNDSMATCH}(\mathcal{F}_{VC_2}(r), ac)$ 
19:      if  $\text{perfectMatch} \neq \text{null}$  then
20:         $\mathcal{F}_{AC_1}(r) \leftarrow \mathcal{F}_{AC_1}(r) - \{ac\}$ 
21:         $\mathcal{F}_{AC_2}(r) \leftarrow \mathcal{F}_{AC_2}(r) - \{\text{perfectMatch}\}$ 
22:      else if  $\text{attributesMatch} \neq \text{null}$  then
23:         $\text{REPORTDIFFERENCE}(\text{bounds}, ac, \text{attributesMatch})$ 
24:      else if  $\text{boundsMatch} \neq \text{null}$  then
25:         $\text{REPORTDIFFERENCE}(\text{attributes}, ac, \text{boundsMatch})$ 
26:      end if
27:    end for
28:    for all  $u_1 \in \mathcal{F}_{AC_1}(r)$  do
29:       $\text{REPORTDIFFERENCE}(\text{compound}, u_1)$ 
30:    end for
31:    for all  $u_2 \in \mathcal{F}_{AC_2}(r)$  do
32:       $\text{REPORTDIFFERENCE}(\text{compound}, u_2)$ 
33:    end for
34:  end for
35: end procedure

```

---

The approach collates the final set of differences and their respective width sets into a report, which it sends to the developer for manual analysis. Figure 4.4 presents a snippet of the report produced for the motivating example. As discussed in the previous paragraph, it shows how the various types of model differences are reported to the user. The responsive context given by the report aims to help guide and direct the manual analysis performed by the user. This should be beneficial when compared to simply inspecting a web page “blindly” following a code change.



```

Differing bounds for BODY/NAV/UL/LI[1] sibling of BODY/NAV/UL/LI[5] {L}
  Oracle : 768 -> w_{max}
  Test : 812 -> w_{max}

Unmatched in test: BODY/NAV/UL/LI[1] sibling of BODY/NAV/UL/LI[5] {A}
between 768 -> 811

```

**Figure 4.4:** A snippet of a report produced for the example in Figure 4.1.

#### 4.4 EMPIRICAL EVALUATION

This section evaluates the approach for detecting potentially unseen side-effects of code modifications. Using 15 responsive web pages as subjects, it answers the following research questions:

**RESEARCH QUESTION ONE:** *How accurate is the presented approach at detecting the various types of changes made to the source code of a responsive web page? How does it compare to alternative methods?*

**RESEARCH QUESTION TWO:** *How does the “subtlety” of a layout change influence the effectiveness of the approach?*

**RESEARCH QUESTION THREE:** *How do different configuration parameters affect the efficiency and effectiveness of the approach?*

The next section describes the design of the experiments used to answer the research questions. Then, the following section presents and discusses the obtained results.

##### 4.4.1 *Experimental Design*

#### **Subject Web Pages**

To answer the research questions, 15 responsively designed web pages were selected as test subjects. While more web pages could have been selected, 15 subjects is above average when compared to previous empirical evaluations in this research domain. Also, this evaluation primarily investigates whether changes to a web page can be detected by this chapter’s approach. Therefore, it is arguably more important to evaluate on a large and varied number of code changes, as described in the following sections. To obtain a wide variety of

web pages, they were selected from several different sources, such as the showcases of Twitter Bootstrap [19] and Zurb Foundation [161], two popular RWD frameworks. Examples were also taken from well-known companies and collections of examples of good responsive web design (eg., [111]). The subjects were also selected to represent a variety of different web page sizes, domains and potential userbases. For instance, some subjects selected were small and likely to receive minimal web traffic, whereas others were far more complex and may potentially receive thousands of visits per day. By evaluating the approach on subjects that vary in terms of all of these characteristics, the results obtained should be as generalisable as possible.

The 15 subjects selected were: “Aftnoon”, a web site for a design studio; “Annette’s Creations”, an online shop; “Ashton Snook”, the homepage of a visual designer; “Bootstrap”, the homepage for the popular web design framework; “Coursera”, the well-known provider of massive open online courses; “Denon”, a manufacturer of high-end headphones and DJ equipment; “ISSTA 2016”, the web site for a software testing conference; “Name Mesh”, a site that suggests suitable web domains; “Pay Demand”, a web site for businesses to compare rates for credit card processing; “Rebecca Made”, a web developer’s showcase; “Reserve”, the web site of a mobile application that performs restaurant reservations; “Responsive Process”, an educational web site about responsive web design; “Shield”, the site of the responsive template presented in Chapter 2 and finally, “Treehouse”, a platform for technology training. The web sites come from a wide variety of application domains. They also demonstrate various development styles. Some web pages make use of RWD frameworks while others use bespoke HTML and CSS code. This diversity helps to ensure the representativeness of this chapter’s empirical results.

Table 4.1 details the web pages used in the study. Many web developers choose to apply “minification” to the source code, which removes all unnecessary white space to decrease the loading times of the web page. Therefore, all source code must be formatted in an identical fashion to make a fair comparison of the complexity of the subjects. To do this, the HTML and CSS formatting tools available at <http://www.dirtymarkup.com> were used to remove any inconsistencies in the coding style across subjects. Additionally, the table presents the number of CSS declarations and blocks (groups of declarations applied to a selected group of elements) contained within the source code. Finally, the numbers in parentheses present the number of blocks and declarations actually used by the web page, i.e., applied to at least one element on the page. It is clear that

**Table 4.1:** Responsive web pages used in the empirical study.

In the column headings, “LOC” refers to lines of code. Additionally, a CSS block (i.e., “Blocks”) is a group of individual declarations (i.e., “Declarations”) applied to a group of HTML elements through a CSS selector.

WEB SITE NAME	URL	HTML		CSS		
		LOC	DOM NODES	LOC	BLOCKS	DECLARATIONS
Aftnoon	http://aftnoon.com	204	112	1370	459 (37)	1003 (98)
Annette’s Creations	http://annettescreations.com	235	113	7199	1398 (60)	2383 (179)
Ashton Snook	http://www.ashtonsnook.com	407	126	8417	1730 (104)	3218 (293)
BitTorrent	http://bittorrent.com	830	356	6198	1140 (158)	1907 (406)
Coursera	http://coursera.com	646	472	10829	1958 (83)	4515 (176)
Denon	http://denondj.com	281	232	7975	1457 (62)	3244 (189)
Bootstrap	http://getbootstrap.com	292	147	8550	1757 (61)	3199 (152)
ISSTA	http://issta.cispa.saarland	230	196	8185	1912 (84)	3209 (237)
Name Mesh	http://namemesh.com	598	217	2675	2356 (66)	3725 (171)
Pay Demand	http://paydemand.com	181	106	10961	2471 (56)	4942 (92)
Rebecca Made	http://rebeccamade.com	274	150	3645	1094 (34)	1755 (59)
Reserve	http://reserve.com	229	125	6452	1375 (31)	2537 (71)
Responsive Process	http://responsiveprocess.com	266	142	956	166 (34)	379 (115)
Shield	http://www.blacktie.co/demo/shield	606	336	7637	1747 (98)	2999 (287)
Treehouse	http://teamtreehouse.com	1053	406	34951	5958 (111)	12122 (358)

the 15 web pages selected represent a wide range of complexity. In terms of DOM nodes, the smallest web page contained 106 nodes and the biggest had 472. When considering CSS style sheet size, the number of blocks ranged from 166 to as many as 5,958.

### Implementation

To perform this study, the automated approach was implemented as a tool called REDECHECK (Responsive Design Checker, pronounced “Ready Check”). It consists of three core modules, as shown by Figure 4.5. The *model extraction* module renders each version of the web page in a web browser, then extracts the RLG of each. The *model comparison* module then performs the pairwise matching procedure detailed in this chapter. Finally, the *report generator* analyses the model differences and produces the report for the user. For this study, REDECHECK used Selenium (v2.53.1) to drive and interact with an instance of PhantomJS v2.1.1, a headless web browser ideal for use in automated web based tasks. Selenium allows REDECHECK to easily resize the browser window and extract the DOM and thus determine the relative layout of the web page, as rendered in PhantomJS. This study used a standard iMac workstation to run the experiments. It had 8GB of RAM and was running MacOS Sierra. This shows

ReDeCHECK can run on an “everyday” machine with no high-specification requirements.

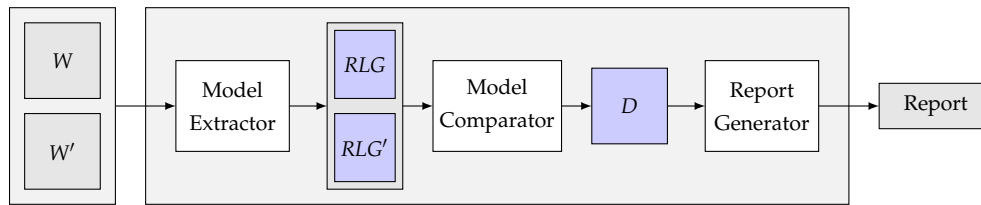


Figure 4.5: High level structure of ReDeCHECK.

### Incrementally Modifying a Web Page

The usage scenario for this chapter’s approach requires two consecutive versions of a web page as input. One contains an incremental code modification made by the developer. In other words, it focusses on the development stage of the software lifecycle. Unfortunately, evaluating the approach on real instances of incremental changes is almost impossible, as obtaining examples of such changes is highly problematic. For this reason this study used mutation analysis to introduce small code changes into the subject web pages. As discussed in Chapter 2, previous approaches to mutation analysis of web pages by Praphamontripong and Offutt [119] only proposed operators targetting functional aspects of HTML and JSP code, while Mahajan et al. [89, 90] did not specifically target layout attributes. This study needed to mutate the aesthetic properties of HTML and CSS used to control the appearance and layout of a web page. Therefore, this section describes an automated technique for introducing small modifications into a web page. This approach mimics what a developer might do in practice, and consists of eight mutation operators.

Before running the automated approach, all the HTML and CSS files required to render each of the fifteen web pages “offline” were downloaded. This then allowed the automated mutation technique to generate modified versions of each web page. To ensure the empirical results were generalisable, it was important the mutation analysis approach created a wide variety of code modifications. Therefore, the eight operators targetted various constructs of both the CSS and HTML of web pages. Table 4.2 describes each operator and presents “before and after” code snippets demonstrating examples of the modifications they generate. The remainder of this section details how each operator works and discusses the challenges encountered during their implementation.

With the ease of development offered by RWD frameworks such as Bootstrap and Foundation, nowadays many web developers choose to make use of the

**Table 4.2:** Descriptions and examples of the incremental code mutation operators used. In this table, the HTML or CSS source code in the “Before” column corresponds to an example of the original version of the web page as it was downloaded, while the “After” column’s code is that which results from applying the specified operator. Note that the first four operators modify the HTML of a page while the final four change its CSS.

OPERATOR NAME	DESCRIPTION	BEFORE	AFTER
<b>Class Addition</b>	Adds a class to an element	<code>&lt;div class="col-xs-12"&gt;</code>	<code>&lt;div class="col-xs-12 col-sm-6"&gt;</code>
<b>Class Deletion</b>	Removes a class from an element	<code>&lt;div class="col-xs-12 col-sm-6"&gt;</code>	<code>&lt;div class="col-sm-6"&gt;</code>
<b>Class Exchange</b>	Replaces a class with another	<code>&lt;div class="col-xs-12 col-sm-6"&gt;</code>	<code>&lt;div class="col-xs-12 col-sm-4"&gt;</code>
<b>Textual Content</b>	Increases/decreases amount of text in an element	<code>&lt;h1&gt;Welcome&lt;/h1&gt;</code>	<code>&lt;h1&gt;Welcome to my page&lt;/h1&gt;</code>
<b>Declaration Value</b>	Modifies value of a declaration	<code>li { width: 100px }</code>	<code>li { width: 105px }</code>
<b>Declaration Unit</b>	Modifies the unit of a declaration’s value	<code>div { width: 50px }</code>	<code>div { width: 50% }</code>
<b>Query Expression</b>	Modifies the media query’s expression	<code>@media (min-width: 640px)</code>	<code>@media (max-width: 640px)</code>
<b>Query Breakpoint</b>	Modifies the media query’s numeric value	<code>@media (min-width: 992px)</code>	<code>@media (min-width: 990px)</code>

pre-defined styles. They do this through the application of CSS classes to HTML elements. These classes cover a full range of styling aspects, such as grid-based layouts to icons and typography. However, making sure the correct styles are applied to the correct elements, at the correct viewport widths, is a difficult task. If not done properly, it can lead to severe layout issues. Because of this, three operators targetted the assignment of these classes to HTML elements. Firstly, *class addition* may, for example, add the class "col-sm-6" to an element, while *class deletion* might remove "col-xs-12". The third operator, *class exchange*, essentially combines the two. For instance, it might replace the classes "col-xs-12 col-sm-6" with "col-xs-12 col-sm-4". The final operator targetting HTML code, *textual content*, modifies the text contained within an element. While initially this operator may not seem to be layout related, in some cases an increase or decrease in the amount of text inside an element can lead to an element becoming bigger or smaller. This can in turn make it susceptible to layout failures.

While RWD frameworks provide a good starting point, many web developers choose to customise the pre-defined CSS, or even write their own styles from scratch. To reflect both of these approaches, the remaining four mutation operators target the CSS of a web page. The first two address the modification of individual CSS declarations, as shown by the code snippets in Figure 4.6. The first operator changes a declaration’s value, which can be either numerical or textual, as shown by part (b) of Figure 4.6. The second changes its unit, as shown by part (c) of Figure 4.6, but can only do so on certain numerical properties. The operators modify numerical values at random by a value of  $\pm 1 - 10$ . Therefore, the *declaration value* operator might change the width of an element from 50px to 53px. Meanwhile, the *declaration unit* operator might change a static width declaration, 50px, to a fluid one, 50%.

<pre>@li {   width: 50px; }</pre>	<pre>@li {   width: 53px; }</pre>	<pre>@li {   width: 50%; }</pre>
a) Original version	b) Value mutation	c) Unit mutation

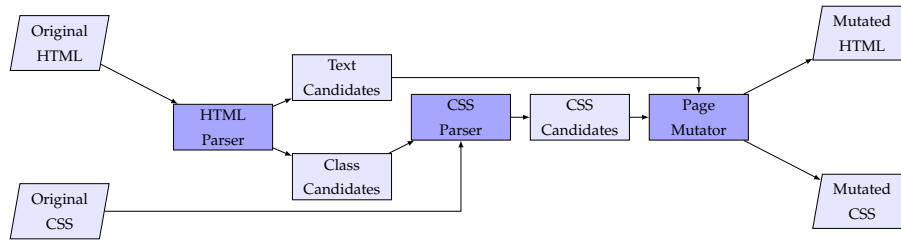
**Figure 4.6:** Examples of the two mutation operators for CSS declarations.

Rather than targetting single declarations, the remaining two operators change the way whole groups of style declarations are applied to elements. They do this by modifying the media queries used to control when to apply the styles. Each operator focusses on a different part of the media query. As with the declaration operators, the code snippets in Figure 4.7 show the operators in action. The first operator, *query expression*, changes the expression used. For example, changing `min-width` to `max-width` applies the rules at a completely different set of viewport widths (part b of the figure). The second operator, *query breakpoint*, then targets the breakpoint of the query. Increasing it from `768px` to `770px` again changes the layout behaviour of the web page (part c of the figure).

<pre>@media(min-width: 768px) {   div { width: 400px; } }</pre>	<pre>@media(max-width: 768px) {   div { width: 400px; } }</pre>	<pre>@media(min-width: 770px) {   div { width: 400px; } }</pre>
a) Original version	b) Query mutation	c) Breakpoint mutation

**Figure 4.7:** Examples of the two mutation operators for media queries.

Figure 4.8 shows the overall approach for injecting the code changes. The first step is a static analysis of the HTML, which identifies the classes applied to each web page element. It stores these as class modification candidates. It also identifies the elements containing text and stores them as the set of text candidates. Next, the approach parses and filters the CSS files to extract the set of CSS candidates. By checking whether the selectors used in each CSS block are present in the class candidate set, the approach ensures the web page applies the selected CSS block to at least one element on the page. This mitigates the risk of equivalent mutants, in which the code modification is inserted into a part of the CSS not actually used by the web page. These modifications obviously do not introduce an observable layout change. Additionally, the approach also prunes the candidate blocks so they only contain the declarations focussing on layout, such as `width`, `margin` and `padding`. This is because modifying non-layout properties such as `background-color` would not impact the web page's layout, only its visual appearance. Then, the approach selects a random operator and a random candidate from the relevant set. It then selects the modification itself



**Figure 4.8:** The high-level structure of the automatic code modification approach.

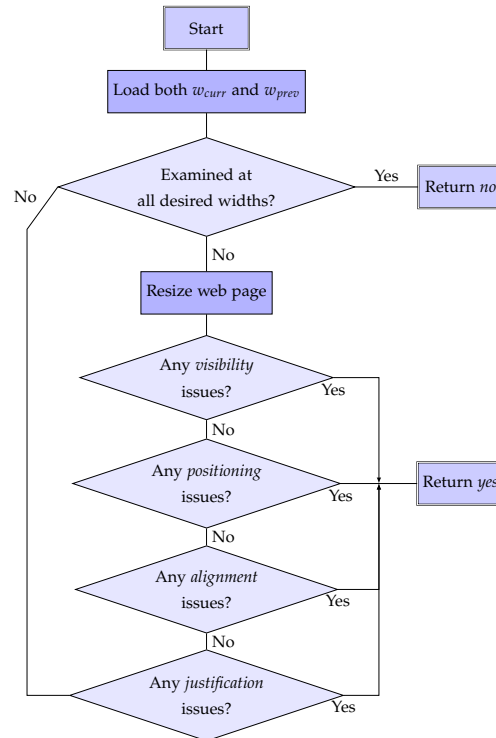
at random, to obtain as wide a variety of layout changes as possible. Finally, it writes the modified HTML and CSS files to disk, ready for use in the experiments.

### Comparison Methods

Due to the lack of previous research on testing responsive web pages, obtaining a baseline against which to compare REDECHECK is problematic. As described in Chapter 2, the main approach used by developers in practice is “spot-checking”. Therefore, this study compared the proposed technique to two baseline approaches based upon this technique. The first was performed manually by three human testers, while the second re-purposed X-PERT, a cross-browser inconsistency tool first introduced in Chapter 2. Both approaches check the layout of the web pages under test at six different viewport widths. These were the pre-set widths advocated by two widely used developer tools, Viewport Resizer [139] and Window Resizer [155]. The six chosen widths were 480, 600, 640, 768, 1024 and 1280 pixels. They not only encompass a large variety of responsive layout behaviour, but also represent a range of common devices in popular usage.

**MANUAL TECHNIQUE:** Henceforth referred to as SPOTCHECK-MANUAL, this technique involves humans comparing the two versions of the web page. At each selected viewport width, the human checks to see if any layout changes are evident. For this empirical study, three human participants performed this technique, namely myself, my supervisor and our collaborator. To mitigate subjectivity in the results, all participants followed the same classification procedure. We also discussed our individual decisions to reach a consensus on the final classification.

**AUTOMATED TECHNIQUE:** As previously discussed, X-PERT—and the alignment graph (AG) underpinning its layout testing approach—is unsuitable for testing responsive layout. It only operates at a single viewport resolution and does not consider responsive design. However, with a small engineering modification,



**Figure 4.9:** The manual procedure for determining if a modified web page contains a layout change.

The individual following this process will ultimately answer *yes* or *no* to the question “does the modified page contain an observable layout change?” In this diagram, a decision is depicted as a diamond and a rectangle denotes a manual step. This iterative process stops when the person performing these steps has either examined the web page at all of the chosen viewport widths and has not found any changes (thus answering *no*) or, alternatively, identifies the first change in the page’s layout (thereby responding with *yes*).

it can compare the layout of two different versions of a web page at the same resolution in a single browser and report the differences. This is in contrast to its original approach, where it compares the same web page in different browsers. This chapter henceforth refers to the implementation of this automated spot-checking technique as SPOTCHECK-AG.

### Manual Classification

This section presents a manual classification procedure which determines whether a given version of a web page contains observable layout changes or not, in relation to the original version. Explicitly defining this process should mitigate any risks of inconsistency between different humans. The procedure was also used to verify the correctness of the results of both REDECHECK and SPOTCHECK-AG. For instance, these automated approaches might fail to report actual layout changes, or incorrectly report layout changes when there are none.



Figure 4.9 presents a flowchart illustrating the overall process. A human tester inspects a pair of web pages at a series of viewport widths and checks for layout changes in the following categories:

- 1. Visibility:** Does the visibility of any element differ between the two versions of the page? For instance, at a particular viewport width, is an element visible in one version but hidden in the other?
- 2. Position:** Does the relative layout of two neighbouring elements differ between the two versions? For example, are two elements rendered side-by-side in one version but stacked one above the other in the other version?
- 3. Alignment:** Is the alignment of a pair of elements different in the two versions? For example, given two elements that aligned on their left edges, do they no longer align in this way?
- 4. Justification:** Is an element no longer justified in the same way within its container? For instance, was an element initially right-justified but is now left-justified?

If a human observed a layout change falling into one of these four categories during the web page inspection, the page contains a layout change. If no layout change was observed at the first viewport width, then they repeated the process at the next viewport width, and so on. If, after inspecting all viewport widths, the human had not found any observable difference between the two versions of the web page, they concluded that it contained no layout changes.

## Methodology

To begin the empirical evaluation, the automated code mutation approach described earlier created 30 modified versions of each of the 15 subject web pages. These 450 modified web pages were then used to answer the three research questions.

**RESEARCH QUESTION ONE:** The main focus of this research question was to compare the performances of `ReDeCheck`, `SpotCheck-Manual` and `SpotCheck-Ag`. As the manual analysis required — both for running `SpotCheck-Manual` and determining the gold standard for each web page— is quite intensive, the experiment for this question used a smaller subset of modified pages. I randomly selected four examples from each web page, with special care made to ensuring each of the eight operators were similarly represented, resulting in a subset of 60 web pages.

In this study, REDECHECK used a step size of 60 pixels and sampled from 400 to 1400 pixels. This ensured it considered a wide range of devices and their associated viewport widths during the experiments. Additionally, as the six viewport widths selected for the two baseline approaches fall within the viewport range tested by REDECHECK, the comparison between them is a fair one.

Different browsers and operating system produce subtle differences in the rendering of web pages. Therefore, the manual classification procedure used screenshots automatically captured by PhantomJS. This ensured a degree of consistency and allowed for easy replication in any future studies. Also, given the subjective nature of manual classifications, the three human testers first performed the procedure individually. They then discussed any web pages for which there were disagreements to reach a consensus on whether each web page contained a layout change.

Layout changes can manifest at unpredictable viewport widths, such as those lying between the ones selected for spot-checking in this study. The results produced by SPOTCHECK-MANUAL therefore cannot reliably constitute a “gold standard”. Further analysis is required before the results of the three main techniques can be classified as *true positive*, *false positive*, *true negative* or *false negative*. This involved taking each modified web page for which SPOTCHECK-MANUAL had returned a negative classification i.e., “does not contain a layout change”. We (my supervisor, colleague and I) then potentially inspected each page at every viewport width within the 400 – 1400 pixel sample range. This produced an accurate gold standard classification. As with SPOTCHECK-MANUAL, we discussed the results as a committee to ensure agreement. For each technique, if it reported an observable layout change and the gold standard declared the web page contained one, then the result is a *true positive*. If no change was observed then it is a *false positive*. If a tool reported no change and no change was present, then the result is a *true negative*, else it is a *false negative*.

RESEARCH QUESTION TWO: Naturally, some layout changes will be harder to detect than others. This can be due to either their visual impact — how significant the layout change is on the page — or the number of viewport widths at which the layout change is evident. The visual impact of a modification may have a significant effect on a human manually checking a web page. It should not detrimentally impact an automated approach like REDECHECK or SPOTCHECK-AG. The number of viewport widths, however, may impact the ability of automated tools to detect changes. This question investigated this by comparing the change detection capability of REDECHECK and SPOTCHECK-AG. It used the

number of viewport widths at which a layout change was observable as a metric of “subtlety”, where the lower the number, the more subtle the change.

Before performing the experiments, an automated approach determined the subtlety of each modified web page. This approach exhaustively compared the DOM of the two versions of the web page at every viewport width in the sample range from RQ1. For each width, given a pair of DOMs  $(d_o, d_m)$ , the approach attempted to match each element in  $d_o$  to an element in  $d_m$  using the element’s XPath identifier. It then compared the coordinates of each pair of matched elements. If there were any unmatched elements, or matched elements exhibiting differing coordinates, then the approach deemed the code modification to have introduced a layout change to the web page at that viewport width. To obtain more generalisable results, this research question used the full set of 450 modified web pages.

After determining the subtlety of a modification on a web page, a value henceforth referred to as #DW, the experiment excluded all web pages for which #DW = 0. These modifications had no impact upon the page’s layout and therefore were “equivalent” mutants. The remaining web pages were then grouped into “buckets” depending on their value of #DW, such as 1, 2-3, 4-5, 6-10, 11-50, 51-100, 101-300 or more than 500. Intuitively, the fewer viewport widths at which a layout change is observable, the more subtle it is. The experiment itself ran both REDECHECK and SPOTCHECK-AG on the web pages, observing and recording whether each detected any layout changes. However, unlike RQ1, the correctness of the produced results was not verified as the manual effort required to do so for all web pages like in RQ1 would have been too high. Instead, given #DW > 0 for a given web page, any layout difference reported by either REDECHECK or SPOTCHECK-AG constituted a correct detection of the injected layout change.

**RESEARCH QUESTION THREE:** In the incremental development scenario presented in Figure 4.2, it is vital REDECHECK runs efficiently. This permits a developer to quickly receive the feedback following their code modifications. If REDECHECK is too slow, developers will end up waiting too long and it will not provide its intended high level of support. Therefore, it is prudent to identify the optimal parameter configuration under which to run REDECHECK. Using the full pool of 450 modified web pages, this question evaluated both the performance and efficiency of REDECHECK using different combinations of sampling techniques and step sizes.

The execution time of a tool or technique is often proportional of the worst-case time complexity of the main algorithm underpinning the technique [81]. As such it is often used as the primary efficiency metric in software engineering research. Therefore, this question began with an investigation in the context of REDECHECK to ascertain how much execution time various components and processes were responsible for. Analysis of the breakdown of REDECHECK's performance discovered the main bottleneck in the tool pipeline was the browser used to render and interact with the web pages under test. In fact, the actual processing and analysis of the extracted DOMs to build the RLGs took an almost negligible amount of time. Interestingly, this was regardless of the complexity of the web page under test. Resizing the browser and querying the DOM at the various viewport widths accounted for the vast majority of the execution time. A correlation between the complexity of the web page under test and the execution time was expected, with a web page containing more DOM nodes requiring a longer execution time. Instead, the execution correlated more with the number of viewport widths sampled by REDECHECK to extract the RLG. Therefore, this research question used it as the primary efficiency metric. As with the previous two questions, the effectiveness metric was the proportion of injected layout changes correctly detected by REDECHECK under each configuration.

The set of initial sample widths,  $\mathcal{S}$ , obtained on line 3 of Algorithm 1, is dictated by two main parameters. The first of these is the sampling technique. The sampling technique presented in Chapter 3 combines two separate techniques: a systematic uniform sample and a boundary sample using programmed breakpoints. This chapter hereto refers to this approach as REDECHECK-COMBINED. However, the initial published version of this work [149], only used the uniform sample as the initial set of sample widths. This question refers to this approach as REDECHECK-INTERVAL, and it serves as the baseline for this experiment. Additionally, a brute-force sampling approach, REDECHECK-EXHAUSTIVE, that samples the web page at every single viewport width within the same range, was used as a second baseline. This technique is essentially REDECHECK-INTERVAL with a step size of 1. Given that REDECHECK-EXHAUSTIVE will sample all of the responsive layout exhibited by a web page within the specified viewport range, it will provide an upper bound for the number of layout changes that the other two competing approaches can detect.

The other controlling parameter is the step size, which controls how large the increments are in the sampling process. For the experiments in RQ1 and RQ2, a value of 60 pixels was used as during the development of REDECHECK it

was observed to balance effectiveness and efficiency well across a variety of web pages. However, should a developer want to sample the web page at a finer granularity, they could use a step size of 10, 20 or 40 pixels, for example. Furthermore, if they feel the number of layout changes in the web page under test is small, they could sample less frequently, using a step of 80, 100 or even 150 pixels. This experiment ran REDECHECK on the pool of mutants with a variety of both small and large step sizes: 10, 20, 40, 60, 80, 100, 150, 200 and 500 pixels. For each modified page under each configuration, the experiment recorded the following three pieces of information:

1. Whether REDECHECK correctly detected the injected layout change.
2. The number of viewport widths REDECHECK sampled to extract the RLG of the modified version.
3. The execution time of REDECHECK.

By evaluating REDECHECK in terms of these three metrics using the full pool of 450 mutants, the results should provide a useful recommendation of the best configuration parameters across a wide range of web pages.

### Threats to Validity

As with all empirical evaluations, any threats to the validity of the results obtained must be considered and mitigated as much as possible. This section details the main threats identified and the steps taken to mitigate them.

**GENERALISABILITY OF WEB PAGES:** As with most software engineering studies, the choice of test subjects is always a threat, as any results obtained may not generalise to other subjects. To mitigate this, the 15 web pages used in this study came from different sources and varied in domain and complexity. They also varied in implementation style, with some utilising popular front-end design frameworks while others used bespoke CSS.

**TOOL IMPLEMENTATION:** Errors in the implementation of REDECHECK are also a significant risk to the validity of the results. Therefore, extensive unit testing was employed on the main components of the tool to verify the performance of the tool. Additionally, during the development of the tool, the RLGs extracted for some subjects and the difference reports produced were manually analysed to further establish confidence in the tool's correctness. Finally, REDECHECK makes use of several third-party libraries, such as the JSoup HTML parser and the JStyleParser library for CSS manipulation. Therefore, these were

also thoroughly tested to mitigate the risk of errors in these external sources compromising the empirical results.

**REALISM OF GENERATED MUTANTS:** Figure 4.2 illustrated a proposed usage scenario for this chapter’s approach, in which a developer implements a code change. However, due to the difficulty in obtaining real examples of code modifications, this chapter’s experiments used an automated code mutation approach to introduce incremental changes into a web page. This leads to another threat to validity, because if the changes introduced are not representative of those made by developers in a real development environment, it could compromise the empirical results. While the eight mutation operators certainly did not insert every type of code modification possible, they focussed on aspects of HTML and CSS specifically targeting the web page’s layout. Furthermore, by randomly generating the modifications, the 450 code modifications produced should represent a wide range of layout changes a developer could implement. To enable replication and analysis of the experiments in this chapter, both the mutated web pages and the results are available online [144, 145].

**MANUAL CLASSIFICATION:** As previously described, some of the experimental procedure in this chapter involves humans manually analysing a web page to determine whether or not a layout change was evident. While the ideal option would be a consistent automated approach, no such approach exists. Thus, previous work on detecting presentation issues in web pages performed this task manually [7, 33, 105, 123]. As the manual analysis was not performed under controlled conditions, the data may not be representative of the real-world procedure performed by developers. However, the human participants followed a fully defined manual procedure designed to match the intuition used by real-world developers when checking for layout issues. Furthermore, to mitigate the risk of subjectivity affecting the results, the three participants discussed the individual decisions as a committee to produce a final classification. Finally, it is worth noting that the majority of the manual analysis results are solely anecdotal evidence for the benefits of using REDECHECK, rather than the main empirical contribution.

#### 4.4.2 *Empirical Results*

**RESEARCH QUESTION ONE:** Table 4.3 presents the results of applying REDECHECK, SPOTCHECK-AG and SPOTCHECK-MANUAL to the selected 60 modified web pages.

**Table 4.3:** Results summarising how REDECHECK, SPOTCHECK-AG and SPOTCHECK-MANUAL detect the incremental code modifications.

In the column headings, each method is represented by its first letter (e.g., “R” stands for “REDECHECK”, “S-AG” denotes the use of SPOTCHECK-AG and “S-MAN” corresponds to the manual classification process of SPOTCHECK-MANUAL).

MUTATION OPERATOR	TRUE POSITIVE			TRUE NEGATIVE			FALSE POSITIVE			FALSE NEGATIVE		
	R	S-AG	S-MAN	R	S-AG	S-MAN	R	S-AG	S-MAN	R	S-AG	S-MAN
RULE-VALUE	4	2	3	0	0	1	1	1	0	0	2	1
MQ-EXPRESSION	5	4	5	2	2	2	0	0	0	0	1	0
HTML-CONTENT	9	7	8	0	0	0	0	0	0	0	2	1
BREAKPOINT-CHANGE	7	5	4	1	1	1	0	0	0	0	2	3
CLASS-DELETION	6	5	6	2	2	2	0	0	0	0	1	0
RULE-UNIT	6	5	5	3	2	3	0	1	0	0	1	1
CLASS-EXCHANGE	5	5	5	1	0	1	0	1	0	0	0	0
CLASS-ADDITION	6	6	6	2	1	2	0	1	0	0	0	0
<b>Total</b>	<b>48</b>	<b>39</b>	<b>42</b>	<b>11</b>	<b>8</b>	<b>12</b>	<b>1</b>	<b>4</b>	<b>0</b>	<b>0</b>	<b>9</b>	<b>6</b>

The results are separated out by mutation operator (one for each row) and then by classification (one column for true positive, one for true negative etc.). The results show REDECHECK correctly identified all 48 layout changes introduced. SPOTCHECK-AG and SPOTCHECK-MANUAL performed significantly worse, detecting only 39 and 42 changes, respectively. This reduced recall was primarily due to the layout changes not being visible at any of the viewport widths tested by SPOTCHECK-AG and SPOTCHECK-MANUAL.

Coupled with this high recall, REDECHECK also demonstrated a high degree of precision. It reported just a single misleading false positive result. After investigating this false positive result by manually inspecting the two versions of the web page, the code modification was found to not have any impact whatsoever on the visual appearance of the web page. However, it had altered the underlying DOM enough to cause a difference in the extracted RLG. This highlights a potential risk with REDECHECK, as it uses the DOM rather than the visual appearance of the web page. However, while SPOTCHECK-MANUAL produced no false positives, SPOTCHECK-AG reported 4, suggesting the shortcoming of REDECHECK is minimal.

Although only performed by three human testers, and not under controlled conditions, the experiment did highlight the two main problems with manual web checking. The first of these is its labour intensiveness. When considering the times observed for each tester, averaged across the 15 web pages, Annette’s Creations required the least time to test with a shortest observed time of 37 seconds. Meanwhile, Shield took the longest, with one tester taking on average more than 10 minutes to perform the manual analysis procedure. These results

are indicative of the general trend observed, in which an increased web page complexity led to a longer analysis time. Intuitively, this is to be expected, as larger, more complex web pages contain more individual elements to check. In some cases, they also require a significant amount of vertical scrolling to view the whole of the web page due to the layout implemented by the developer. The two aforementioned test subjects are again excellent examples of this. Annette's Creations requires very little scrolling, even at narrow viewport widths, as there is so little content displayed on the page. In contrast, Shield — which the developers designed specifically as a “single-page” web site template — requires a huge amount of scrolling at narrow viewport widths. It even requires a reasonable amount of scrolling at the widest viewport widths. The results also showed significant variance in the amount of time required by different testers. This suggests some developers/testers may require more time than others when checking in practice. Importantly, it is worth noting that the execution time of REDECHECK was lower than even the fastest human during the experiments.

The other main problem highlighted by our experiments was the subjective and therefore error-prone nature of manual web page checking. As mentioned previously, each human performed the manual testing individually and then discussed their decisions as a committee. During this discussion, the three testers discovered their initial classifications differed for 21 of the 60 modified web pages. This suggests humans may disagree on what constitutes a layout change. This emphasises just how subjective and error-prone manual checking can be. It also provides further empirical support for this chapter's approach, as a fast, effective and reliable method for detecting the unseen side-effects of code modifications would clearly be hugely beneficial to developers in the real world.

**Conclusion for RQ1** The results indicate this chapter's approach can accurately and precisely detect unseen side-effects of code changes, with no false negatives and just a single false positive result. They also indicate the proposed approach is superior to both SPOTCHECK-AG and SPOTCHECK-MANUAL, with higher levels of both recall and precision, highlighting the shortcomings of spot check-based testing approaches.

**RESEARCH QUESTION TWO:** Table 4.4 presents the change detection results for both REDECHECK and SPOTCHECK-AG when executed on the full set of 450 modified pages. Each row represents one “bucket” and the column #LC represents the number of injected layout changes placed in each bucket. Subtle mutants are



**Table 4.4:** The effectiveness of REDECHECK and SPOTCHECK-AG at detecting layout changes that vary according to how “subtle” they are.

#DW stands for the number of distinct viewport widths at which the layout change is evident; we say that a fault is less subtle if it is visible at a greater number of viewport widths. The remainder of the table gives the number of visible layout changes and the number of these that are detected by REDECHECK and SPOTCHECK-AG, respectively.

#DW	#LC	REDECHECK		SPOTCHECK-AG	
		Detected	Recall	Detected	Recall
1	5	5	1	1	0.2
2-3	9	9	1	4	0.4444
4-5	11	8	0.7273	2	0.1818
6-10	11	10	0.9091	9	0.8182
11-50	4	4	1	3	0.75
51-100	5	5	1	3	0.6
101-300	26	22	0.8462	20	0.7692
301-500	29	26	0.8966	24	0.8276
501+	198	176	0.8889	176	0.8889
Total	298	264	0.8859	242	0.8121

those for which the value of #DW is small, while a large #DW value indicates a layout change is more “obvious”. The results show REDECHECK was superior to SPOTCHECK-AG for all nine buckets. In some cases, it is by a significant margin, such as for the 4-5 viewport width bucket. Here, REDECHECK detects 8 out of the 11 inserted changes, compared to just 2 for SPOTCHECK-AG. The results also suggest the number of viewport widths at which a layout change is evident has little to no impact on the performance of REDECHECK. It exhibits consistently high recall across the various buckets. In contrast, the performance of SPOTCHECK-AG was fairly inconsistent across buckets. Despite this, the recall values do generally trend upwards as the value of #DW increases. I expected this, however, as intuitively the more viewport widths at which a layout change is observable, the higher probability that one of the selected spot check widths will allow the tester/technique to observe the change.

For layout changes with a low degree of subtlety — those visible at more than 100 viewport widths — the performances of REDECHECK and SPOTCHECK-AG were comparable, with high levels of recall achieved by both. However, these mutants would likely be easily detected by a human performing a very simple

checking process. It is therefore arguable that layout changes visible at few viewport widths are more important for evaluating the relative performance of the two techniques. These are the changes human testers are most likely to miss. For these mutants, REDECHECK's superiority over SPOTCHECK-AG is particularly pronounced. For instance, consider the layout changes visible at 10 or fewer viewport widths (i.e., the first four buckets). REDECHECK detected 32 out of these 36 layout changes, for a recall score of 88.9%. These results are very promising, as they show the proposed approach can accurately detect layout changes humans would likely miss. For the same set of pages, SPOTCHECK-AG failed to identify 20 layout changes, resulting in a substantially lower recall of 44.4%. This lends strong empirical support to this chapter's proposed approach of sampling the web page and constructing the RLG to model its layout, rather than the static viewport inspection advocated by SPOTCHECK-AG and a range of other common developer tools.

**Conclusion for RQ2** The performance of REDECHECK does not appear to be linked to the value of #DW, as it achieved consistently high recall across the various buckets. The results also suggest spot-checking techniques such as SPOTCHECK-AG struggle to detect layout changes that occur at only a small number of viewport widths. In other words, for subtle mutants the benefits of using the proposed approach are particularly pronounced.

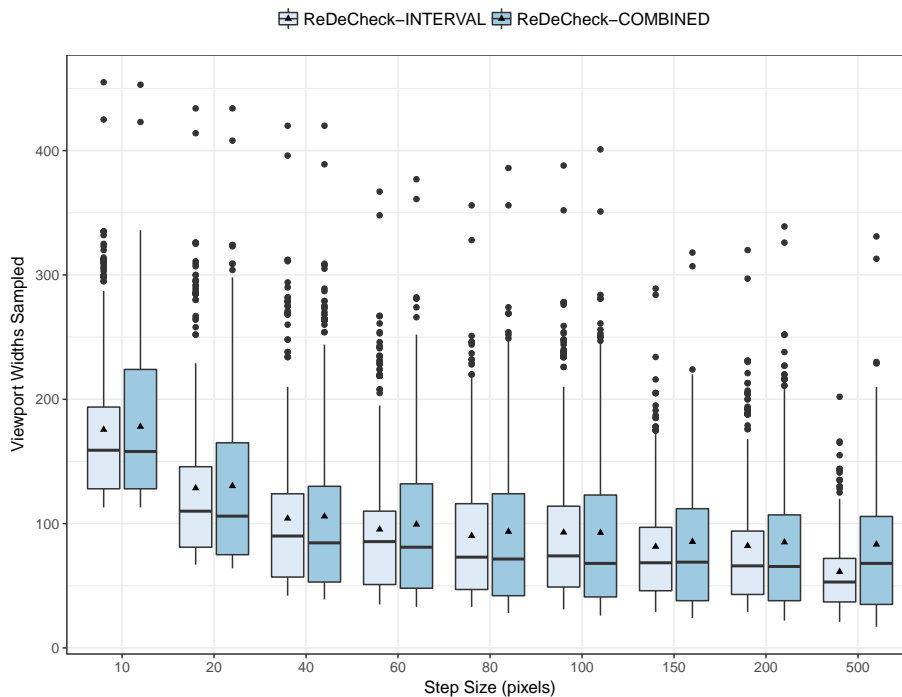
RESEARCH QUESTION THREE: Table 4.5 presents the change detection results for each combination of step size and sampling technique. It also presents the results on a subject by subject basis. For each step size, the column headings I and C represent the REDECHECK-INTERVAL and REDECHECK-COMBINED sampling techniques, respectively. The column heading "Changes" presents the number of layout changes introduced into each subject. The results clearly show very little variance in the change detection ability of REDECHECK when using either REDECHECK-INTERVAL or REDECHECK-COMBINED as the sampling technique. There are, however, a couple of results of interest. Firstly, for Ashton Snook and Pay Demand, REDECHECK-COMBINED outperformed REDECHECK-INTERVAL across all nine step sizes. This suggests the addition of breakpoints into the initial sample set allowed REDECHECK to detect more changes. The "unintelligent" sampling of REDECHECK-INTERVAL can miss layout changes occurring inbetween or close to breakpoints. Secondly, REDECHECK-INTERVAL demonstrated a reduction in effectiveness as the step size increases for five of the web pages. In comparison, this only occurred for one subject when using REDECHECK-

**Table 4.5:** REDECHECK’s effectiveness at detecting layout changes at various step sizes. In this table, the label “I” stands for the REDECHECK-INTERVAL method and “C” denotes the use of REDECHECK-COMBINED.

WEB SITE NAME	Changes	10px		20px		40px		60px		80px		100px		150px		200px		500px		E
		I	C	I	C	I	C	I	C	I	C	I	C	I	C	I	C	I	C	
Aftnoon	15	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
Annette’s Creations	14	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	7	8	8
Ashton Snook	16	13	14	13	14	12	14	12	14	12	14	12	14	12	14	12	14	11	14	14
BitTorrent	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
Coursera	10	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
Denon	12	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
Bootstrap	29	28	28	28	28	28	27	28	28	28	27	28	27	28	27	28	27	28	26	28
ISSTA	19	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	11	11	11	12
Name Mesh	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
Pay Demand	29	26	27	26	27	26	27	26	27	26	27	26	27	26	27	26	27	26	27	27
Rebecca Made	21	20	20	20	20	19	19	20	20	19	19	20	20	20	20	20	20	20	20	20
Reserve	27	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
Responsive Process	26	19	19	19	19	19	19	19	19	19	19	19	18	19	18	19	18	19	17	19
Shield	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
Treehouse	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

COMBINED. The final interesting result is that of Rebecca Made, where sampling at either 40 pixels or 80 pixels causes one layout change to be missed. Finally, the results for REDECHECK-EXHAUSTIVE illustrate no benefit over the other two approaches, as it detects no additional changes. This provides empirical support for REDECHECK’s approach of sampling a web page and searching for the breakpoints at which the layout changes, rather than sampling at every single viewport width.

Given the similarity in effectiveness, recommending a set of configuration parameters becomes easier. Execution time or effort is the sole consideration, rather than striking a balance between effectiveness and efficiency. This question therefore immediately discards REDECHECK-EXHAUSTIVE. It provides no effectiveness benefit while being substantially more labour-intensive than REDECHECK-INTERVAL or REDECHECK-COMBINED. For example, execution times for REDECHECK-EXHAUSTIVE ranged from 89 seconds to 148 seconds, with a mean time of 108 seconds. In contrast, REDECHECK-INTERVAL and REDECHECK-COMBINED exhibited an average execution time of less than 30 seconds for all step sizes investigated. The brute-force nature of REDECHECK-EXHAUSTIVE is the primary reason for this. By sampling at all viewport widths — 1001 in this case — with no regard for page complexity or the number of layout changes exhibited by the web page, a large amount of computation is wasted. When using the alternative techniques, some RLGs required sampling at fewer than 100 viewport

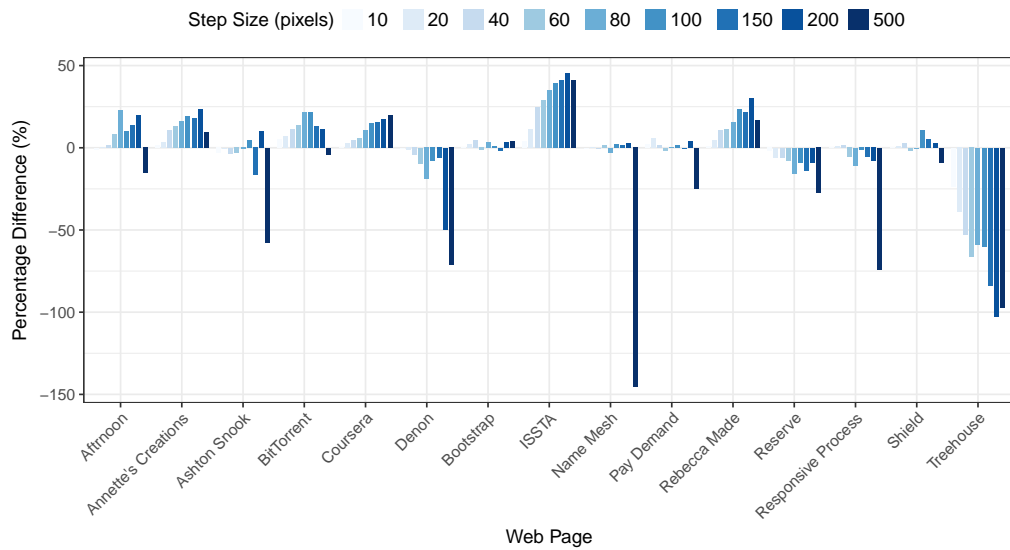


**Figure 4.10:** The number of viewport widths sampled by REDECHECK-INTERVAL and REDECHECK-COMBINED.

The boxplots show the data across all 26 web pages and for the 9 studied step sizes. Light grey boxes represent REDECHECK-INTERVAL, while dark grey ones represent REDECHECK-COMBINED.

widths: a reduction in effort of over 90%. Furthermore, responsive web design by nature results in a degree of consistency between similar viewport widths, as layout changes do not occur very frequently in well designed web pages. REDECHECK-INTERVAL and REDECHECK-COMBINED can easily capture these layout changes, meaning REDECHECK-EXHAUSTIVE samples at an unnecessarily fine granularity. Finally, given the iterative development scenario envisaged for REDECHECK, users are likely to consider REDECHECK-EXHAUSTIVE far too slow. The tool would disrupt their workflow every time it checks for unintended side-effects. This may be a sacrifice they are not willing to make. While REDECHECK-COMBINED and REDECHECK-INTERVAL do not provide instantaneous feedback to the developer, the value they provide by highlighting potential unintended issues should counteract any issues a user may have with waiting a few seconds for the change detection to be performed. Furthermore, in other types of software development, it is not unusual for a developer to wait a few seconds after making a code change for a unit test suite to run and provide feedback on the code modification made.

After discarding REDECHECK-EXHAUSTIVE, this question next compared the remaining two sampling techniques. Figure 4.10 presents a box-plot showing the



**Figure 4.11:** The percentage difference between the number of sampled viewport widths for REDeCHECK-COMBINED and REDeCHECK-INTERVAL.

In this graph, positive values indicate that REDeCHECK-COMBINED samples fewer viewport widths than the REDeCHECK-INTERVAL method. Alternatively, a negative value reveals that REDeCHECK-COMBINED samples more viewport widths than REDeCHECK-INTERVAL.

distributions of the number of viewport widths sampled using each technique. As one might expect, using a very small step size such as 10 pixels results in a much higher number of sampled widths for both techniques. For instance, the mean values were 176 for REDeCHECK-INTERVAL and 180 for REDeCHECK-COMBINED. Using a 20 pixel step size resulted in means of 129 and 130, respectively. There is a clear trend of the number of sampled widths decreasing as the step size increases. However, the trend reaches a plateau at around the 60 pixel step size. Again, this was expected, as initial experimentation using REDeCHECK suggested 60 pixels was a sensible step size to balance effectiveness and efficiency. The most interesting observation was the very close similarity between the two techniques. Prior to the experiments, the expectation was that REDeCHECK-COMBINED would require fewer viewport widths to be sampled in order to create the RLG. The addition of the extracted breakpoints into the initial sample set should theoretically remove the need for many costly binary searches. There are two likely reasons for this similarity. The first was the majority of layout changes detected during the sampling process occurred at widths not programmed in the CSS. This would mean REDeCHECK performed the binary searches regardless of sampling technique. The second concerned the cost overhead of also sampling at the breakpoint viewport widths using REDeCHECK-COMBINED. It could potentially be so large that any saving obtained only counteracted the initial overhead, rather than producing an overall benefit.

When considering the collection of test subjects as a whole, the two sampling techniques are very similar. However, this question then investigated how the performance of each varied on a web page by web page basis. Figure 4.11 presents the difference between the number of viewport widths sampled by each configuration on each web page. In the figure, positive values indicate REDECHECK-COMBINED required fewer viewport widths (i.e., provided an efficiency improvement). Meanwhile, negative values indicate adding in the extracted breakpoints made REDECHECK-COMBINED less efficient than REDECHECK-INTERVAL. The two subjects demonstrating the two polar opposite impacts are ISSTA and Treehouse. For ISSTA, REDECHECK-COMBINED was substantially more efficient, while for Treehouse REDECHECK-COMBINED required much more effort than REDECHECK-INTERVAL. Further manual analysis of the subjects revealed ISSTA almost no layout changes at viewport widths not programmed as breakpoints in the CSS. This meant REDECHECK-COMBINED required much less effort to extract the RLG. On the other hand, Treehouse contained a large amount of breakpoints in its CSS. This caused any reduction in binary searches to be completely overshadowed by the increase in the initial effort overhead. The remainder of the web pages demonstrated less severe effects, with the majority demonstrating REDECHECK-COMBINED resulted in either equivalent or marginally increased efficiency.

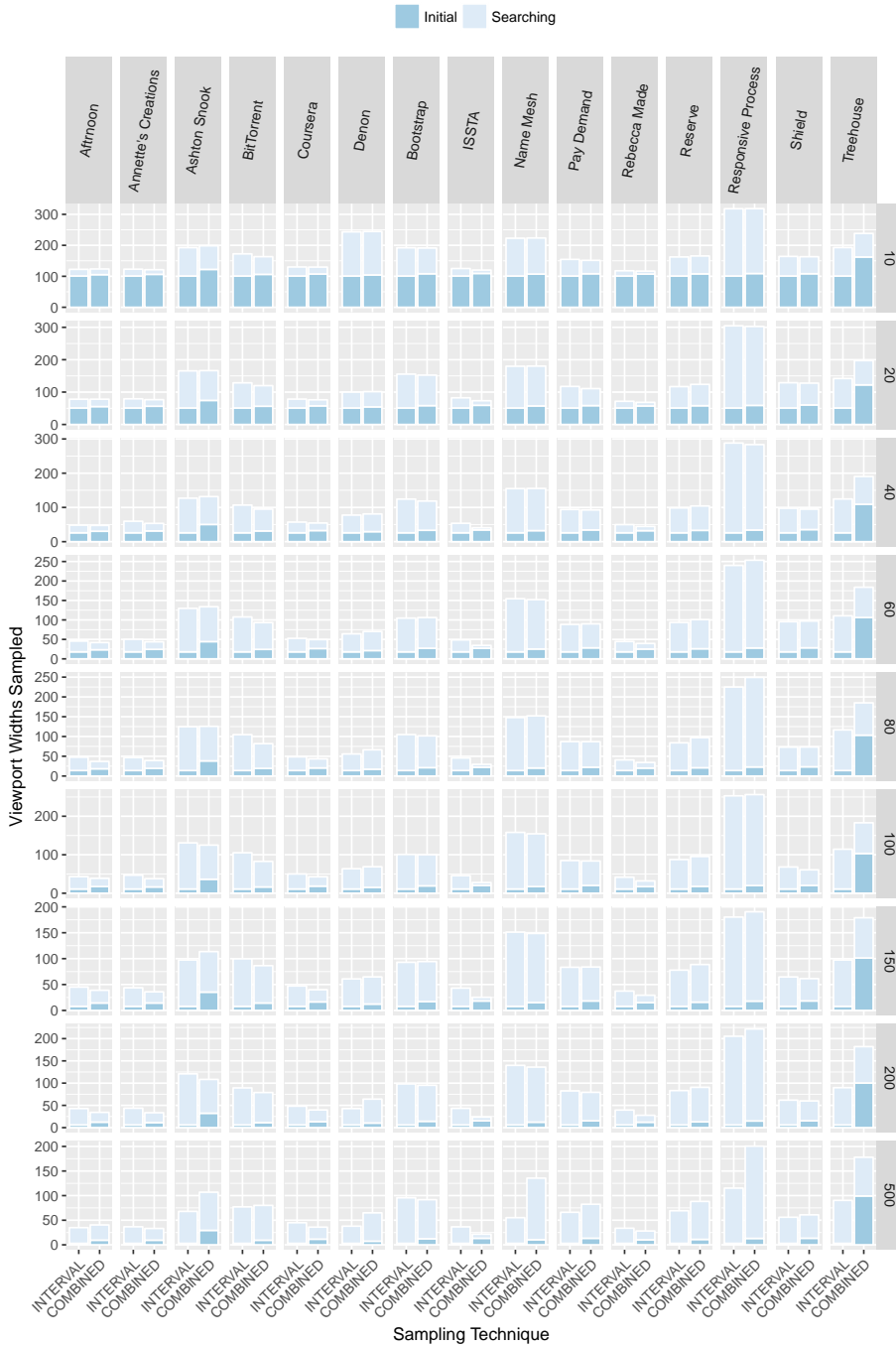
Next, this question further investigated how REDECHECK-INTERVAL and REDECHECK-COMBINED sample a web page to extract an RLG. There are two stages to sampling. These are the *initial* sample and the subsequent *binary searching*. Figure 4.12 shows how many of the total sampled viewport widths are attributable to the two stages. The dark blue bars represent the initial sample, and the light blue bar the binary searching. The data is also separated out by both web page and step size to allow more in-depth analysis of specific results. Along with ISSTA, subjects such as Afrnoon, Annette's Creations, BitTorrent, Coursera and Rebecca Made demonstrate the ideal scenario envisioned when developing the REDECHECK-COMBINED sampling technique. For these, the sampling effort saved due to fewer binary searches more than outweighs the cost of initially sampling at the extra viewport widths. The figure also emphasises the aforementioned problem with Treehouse, where the substantially larger dark blue bars for REDECHECK-COMBINED show the cost of adding the breakpoints to the initial sample. Finally, Ashton Snook, Pay Demand and Shield show how the saving on binary searches sometimes only allows REDECHECK-COMBINED to achieve efficiency equal to that of REDECHECK-INTERVAL.

In the scenarios in which the efficiency of the two techniques is similar, there is no quantitative evidence to support the choice of one over the other. Using REDECHECK-COMBINED is likely to produce an RLG that better represents the responsive layout of the web page. Meanwhile, REDECHECK-INTERVAL has the potential to miss layout behaviour occurring between consecutive sample widths. For this reason, the use of REDECHECK-COMBINED over REDECHECK-INTERVAL is preferable when their efficiencies are equivalent.

Following the observations of the trends illustrated by Figure 4.11 and Figure 4.12, statistical hypothesis tests were run on the efficiency data for the two sampling techniques. To begin, the Mann–Whitney  $U$ -test compared the distributions of the two techniques for the nine step sizes investigated. This non-parametric test was used since the normality of the sample means was not guaranteed, as required by parametric statistical tests such as the  $t$ -test. These statistical tests served two purposes. The first was to determine if REDECHECK-COMBINED sampled significantly fewer viewport widths than REDECHECK-INTERVAL. The second was to determine if different step sizes caused an individual technique to sample viewport widths in a manner that is statistically significantly different.

To complement the significance tests, the non-parametric  $\hat{A}_{12}$  statistic of Vargha and Delaney [138] was also employed. This computes effect sizes, which determine the average probability that one approach “outperforms” another. Following the guidelines of Vargha and Delaney, an effect size was “large” if the value of  $\hat{A}_{12}$  is  $< 0.29$  or  $> 0.71$ , “medium” if  $\hat{A}_{12}$  is  $< 0.36$  or  $> 0.64$  and “small” if  $\hat{A}_{12}$  is  $< 0.44$  or  $> 0.56$ . Values of  $\hat{A}_{12}$  close to 0.5 show no practical effect. This study interprets the effect size values in the following fashion. When comparing REDECHECK-COMBINED and REDECHECK-INTERVAL, a value near 1 means REDECHECK-COMBINED is likely to be more efficient than REDECHECK-INTERVAL. Conversely, a value close to 0 indicates REDECHECK-INTERVAL is better. When comparing REDECHECK’s use of different step sizes, a value near 1 means that the larger step size is preferred, while a value near 0 means that the smaller step size is better on average.

Part a) of Table 4.6 shows the results for comparing REDECHECK-COMBINED and REDECHECK-INTERVAL. The  $p$ -values show that for eight step sizes there is no evidence to suggest the techniques demonstrate significantly different efficiencies. The only exception to this is 500 pixels. These values are consistent with the box-plots in Figure 4.10, where there is little difference evident between the two techniques. The  $\hat{A}_{12}$  test results show a small efficiency benefit when using



**Figure 4.12:** A breakdown of the number of viewport widths sampled initially, and then during binary searches to complete the RLG, shown in dark blue and light blue, respectively.



**Table 4.6:** The statistical hypothesis test results.

The left-hand table shows the comparison of REDECHECK-COMBINED and REDECHECK-INTERVAL at each step size, while the right-hand table shows the comparison of consecutive step sizes using the REDECHECK-COMBINED sampling approach.

Step Size (pixels)	$p$ -value	$\hat{A}_{12}$	Step Size (pixels)	$p$ -value	$\hat{A}_{12}$
10	0.81991	0.496	10 vs 20	0	0.751
20	0.38434	0.517	20 vs 40	0	0.644
40	0.29856	0.52	40 vs 60	0.03086	0.542
60	0.61107	0.51	60 vs 80	0.01031	0.549
80	0.25719	0.522	80 vs 100	0.33473	0.519
100	0.06289	0.536	100 vs 150	0.11474	0.53
150	0.39921	0.516	150 vs 200	0.41891	0.516
200	0.12911	0.529	200 vs 500	0.63447	0.509
500	0.00013	0.426			

(a) Comparing sampling techniques

(b) Comparing step sizes for REDECHECK-COMBINED

REDECHECK-COMBINED, as all but two of the values are above 0.5. However, none of the effect sizes are large enough to be significant.

After establishing the small efficiency benefit of using REDECHECK-COMBINED over REDECHECK-INTERVAL, further statistical tests were conducted. These aimed to establish which step size was the most efficient when using REDECHECK-COMBINED. Part (b) of Table 4.6 presents the results of comparing the distributions for consecutive pairs of step sizes. The results again mirror the trends shown by Figure 4.10. The tests found very low  $p$ -values for the pairs 10 pixels-20 pixels, 20 pixels-40 pixels, 40 pixels-60 pixels and 60 pixels-80 pixels. These correspond to the downward trend in the number of sampled viewport widths as the step size increases. Meanwhile, the values of 0.335 and 0.115 for the pairs 80 pixels-100 pixels and 100 pixels-150 pixels also correspond to the observed plateau. Interpreting the  $\hat{A}_{12}$  values as probabilities, the results show 20 pixels is highly probable (75.1%) to be more efficient than 10 pixels. Likewise, 40 pixels will outperform 20 pixels 64.4% of the time. For the comparisons between 40 pixels-60 pixels and 60 pixels-80 pixels, the respective values of 0.542 and 0.549 indicate the benefits of using the latter step sizes have no significant effect size, but they are almost classified as small. These results suggest the choice of 60 pixels over 40 pixels for the experiments in RQ1 and RQ2 was sensible.

However, they also suggest the previous experiments could potentially have used 80 pixels instead. Finally, as the values for the comparisons between the larger step sizes are all near to 0.5, it is clear there was no benefit to using 100 pixels or 150 pixels in place of 60 pixels.

**Conclusion for RQ3** For the collection of layout changes used in this study, the sampling technique and step size used has little impact on the change detection ability of the approach. REDECHECK-COMBINED did however demonstrate a boost in efficiency over REDECHECK-INTERVAL. Statistical analysis also supported the use of 60 pixels as a global step size which should perform well across a wide variety of web pages.

#### 4.4.3 Discussion

During this chapter's empirical evaluation, several qualitative benefits were observed. Firstly and most importantly, the automatic technique removed the need for a developer to select the viewport widths at which to check the web page. This selection process has been shown to be error-prone and subject to missing layout changes. Theoretically, a developer could simply resize the browser randomly and examine the layout in the hope of observing any layout issues that may be present. However, this approach is likely to be ineffective and inconsistent. There is no way of telling the viewport widths at which any existing layout issues are evident. This means many of them could go undetected (as shown by a later experiment detailed in Chapter 5). The modelling performed by REDECHECK is particularly useful, since many observed layout changes manifested at unpredictable viewport widths. These included those not covered by popular devices or advocated by popular tools and those between the common breakpoint widths used by both developers and RWD frameworks.

The second key benefit of the approach is the reports provided to the user. These are likely to be highly useful, providing guidance to developers and testers when performing more in-depth manual checking of their web sites. As the RLG models the full responsive layout of a web page, the output of the approach provides considerably more contextual information than other competing approaches. These only report a simple binary decision as to whether any changes in a page's layout have been found. REDECHECK informs the developer of *where* the change occurred (which elements and layout relationships

changed), *how* the layout changed and *when* (at which viewport widths are the changes observable). As further development of REDECHECK is undertaken, these benefits are likely to become more pronounced. For instance, REDECHECK could provide an interactive, graphical report highlighting the differing parts of the RLG. Another option is a screenshot-based approach showing the change in layout of the modified elements. Either of these improvements could significantly increase the usability of the approach.

As previously discussed, the approach failed to detect a small number of the injected layout changes. However, manual analysis of the offending modifications found almost all of§ of these false-negative results occurred because the injected code modification impacted the underlying DOM in such a subtle manner that it did not change the extracted RLG. A tiny shift in the position of an element, perhaps made by changing its padding or margin by a small amount, is a common example of this. While the experiment must report these issues as false-negatives, their visual impact is so small humans would be highly unlikely to detect them by manually checking the layout of the web page. End users are even less likely to observe them, suggesting the shortcoming in the approach is minimal.

Finally, as most modern web sites consist of multiple pages, a tester wishing to check for layout changes across an entire site would need to run REDECHECK on a page-by-page basis. While this task is currently a manual one and may be cumbersome, the process is fully amenable to automation. Checking the layout across different web pages within the same web site is vital, as CSS files are almost always shared between pages to provide a consistent look-and-feel to a site. Therefore, it is possible a CSS modification which causes no changes on one page could have a drastic impact on another page.

#### 4.5 CONCLUDING REMARKS

Building upon the responsive layout graph introduced in Chapter 3, this chapter presented an approach for automatically detecting unseen layout side-effects following small changes to a responsive web page. The approach compared the RLGs of two versions of a web page to detect any layout changes. The approach was implemented in a tool named REDECHECK, and evaluated in the context of three main research questions. The first of these found that the proposed approach outperforms both manual and automated spot-checking baselines,

with a higher number of true positive results and fewer false positive results. The second question investigated the performance of both the proposed approach and the automated baseline on layout changes of varying subtlety. The results showed that while changes with little subtlety produce comparable performance between the two techniques, on “subtle” layout changes the proposed approach outperforms the automated baseline by a substantial margin. Finally, the third research question investigated the effectiveness and efficiency of the approach under a wide variety of different sampling techniques and step sizes. The results demonstrated that the configuration parameters chosen have very little impact on the approach’s ability to detect layout changes. Subsequent experiments then showed that while there are various options regarding the configuration of `ReDeCheck`, the `ReDeCheck-Combined` sampling technique first presented in Figure 3.2 with a step size of 60 pixels is a sensible choice which should perform well on a wide variety of web pages.

---

## DETECTING COMMON TYPES OF RESPONSIVE LAYOUT FAILURES

---

Chapter 3 introduced the responsive layout graph (RLG) as a way to model a web page’s responsive layout. Then, Chapter 4 showed how comparing the RLGs of two versions of a web page can highlight unintended layout side-effects caused by code modifications to the developer quickly and effectively. However, this approach suffers from three main problems. Firstly, the developer is responsible for analysing all of the reported layout differences. They must decide whether each was intentional or not. This makes the approach still potentially labour-intensive. Secondly, it is only capable of detecting presentation failures of a *regression* nature. This renders it useless if a previous version of a web page is unavailable. Also, if the previous version is significantly different to the current one, it would report an overwhelming number of model differences. Finally, if a failure is present in both versions of a web page, it would not regard it as a difference. It would therefore not bring it to the attention of the developer, meaning the failure could continue to manifest in the live version of the web page.

This chapter addresses these problems by first describing five different types of *responsive layout failure (RLF)*. These are presentation failures that occur intermittently across the full range of viewport widths at which someone could view a web page. Two examples are overlapping elements and elements protruding outside of the visible viewport. This addresses the issue of distinguishing between intended layout changes and “real” layout failures. Next, this chapter presents an approach that can detect these failures. To address the oracle problem (i.e., the previous version of the web page), it does not require an explicit oracle, such as a previous version of the web page, or a screenshot showing the intended layout of the web page. Instead, it works by first building the RLG of just the latest version of the web page under test. Then, a series of algo-

rithms searches the RLG for patterns signifying the five different types of RLF. Finally, it reports any identified failures to the developer. This also tackles the third problem with the previous chapter's approach. By leveraging implicit oracle information rather than an explicit oracle, failures present in both versions of a web page can be detected. I implemented the approach into the existing REDECHECK tool. Users now have the choice of running REDECHECK in one of two testing modes: the *regression checking* approach of Chapter 4 or the common failure detection approach presented in this chapter.

The approach is then evaluated in three main experiments. The first of these investigates the effectiveness of the approach at detecting common RLFs in web pages. Using a corpus of 26 randomly selected responsive web pages, the results show that these RLFs are prevalent in real-world web pages. The second experiment compares the proposed approach to several common RWD spot-checking tools. The final experiment investigates the efficiency of the approach, as the execution time must be short enough to not detrimentally impact the productivity of a developer.

Following on from the main empirical evaluation, the chapter concludes by presenting several small modifications to the approach to improve its accuracy and precision followed by a smaller evaluation to assess the effects of the improvements.

The key contributions of this chapter are:

1. A categorisation of five common types of *responsive layout failure (RLF)* discoverable without the need for explicit oracles.
2. A collection of four algorithms that automatically analyse the RLG of a web page in order to detect the five types of RLF. These algorithms have been implemented as a module of REDECHECK.
3. An empirical evaluation of 26 randomly selected production web pages, showing that the RLF types identified are prevalent in live sites and the algorithms are capable of detecting them and reporting them to the developer.
4. Modifications to the RLG model and failure detection algorithms that reduce the quantity of misleading false positive results reported by the approach.

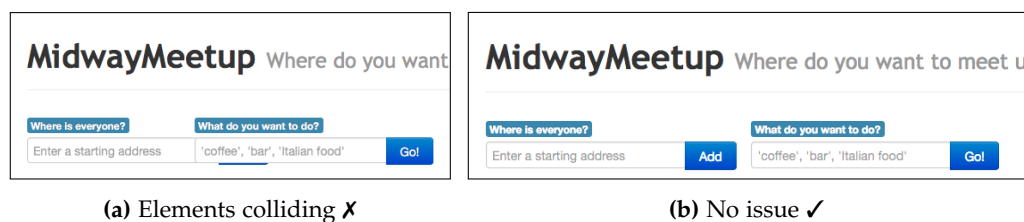
## 5.1 A CATEGORISATION OF RESPONSIVE LAYOUT FAILURES

This section defines five distinct types of *responsive layout failure (RLF)*. Each of these are problematic for developers of responsive web sites and users who browse them. These failure types were identified primarily through experience of browsing the web on different types of device and experiencing issues while doing so, although my knowledge of web development also played a part by allowing me to envision potential failures that could be introduced by modifying the code of a web page in some way.

Each type of RLF is generally caused by the misapplication of one or more of the three RWD “ingredients”: grid-based layouts, flexible media and media queries. For each category of RLF, this section presents a real-world example to show the visual impact of each failure type on the aesthetics of a web page.

### Element Collision

As viewport widths narrow, developers commonly shift horizontally-aligned elements closer together to cope with the reduced horizontal space. However, as the viewport contracts further, elements can collide with one another and their contents begin to overlap. The effects of this can be purely aesthetic, such as overlaid images. They can also potentially be much more damaging, if, for example, the overlap obscures crucial text or navigational links. Figure 5.1 presents an example of this, taken from the web page MidwayMeetup ([www.midwaymeetup.com](http://www.midwaymeetup.com)).



**Figure 5.1:** An example of an element collision.

At the wider viewport width shown in part (b), the two input forms are displayed comfortably side-by-side. As the viewport narrows, the two forms collide and the right-hand form is overlaid above the left-hand one. This renders the left-most submit button “Add” unclickable, as shown by part (a). This demonstrates a collision failure affecting both the visual appearance and functionality of a web page. For users browsing this web page on a device where

the failure manifests, the unclickable button prevents a user from engaging in the primary function of the web page, making it completely useless.

### Element Protrusion

Another key concern for developers of responsive web pages is ensuring that as the viewport width reduces, the elements on the web page resize to remain wide enough to contain their contents. Conversely, the contained elements must remain small enough to fit within their containers. Fluid grid-based layouts and flexible media aid developers in this task. However, if incorrectly implemented, elements may protrude outside of their containers into neighbouring areas of the web page. Figure 5.2 presents an example of an element protrusion from the web page PDFescape (www.pdfescape.com).

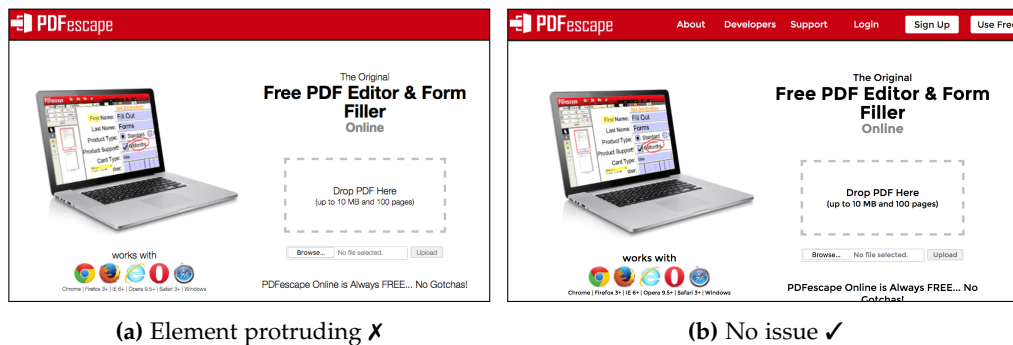


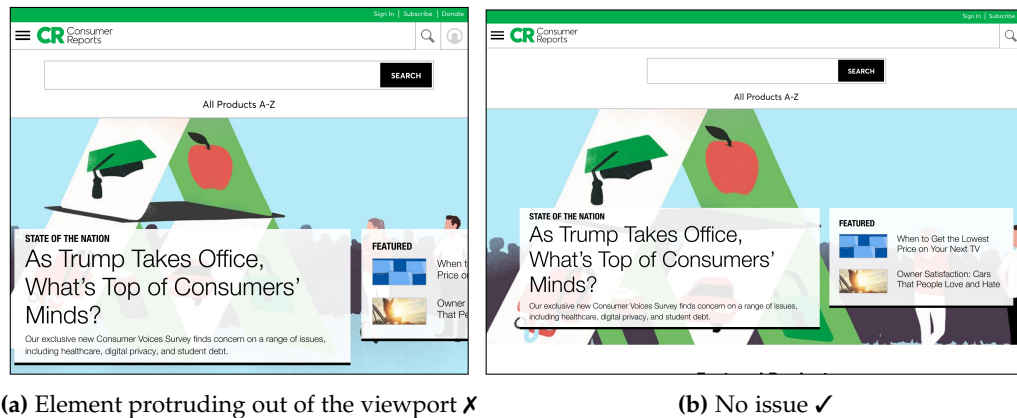
Figure 5.2: An example of an element protrusion.

At the wider viewport shown by part (b), the navigation bar at the top of the page contains a row of links. However, at the narrower viewport width of part (a), the set of links are too wide to fit alongside the web page logo in the navigation bar. This causes them to protrude out into the main content of the page. Ordinarily, the effect would be purely aesthetic, as the links would still remain clickable in their new location. However, in this scenario, the container has the overflow: hidden CSS property. This hides any overflowing content from view, meaning the links become not only invisible but also unclickable. This makes navigation of the page very difficult on devices where this failure manifests.

### Viewport Protrusion

When viewport widths narrow, elements may not only protrude out of their containers, but also outside of the web page's root presentational element, the body tag. This means they are rendered outside of the viewable portion of the page i.e., the viewport. These failures can again be purely aesthetic or in some cases, detrimental to the page's functionality.





**Figure 5.3:** An example of a viewport protrusion.

ConsumerReports ([www.consumerreports.org](http://www.consumerreports.org)) exhibits a viewport protrusion, as illustrated by Figure 5.3. At the wider viewport width of part (b), the page renders the two content tiles containing the featured articles side by side within the banner. However, at the narrower width of part (a) part of the right-hand tile has protruded outside of the viewport. This makes access to the headline content limited. Users do have the option of scrolling horizontally to access the partially hidden content. However, nowadays many of them expect to only scroll vertically when browsing a web page. Therefore, many end users may simply disregard the hidden content.

### Small-Range Layouts

As discussed in previous chapters, building responsive web pages is difficult for developers, with hundreds if not thousands of CSS rules. A series of media queries activate and deactivate these rules. More than one media query can evaluate to true at the same viewport width. For instance, the queries (`min-width: 768px`) and (`max-width: 1023px`), will both be activated for viewports between 768 and 1023 pixels. Because of this, the logic controlling which sets of rules to apply at which viewport widths can quickly become complex. As such, developers can easily make errors and apply certain rules at unintentional viewport widths. This is especially true when they mix the use of both the `min-width` and `max-width` queries. Suppose a developer uses the media query (`max-width: 768px`) to encode the layout for mobile devices, and another (`min-width: 768px`) to handle the tablet and desktop layout. Both will evaluate to true at the viewport width of 768 pixels. Media queries “clashes” such as this can cause unusual layout behaviour, as two sets of CSS rules apply to some elements when only

one should be. Due to the small number of viewport widths at which this type of failure is observable, they can be very difficult to detect manually.

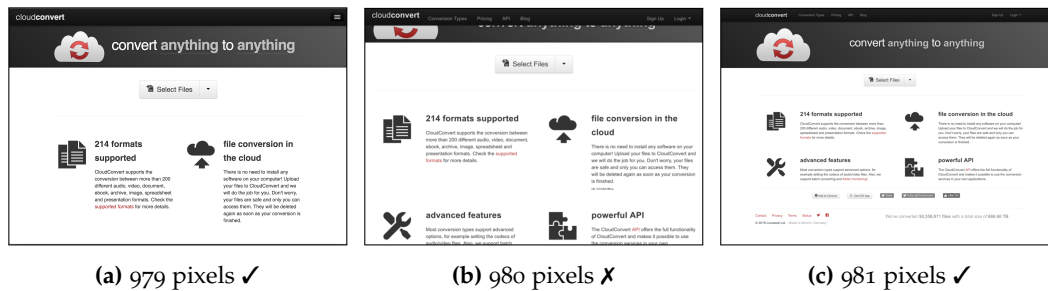


Figure 5.4: An example of a small-range layout.

Figure 5.4 presents an example of a small-range layout from the web page Cloud-convert (cloudconvert.com). At the narrower viewport width of part (a) and the wider viewport width shown in part (c), there are no issues with the layout. However, at the sole viewport width of 980 pixels in part (b), an error in the CSS media queries causes the page’s navigation bar and banner to overlap, obscuring the company’s logo and slogan.

**Wrapping Elements** As previously discussed, when elements are not wide enough to hold their contents, said contents can protrude out of the container. However, if the container remains “tall” enough, or has CSS properties making its height flexible, horizontally-aligned elements may instead “wrap” onto a new row. While not affecting a web page’s functionality, this additional row of elements is an often undesirable presentational effect.

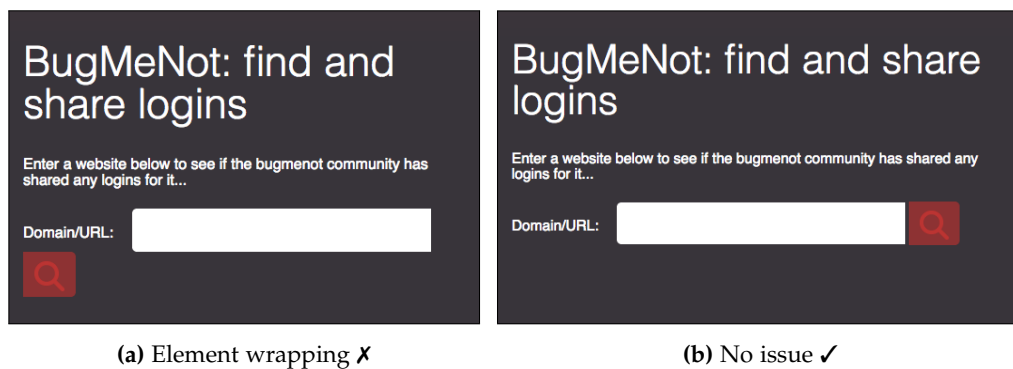


Figure 5.5: An example of a wrapping failure.

Figure 5.5 presents BugMeNot (bugmenot.com), in which a wrapping failure can be observed. In part (b) of the figure, the viewport is wide enough for the browser to render the three form components — the label, input box, and sub-

mit button — in a single row. However, due to a lack of responsiveness, as the viewport narrows the submit button wraps onto a second row.

## 5.2 DETECTING RESPONSIVE LAYOUT FAILURES WITHOUT AN EXPLICIT ORACLE

This section now describes an approach for identifying the five types of responsive layout failure. It begins with a high-level summary of the overall technique, then it presents the individual algorithms for identifying the different categories of failure.

### Summary of Approach

This chapter’s overall approach begins by automatically extracting the RLG of the web page under test, as per the approach outlined in Chapter 3. It then analyses the RLG, searching for specific layout patterns representing responsive layout failures. Finally, it reports these failures to the user via a textual report and a series of screenshots with the offending elements outlined for easy identification.

To detect element collision and element protrusion failures, the RLG models an additional layout attribute “overlapping” on sibling edges, represented by the label *O*. Determining this attribute is very simple. A simple check whether the bounding boxes of the two elements intersect is sufficient.

### Detecting Element Collisions

The detection of element collision failures involves first searching for pairs of elements that are overlapping at one range of viewport widths. The approach then checks if they are not overlapping at the immediately wider viewport width, as there is now enough horizontal space for them. Algorithm 7 formalises this technique. Figure 5.6 presents a simple wireframe example and an RLG fragment to demonstrate the algorithm in practice.

The algorithm begins by iterating through all alignment constraints in the RLG. If it finds one of the *sibling* type with the *overlapping* attribute set (line 4), it investigates it further. Using Figure 5.6 as an example, this stage of the algorithm would find the constraint  $(320, 767, s, \{O\})$  between `img[1]` and `img[2]` as a constraint of interest. Then, the algorithm obtains the alignment constraint at the immediately wider viewport width (line 5). If the function finds such a

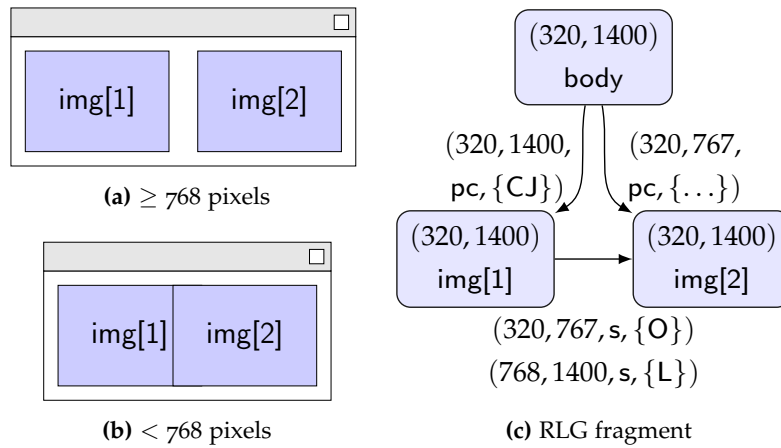


Figure 5.6: A wireframe example of an element collision.

---

### Algorithm 7 Detection of element collision & protrusion failures

---

```

1: procedure DETECTCOLLISIONANDPROTRUSIONFAILURES( $(\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ )
2:   for all  $r = (e_1, e_2) \in \mathcal{R}$  do
3:     for all  $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$  do
4:       if  $t = s \wedge O \in P$  then
5:          $(\dots, P_{wider}) \leftarrow \text{ALIGNMENTCONSTRAINTAT}(e_1, e_2, t, amax+1)$ 
6:         if  $(\dots, P_{wider}) \neq \perp \wedge O \notin P_{wider}$  then
7:           REPORTFAILURE(element-collision,  $\{e_1, e_2\}, \{(amin, amax)\}$ )
8:         else
9:            $a_1 \leftarrow \text{GETANCESTORSAT}(e_1, amax + 1)$ 
10:           $a_2 \leftarrow \text{GETANCESTORSAT}(e_2, amax + 1)$ 
11:          if  $(e_1 \in a_2) \vee (e_2 \in a_1)$  then
12:            REPORTFAILURE(element-protrusion,  $\{e_1, e_2\}, \{(amin, amax)\}$ )
13:          end if
14:        end if
15:      end if
16:    end for
17:  end for
18: end procedure

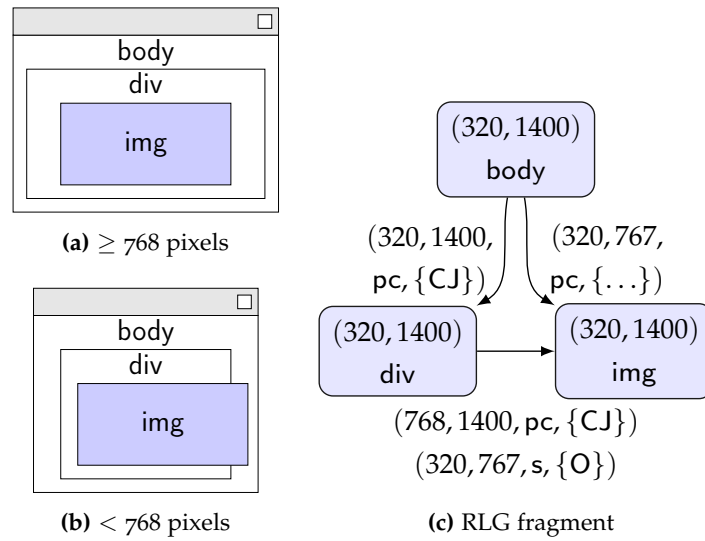
```

---

constraint, the algorithm inspects its attribute set for the overlapping attribute (line 6). If the attribute is **not** present, it signifies the elements are no longer overlapping. In this case, the algorithm reports an element collision failure (line 7). Continuing with the wireframe example, line 6 would obtain the constraint  $(768, 1400, s, \{L\})$ . As the overlapping attribute is not present, Algorithm 7 reports the first constraint to the user as an element collision failure.

### Detecting Element Protrusions

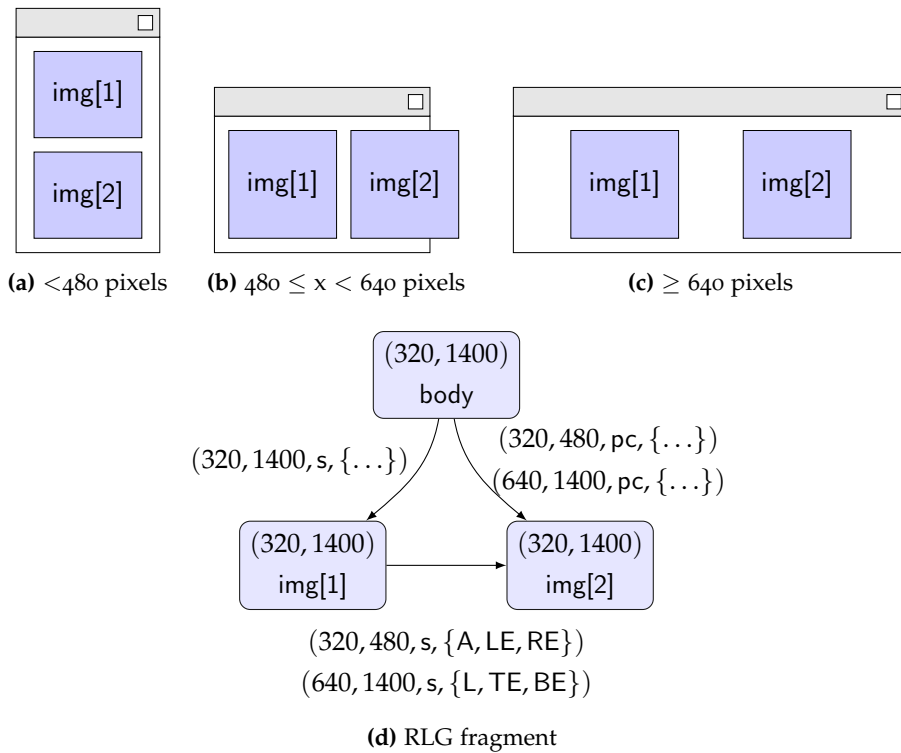
The detection of element protrusions involves querying the RLG in a similar way to the detection of element collisions. This time, the approach does not search for a change in alignment attributes between two elements. Instead, it



**Figure 5.7:** A wireframe example of an element protrusion.

searches for a change in relationship type. The key insight is as follows. Normally the two offending elements are in a parent-child relationship, as one contains the other. However, as the viewport narrows and the “child” starts to protrude, it begins to overlap with its intended parent. Because the detection of element protrusions depends on initially finding a pair of overlapping elements, the approach also uses Algorithm 7, continuing from line 8. Here, it has identified a pair of elements as overlapping, but not because of an element collision failure.

Figure 5.7 presents a simple wireframe example of an element protrusion failure. The initial search for overlapping sibling elements would find the constraint  $(320, 767, s, \{O\})$  between the `div` and `img` elements. If an element protrusion has occurred, one of the two elements is a child of the other (or some ancestor of the other), at the immediately wider viewport width. Therefore, the algorithm inspects the RLG’s alignment constraints, traversing them to obtain the ancestry of the two overlapping elements at the wider viewport width (lines 9-10). In this case, the constraint  $(320, 1400, pc, \{CJ\})$  between `body` and `div` means `div`’s ancestry set is  $\{body\}$ . Similarly, the constraint  $(768, 1400, pc, \{CJ\})$  between `img` and `div` means `img`’s ancestry is  $\{div\}$ . The algorithm then checks the ancestry sets. If  $e_1$  is an ancestor of  $e_2$ , or vice versa (line 11), it reports an element protrusion failure at the viewport widths where the two elements are overlapping. In this case, when inspecting the ancestry of  $e_2$ , the algorithm sees that `div` is an ancestor of `img` and therefore reports the protrusion to the user.



**Figure 5.8:** Example of an viewport overflow failure, and its corresponding RLG fragment.

### Detecting Viewport Protrusions

Viewport protrusion failures are essentially element protrusions of the root HTML element, the body tag. Despite this, the approach used to detect them is very different. The RLG uses a tree-based structure to organise its constituent nodes. Therefore, every element generally has a parent node for every viewport width at which it is visible on the web page. The exception to this rule is when an element has protruded out of the viewport altogether. Here, it is not contained by any element and is therefore “parentless”. Additionally, elements protruding out of the viewport are not classed as siblings of the body element. While in a web page, the body tag has the html tag as a parent, in the RLG the html tag is not modelled as the body element is the root *presentation* element. In other words, the body is highest element in the tree that represents the appearance of the web page. Because of this, body has no parent, meaning there is no common, shared parent.

Figure 5.8 presents a simple example. The `img[2]` element is contained within the body element at viewport widths of 640 pixels or more. At narrower viewport widths however, it protrudes out of the viewport’s right-hand side. Finally, at very narrow viewport widths (less than 480 pixels), a shift to a single-column

**Algorithm 8** Detection of viewport protrusion failures

---

```

1: procedure DETECTVIEWPORTPROTRUSIONFAILURES( $(\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ )
2:   for all  $e \in \mathcal{E}$  where  $e \neq \text{body}$  do
3:      $S \leftarrow \emptyset$ 
4:     for all  $r = (e_1, e_2) \in \mathcal{R}$  where  $e_2 = e$  do
5:       for all  $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$  do
6:         if  $t = \text{pc}$  then
7:            $S \leftarrow S \cup \{(amin, amax, t, P)\}$ 
8:         end if
9:       end for
10:    end for
11:    if  $S \neq \emptyset$  then
12:       $L \leftarrow \text{SORTBYASCENDINGMINIMUMRANGEVALUES}(S)$ 
13:       $gmin \leftarrow w_{min}$ 
14:      while  $\text{HASNEXT}(L)$  do
15:         $(amin, amax, t, P) \leftarrow \text{NEXT}(L)$ 
16:         $gmax \leftarrow amin - 1$ 
17:        if  $gmax < gmin$  then
18:           $VR \leftarrow \text{VISIBLERANGES}(e, (gmin, gmax))$ 
19:          if  $VR \neq \emptyset$  then
20:             $\text{REPORTFAILURE}(\text{viewport-protrusion}, \{e\}, VR)$ 
21:          end if
22:        end if
23:         $gmin \leftarrow amax + 1$ 
24:      end while
25:       $gmax \leftarrow w_{max}$ 
26:       $VR \leftarrow \text{VISIBLERANGES}(e, (gmin, gmax))$ 
27:      if  $VR \neq \emptyset$  then
28:         $\text{REPORTFAILURE}(\text{viewport-protrusion}, \{e\}, VR)$ 
29:      end if
30:    end if
31:  end for
32: end procedure

```

---

layout means the element is no longer protruding. Algorithm 2 shows the approach for detecting viewport protrusion failures. It checks that for every viewport width at which an element is visible on the web page, it is the child of some other element in the RLG.

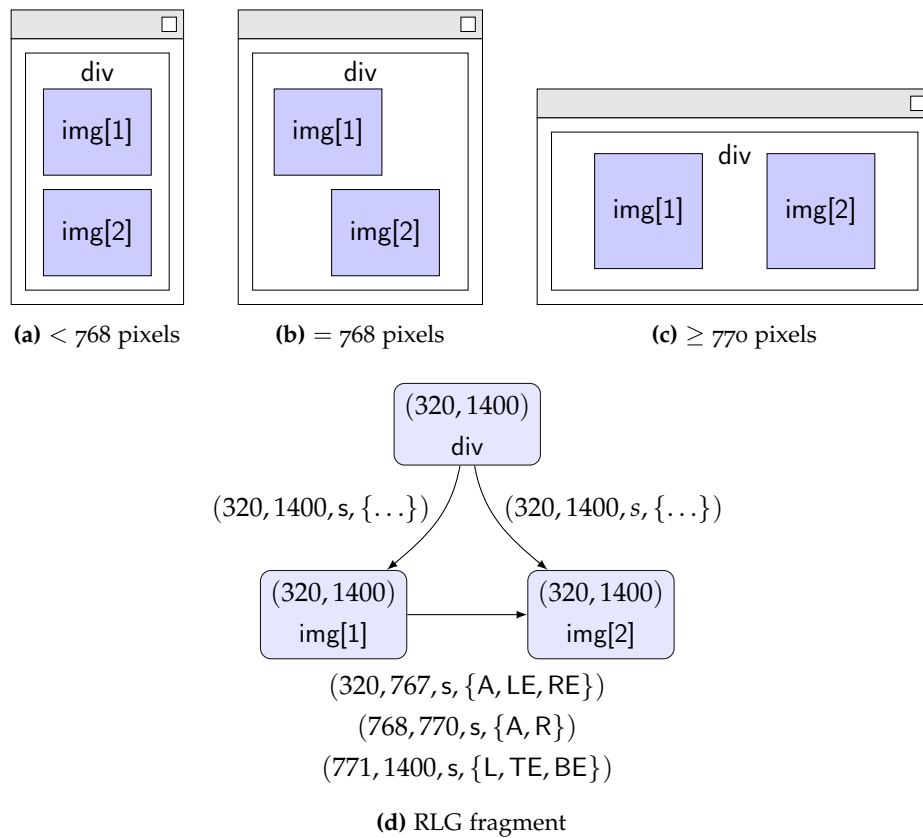
The algorithm begins by iterating through every element in the RLG, except the body element. For each element  $e$ , it finds all the parent-child alignment constraints for which  $e$  is the child, and adds them into a set  $S$  (lines 3-10). In this example, when  $e$  is `img[2]`,  $S$  would contain the constraints  $(320, 480, \text{pc}, \{\dots\})$  and  $(640, 1400, \text{pc}, \{\dots\})$ . Next, the algorithm only investigates further if  $S$  is non-empty (line 11). If an element protrudes outside of the viewport for every single viewport width, the developer is likely holding it offscreen intentionally. In this case, the algorithm should not report any failures. If  $S$  is non-empty, the algorithm sorts its contents into a list  $L$  by the function call on line 12. This

function places them in ascending order based on their lower bound values (i.e.,  $amin$  for an alignment constraint of the form  $(amin, amax, t, P)$ ). This guarantees the algorithm will analyse the constraints in consecutive viewport order. If an element never protrudes out of the viewport, the upper bound of one constraint will be one pixel less than the lower bound of the succeeding one. The key insight of the algorithm is that any “gaps” between the bounds of the sorted alignment constraints — ranges of viewport widths where the element has no parent in the RLG — where the element is visible must indicate the element has protruded out of the viewport.

The main loop of the algorithm attempts to find these “gaps”. It begins by iterating through each alignment constraint in  $L$ . For each constraint, it forms a range  $(gmin, gmax)$  modelling the gaps between the current and previous constraints. Initially, the value of  $gmin$  is set to  $w_{min}$  (line 13). It is then updated to be the upper bound of the previous alignment constraint  $+1$  (line 23). Meanwhile,  $gmax$  is iteratively set to the lower bound of the current constraint  $-1$  (line 16). For instance, when analysing the first alignment constraint from Figure 5.8,  $gmin$  and  $gmax$  would have values of 320 and 319, respectively. Next, the algorithm checks whether the current range represents a gap (i.e.,  $gmax > gmin$ , line 17). If there is no gap, for example if a constraint finishes at 767 and the next begins at 768, then  $gmin$  and  $gmax$  have values of 768 and 767, respectively. In this case, the algorithm moves on to the next constraint. However, if a gap occurs, as is the case with the example, Algorithm 8 calls the function `VISIBLERANGES`. This returns the ranges of viewport widths between  $gmin$  and  $gmax$  at which  $e$  is visible (line 18). When the algorithm analyses the second constraint,  $gmin = 481$  and  $gmax = 639$ , so it calls `VISIBLERANGES`.

If the result of calling `VISIBLERANGES`,  $VR$ , is not empty (line 19), then the algorithm reports a viewport protrusion failure for the ranges returned (line 20). The `img[2]` element is visible for the entire range at which it has no parent. Therefore, the algorithm reports a viewport protrusion failure between 481 pixels and 639 pixels. The final steps (lines 25-29) perform the same checks as the main loop, but between the upper bound of the final constraint and the largest viewport width modelled by the RLG,  $w_{max}$ . This makes sure the element does not protrude out of the viewport again at wide widths. In this case, there exists no gap between the final constraint and  $w_{max}$ , so the algorithm takes no further action.





**Figure 5.9:** Example of a small-range failure, and its corresponding RLG fragment.

### Detecting Small-Range Layouts

To detect small-range layout failures the approach searches through every alignment constraint in the RLG and inspects its lower and upper bounds. Algorithm 9 shows this process in detail. Firstly, the algorithm checks to see if the range of widths falls below a threshold value *thres* (line 4). In the empirical evaluation presented later in this chapter, this value is 5, but the algorithm could use other values. Next, if it finds a small-range constraint, the algorithm investigates whether an alignment constraint exists between the same two nodes but with differing alignment attributes, at both immediately wider and narrower viewport widths (lines 5-6). If it finds such constraints (line 7), then the algorithm deems the small-range layout to be representing a small-range layout and reports it to the developer. Using the example presented in Figure 5.9, line 4 identifies the constraint  $(768, 770, s, \{A, R\})$ . Then, the function calls return the constraints  $(320, 767, s, \{A, LE, RE\})$  and  $(771, 1400, s, \{L, TE, BE\})$ , respectively. As the algorithm found both preceding and succeeding constraints, it reports a small-range layout failure.

**Algorithm 9** Detection of small-range layout failures

---

```

1: procedure EXTRACTSMALLRANGLAYOUTS( $(\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ )
2:   for all  $r = (e_1, e_2) \in \mathcal{R}$  do
3:     for all  $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$  do
4:       if  $amax - amin \leq thres$  then
5:          $existsNarrower \leftarrow EXISTSAT(e_1, e_2, t, amin - 1)$ 
6:          $existsWider \leftarrow EXISTSAT(e_1, e_2, t, amax + 1)$ 
7:         if  $existsNarrower \wedge existsWider$  then
8:            $REPORTFAILURE(\text{small-range-layout}, \{e_1, e_2\}, \{(amin, amax)\})$ 
9:         end if
10:      end if
11:    end for
12:  end for
13: end procedure

```

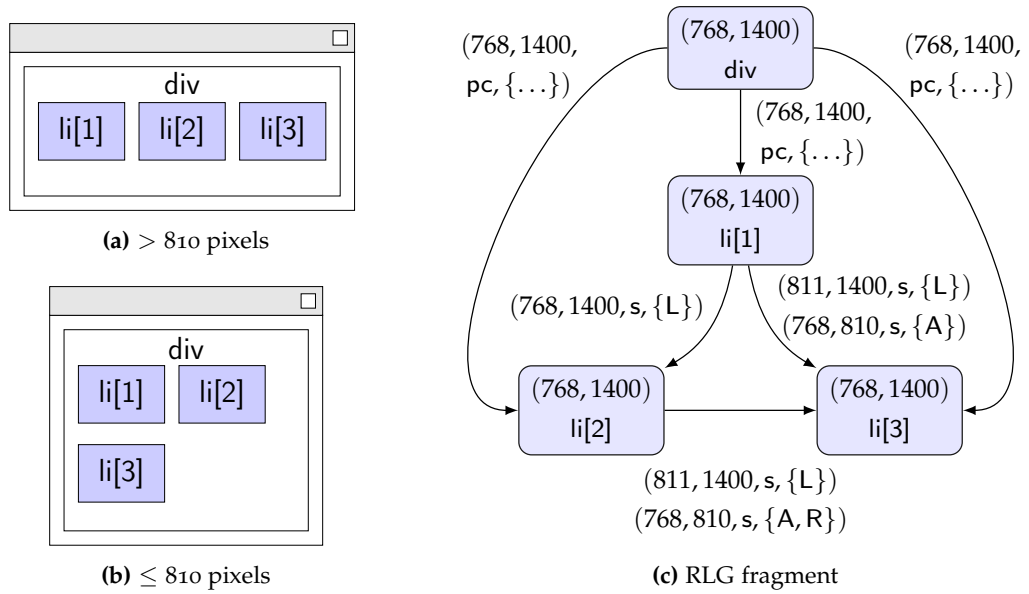
---

**Detecting Wrapping Elements**

To detect wrapping failures, Algorithm 10 analyses the alignment constraints between neighbouring elements. It infers which ones the web page lays out in horizontal rows for different viewport ranges. The insight of the approach is that if an element  $e$  is rendered in a row with its neighbours at one viewport range, but at a narrower viewport width the same row still exists, but no longer contains  $e$ , then a wrapping failure has occurred.

Algorithm 10 analyses each element  $e$  in turn. It begins by obtaining the list of its children  $C$ , by inspecting the alignment constraints for which  $e$  is the parent (lines 3-10). Next, the algorithm finds all of the sibling alignment constraints between the children and adds them into a set,  $S$  (lines 11-16). Using Figure 5.10 as an example, when analysing `div`,  $C$  would contain the three `li` elements. Likewise,  $S$  would contain the five alignment constraints between them. Then, the function on line 17 extracts the lower and upper bounds of these constraints. It then returns a list of viewport ranges  $L$ , sorted into ascending order (line 17). Normally, these ranges simply match the individual bounds of each alignment constraint. However, as elements rearrange dynamically the ranges often intersect, in which case the intersecting ranges are “spliced”. This creates a series of ranges such that each viewport width only falls within a single range. For instance, some of the constraints in the example intersect, so  $L$  would contain two ranges (768 – 810) and (811 – 1400). Therefore, an alignment constraint can intuitively be present in multiple viewport ranges. For instance, the constraint (768, 1400,  $s, \{L\}$ ) would be part of both the (768 – 810) and (811 – 1400) range.

The algorithm then iterates through successive pairs of viewport ranges to detect any wrapping failures present. It achieves this by first determining which elements in  $C$  are children of  $e$  for the two viewport ranges in question (lines



**Figure 5.10:** Example of a wrapping failure, and its corresponding RLG fragment. At narrow viewport widths, elements intended to align next to each other on a single row wrap to produce a second row.

22-23). It then calls the function `GETCHILDRENINROWS` to work out the relative layout of those children in each range (24-25). In this function, if a pair of elements  $e_1, e_2$  have relative alignment attributes L or R (i.e. one is to the left/right of the other) and does not have either the attributes A or B, they are in a row. If the constraint in question did contain A or B, the elements would intuitively *not* be in a row, as  $e_1$  cannot be above/below  $e_2$  if they were horizontally aligned. Finally, if  $IR_c$  contains at least two elements (i.e., a row exists at that viewport range), and a wrapped element  $w$  is not a member of  $IR_c$  but is a member of  $IR_n$ , then the algorithm reports a wrapping failure (lines 26-30).

As a worked example, when considering the range (768 – 810), `GETCHILDRENINROWS` would return  $\{li[1], li[2]\}$  as  $IR_c$ . The constraint  $(768, 1400, s, \{L\})$  clearly indicates that they are in a row, while the constraint  $(768, 810, s, \{A\})$  between  $li[1]$  and  $li[3]$  shows  $li[3]$  is not in that row. Similarly, when considering the wider range, the constraint  $(811, 1400, s, \{L\})$  shows that  $li[3]$  is now in the row, meaning  $IR_n$  is the set  $\{li[1], li[2], li[3]\}$ . As  $IR_c$  and  $IR_n$  show the presence of a row at both viewport ranges, with a single element,  $li[3]$ , not in the row at the narrower viewport width, the algorithm reports a wrapping failure between 768 pixels and 810 pixels.

**Algorithm 10** Detection of wrapping failures

---

```

1: procedure EXTRACTSMALLRANGLAYOUTS( $(\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ )
2:   for all  $e \in \mathcal{E}$  do
3:      $C \leftarrow \emptyset$ 
4:     for all  $r = (e_1, e_2) \in \mathcal{R}$  where  $e_1 = e$  do
5:       for all  $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$  do
6:         if  $t = pc$  then
7:            $C \leftarrow C \cup \{e_2\}$ 
8:         end if
9:       end for
10:    end for
11:     $S \leftarrow \emptyset$ 
12:    for all  $r = (e_1, e_2) \in \mathcal{R}$  where  $e_1 \in C \wedge e_2 \in C$  do
13:      for all  $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$  where  $t = s$  do
14:         $S \leftarrow S \cup (amin, amax, t, P)$ 
15:      end for
16:    end for
17:     $L \leftarrow \text{GETCHILDANGESORTEDBYASCENDINGMINIMUMRANGEVALUES}(S)$ 
18:     $i \leftarrow 0; len \leftarrow \text{LENGTH}(L)$ 
19:    while  $i < len - 1$  do
20:       $(rmin_c, rmax_c) \leftarrow L[i]$ 
21:       $(rmin_n, rmax_n) \leftarrow L[i + 1]$ 
22:       $C_c \leftarrow \text{GETCHILDRENINRANGE}(e, C, (rmin_c, rmax_c))$ 
23:       $C_n \leftarrow \text{GETCHILDRENINRANGE}(e, C, (rmin_n, rmax_n))$ 
24:       $IR_c \leftarrow \text{GETCHILDRENINROWS}(C_c, (rmin_c, rmax_c))$ 
25:       $IR_n \leftarrow \text{GETCHILDRENINROWS}(C_n, (rmin_n, rmax_n))$ 
26:      for all  $c \in C_c$  do
27:        if  $|IR_c| \geq 2 \wedge c \notin IR_c \wedge c \in IR_n$  then
28:          REPORTFAILURE(wrapping,  $\{e, c\}$ ,  $(rmin_c, rmax_c)$ )
29:        end if
30:      end for
31:       $i \leftarrow i + 1$ 
32:    end while
33:  end for
34: end procedure
35:
36: procedure GETCHILDRENINROWS( $C_r, (rmin, rmax)$ )
37:    $IR \leftarrow \emptyset$ 
38:   for all  $r = (e_1, e_2) \in \mathcal{R}$  where  $e_1 \in C_r \wedge e_2 \in C_r$  do
39:     for all  $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$  where  $t = s$  do
40:       if  $amin \leq rmin \wedge amax \geq rmax$  then
41:         if  $L \in P \vee R \in P \wedge A \notin P \wedge B \notin P$  then
42:            $IR \leftarrow IR \cup \{e_1, e_2\}$ 
43:         end if
44:       end if
45:     end for
46:   end for
47:   return  $IR$ 
48: end procedure

```

---

### 5.3 EMPIRICAL EVALUATION

This section evaluates this chapter's approach in terms of both its ability to detect responsive layout failures and its efficiency. It uses a collection of 26 responsively designed production web pages, in order to answer the following three research questions:

**RESEARCH QUESTION ONE:** *How effective are the four RLG analysis algorithms at detecting responsive layout failures?*

**RESEARCH QUESTION TWO:** *How does the proposed approach compare to spot-checking approaches commonly used in real-world development?*

**RESEARCH QUESTION THREE:** *How long does the proposed approach take to run?*

The remainder of this section presents the experimental design employed in this study, then the obtained empirical results and a discussion of the main interesting points.

#### 5.3.1 Experiment Design

##### **Subject Web Pages**

It is vital the experimental results are not adversely biased in any way. Therefore, a random URL generator, [randomusefulwebsites.com](http://randomusefulwebsites.com), was used to select 25 responsively designed web pages. Unfortunately, not all web pages served implemented responsive design. This required a degree of manual analysis to select only those web pages in the scope of the study. For instance, a web page exhibiting no responsive behaviour may exhibit layout issues. However, these failures are not caused by errors in the implementation of a responsive design. To select only those subjects within the scope of the study, the viewport was resized while browsing each web page and the layout behaviour observed. If the web page demonstrated RWD principles (i.e., used grid-based layouts and media queries), then it was saved for use in the study. As mentioned in Chapter 2, many web pages use RWD frameworks, but it is important to stress that simply importing such a framework did not guarantee inclusion in the study. This is because an import does not indicate correct application. This process repeated until 25 web pages had been selected. While a larger pool of subject web pages could potentially have been used, 25 was chosen to obtain a wide enough

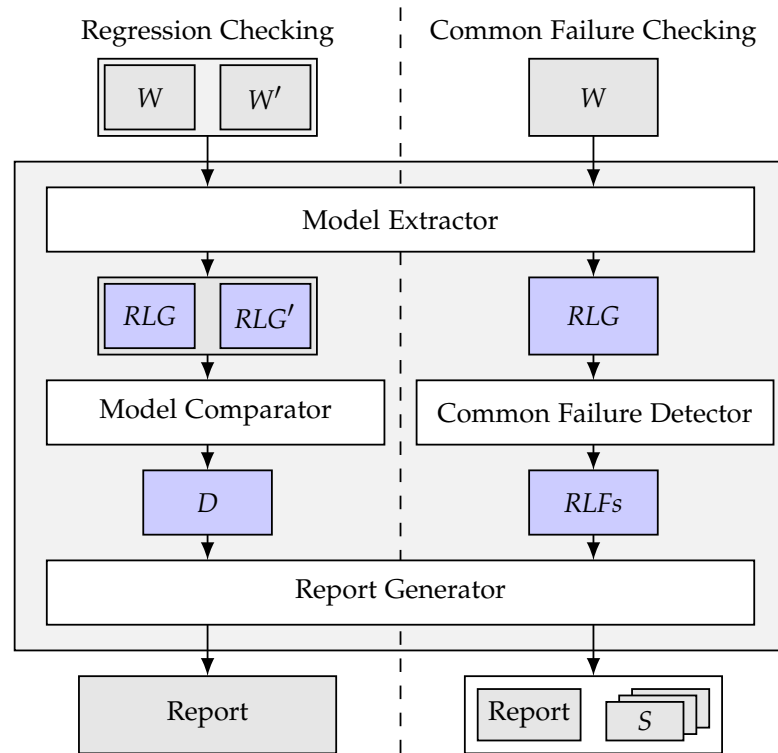
**Table 5.1:** Web Pages Used in the Empirical Study

WEB SITE NAME	URL	# HTML ELEMENTS	# CSS DECLARATIONS
3-Minute Journal	www.3minutejournal.com	79	3354
Accountkiller	www.accountkiller.com/en	343	559
Airbnb	www.airbnb.com	1469	5638
BugMeNot	bugmenot.com	41	237
Cloudconvert	cloudconvert.com	907	2831
ConsumerReports	www.consumerreports.org	1037	6295
CoveredCalendar	www.coveredcalendar.com	147	5131
Days Old	www.daysold.com	65	1033
Dictation	dictation.io	194	166
Duolingo	www.duolingo.com	816	16929
Honey	www.joinhoney.com/install	460	3249
Hotel WiFi Test	www.hotelwifitest.com	358	4258
Mailinator	www.mailinator.com	279	5086
MidwayMeetup	www.midwaymeetup.com	85	2942
Ninite	ninite.com	640	2721
PDFescape	www.pdfescape.com	176	794
PepFeed	www.pepfeed.com	342	4563
Pocket	getpocket.com	663	5203
Rainy Mood	rainymood.com	88	50
RunPee	runpee.com	437	7273
StumbleUpon	www.stumbleupon.com	283	8530
Top Documentary Films	topdocumentaryfilms.com	410	702
Usersearch	usersearch.org	865	1495
What Should I Read Next	www.whatshouldireadnext.com/search	111	852
Will My Phone Work	willmyphonework.net	781	2022
Zero Dollar Movies	zerodollarmovies.com	246	1802

sample of web pages to obtain generalisable results without making the manual effort required to analyse the empirical results unmanageable. A 26th web page, ConsumerReports, was also added as this subject acted as a motivating example in a publication of this work [147]. Table 5.1 details the selected web pages, including their URLs, the number of HTML elements contained within them, and the number of CSS declarations used to implement their responsive layout.

## Methodology

**IMPLEMENTATION:** The RLF detection algorithms were implemented into a new module of REDECHECK. The tool was also modified to allow users to select which “mode” of checking REDECHECK performed. From this point forward, this thesis refers to the approach presented in Chapter 4 as *regression check-*



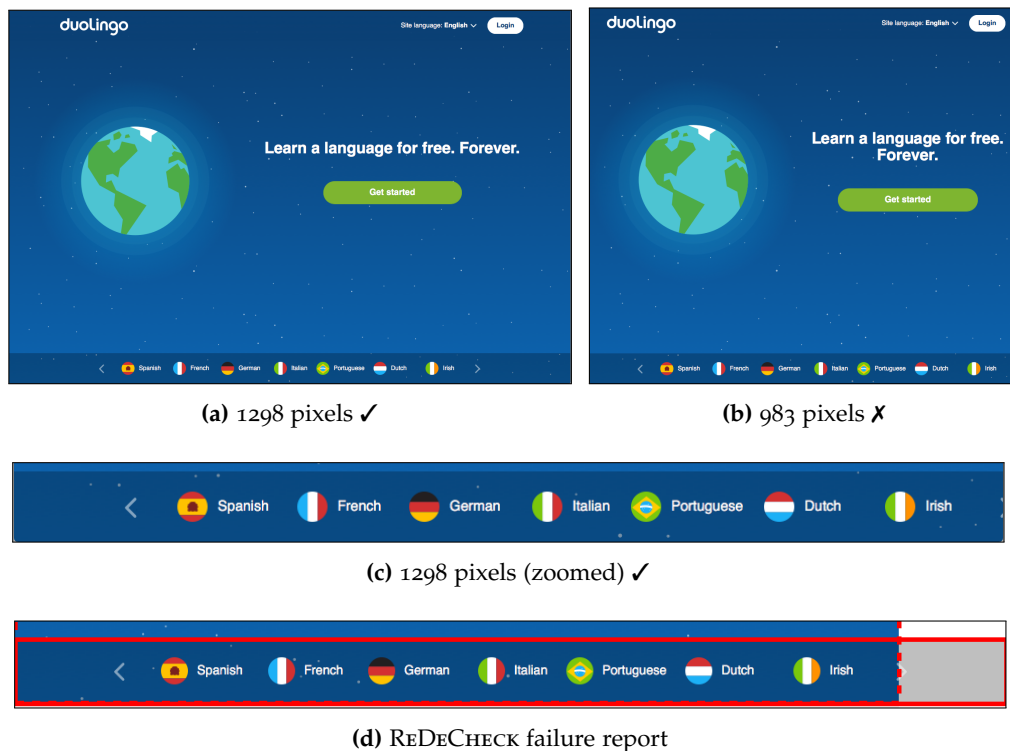
**Figure 5.11:** The high-level structure of the REDECHECK tool. To the left of the dashed vertical line, the modules support regression checking, to the right, common failure checking.

ing and the approach described in this chapter as *common failure detection*. Figure 5.11 illustrates the updated high-level architecture of REDECHECK.

The report generator module again produces a single textual report when in common failure detection mode. However, it also produces a series of annotated screenshots showing each detected RLF with the offending elements highlighted in the web page. This should make it easier for a developer to find and diagnose issues. Figure 5.12 presents an example, where REDECHECK identifies a viewport protrusion in Duolingo and outputs the highlighted screenshot to the developer.

**CONFIGURATION:** REDECHECK was configured to sample the web page under test across a viewport range of 320–1400 pixels. This again ensured the RLG covered devices ranging from small smartphones to large, widescreen desktops. REDECHECK was also set by the configuration parameters to render the web page in the Firefox browser. All the experiments used an iMac with 8GB of RAM and MacOS 10.12 Sierra as the operating system.

**RESEARCH QUESTION ONE:** To answer this first research question, each RLF reported by REDECHECK was analysed and classified as one of three distinct



**Figure 5.12:** An example report screenshot from REDECHECK.

The failure is from *Duolingo*, where a carousel of languages is correctly centered (parts (a) and (c)), before protruding outside the viewport as the width narrows, obscuring the right-hand arrow (parts (b) and (d)). Finally, part (e) shows a report, produced by REDECHECK, highlighting the failure to the developer using dashed and solid red boxes.

types. True positives (TPs) are failures which reveal clearly evident RLFs when a user views the web page at one of the reported “faulty” viewport widths. For instance, the examples presented by Figures 5.1 through 5.5 show examples of TPs detected as part of this study. In contrast, FPs do not reveal failures of any kind, either in the visual appearance of the web page or its underlying DOM structure.

This RQ defines a third category of reported failure, *non-observable issues* (NOIs), which lie somewhere between TPs and FPs. They do not manifest visually in the aesthetics of the web page. Instead they reveal potential issues at the DOM level, detectable through the use of developer tools such as Firebug [115]. For instance, two elements may overlap/collide at the DOM coordinate level. Due to certain styling rules, such as transparent edges (i.e., no visible border) or large amounts of padding, they may present no visible issue to a user when viewing the web page. Given that they do not manifest visually, NOIs do not represent serious problems like those classified as TPs. However, as they can highlight issues which could potentially manifest as TPs at a later time, they are useful to



developers and are therefore different to FPs. They could be considered similar to the output of linting tools [75]. These are software tools that point out source code that might cause potential issues during maintenance or execution on different platforms. NOIs can inform web developers of structural issues or other factors related to CSS code that may negatively affect ease-of-modification or the ways in which different web browsers may render pages.

Unfortunately, classifying presentation and layout failures in web pages is a task that falls on the shoulders of humans (c.f. [7, 33, 105, 123]). Therefore, I performed the initial classification of all the reported failures. I then reviewed those classifications with my supervisor. This mitigated any potential subjectivity stemming from individual perceptions as to what constitutes a true positive, false positive or non-observable issue. To invite discussion on the nature of layout failures in web pages, the reasoning behind the final classifications is available in an online results archive, along with all of the produced failure reports and screenshots [4].

When applied together, the detection algorithms might report a failure more than once for different RLF categories. For example, an element collision may also be a small-range layout. The approach may also report related failures involving common HTML elements that are likely to emanate from a single defect in the web page's source code. To summarize REDECHECK's ability to reveal *distinct* RLFs, therefore the set of TPs for each page were manually analyzed to determine the number of discrete, observable failures evident. Furthermore, multiple failure reports may be produced by REDECHECK for the same viewport range. In practice, a developer would not need to examine each report individually. Instead, they would view the web page within each distinct viewport range to check for RLFs, as multiple failures could be visible in the same viewport range. The results therefore also detail the number of distinct viewport ranges for all of the failure reports produced by REDECHECK for each web page.

**RESEARCH QUESTION TWO:** This experiment first involved selecting a series of spot-checking tools to compare to REDECHECK. Following a Google search for "responsive web testing tool", the top four results were selected. These were KERSLEY'S RWD tool [78], RESPONSINATOR [120], RESPONSIVEDSIGNCHECKER [121], and VIEWPORTRESIZER [139]. The fifth tool selected was the popular integrated "Responsive Design View" utility of the Firefox browser's developer tools [3]. These tools advocate checking a web page's layout at a wide range of viewport widths, corresponding to the viewport width (in either portrait or landscape

orientation) of a device in common use. For instance, Kersley's tool suggests a developer checks their web page at 4 viewport widths in the 320–1400 pixel range. VIEWPORTRESIZER advocates more detailed checking, recommending 12 different widths. In total, the 5 tools suggested checking at 21 different widths. Finally, this question also investigates the performance of a sixth approach, in which 21 viewport widths were randomly selected. This random approach tries to complement the more device-specific widths already chosen.

The spot-checking process itself involved first resizing the Firefox browser to each advocated viewport width. Then, at each width, whether any of the RLFs previously detected by REDECHECK were detectable at that viewport width was recorded. As with RQ1, the initial findings were then checked as a committee to ensure correctness.

**RESEARCH QUESTION THREE:** The time taken for REDECHECK to perform its analysis and produce its report can sometimes be affected by issues such as the browser load-up and interaction time. Therefore, the timing experiment consisted of 30 trials for each of the 26 web pages, to mitigate any bias in the results.

### Threats to Validity

As with all empirical evaluations, any threats to the validity of the obtained results must be considered and mitigated as much as possible by the methodology. This section details the main identified threats and the steps taken to mitigate them.

**GENERALISABILITY OF WEB PAGES:** The choice of test subjects is always a validity threat, as results may not generalise to other web pages. This was mitigated by using a random URL generator, [randomusefulwebsites.com](http://randomusefulwebsites.com). The resulting subject web pages varied drastically in both domain and complexity. For instance, one of the smallest subjects, Days Old, provided a simple calendar function, while large applications such as Duolingo provide a fully-fledged online language learning platform.

**MANUAL CLASSIFICATION:** As with the evaluation in Chapter 4, the manual classification of results presents another threat. However, as no automated approach for the task currently exists, humans must do the task manually. To mitigate the risk of subjectivity affecting the classifications, a committee verified the initial decisions. The individual failure reports and screenshots produced by REDECHECK were also published online, along with the classifications as-

signed to them. This will allow others in the community to inspect them and provide any feedback if they so wish.

**SPOTCHECKING:** Due to the lack of a human study, the spot-checking results presented as part of RQ2 were not obtained by using real-world developers. However, given the viewport widths selected are those advocated by commonly used RWD testing tools, RQ2 should give a good insight into the failure detection capabilities of manual spotchecking in practice. In fact, given that human testers can potentially fail to identify layout issues when manually inspecting web pages, the results in RQ2 are likely to be an upper bound.

**MACHINE RELATED PERFORMANCE BIAS:** Given the time taken for REDECHECK to perform its analysis of a web page can vary due to browser load-up and interaction time, the timing results in RQ3 represent 30 repeat trials on each subject web page.

**BROWSER SPECIFIC FAILURES:** There are often subtle differences in how different browsers render web pages. The diverse body of work targeting cross-browser incompatibilities is clear evidence of this. Therefore, the use of Firefox in the configuration of REDECHECK constitutes another validity threat. However, given Firefox's popularity and widespread use among not only the web development community but the general public as well, the results produced using Firefox should be representative. However, to ensure any failures detected were not specific to Firefox, their presence was manually confirmed on both the latest versions of Safari and Chrome.

**LACK OF COMPARISON TO RELATED APPROACHES:** As discussed in Chapter 2, several approaches to detecting presentational issues have been previously proposed. This study did not compare them to REDECHECK for a variety of reasons. Firstly, the 26 web pages in the study were simply downloaded from the live web. This means they do not come with oracle images or layout specifications, ruling out tools such as WEBSEE and CORNIPICKLE. Furthermore, the detailed intentions of the developers of the web pages are unknown. This makes manually creating such oracles in order to use these tools a threat in itself to the validity of any results stemming from their use. Finally, techniques for cross-browser incompatibility detection are unsuitable as they rely on one browser representing the "correct" layout. In contrast, the proposed approach uses *implicit* oracle information to detect responsive layout failures.

**TOOL IMPLEMENTATION:** Bugs in the implementation of the described approach present a large risk to the validity of any results. Therefore, as with the version

**Table 5.2:** Results for the RLF Detection Approach.

This table records the number of failure reports produced by REDECHECK, categorized as true positives (TP), false positives (FP) or non-observable issues (NOI) for each RLF type. The 5 main column headers represent the 5 failure types: element collision (EC), element protrusion (EP), viewport protrusion (VP), small-range (SR) and wrapping (W). A “-” indicates that REDECHECK produced no reports for a particular page and category. The “Distinct Viewport Ranges” column records the number of distinct viewport ranges that a developer would need to examine to verify all of the failure reports produced for a particular web page. The “Distinct RLFs” column gives the number of TPs (i.e., actual RLFs) detected by REDECHECK that manual analysis subsequently revealed to be distinct in a page’s layout.

WEB PAGE	EC			EP			VP			SR			W			DISTINCT VIEWPORT RANGES	DISTINCT RLFs
	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI		
3-Minute Journal	-	-	1	-	-	2	8	-	-	-	1	-	-	-	-	12	2
Accountkiller	-	-	-	-	-	-	-	-	-	147	5	-	2	-	-	4	3
Airbnb	-	-	1	-	-	4	-	-	4	-	2	-	2	-	-	9	2
BugMeNot	-	-	-	1	-	3	2	-	-	-	-	-	1	-	-	7	4
Cloudconvert	1	-	-	-	-	-	-	-	-	1	-	-	-	-	-	1	1
ConsumerReports	-	-	7	1	-	3	9	-	3	-	1	-	-	-	-	16	4
CoveredCalendar	-	-	-	-	-	-	-	-	3	-	-	-	2	-	-	3	2
Days Old	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	1	0
Dictation	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	1	0
Duolingo	-	-	1	-	-	-	2	-	2	-	1	-	-	2	-	7	1
Honey	-	-	-	-	-	8	-	-	2	-	3	-	-	-	-	8	0
Hotel WiFi Test	-	-	-	-	-	-	1	-	-	-	2	-	-	-	-	3	1
Mailinator	-	-	1	-	-	-	-	-	-	-	2	-	-	-	-	2	0
MidwayMeetup	1	-	-	-	-	1	-	-	1	-	-	-	-	-	-	3	1
Ninite	-	-	-	-	-	-	-	-	-	-	-	-	1	1	-	2	1
PDFescape	-	-	-	1	-	5	1	-	3	-	-	-	-	-	-	8	2
PepFeed	4	-	3	-	-	2	1	-	1	2	14	-	1	-	-	20	6
Pocket	-	-	2	-	-	3	-	-	-	-	3	-	-	-	-	5	0
Rainy Mood	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
RunPee	-	-	-	-	-	-	-	-	-	-	5	-	-	1	-	6	0
StumbleUpon	1	-	-	-	-	-	-	-	-	-	-	-	-	1	-	2	1
Top Documentary Films	-	-	7	-	-	4	-	-	-	-	2	-	-	-	-	10	0
Usersearch	-	-	1	-	-	-	-	-	-	-	-	-	1	-	-	2	1
What Should I Read Next	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	1	0
Will My Phone Work	1	-	-	-	-	1	-	-	-	2	-	-	-	-	-	2	1
Zero Dollar Movies	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-	2	0
<b>Total</b>	8	0	24	3	0	36	24	0	23	152	43	0	10	5	0	137	33

of REDECHECK implemented in Chapter 4, regular automated unit testing of the individual methods and modules of REDECHECK was performed. Detailed manual verification of the results produced using web pages in which I had manually introduced RLFs was also employed, to establish confidence in the approach’s implementation.

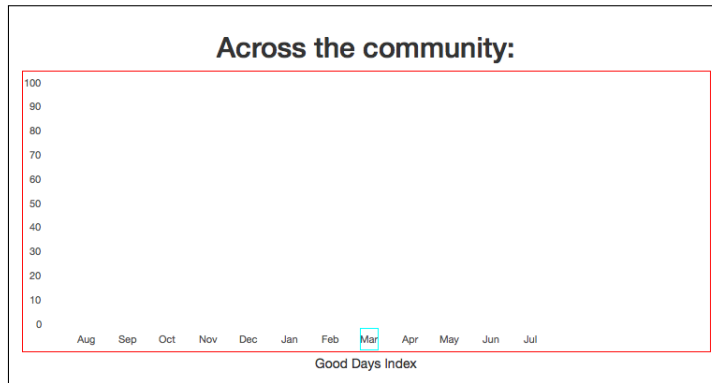
### 5.3.2 Empirical Results

RESEARCH QUESTION ONE: Table 5.2 presents the classification of the failures reported by the approach, broken down by both web page and failure type. As the results show, the approach found true positives (i.e., actual RLFs) in 16 of the 26 web pages in the experiment. Additionally, it found at least one TP of all five failure types. This suggests the types of failures proposed by this chapter are prevalent in real-world web pages and that developers of responsive web pages do indeed struggle to create pages free from layout failures. Interestingly, REDECHECK found RLFs in well-known web pages such as Airbnb, Consumer-Reports and Duolingo. This is particularly compelling for two reasons. Firstly, these web pages are popular and likely receive a large amount of user traffic. Therefore, any RLFs could potentially harm the browsing experience of thousands of end users. This in turn could cause significant detrimental impacts to the respective organisations. Secondly, due to the size of the organisations in question, the web pages are likely to have undergone a thorough in-house testing process. In these cases, the checking has failed to detect the presence of the identified RLFs. This suggests that as well as being prevalent in web pages, RLFs are also often missed by highly skilled web professionals.

In total, the approach produced 197 true positive failure reports. Following manual analysis these reduced down to 33 distinct RLFs, as shown by the final column of Table 5.2. For example, the eight viewport protrusion failures reported for 3-Minute Journal involve neighbouring elements and conflate down to two distinct RLFs. Accountkiller represents a more exaggerated example, in which 147 individual small-range failures all relate to a single distinct RLF. In this instance, the web page contains a large grid of elements with alignment constraints describing the layout between each pair. For a few viewport widths, the approach detects a change in layout where the web page does not lay out the elements in a grid-like manner. This results in a large number of alignment constraints that hold true for just a few viewport widths. This causes the small-range detection algorithm to trigger failure reports for each individual constraint.

The approach also reported some false positives, the majority of which were from the small-range detector. Analysis of these FPs revealed that the most common cause was “coincidental” layout attributes on alignment constraints. Figure 5.13 presents an example in which the offending element “Mar” (outlined in blue) has no CSS attributes dictating its horizontal alignment within

its parent (outlined in red). However, it has coordinates which the approach interprets as being *centre-justified* for a small number of viewport widths.



**Figure 5.13:** A false positive small-range layout.

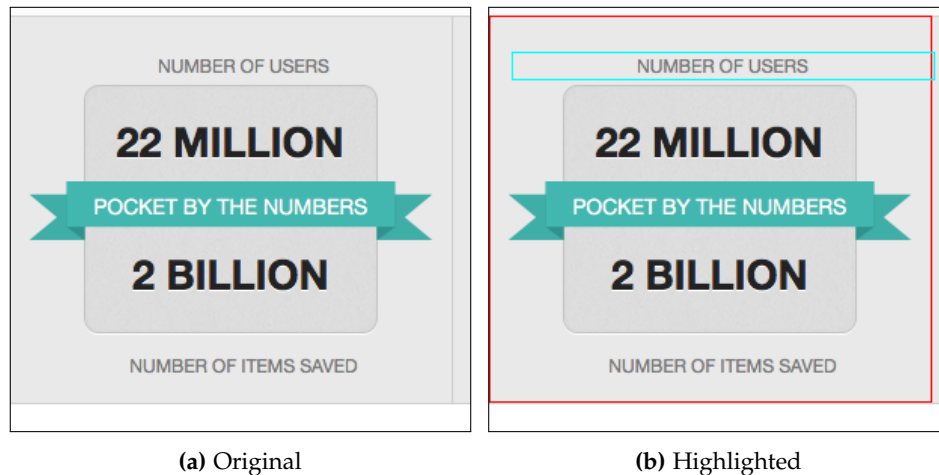
The wrapping detection algorithm also reported a small number of FPs, which were due to one of two things. Firstly, the algorithm could have incorrectly identified a set of elements as being in a row. This caused the wrapping detector to report a failure when a shift in layout caused one of the elements to no longer be a part of the row. Alternatively, a legitimate shift in layout could trigger the reporting of a failure, as illustrated by the example from StumbleUpon in Figure 5.14. Here, the approach has detected one element (highlighted in red) that it thinks has wrapped onto a new row from the other elements (highlighted in blue). However, it is clear that the behaviour shown in Figure 5.14 is intentional and therefore the developer does not need to be alerted to it.



**Figure 5.14:** A false positive wrapping failure.

Additionally, the element collision, element protrusion and viewport protrusion detection algorithms reported several non-observable issues. While these did not manifest visually in the appearance of the web page, several of them represented significant collisions or protrusions at the DOM level. For instance, large amounts of padding applied to elements often caused DOM-level element collisions. Similarly, the CSS property `overflow: hidden` resulted in several DOM-level element protrusions. Following future changes to the web page's CSS, these failures could transform into visually evident TPs, demonstrating the importance of reporting them to the developer. Figure 5.15 presents an example of an NOI from this study. In part (a), the heading element "Number

of Users” seems to be correctly displayed within the main content tile. However, after highlighting the elements in part (b), it becomes clear there is a small underlying issue. The heading is in fact not centre-justified within the tile and instead protrudes slightly.



**Figure 5.15:** A non-observable element protrusion.

With the large number of failure reports produced by REDECHECK for some web pages, one might think that the manual analysis required to inspect each failure report would render REDECHECK very labour-intensive. Instead, a developer can do so with minimal effort even when several failure reports require inspection, as the reports often repeat the same viewport ranges. The “Distinct Viewport Ranges” column in the results table shows this effect. It generally occurs when different algorithms report the same distinct failure or a single algorithm reports related elements as individual failures at the same range of viewport widths. Rather than inspecting each reported failure sequentially, in practice a developer could inspect the textual report to obtain the distinct viewport ranges. They could then view the web page at each of those and confirm any reported failures, as multiple failures could be visible at each viewport width. This essentially means the human effort investment required for using REDECHECK is not correlated with the pure number of reported failures. Instead the number of distinct viewport ranges representing the failures indicates the manual effort required. Table 5.2 shows REDECHECK detected 33 distinct RLFs and reported failures at 137 distinct viewport ranges. Therefore, on average a developer would only need to inspect a web page at an average of 4.2 different viewport widths to observe each distinct RLF. This is particularly compelling as the potential benefits of detecting TP failures — actual RLFs in the web page—

more than outweighs the cost of performing the manual analysis. This is especially true when compared to the alternative of doing everything manually.

Finally, there appears to be no relation between web page complexity and the number of failures contained within it. For instance, while simple web pages such as Rainy Mood contain no failures, other small web pages do. As an example, BugMeNot evinced four distinct RLFs. Similarly, very large, complex web pages such as Airbnb contain relatively few failures in comparison to a medium-complexity web page such as PepFeed (two for Airbnb vs. six for PepFeed). A likely cause for this could be that as the more complex web pages often belong to well-known organisations with large userbases, they would have undergone a significantly more thorough checking process. This would likely have detected many of the RLFs present in the pages, before they went live. In contrast, smaller web pages may have only received a small, perhaps insufficient level of checking. Regardless, as REDECHECK found RLFs in web pages with diverse complexities and domains, identifying them in an accurate manner is clearly an important task.

**Conclusion for RQ1** The results show REDECHECK can accurately detect RLFs of all five types, detecting 33 distinct failures in total. Furthermore, given that 16 of the 26 web pages studied contained at least one RLF, the results also suggest the proposed failure types are prevalent in real-world responsive web pages.

RESEARCH QUESTION TWO: Table 5.3 presents the results of the spotcheck testing procedure, by inspecting the viewport widths advocated by each tool. Firstly, it is important to note the spot-checking process detected no new failures. This suggests there were no false negative results for REDECHECK. This research question can therefore fairly compare the detection results presented in the table against those of REDECHECK in RQ1.

The results show that spot-checking detected substantially fewer RLFs than REDECHECK, with recall scores ranging from 66% for KERSLEY'S and RESPON-SINATOR to 81% for Firefox RWD View. As the widths tested by these approaches correspond to those commonly used by popular devices, this shows REDECHECK detected actual RLFs occurring at viewport widths which could affect the browsing experience of a large number of end users. Another key consideration is that while the spotchecking tools display the web page at the various widths, the human tester is still responsible for performing the manual



**Table 5.3:** Results of the spot-checking process.

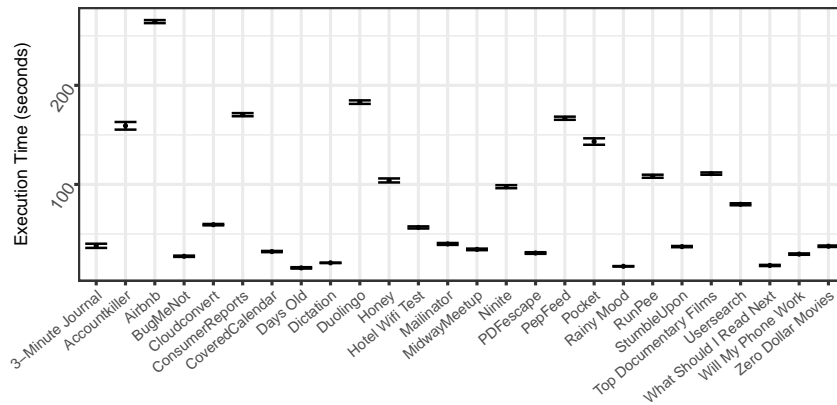
The “RLFs Detected” column denoted how many of the 33 RLFs identified in RQ1 were observable using each approach, while the “Recall” column presents this value as a percentage (eg. 22 out of 33 RLFs = 66% recall).

TOOL/METHOD	RLFs DETECTED	RECALL
Kersley’s	22	66%
Responsinator	22	66%
Responsive Design Checker	23	69%
Viewport Resizer	26	78%
Firefox Responsive Design View	27	81%
Random	24	72%
Detected using at least one tool/method	28	85%

inspection to detect the failures. In contrast, REDECHECK reduces this manual effort considerably. This is especially true for tools that advocate checking a web page’s layout at a large number of viewport widths. For instance, while KERSLEY’S tool only tests at 4 viewport widths, VIEWPORTRESIZER advocates testing at 12 viewport widths. This makes its usage potentially very labour-intensive. Finally, even if a human used all five tools in combination with random checking, the results show that this approach would still miss five of the RLFs detected by REDECHECK.

**Conclusion for RQ2** The results show spot-checking detects fewer failures than REDECHECK. Across the different tools, between 19 and 34% of failures went undetected. This highlights the problems with using spot-checking as failures can often manifest at widths not selected for checking. This result also provides empirical support for the automated approach presented in this chapter.

RESEARCH QUESTION THREE: Figure 5.16 presents the median execution times recorded for REDECHECK across the 30 trials on the 26 subject web pages used in this study. REDECHECK analysed all but one subject within a median of around three minutes. Over half of the subjects — 15 in fact — required less than a minute. This result shows REDECHECK can accurately detect RLFs quickly enough to not be detrimental to a web developer’s workflow, as in software development, large unit test suites can take similar amounts of time to run. Airbnb required the most time at around 4.5 minutes. However, as shown by Table 4.1, it is the most complex web page in the study in terms of the number of



**Figure 5.16:** Execution times for REDECHECK

When not obscured due to a small inter-quartile range (IQR), a small circle denotes the median of the timings across the 30 trials and the upper and lower hinges of the error bars, respectively, designate the median value added to and subtracted from the IQR of the executing timing data.

elements. Therefore it is likely to require more time as the size of the extracted RLG will be much larger and the analysis will take longer. Despite this, the benefits of detecting RLFs, such as the two wrapping failures detected in Airbnb by REDECHECK, should more than outweigh the time investment required to use the approach. The figure shows that increased execution time is the result of either a) high web page complexity (e.g. Airbnb), b) large number of reported failures (e.g. Accountkiller and PepFeed) or c) a combination of the two (e.g. ConsumerReports).

**Conclusion for RQ3** The graph shows the approach can detect RLFs quickly enough to not be disruptive to a developer. REDECHECK processed all but one subject in less than 3 minutes, meaning in a real development scenario, developers can quickly obtain useful feedback on their responsive web pages.

The results of this evaluation demonstrate the potential of the proposed approach for identifying RLFs without an oracle. The next section describes a series of refinements aiming to make the approach more accurate and precise.

#### 5.4 REFINING THE RLF DETECTION APPROACH

The main shortcoming of the original approach proposed in this chapter is the “noisiness” of the small-range layout failure detector. The empirical evaluation found that false positive results were produced by this detector for 13

of the 26 subject web pages, with a total of 43 false positives. For one extreme case, PepFeed, the detector reported 14 false positive results, which in practice would mean a developer analysing the results would likely waste a lot of effort on these failures that could be better spent debugging other true positives or non-observable issues. Therefore, this section seeks to refine this detection algorithm so that it demonstrates improved precision with comparable recall to the original algorithm, i.e., report fewer false positives while reporting a similar (ideally the same) number of true positives. While this chapter could also have addressed the small number of false positive results reported by the wrapping element detector, the fairly high level of precision exhibited by the current algorithm means that modifying the algorithm to filter out the edge cases that caused the false positives would likely cause “overfitting” and potentially make the algorithm less generalisable to other web pages.

As the first set of empirical results show, the element collision, element protrusion and viewport protrusion detection algorithms produce a number of non-observable issue reports — 83 in fact. While many of these do not represent potential underlying issues the developer should be aware of, some of them do, which is how the study justified presenting them to the developer as a category separate to false positives. To increase the benefit of these reports to the developer, one option would be to develop an approach that can determine how likely a non-observable issue is to represent an important underlying issue, as these could be reported to the developer before those that are less likely. However, this option is out of scope of this thesis and is instead listed as a potential direction of future work, described in more detail in Chapter 8.

#### 5.4.1 *RLG Definition Modifications*

When analysing the false positive reports generated by the small-range failure detector, there was one characteristic shared by a large proportion of the failures. The attributes that changed on the small-range alignment constraint were often those that a developer often does not explicitly encode in the CSS of a web page. The vertical alignment of a child element within its parent is probably the most common example of this. Developers frequently program an element to be left, centre or right justified to make sure collections of elements are laid out correctly, but given the constantly changing viewport widths and the dynamic readjustment that accompanies it, vertical alignment is less of a priority.

**Algorithm 11** Detection of small-range layout failures

---

```

1: procedure EXTRACTSMALLRANGLAYOUTS( $(\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ )
2:   for all  $r = (e_1, e_2) \in \mathcal{R}$  do
3:     for all  $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$  do
4:       if  $amax - amin \leq thres$  then
5:          $(\dots, P_{prev}) \leftarrow \text{EXISTSAT}(e_1, e_2, t, amin - 1)$ 
6:          $(\dots, P_{next}) \leftarrow \text{EXISTSAT}(e_1, e_2, t, amax + 1)$ 
7:         if  $(\dots, P_{prev}) \neq \perp \wedge (\dots, P_{next}) \neq \perp$  then
8:           if  $|(P_{prev} \setminus P) + (P \setminus P_{prev})| \geq 2 \wedge |(P_{next} \setminus P) + (P \setminus P_{next})| \geq 2$  then
9:             REPORTFAILURE(small-range-layout,  $\{e_1, e_2\}, \{(amin, amax)\}$ )
10:          end if
11:        end if
12:      end if
13:    end for
14:  end for
15: end procedure

```

---

Because of this, many of the vertical alignment attributes extracted and assigned to constraints in the RLG are coincidental rather than intended by the developer, making it very unlikely that any small-range layout where such an attribute is changed is an actual RLF. In an attempt to exploit this insight and improve accuracy, the RLF detection approach in this section uses a modified RLG in which no vertical alignment in parent-child relationships is modelled, leaving only the horizontal alignment attributes.

#### 5.4.2 Improving the Small-Range Detection Algorithm

The modifications made to the small-range layout detection algorithm first presented in Algorithm 9 are fairly small, but should work in conjunction with the RLG definition modifications described by the previous section to produce a significant improvement in the accuracy of the approach. Algorithm 11 illustrates the new version of the approach.

The key change is the additional *if* statement on line 8, which inspects the three identified alignment constraints, rather than simply reporting a small-range layout failure if constraints either side of the small-range one are found. It compares the attributes of the preceding and succeeding constraints to the small-range constraint found, in order to determine whether the layout change described by the constraints is “significant”, or just a coincidental attribute labelling that the web developer need not worry about. This is achieved using the *set difference* operator to calculate the number of attributes that are different between the pairs of attribute sets. If both comparisons show at least two at-

**Table 5.4:** Additional Web Pages added to the Empirical Study

WEB SITE NAME	URL	# HTML ELEMENTS	# CSS DECLARATIONS
Eat This Much	<a href="https://www.eatthismuch.com/">https://www.eatthismuch.com/</a>	806	7218
Forvo	<a href="https://forvo.com/">https://forvo.com/</a>	583	14438
Google Maps SVP	<a href="http://www.brianfolts.com/driver/">http://www.brianfolts.com/driver/</a>	267	2832
Hours Of	<a href="http://www.hoursof.com/">http://www.hoursof.com/</a>	1257	2601
Khan Academy	<a href="https://www.khanacademy.org/">https://www.khanacademy.org/</a>	847	3076
Memrise	<a href="https://www.memrise.com/">https://www.memrise.com/</a>	268	2238
Retail Me Not	<a href="https://www.retailmenot.com/">https://www.retailmenot.com/</a>	1335	1543
Similar Sites	<a href="https://www.similarsites.com/">https://www.similarsites.com/</a>	477	5518
Startup Stash	<a href="http://startupstash.com/">http://startupstash.com/</a>	730	10823
Tiiime	<a href="http://tii.me/">http://tii.me/</a>	79	535

tributes different, then the algorithm deems the layout change significant and reports the small-range failure in the same way as before. However, say, for instance, a single attribute had “toggled” in the small-range constraint (attribute not present before and after, but present for a tiny range of viewport widths), the updated version of the algorithm would no longer report the failure, making the overall approach far more accurate and more usable for end users.

### 5.4.3 Empirical Evaluation

**SUBJECTS:** To evaluate the effects of the modifications, this study used two collections of responsive web pages. The first of these is the collection of 26 subjects obtained from [randomusefulwebsites.com](http://randomusefulwebsites.com) earlier in this chapter. The second contains an additional 10 web pages collected during the initial subject gathering phase but not selected for use in the initial evaluation. These were used to avoid the approach being overfitted to the initial pool of web pages. Table 5.4 presents the details of these additional web pages.

**RESULTS:** Table 5.5 presents the classification of the failures reported by the approach, broken down by both web page and failure type. The first key observation is that the modifications to the detection approach had a profound impact on the accuracy of the small-range detector, reducing the number of false-positive results from 43 to 4. Furthermore, the approach achieved this with no loss of recall, as all the true positive small-range RLFs found by the original version of the approach are also detected by the new version, albeit with some individual true positives no longer reported.

**Table 5.5:** Failure Detection Results following the modifications to the approach.

WEB PAGE	EC			EP			VP			SR			W			DISTINCT VIEWPORT RANGES	DISTINCT RLFs
	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI		
3-Minute Journal	-	-	1	-	-	2	8	-	-	-	-	-	-	-	-	11	2
Accountkiller	-	-	-	-	-	-	-	-	-	119	-	-	2	-	-	3	3
Airbnb	-	-	1	-	-	3	-	-	4	-	-	-	2	-	-	6	2
BugMeNot	-	-	-	1	-	3	2	-	-	-	-	-	1	-	-	7	4
Cloudconvert	1	-	-	-	-	-	-	-	-	1	-	-	-	-	-	1	1
ConsumerReports	-	-	7	1	-	3	9	-	3	-	-	-	-	-	-	15	4
CoveredCalendar	-	-	-	-	-	-	-	-	3	-	-	-	2	-	-	3	2
Days Old	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	1	0
Dictation	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	1	0
Duolingo	-	-	1	-	-	-	2	-	2	-	-	-	-	2	-	6	1
Honey	-	-	-	-	-	8	-	-	2	-	-	-	-	-	-	9	0
Hotel WiFi Test	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	1	1
Mailinator	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	1	0
MidwayMeetup	1	-	-	-	-	1	-	-	1	-	-	-	-	-	-	3	1
Ninite	-	-	-	-	-	-	-	-	-	-	-	-	1	1	-	2	1
PDFescape	-	-	-	1	-	5	1	-	3	-	-	-	-	-	-	8	2
PepFeed	4	-	3	-	-	2	1	-	1	1	1	-	1	-	-	12	6
Pocket	-	-	2	-	-	3	-	-	-	-	-	-	-	-	-	3	0
Rainy Mood	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
RunPee	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1	0
StumbleUpon	1	-	-	-	-	-	-	-	-	-	-	-	-	1	-	2	1
Top Documentary Films	-	-	7	-	-	4	-	-	-	-	1	-	-	-	-	9	0
Usersearch	-	-	1	-	-	1	-	-	-	-	-	-	1	-	-	3	1
What Should I Read Next	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	1	0
Will My Phone Work	1	-	-	-	-	1	-	-	-	-	-	-	-	-	-	2	1
Zero Dollar Movies	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
TOTAL																111	33

Given there were no explicit changes to any of the other detection algorithms, the remainder of the results are also identical to those presented in the previous chapter. There was no reduction in recall for any of the failure types, but the changes to the RLG definition appears to have caused a couple of additional NOI results to have been reported by the element protrusion algorithm. Given the significant reduction in false positives achieved, these extra reports do not cause any significant increase in the manual effort required by a user of REDECHECK. In fact, a developer would now need to inspect at an average of just 3.4 distinct viewport ranges to observe each RLF.

Table 5.6 shows the RLF detection results for the additional ten web pages. It shows that 7 of the 10 web pages contained RLFs identified by REDECHECK. Furthermore, the results show that failures of all five types were found in this secondary pool of subject web pages. This provides further empirical support not only for the approach used by REDECHECK to detect RLFs, but also the

**Table 5.6:** Failure Detection Results on the additional pool of subjects.

WEB PAGE	EC			EP			VP			SR			W			DISTINCT VIEWPORT RANGES	DISTINCT RLFs
	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI		
Eat This Much	-	-	5	1	1	4	1	-	1	-	-	-	-	1	-	10	2
Forvo	-	-	-	-	-	3	-	-	-	11	-	-	4	-	-	8	5
Google Maps SVP	-	-	-	1	-	1	-	-	-	-	-	-	-	-	-	2	1
Hours Of	-	-	-	-	-	1	-	-	1	-	-	-	-	-	-	2	0
Khan Academy	1	-	-	-	1	2	-	-	2	-	-	-	-	-	-	6	1
Memrise	-	1	-	1	1	1	-	2	2	-	-	-	-	1	-	6	1
Retail Me Not	2	-	-	-	-	30	-	-	-	-	-	-	-	4	-	11	2
Similar Sites	-	-	-	-	-	1	3	-	2	-	-	-	-	-	-	3	3
Startup Stash	1	-	1	1	-	14	-	-	1	-	-	-	1	-	-	6	2
Tiiime	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	1	0
TOTAL	4	1	6	4	3	58	3	2	9	11	0	0	5	6	0	55	17

real-world need for such a tool, as developers clearly struggle to implement responsive web sites that do not suffer from aesthetic issues of some kind.

In total, 27 true positive failures were reported by the approach. Following the previously outlined manual analysis procedure, these failures conflated down to 17 distinct RLFs, as shown by the final column of Table 5.6. For instance, three viewport protrusions reported for Similar Sites all occur at similar viewport widths and conflate to a single distinct RLF, while the 11 small-range failures reported for Forvo also stem from a single distinct RLF, in a similar manner to the Accountkiller failure described in Chapter 5.

In contrast to the rather high proportion of false positives reported by the initial version of the approach, only twelve false positives were reported by the updated version of the approach on the new collection of web pages, averaging to just over one FP per web page. Crucially, not a single FP was reported by the small-range detection algorithm, suggesting the modifications presented in this chapter are generalisable to other web pages rather than being “overfitted” to the initial pool of 26 web pages. The majority of FPs were produced by the wrapping detector and were caused by genuine layout shifts being interpreted as failures by the approach; improving this algorithm is a potential avenue of future work. Eat This Much reported an element protrusion for an element that the approach determined was protruding outside of its parent, when actually it was simply overlapping with a “Start a help chat” tab that was intentionally rendered at the bottom of the viewport window. Khan Academy evinced a viewport protrusion where some content was indeed obscured, suggesting the failure was a true positive, but the consensus among the committee that performed the manual classification was that it was an intended developer effect

and therefore a false positive. Finally, and perhaps most interestingly, Memrise reported failures involving an animated cloud that moved about in the banner of the web page as the user browsed. This caused problems for the approach, as the element's location at one viewport width when its layout was extracted was drastically different to its location at a subsequent viewport width. Future development of the approach could potentially investigate methods of identifying similar elements and dealing with them appropriately.

As with the previous iteration of the approach, there were some non-observable issues reported by the element collision, element protrusion and viewport protrusion algorithms, with a couple of subject web pages reporting quite a large number. For instance, in Retail Me Not there were 30 NOIs reported. However, these were caused by a recurring underlying issue in the CSS of the web page that caused very similar issues to be reported for many elements. However, given the number of failures reported, a human using REDECHECK may assume that the web page contains a large amount of actual RLFs, so future work could look at either automatically classifying NOIs as such, or removing them from the report completely if the approach does not think the developer needs to know about them.

In this experiment, 17 distinct RLFs were identified while failures were reported at 55 distinct viewport ranges. Therefore, a user would have to inspect on average 3.2 distinct viewport widths to observe each of the RLFs found. This is slightly lower than the mean value of 3.4 observed for the initial set of web pages, providing further empirical support for the expected real-world usability of the approach.

**Conclusion** The results show the modifications to REDECHECK produce a substantial increase in accuracy with no loss of recall. The results also show that responsive layout failures were prevalent again in the second set of responsive web pages. They showed REDECHECK was capable of accurately detecting them, with minimal false positives, especially from the small-range detector. However, many NOIs were reported for certain web pages, which reduce the overall usability of the approach.



## 5.5 CONCLUDING REMARKS

This chapter began by introducing five distinct types of responsive layout failure (RLF) that negatively impact the aesthetics of modern web pages. These were element collision, element protrusion, viewport protrusion, small-range layouts and wrapping elements. The chapter then went on to present an automated approach for detecting these RLFs. This approach used four RLG analysis algorithms which required no oracles, instead leveraging implicit oracle information and searching for patterns representing the failures. I implemented the approach as a new module of the `REDECHECK` tool, which reports any detected failures to the developer via a textual report and annotated screenshots.

This chapter then evaluated the approach using 26 randomly selected responsive web pages of varying complexity from a wide range of domains. The approach was able to detect a large number of actual RLFs. 16 of the 26 web pages studied contained at least one failure, with a total of 33 distinct failures. While the approach did report some false positives and “non-observable issues”, the overall effort required to use it was easily outweighed by the benefits of finding potentially damaging RLFs. The results also showed that the approach outperforms a selection of popular spot-checking techniques. Timing experiments found that the approach operates in a relatively short amount of time. This makes it feasible for web developers to easily integrate it into their programming toolbox.

Following on from this evaluation, this chapter presented a series of changes to the RLF detection approach that aimed to make the approach more precise. The empirical results showed that the modifications had a positive impact on the precision of the tool, producing fewer false positives while detecting the same number of RLFs as before. They also found RLFs to be prevalent in an additional 10 web pages not seen before, with an extra 15 RLFs identified by `REDECHECK`.



---

## GROUPING RELATED FAILURES TOGETHER

---

The previous chapter demonstrated how five distinct categories of responsive layout failure (RLF) can be detected using the RLG of the web page under test, leveraging implicit oracles rather than relying on an explicit one. However, with the proposed approach, the process of determining which reported RLFs were related to each other, i.e., stemming from the likely same root cause in the underlying code, was a purely manual process.

This chapter addresses this problem by presenting an automated approach that uses three metrics of similarity to group failures together to make it easier for developers to diagnose problems in the web page.

The key contributions of this chapter are:

1. An algorithm that groups related responsive layout failures together.
2. An empirical evaluation on a large collection of real-world responsive web pages, showing the effectiveness of the approach and that humans generally agree with the reported failures and the groupings produced.

### 6.1 GROUPING INDIVIDUAL REPORTS INTO DISTINCT RLFs

In Chapter 5's empirical evaluation, the true positive failure reports were manually investigated in conjunction with my supervisor and grouped into related failures to ascertain the number of *distinct RLFs* evident in each web page. However, a developer is likely to find it more useful if this process is automated, as they can simply investigate each distinct RLF and all of its constituent individual failures in order to diagnose and fix any underlying layout faults, rather than expending effort manually grouping related failures.

**Algorithm 12** Grouping Reports Together

---

```

1: procedure GROUPFAILURES( $F$ )
2:    $\mathcal{G} \leftarrow \{\}$ 
3:   for  $f \in F$  do
4:      $highestSimilarity \leftarrow 0$ 
5:      $closest \leftarrow null$ 
6:     for  $g \in \mathcal{G}$  do
7:        $similarity \leftarrow \text{CALCULATESIMILARITY}(f, g)$ 
8:       if  $similarity > highestSimilarity \wedge similarity > t$  then
9:          $closest \leftarrow g$ 
10:         $highestSimilarity \leftarrow similarity$ 
11:       end if
12:     end for
13:     if  $closest \neq null$  then
14:        $closest \leftarrow closest \cup \{f\}$ 
15:     else
16:        $\mathcal{G} \leftarrow \mathcal{G} \cup \{f\}$ 
17:     end if
18:   end for
19:   return  $\mathcal{G}$ 
20: end procedure

```

---

6.1.1 *Grouping Approach*

Algorithm 12 presents the approach for automatically grouping together related failures. After initialising the set of grouped failures  $\mathcal{G}$  (line 2), the algorithm iterates through each detected RLF  $f$  and identifies the currently identified group of failures closest to it. To do this, the algorithm computes the similarity (line 7) between  $f$  and each group of failures,  $g$ , currently identified by the grouping process. If the similarity is greater than the current highest similarity observed and above a similarity threshold  $t$  (line 8), then the values of both the closest group and the highest similarity are updated (lines 9 and 10). If a sufficiently similar group is identified in the previous step,  $f$  is added to the relevant group,  $closest$  (line 14). However, if a group is not found,  $f$  is not added and instead a new group containing just  $f$  is created and added to  $\mathcal{G}$  (line 16). Once all the failures have been grouped, the algorithm returns the final grouping  $\mathcal{G}$  to the developer.

6.1.2 *Computing Similarity*

The process for computing the similarity between an individual failure and a collection of already grouped failures is shown by Algorithm 13. At a high level,

**Algorithm 13** Calculating Similarity Between Failure and Group

---

```

1: procedure CALCULATESIMILARITY( $(e, b, t), g$ )
2:    $total \leftarrow 0$ 
3:   for  $(e_g, b_g, t_g) \in g$  do
4:      $boundsSim \leftarrow \text{GETBOUNDSIMILARITY}(b, b_g)$ 
5:      $typeSim \leftarrow \text{GETTYPESIMILARITY}(t, t_g)$ 
6:      $elementSim \leftarrow \text{GETELEMENTSIMILARITY}(e, e_g)$ 
7:      $total \leftarrow total + boundsSim + typeSim + elementSim$ 
8:   end for
9:   return  $total / |g|$ 
10: end procedure

```

---

Algorithm 13 iterates through each failure in the group  $g$  and computes three measures of similarity before incrementing a similarity total. These measures were chosen as they represent the individual features of the RLF introduced in the previous chapter and closely mirror the thought processes used to manually group RLFs together. The measures used are as follows:

- *Bounds* - Two RLFs that manifest at similar ranges of viewport widths are more likely to be related. For instance, a failure occurring between 320 pixels and 400 pixels is more than to be related to one occurring between 350 pixels and 420 pixels than one that is present between the viewport widths of 1200 pixels and 1400 pixels.
- *Type* - Two RLFs that are of the same type are more likely to be related, i.e., two element collisions are more likely to be related than a viewport protrusion and a wrapping failure.
- *Element* - Two RLFs involving elements close to each other on the web page are more likely to be related. For example, an RLF in the web page's header is not very likely to be related to an RLF occurring in the web page's footer.

Finally, it normalises the total by dividing it by the number of failures contained within  $g$ , essentially computing the average similarity between the failure in question and the individual failures of  $g$ . This ensures groups of different sizes can be compared to the individual failure in a consistent and fair manner. This section now goes on to describe how the three separate similarity measures are computed.

**Viewport Similarity**

The first metric used by Algorithm 13 is the similarity in the range of viewport widths at which a pair of failures are observable. Algorithm 14 shows how the

**Algorithm 14** Calculating Viewport Similarity Between Two Failures

---

```

1: procedure GETBOUNDSSIMILARITY( $(l_1, u_1), (l_2, u_2)$ )
2:   if  $l_1 = l_2 \wedge u_1 = u_2$  then
3:     return 1
4:   else if  $l_1 = l_2 \vee u_1 = u_2$  then
5:     return 0.4
6:   else if  $l_2 < u_1 < u_2 \vee l_1 < u_2 < u_1$  then
7:     return 0.2
8:   else
9:     return 0
10:  end if
11: end procedure

```

---

various levels of similarity are scored. Firstly, if the two failures have identical bounds, then the algorithm assigns a similarity score of 1. Next, if either the lower or upper bounds are identical, then a score of 0.4 is given by the algorithm. Penultimately, if the two viewport ranges intersect, the algorithm gives a score of 0.2. Finally, if none of the previous conditions are satisfied then the algorithm simply assigns a value of 0.

**Error Type Similarity**

When comparing two failures, the type of failure can often provide insight as to whether the failures are related. Therefore, Table 6.1 shows how the various combinations of failures types are ranked by the similarity algorithm.

	EC	EP	VP	SR	W
EC	1	0.3	0	0.4	0
EP	-	1	0.5	0.4	0
VP	-	-	1	0.4	0
SR	-	-	-	1	0.4
W	-	-	-	-	1

**Table 6.1:** Failure Type Similarity

In this table, acronyms are used for the column and row headings. EC represents element collision, EP is element protrusion, VP is viewport protrusion, SR is small-range and finally, W represents wrapping elements.

Intuitively, if the two failures have the same type, then they have the highest probability of stemming from the same root cause and therefore are assigned a similarity score of 1. Due to the similarities between how element collisions and element protrusions manifest, the combination of the two has a score of 0.3. However, this is not a frequent occurrence, hence the low similarity measure

**Algorithm 15** Calculating Similarity Between Elements

---

```

1: procedure CALCULATEELEMENTSIMILARITY( $e_f, e_g$ )
2:    $total \leftarrow 0$ 
3:   for  $n_f \in e_f$  do
4:     for  $n_g \in e_g$  do
5:        $editDistance \leftarrow \text{GETEDITDISTANCE}(n_f, n_g)$ 
6:        $total \leftarrow total + editDistance$ 
7:     end for
8:   end for
9:   return  $total / (|e_f| * |e_g|)$ 
10: end procedure

```

---

assigned. Element protrusions can potentially progress to being viewport protrusions. Through observations of examples, this scenario is more likely than the previous combination of element collision and protrusion, so this combination is given a higher similarity measure of 0.5. Finally, as any element collision, element protrusion, viewport protrusion or wrapping failure can manifest at a small range of viewport widths, any combination of a small-range with another failure type is given a score of 0.4. These values were developed through trial and error on test web pages, so it is possible that different values will lead to significantly different groupings.

**Element Similarity**

The final similarity metric used by the grouping algorithm is element similarity, which represents the idea that faulty elements that are closer to each other on the web page are more likely to belong to the same distinct RLF than two elements that are in completely different parts of the page. Algorithm 15 presents the approach used by the grouping technique for comparing the faulty elements of two failures (as called by line 6 of Algorithm 13).

Algorithm 15 iterates through the faulty elements from both failures passed as input, which results in the similarity for every possible combination of elements being computed. This calculation, performed by the GETEDITDISTANCE function on line 5, compares the XPath expressions used to identify the two elements using the popular Levenshtein distance metric [85]. This calculates the number of string operations — character additions, deletions and substitutions — required to convert one XPath to the other. For instance, `body/div/li[1]` and `body/div/li[2]` have a Levenshtein distance of 1, as only a single substitution is required to convert one into the other. To account for XPath expressions of different lengths, the function returns a percentage rather than a discrete integer value, as intuitively two XPath expressions of length 50 with an edit distance of 3 are more similar than a pair of length 10 with the same edit dis-

tance. The algorithm tracks the sum of these similarities in a variable, *total*, which is normalised in the final step (line 9) by dividing it by the number of element comparisons made i.e., the product of the number of elements in the two failures.

## 6.2 EMPIRICAL EVALUATION

To evaluate the distinct RLF grouping technique presented in this chapter, this section applies it to a collection of responsive web pages to answer the following two research questions.

RESEARCH QUESTION ONE: *How effective is the technique for grouping related failures together?*

RESEARCH QUESTION TWO: *Do human users agree with the failures reported and the distinct RLF groupings produced by the approach?*

### 6.2.1 Experiment Design

#### Subject Web Pages

The research questions in this study used the same two pools of subject web pages as Chapter 5. However, it is important to note that the grouping technique was developed **before** the final experiment in the previous chapter that used the extra ten web pages. As such, these additional web pages did not provide insight into the best way to automatically group the failures, unlike the original 26 subjects. Therefore, the grouping results presented later are not at risk of “overfitting”.

#### Methodology

IMPLEMENTATION: The technique for grouping individual reported failures into distinct RLFs automatically was implemented into the *report generator* module of REDECHECK. This update results in the textual report output by REDECHECK showing the failures grouped together, rather than simply listed one after the other, while the highlighted screenshots were reorganised into a directory structure reflecting the determined grouping.



**RESEARCH QUESTION ONE:** For this question, which evaluates the effectiveness of the grouping algorithm, the output of `REDeCHECK` was automatically and systematically compared against a gold standard. During the previous evaluation, all of the failures reported by `REDeCHECK` were manually analysed to determine which failures were related. Each true positive reported was assigned to a unique “distinct RLF ID”, while all non-observable issues and false positives were simply assigned the mark ‘-’ to signify they do not belong to a distinct RLF. For instance, 3-Minute Journal evinced 8 true positive reports that were manually grouped into 2 distinct RLFs, so the gold standard was {1,1,2,1,1,1,2,1,-,-,-}. The automatic comparison approach compared the produced grouping to the gold standard, using the following guidelines:

1. All of the failures for each distinct RLF should be placed in the same group (e.g., the six failures for RLF 1 above were all in a single group).
2. No group should contain failures from multiple distinct RLFs (e.g., one of the failures from RLF2 was not grouped with the failures for RLF1).

If the comparison approach found no issue with a group, then it marked it as “correctly grouped”. If not, it was “incorrectly grouped”. A correct group may contain FPs and NOIs as well as the important TPs. Ideally, it would not, as a group containing only related TPs is likely to be more beneficial to a developer, as they will not have to manually determine which failures they need to deal with. Therefore for each correct group, the comparison also checked whether it contains any FPs or NOIs. If it does not, then the comparison considers it to be a “perfect group”.

**RESEARCH QUESTION TWO:** To ascertain the extent to which real end users agree both with the individual failures and overall distinct groupings reported by `REDeCHECK`, this research question used a human study.

While many previous software engineering human studies were performed in controlled environments, the tasks involved in this study did not require this and were therefore run in a crowdsourced manner. Both postgraduate and undergraduate students were invited to participate in the study. Given the lack of need for experience in web development, or in fact programming experience of any kind, prospective participants were not required to complete any kind of qualification test in order to join the study. Prior to the experiment, participants were introduced to the five different types of responsive layout failure and how it is important for developers to detect them. In total, 10 postgraduate students and 1 undergraduate student took part in the study.

**Question 1**

Do you think there is an RLF on this page?

Yes

No

Please explain your answer.

Your answer

**NEXT**

Page 1 of 21

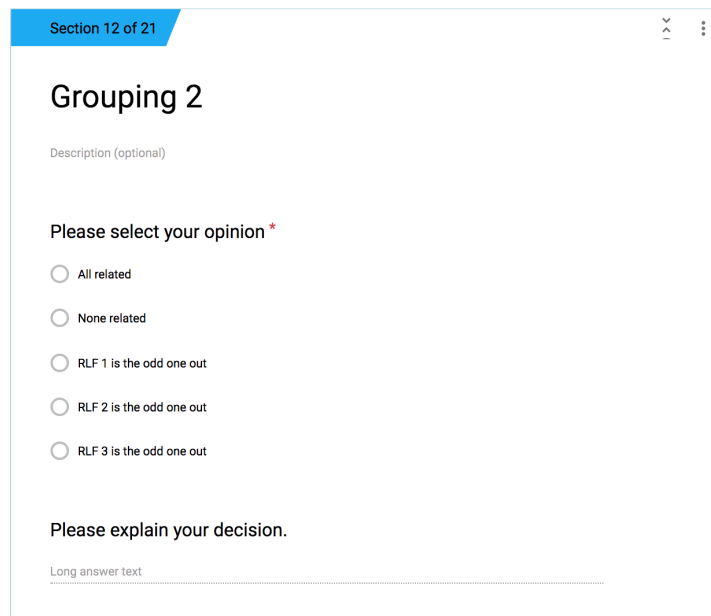
Never submit passwords through Google Forms.

**Figure 6.1:** An example response form for part one of the human study.

The experiment itself consisted of two main tasks, with no set time limit so each participant could complete it in their own time. The first task required the participants to inspect a series of screenshots and state whether they felt an RLF was evident. This allowed for the validation of not only the types of layout failure currently detectable by REDECHECK but also the manual classifications made during the previous empirical evaluation. For instance, if the majority of participants agreed with the failures reported by REDECHECK that were initially classified as true positives, then it is clear that REDECHECK is capable of detecting RLFs that human users actually deem important. Participants responded with a simple binary “Yes” or “No” decision as to whether they thought the screenshots showed an RLF, and also had the opportunity to elaborate on their decision, as shown by Figure 6.1.

To mitigate any bias that could potentially arise from the selection of failures for the study, an automated selection function was implemented that randomly selected failures based on probability. Firstly, the approach read in the manual classifications of all of the failures and stored them into three sets, representing the three possible classifications. Then, it selected one of the three sets, from which it randomly selected a failure to add to the study. The selection process selects true positive failures with a probability of 0.6, and false positives or non-observable issues with a probability of 0.2 each. While equal probabilities could have been assigned to each category, the primary purpose of this study was to evaluate whether humans agree with the *actual* RLFs detected by REDECHECK, so selecting a larger proportion of true positive results is a sensible choice.

For the second experiment, the participants were asked to inspect a group of failure screenshots and decide whether they think they were related, allowing the grouping algorithm presented earlier in this chapter to be evaluated. For each grouping presented to them, they had the option of selecting “All failures were related”, “none were related”, or selecting which individual failure in the grouping was the “odd one out”. Figure 6.2 shows an example of the response form. As with the previous task, if the majority of participants agreed with the groupings produced by REDECHECK, then it provides empirical support for the overall approach and for the manually curated “distinct RLF groupings”.

The image shows a screenshot of a survey form titled "Grouping 2". At the top left, it says "Section 12 of 21". Below the title, there is a field for "Description (optional)". The main question is "Please select your opinion \*", followed by five radio button options: "All related", "None related", "RLF 1 is the odd one out", "RLF 2 is the odd one out", and "RLF 3 is the odd one out". Below the options, there is a text prompt "Please explain your decision." and a "Long answer text" input field.

**Figure 6.2:** An example response form for part two of the human study.

As with the failure selection in part one of the human study, an automated approach was used to randomly select the groupings presented to the participants. To begin, the approach randomly selected one of the distinct RLFs. Then, the approach decided whether to keep the grouping as it is, or modify it to create an incorrect grouping. It used a probability of 0.7 to stay with the original grouping, and 0.3 for creating a “mutated” grouping. When this occurred, the approach randomly selected an additional failure that was reported for the web page in question and added it to the group. If the initial grouping contained just a single failure, however, the approach added two other reported failures. This is because presenting a grouping of two failures to the participants makes it very difficult for them to select the “odd one out”, as technically both failures could be considered to be the odd one out, which would make analysing their responses problematic.

### Threats to Validity

**USE OF STUDENTS:** All of the participants in the study were students, and previous research has investigated how respective students are when compared to professionals, especially when used in experiments such as the human study presented in this chapter [25, 70, 127]. These papers found that in many scenarios, students can be used in place of industrial participants and not impact the validity of the obtained results. This is especially true when performing or evaluating a new approach that neither the students or professionals have seen before, or performing relatively small tasks of judgement, such as the tasks in this study. However, the experiment required no prior knowledge of web development, or indeed any programming experience whatsoever. It simply investigated human's subjective views on the visual appearance of a group of web pages, therefore their opinions should be generalisable to other web users with a wide variety of experience.

**FAILURE SELECTION:** For the two tasks in the human study, it was possible that the selection of the individual failures in part one and the groupings of failures in part two represent a potential validity threat. To mitigate this, they were randomly selected to obtain as wide a variety of subjects as possible for the study.

**CREATION OF GOLD STANDARD:** In this evaluation, the gold standards against which REDECHECK's output and the participants in the human study are compared were manually generated. This is obviously a significant threat to the validity of any obtained results. However, as discussed in Chapter 5, the gold standard was generated by a committee to mitigate the risk of any subjectivity bias from individual opinions.

#### 6.2.2 Empirical Results

**RESEARCH QUESTION ONE:** Table 6.2 shows the results of the implemented grouping approach when compared to the gold standard. The results show that the vast majority of failures were correctly grouped by the automated approach. For some web pages, including Hotel WiFi Test, MidwayMeetup, Ninite and StumbleUpon, it is actually impossible for the approach to group the failures incorrectly, according to the rules used by the grouping comparator. For these

Web Page	TPs Reported	Distinct RLFs	Correct Groups	Perfect Groups
3-Minute Journal	8	2	2	2
Accountkiller	121	3	3	1
Airbnb	2	2	0	0
BugMeNot	4	4	4	4
Cloudconvert	2	1	1	1
ConsumerReports	10	4	3	2
CoveredCalendar	2	2	2	2
Duolingo	2	1	1	0
Hotel WiFi Test	1	1	1	1
MidwayMeetup	1	1	1	1
Ninite	1	1	1	1
PDFescape	2	2	2	2
PepFeed	7	6	4	3
StumbleUpon	1	1	1	1
Usersearch	1	1	1	1
Will My Phone Work	1	1	1	1
Eat This Much	2	2	2	0
Forvo	15	5	5	5
Google Maps SVP	1	1	1	1
Khan Academy	1	1	1	1
Memrise	1	1	1	0
Retail Me Not	2	2	2	2
Similar Sites	3	3	1	0
Startup Stash	3	2	2	1

**Table 6.2:** Grouping results for all web pages containing at least one RLF.

web pages RECHECK reported a single TP, and therefore the group cannot be “missing” any related failures or contain any unrelated TPs.

Because of this, arguably the more important subjects to analyse are those with multiple TPs and/or multiple distinct RLFs. For some subjects such as Airbnb, CoveredCalendar and PDFescape, each distinct RLF contained just a single failure. This provided an easier task for the grouping algorithm as it only has to keep unrelated TPs apart, rather than grouping related TPs together. However, for others such as 3-Minute Journal, ConsumerReports and Cloudconvert, some distinct RLFs were made up of multiple individual failures that the approach had to group together. For instance, on ConsumerReports, the approach had to group 7 related failures into a single group while keeping the other 3 failures corresponding to the 3 remaining RLFs separate. 3-Minute Journal was perhaps

even more complex, as the 2 distinct RLFs were represented by 6 and 2 individual failure reports, respectively. Nevertheless, despite these difficulties, the results show the grouping approach was highly effective.

The only exceptions were the two RLFs reported for each of Airbnb, PepFeed and Similar Sites. To investigate the cause of the differences between the groupings produced by REDECHECK and the gold standard, the logs of the grouping process were analysed, which showed the similarity scores observed for each failure-group pair. For Airbnb, this revealed that as the two distinct failures were of the same type (element wrapping), occurred at similar narrow viewport widths and involved elements that were close together on the web page and therefore possessed relatively similar XPath expressions, the grouping approach calculated a similarity score that was sufficiently high to cause the two failures to be grouped together. For PepFeed, the incorrect grouping was caused by one of the failures from one distinct RLF to be relatively similar in terms of type, bounds and elements to another group of failures representing a different distinct RLF. The failure was then “incorrectly” added to the group, causing the failures for both distinct RLFs to be regarded as incorrectly grouped. Finally, for Similar Sites, two of the three viewport protrusions which had been classified as distinct were placed into a single group by the approach. This was due to the protruding elements having fairly similar XPath expressions and protruding at similar viewport widths.

The table also shows the approach achieved “perfect” groupings on the majority of subjects. In some cases achieving this was straightforward, as the approach only reported TPs and therefore achieving correct groupings automatically led to perfect groupings. A good example of this is Accountkiller, where only TPs from three distinct RLFs were reported and subsequently grouped. For others, it was more complex due to the NOIs and FPs that the approach had to keep separate from the key groups. On ConsumerReports, for instance, there were 13 NOIs for the approach to contend with. Despite this, two of the three correct groups reported by the approach were “perfect”.

**Conclusion for RQ1** The results show that the proposed automated grouping approach can accurately group failures into distinct RLFs, with only a couple of incorrectly grouped failures. In the majority of cases, the approach produced “perfect” groupings, in which only related TPs were present in the key groups. This is likely to be highly useful to developers as they

can see the various manifestations of the same root cause error easily and quickly.

**RESEARCH QUESTION TWO:** This section presents the results obtained during the human study described earlier in this chapter. For reference purposes, the screenshots provided to the participants in the study are included in an appendix at the end of this thesis.

### **Part One**

Figure 6.3 shows the participant responses to the ten questions in part 1 of the study. This section discusses each of the ten questions individually. As each of the ten RLFs were reported by REDECHECK and then manually classified, this section compares the participant majority to the human generated gold standard, referred to as the *expected result*. Finally, the section ends by discussing the results as a whole.

#### **Q1:**

Participant Majority: No Failure

Gold Standard: No Failure (failure classified as a NOI)

For this first question, the screenshots showed a NOI in which a button protruded out of the viewport very slightly. The results showed that only 2 participants reported a failure for this question, highlighting the subtlety of the failure. It also emphasises the ability of REDECHECK to detect issues with a low visual impact. One of the two people who reported a failure actually reported a failure in a different section of the web page. Interestingly, one person observed the failure but answered “No” as they stated it “looks a little odd” but did not consider it a failure.

#### **Q2:**

Participant Majority: No Failure

Gold Standard: No Failure (failure classified as a NOI)

This screenshot contained an invisible element collision that the previous chapter classified as a NOI. All but one participant therefore correctly said there were no RLFs present in the web page. Interestingly however, one person said there was a wrapping failure evident, but did not say where.

**Q3:**

Participant Majority: No Failure

Gold Standard: No Failure (failure classified as a NOI)

This question presents a small element protrusion NOI to the participants. The majority (8 of 11) of participants reported no failure, but three people reported an element collision in a different part of the web page. The potential failure was therefore analysed to determine why REDECHECK had not reported it. It turns out the “overlapping” elements were always in a parent-child relationship and therefore were never reported as a collision failure. This was caused by the child element being rendered in the right-hand side padding of the parent element and then being forced closer to the main content as the viewport narrowed.

**Q4:**

Participant Majority: Failure Present

Gold Standard: Failure Present (failure classified as a TP)

This question presented a screenshot obviously containing an RLF, causing all eleven participants to report an issue. Interestingly, some participants simply commented on the general “ugly” or “weird” layout presented, but only one participant actually mentioned the specific element protrusion failure reported by REDECHECK. This is however understandable, as the protruding element’s parent has no visible border while the other neighbouring elements exhibit layout behaviour more likely to attract the attention of an end user.

**Q5:**

Participant Majority: Failure Present

Gold Standard: Failure Present (failure classified as a TP)

This screenshot showcased a fairly obvious wrapping failure, in which a whole section of the web page’s footer wrapped onto a new line. 8 of the 11 participants correctly stated there was a failure on the web page. The other 3 participants deemed the wrapping behaviour to be acceptable or even intended behaviour. For instance, one said “the last menu wraps to the next line successfully for the lower resolution”, suggesting there is some subjectivity when it comes to what constitutes a wrapping failure.



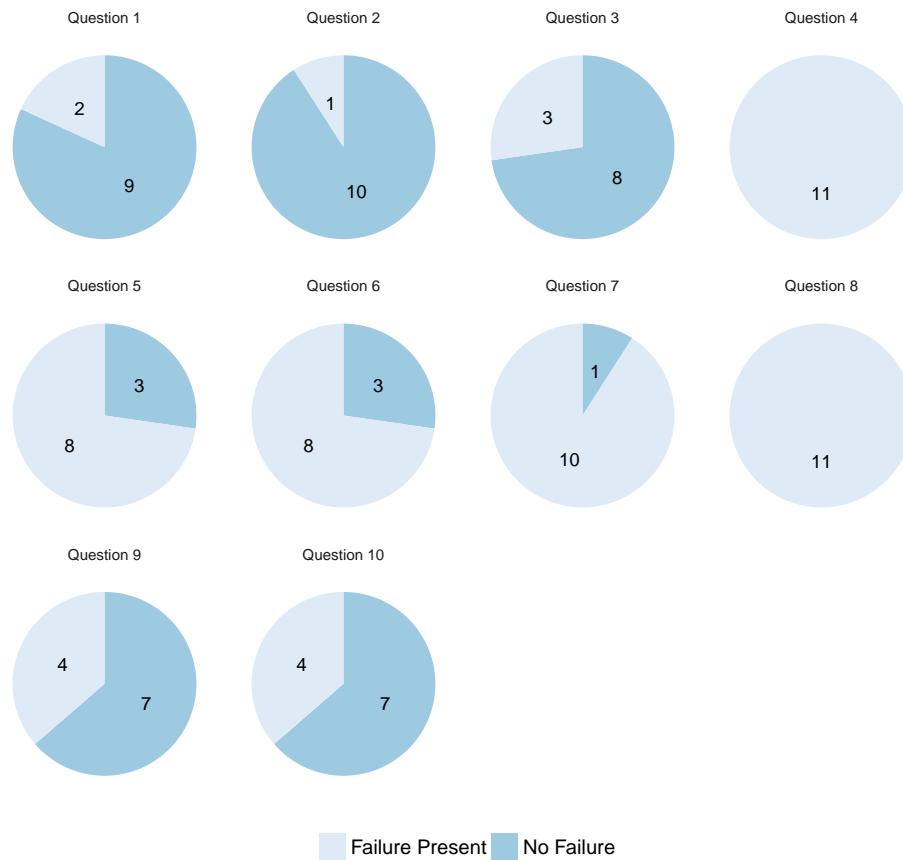


Figure 6.3: Responses for Part One.

#### Q6:

Participant Majority: Failure Present

Gold Standard: Failure Present (failure classified as a TP)

This question presented a web page demonstrating a slightly unusual grid-based layout for a small range of viewport widths. However, at the specific viewport width of the RLF, there is nothing “blatantly” incorrect about the layout. The results are therefore perhaps unsurprising, with some 8 people reporting RLFs and 3 not. Of the 8 who reported an RLF, 4 reported an RLF in a different part of the page, suggesting they considered the rest of the web page to be error-free. My analysis found that this other RLF was not detected by REDECHECK as it’s visual appearance was not reflected in the underlying DOM structure used by REDECHECK. The results also emphasised the advantage of REDECHECK sampling the web page across a wide range of viewport widths.

By inspecting at just individual viewport widths like in this study, some RLFs can be missed.

**Q7:**

Participant Majority: Failure Present

Gold Standard: Failure Present (failure classified as a TP)

For this question, the screenshots presented a clearly evident wrapping failure. All but one participant stated there was a failure present. The one participant who did not justified their decision by the fact that all the same information was available on the “faulty” web page. Given the rather obvious nature of the failure, it comes as little surprise that almost all participants reported it as a failure.

**Q8:**

Participant Majority: Failure Present

Gold Standard: Failure Present (failure classified as a TP)

This question presented the participants a screenshot showing the header of a web page overlapping with the main content. Given the large visual impact of the failure and it’s location right at the top of the web page, it is no surprise that all participants reported the failure.

**Q9:**

Participant Majority: No Failure

Gold Standard: Failure Present (failure classified as a TP)

For this question, the screenshots showed some navigation links wrapping onto multiple rows for a small number of widths as the viewport narrows. However, as the screenshots did not show the layout immediately either side of this small range layout, it is more likely participants would view it as a wrapping failure instead. The majority of the participants stated the wrapping was correct behaviour as there were so many links there was no way of avoiding it. However, 4 participants decided that the wrapping was a failure. This suggests failures of this type can be quite subjective. They generally involve no loss of functionality, but some people feel they compromise the aesthetics of the web page. This subjectivity could depend on which elements wrap. For example, if a navigation link right at the top of the web page wrapped, most people would likely agree

that it is an RLF. In contrast, failures such as this one, where the wrapping occurs in the footer, are perhaps less likely to be considered failures.

#### Q10:

Participant Majority: No Failure

Gold Standard: Failure Present (failure classified as a TP)

This question presented a failure in which the navigation links protruded out of the header. However, due to certain CSS rules, they disappeared completely from view and were unclickable. The results were rather interesting. Firstly, only 4 participants reported a failure, while the other 7 said there was no failure. This highlights the subtlety of the failure and showcases REDECHECK's ability to identify important RLFs that human testers are likely to miss. Quite interestingly, one participant observed the disappearance of the navigation links but stated there was no RLF, with their justification stating they "assumed that the loss of the top menu was intentional".

#### Part One Summary

Table 6.3 shows a summary of the results for part one of the study, including the manual classification assigned to each failure, the expected result and participant majorities. For the 7 questions presenting screenshots containing TPs, i.e. actual RLFs, a majority of the participants reported failures on 5 of them. In most cases, the majorities were large. In fact, the smallest majority was still 5, from an 8-3 vote in favour of an RLF (questions 5 and 6). The results suggest humans strongly agree with the TP RLFs reported by REDECHECK in the previous chapter. This is especially true for RLFs with a large visual impact, where a sizeable or possibly important part of the web page is involved in the failure.

Three of the questions (1, 2 and 3) presented screenshots containing no RLFs. On all of these, the majority of the participants reported no failure present. This is to be expected. FPs represent correct layout behaviour incorrectly reported as failures and NOIs have no visible manifestation, existing purely at the DOM level.

Obviously, this study only used 10 of the many RLFs reported by REDECHECK in the previous experiment. For instance, only 6 of the 50 TP RLFs were used, so the results may not necessarily generalise to the remainder of the reported RLFs. However, the selected failures were randomly chosen and demonstrated a variety of failure types and classifications. Therefore, the results obtained should be generalisable.

Question	Classification	Participant Majority	Expected Result
1	NOI	No Failure	No Failure
2	NOI	No Failure	No Failure
3	NOI	No Failure	No Failure
4	TP	Failure Present	Failure Present
5	TP	Failure Present	Failure Present
6	TP	Failure Present	Failure Present
7	TP	Failure Present	Failure Present
8	TP	Failure Present	Failure Present
9	TP	No Failure	Failure Present
10	TP	No Failure	Failure Present

**Table 6.3:** Summary of the results for part one of the study.

## Part Two

In this section, each question is again discussed individually before the results as a whole are presented. For each question, the participant majority is compared to the human generated expected result and the output of REDECHECK's automated grouping technique.

### Q1:

Participant Majority: All Related

Expected Result: All Related

REDECHECK Result: All Related

This grouping contained a collection of viewport protrusion failures that all involved similar elements at similar viewport widths. During RQ1, we had manually grouped these six failures as all pertaining to the same distinct RLF. REDECHECK's automated grouping approach had also grouped them together. Therefore, it is unsurprising that all but one of the participants said all the reported RLFs were related.

**Q2:**

Participant Majority: Two is the Odd One Out

Expected Result: None Related/Two is the Odd One Out

REDECHECK Result: Two is the Odd One Out

This grouping presented one TP and two NOIs to the participants. The two NOIs (RLFs 1 and 3) are very similar in nature, both being element protrusions occurring at the same viewport widths and involving similar elements. Therefore, some participants could consider them related and state that RLF 2 is the odd one out. In fact, this was the decision REDECHECK made. However, given the elements are not actually the same, some may deem none of the RLFs to be related. Figure 6.4 shows the vast majority (9 of 11) correctly reported RLF 2 as the odd one out. However, interestingly two people said all three were related. This was because they both misinterpreted the two element protrusions as element collisions and reported the three RLFs as related because they were all of the same type.

**Q3:**

Participant Majority: Three is the Odd One Out

Expected Result: None Related/Three is the Odd One Out

REDECHECK Result: None Related

Question 3 presented the participants with an NOI element protrusion, an FP viewport protrusion and a TP element collision. However, the screenshot for RLF 3 (the element collision) appears to be an NOI. It is only when the viewport narrows further that it becomes apparent that it is in fact a TP. This likely impacted people's judgement, and a more illustrative screenshot may have helped the participants make a more informed decision. RLFs 1 and 3 both involve the same element in the web page's footer, which likely led the majority of participants to report RLF 2 as the odd one out. REDECHECK reported all three failures as unrelated.

**Q4:**

Participant Majority: Three is the Odd One Out

Expected Result: None Related

REDECHECK Result: None Related

This question presented three RLFs, a viewport protrusion, a wrapping and an element collision, each of which we initially classified as distinct. REDECHECK also determined all three failures to be distinct and unrelated. However, as RLFs 1 and 2 involved the same faulty elements, the majority (7 of 11) reported them as related. This is despite the fact the two RLFs are of different types and occur at quite different viewport widths. 3 of the participants reported the expected answer that all of them were unrelated. Finally, one person actually reported RLF 1 as the odd one out, because the element wrapping and element collision failures manifested in similar ways and were therefore in their opinion related, even though they involved completely different elements.

**Q5:**

Participant Majority: None Related

Expected Result: None Related/Three is the Odd One Out

REDECHECK Result: None Related

This question presents two NOIs and a single TP wrapping failure to the participants. The two NOIs involved different elements and were of different types. However, they did both manifest at the same viewport width. Therefore, I expected the majority of participants to select either “None related” (REDECHECK reported this decision) or “RLF 3 is the odd one out”. The results show that these combine to produce a large majority, as 5 people selected “None related” and 2 selected “Three”. 3 people said all three RLFs were related because “all of these images look correct” and “I don’t think any of them display any layout faults”. These reasons are unexpected, because even if a human thought not of the screenshots showed RLFs, the fact they all involve different parts of the web page at different viewport widths should mean humans regard them as unrelated.

**Q6:**

Participant Majority: All Related

Expected Result: Three is the Odd One Out

REDECHECK Result: Three is the Odd One Out

This question presented three TP viewport protrusions, two of which we originally grouped together while keeping the other (RLF 3) in a different group. REDECHECK also kept RLF 3 separate from the other two. However, the vast majority of participants said all three RLFs were related. This is understandable,

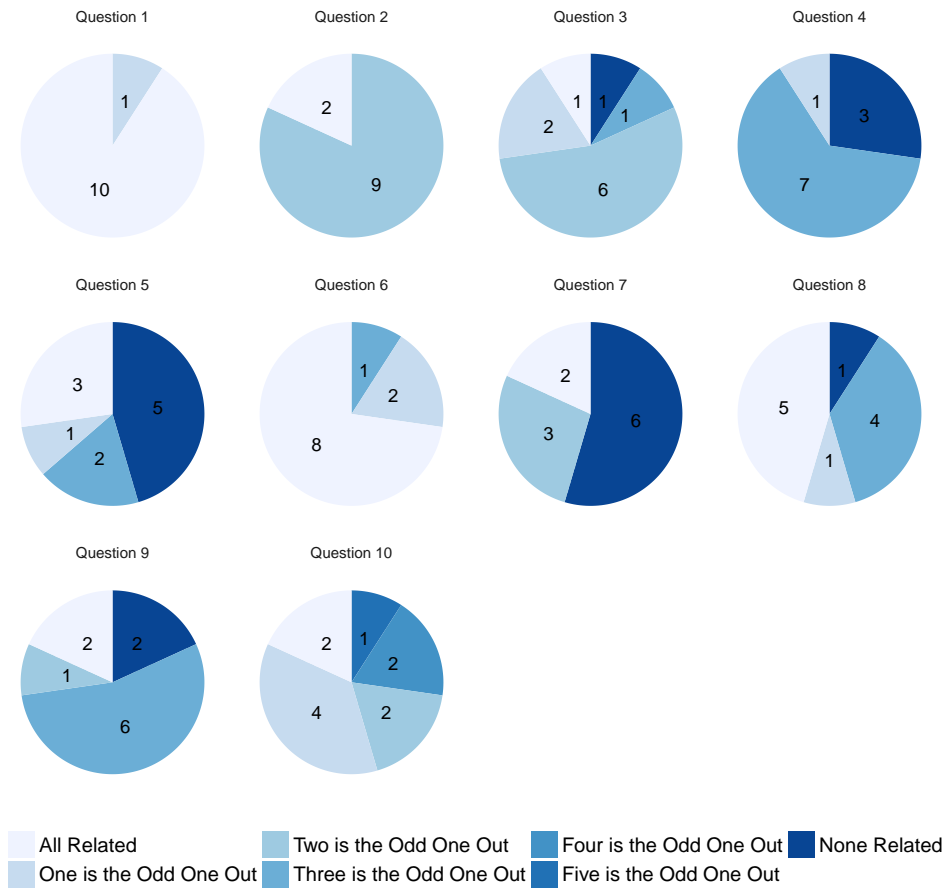


Figure 6.4: Responses for Part Two.

given that the failures are all the same type and involve the same elements. Only one person reported RLF 3 as the odd one out, while two people reported RLF 1 as the odd one out, with their reasons suggesting they may have misinterpreted the images presented to them.

Q7:

Participant Majority: None Related/Two is the Odd One Out

Expected Result: None Related/Two is the Odd One Out

REDeCHECK Result: Two is the Odd One Out

Question 7 involved two NOI viewport protrusions at wide viewport widths and one TP wrapping RLF (RLF 2) at narrow viewport widths. Therefore, the obvious answer is either “None related” or “RLF 2 is the odd one out”, which is the decision REDeCHECK made. Figure 6.4 shows the vast majority of participants agreed, with 6 saying “None related” and 3 saying “RLF 2”. Interestingly,

two people said all the RLFs were related. One gave the reason “All of these look correct” while the other said “I don’t think any of the highlighted images in the three RLFs are properly centralised to be within the middle of the page”. These reasons fail to provide much useful information as to why they answered “All related”.

**Q8:**

Participant Majority: All Related

Expected Result: None Related

REDECHECK Result: None Related

Here, the study presented three TP RLFs to the participants; one wrapping, one viewport protrusion and an element protrusion. Both the manual grouping and REDECHECK’s automated approach classified all three RLFs as unrelated to each other. However, as the wrapping and viewport protrusion (RLFs 1 and 2) involve the same element, 4 of the participants stated they were related and marked RLF 3 as the odd one out. Similarly, as all three failures involved form input boxes, albeit in different parts of the web page, 5 people said all three were related. This is particularly interesting, as both the two most popular decisions among the participants contrast with the gold standard and REDECHECK’s grouping.

**Q9:**

Participant Majority: Three is the Odd One Out

Expected Result: None Related/Three is the Odd One Out

REDECHECK Result: None Related

For this question, the three screenshots presented two NOIs and a TP. The two NOIs were an element collision and element protrusion involving very similar elements while the TP was a wrapping failure in a completely different part of the web page. Therefore, the expected result was for participants to either state RLF 3 (the wrapping) is the odd one out or none of the RLFs are related. REDECHECK determined the failures were all unrelated. The results show the participants agreed with the expected human generated decision. 6 stated RLF 3 was unrelated to the other two, while a further 2 people said they were all unrelated to each other.



**Q10:**

Participant Majority: One is the Odd One Out

Expected Result: Two is the Odd One Out

REDeCHECK Result: Two is the Odd One Out

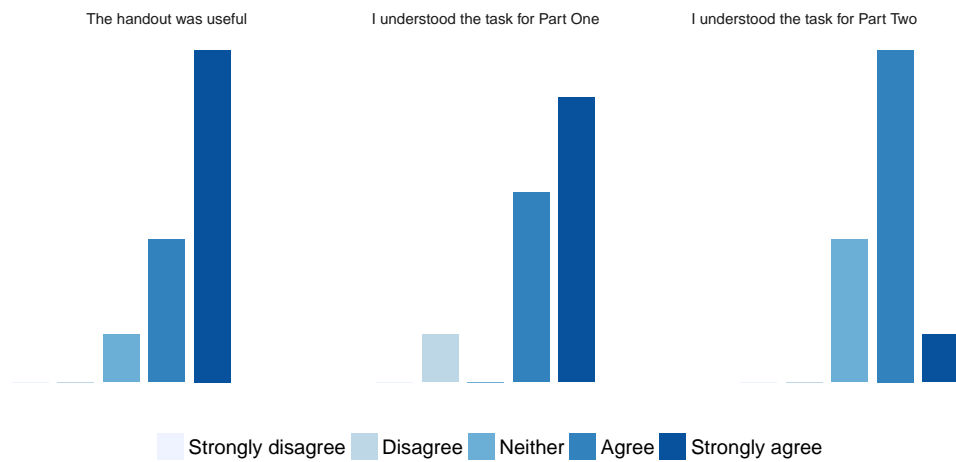
For the final question, the study presented eight TP viewport protrusions to the participants. 7 of these showed various elements in one section of the web page protruding, while RLF 2 showed the banner at the top of the page protruding. Therefore, manual grouping and REDeCHECK grouped 7 of them together, with RLF 2 being the odd one out. However, Figure 6.4 shows there was a wide variety of answers given by the participants. The leading answer was RLF 1 being the odd one out. Revisiting the screenshots and analysing participant's justifications, it is clear why 4 people gave this answer. RLF 1 shows an element that has first protruding out of its parent, and then protruded out of the viewport. Two people did state RLF 2 was the odd one out, giving the same reasons that both the manual grouping and REDeCHECK gave for their decisions. Two people said they were all related as they were all of the same type, suggesting error type was the main determinant in their decision. Finally, two people and one person said the odd one out was RLF 4 and RLF 5, respectively.

**Part Two Summary**

Question	Participant Majority	Expected Result (Human)	REDeCHECK
1	All Related	All Related	All Related
2	Two is Odd	None/Two is Odd	Two is Odd
3	Three is Odd	None/Three is Odd	None Related
4	Three is Odd	None Related	None Related
5	None Related	None/Three is Odd	None Related
6	All Related	Three is Odd	Three is Odd
7	None/Two is Odd	None/Two is Odd	Two is Odd
8	All Related	None Related	None Related
9	Three is Odd	None/Three is Odd	None Related
10	One is Odd	Two is Odd	Two is Odd

**Table 6.4:** Summary of the results for part two of the study.

The participants agreed with the manual groupings for 6 of the 10 groupings presented (questions 1, 2, 3, 5, 7 and 9). For instance, in Question 1 they correctly stated that all of the presented RLFs were related. Similarly, in 5 cases they correctly selected the “odd failure out” or marked all the RLFs presented as



**Figure 6.5:** Responses for Exit Survey.

unrelated. The participants matched the grouping produced by REDECHECK in 4 out of these 6 scenarios, with the exceptions being questions 3 and 9. However, in both of these questions, both “None related” and “Three is Odd” were correct answers (i.e. the expected result), so while the participants and REDECHECK disagreed on their decision, they both produced the expected result.

There were four instances (questions 4, 6, 8 and 10) where the participant majority was different to both the manually created gold standard and the groupings produced by REDECHECK. However, in these cases, the feedback given by the participants helped to explain why they made the decisions they did. Chapter 7 goes on to investigate the root causes of these RLFs to determine which decision was correct. This will highlight any potential shortcomings with the automated grouping approach presented earlier in this chapter.

### Exit Survey

Figure 6.5 presents the results of the exit survey filled in by the participants. It breaks the results down into the three individual questions. The first graph shows participants strongly agreed that the handout was very useful to them, suggesting it did an excellent job of introducing them to responsive layout failures and why they are important to identify.

Next, the feedback shows all but one participant understood the task for part one of the study. The majority of these stated that they “strongly agreed”, suggesting the results for part one should be reliable as the decisions were based on a good understanding of the problem.

Finally, the third bar graph shows that for part two, the majority of participants understood the task they were asked to perform. However, only 1 person stated that they “strongly agreed” with the statement, while 3 said they “neither agreed nor disagreed”. This suggests the second task was in general a less clear than the first one, which could impact the reliability of the results. For instance, some of the reasons given for a particular answer suggested some participants did not fully understand what was meant by two RLFs being “related”. This therefore means their answers may not be an accurate representation of their opinions. Also, the results could also simply show the subjectivity of humans when asked to give their opinions on the aesthetics of a web page. As the experiment in this chapter was only a pilot study, any future studies will involve substantially more participants and will use the additional feedback provided to improve the study, with particular attention being to ensuring the participants understand the notion of related failures. Despite this, the results still suggest the results discussed for part two of the study should again be reliable.

**Conclusion for RQ2** The results show that humans agree with the RLFs reported by REDECHECK, reporting the expected classification in 5 of the 7 scenarios. When presented with screenshots containing either NOIs or FPs, the majority of participants correctly stated there was no RLF present.

The results also show humans agreed with the expected groupings in 6 of the 10 scenarios and agreed with REDECHECK in 4 of those 6 examples. Further analysis of the RLFs in the next chapter will reveal which of the groupings produced was correct.

Finally, the feedback from the exit survey suggests the majority of participants understood the tasks presented to them, meaning the obtained results should be reliable.

### 6.3 CONCLUDING REMARKS

This chapter described an approach that automatically groups RLFs that are related and likely to share the same root cause. The approach used the concepts of element similarity, viewport similarity and failure type similarity to group the failures. By implementing this technique into REDECHECK, it should reduce

the effort required from the user, as this task was performed manually in the previous chapter.

Next, the chapter evaluated the automated grouping approach. It used both sets of responsive web pages from Chapter 5, which contained 26 and 10 web pages, respectively. The grouping approach was shown to be highly effective. Only 7 RLFs were incorrectly grouped, and the vast majority of groupings were “perfect”, meaning groups containing TPs contained only TPs.

Finally, the chapter presented a human study investigating human opinion on both the RLFs reported by REDECHECK and the groupings it produced. Using a pool of 11 computer science students, the study found humans agreed with the majority of RLFs reported by REDECHECK. They also showed that they generally agreed with the manual groupings (6 out of 10 instances) and agreed with the groupings produced by REDECHECK in 4 of those 6 instances.

# 7

---

## A STUDY OF THE ROOT CAUSES OF REAL-WORLD RLFs

---

Chapter 5 investigated responsive layout failures in real-world web pages and found them to be prevalent, with over half of the web pages evincing RLFs. This chapter investigates these RLFs further by implementing potential patches for each failure. By identifying the underlying faults, i.e., the programming mistakes made by the developers of the web pages, this chapter investigates how difficult fixing RLFs is for developers, as well as further evaluating the distinct RLF groupings produced in Chapters 5 and 6. This empirical evaluation forms the key contribution of this chapter.

### 7.1 STUDY DESIGN

This chapter evaluates the root causes and potential fixes implemented in this study in the context of two research questions:

**RESEARCH QUESTION ONE:** *How accurate were the distinct RLF groups of related failures identified in Chapter 5?*

**RESEARCH QUESTION TWO:** *How complex are the patches required to fix the responsive layout failures?*

#### **Subjects**

This study used the initial pool of 26 responsive web pages from Chapter 5. As shown by that chapter, these web pages contained a total of 33 distinct RLFs, with a wide variety of failure types evident.

#### **Methodology**

For this study, the fault fixing process needed to be as consistent as possible across the different RLFs and web pages. Therefore, a simple, methodical ap-

proach was devised which I followed as closely as possible. It began by identifying the specific faulty elements in the source code. Then, domain-specific knowledge was used to inspect particular CSS rules that could be responsible for the failure. This approach was more efficient than exhaustively inspecting each CSS rule applied to the elements. For instance, if an element had wrapped onto a new row, the fault probably involved width or margin rules. Likewise, if a failure occurred at the same viewport width as a significant shift in layout, the fault could be in one of the media query breakpoints.

Once the potential root cause had been identified, a fix was attempted. In some cases, this was done directly in the browser via Firefox developer tools. This utility allows modification of the CSS rules applied to an element to quickly see whether changing a particular rule had a positive impact on the web page layout. However, in most cases the developer tools were simply used to identify the location of the problematic source code. The potential fix was then implemented in the source code and the web page reloaded in the browser. If the failure persisted, the current “fix” was evaluated using domain knowledge to see if it was an improvement. If it was, then further changes were implemented to try and create a full fix. If not, the code was reverted back to its original state and the process began again from the beginning.

Once the failure no longer manifested in the web page, `ReDeCheck` was run on the “fixed” version of the web page. This acted as a sanity check of sorts, to make sure that the failure was successfully fixed at all viewport widths and no new failures had been introduced. To invite discussion on the fixes, they are publicly available. The fixed versions of the web pages are available in the same GitHub repository as the faulty ones [146]. The thought processes behind the fixes are also available as think-aloud screen recordings [5].

### Threats to Validity

While this study was not evaluating a novel approach, there were still some validity threats that had to be mitigated.

**INCONSISTENCY IN FIXING PROCESS:** As the study aimed to compare the different fixes in term of their relative complexities, it is important that the process followed to implement the fixes was consistent. Therefore, an approach was outlined prior to starting the fixing experiment, which was then followed as closely as possible for each RLF. Finally, the study was conducted in a “think-aloud” manner, in which each step and decision taken was explained in real-

time. These recordings are available online for analysis by other researchers or industry professionals [5].

**CORRECTNESS OF FIXES:** Using the outlined fault fixing procedure, the fixes implemented should be the most suitable and “correct” ones for each failure. However, other developers may disagree with these fixes as web page aesthetics are subjective, as previously discussed in Chapter 2. Therefore, the fixes are publicly available for scrutiny and discussion [146].

## 7.2 PATCHING THE FAILURES

This section describes the fixes for each failure identified in Chapter 5. For each one, a figure illustrates the modified code along with “before” and “after” screenshots of the web pages.

### 7.2.1 *Element Collision Failures*

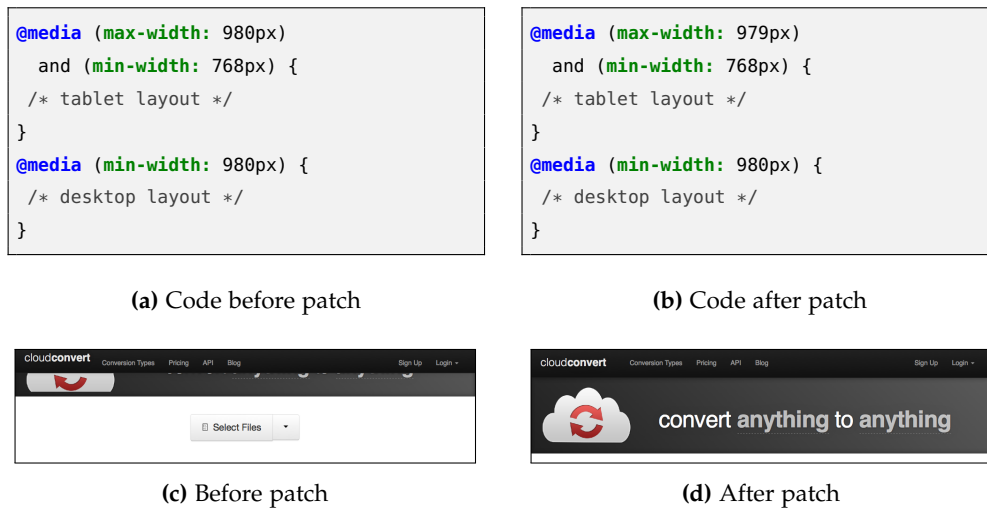
#### **Cloudconvert**

At the single viewport width of 980px, an error in the media queries controlling the switch between the mobile and desktop layouts results in the header bar and the main content banner overlapping. Given the failure occurs at the breakpoint at which the web page’s header toggles between its “mobile” and “desktop” versions, intuitively the root cause of the failure was likely to be the breakpoint values of the media queries controlling the offending elements.

The debugging process identified two faulty media queries with the condition (`max-width: 980px`). These meant at a viewport width of 980 pixels, part of the web page was using the desktop layout (triggered with a (`min-width: 980px`) media query) while the remainder was still using the mobile/tablet design. By changing both conditions to be (`max-width: 979px`), the failure was rectified. Figure 7.1 shows both the CSS code and visual appearance of the web page before and after the patch.

#### **MidwayMeetup**

When the viewport is wide enough, two small forms are displayed side-by-side. However, as the viewport narrows, the responsive layout does not adjust and the two forms begin to overlap. Initially, this obscures only part of one of



**Figure 7.1:** Fixing the Cloudconvert element collision failure.

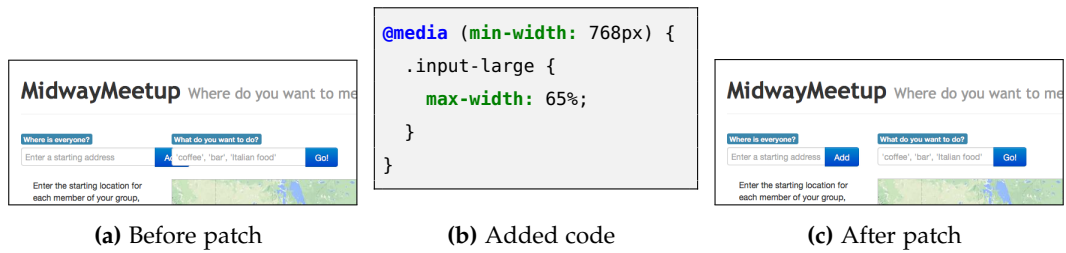
the submit buttons. But, as the viewport gets continually narrower the effects become more pronounced. Eventually, the entire button becomes covered and unclickable. This is a very severe failure as the obscured button controls the primary functionality of the web page. This essentially renders the page unusable on devices where the failure is at its worst. Intuitively, the root cause of the failure is likely to be one or both of the forms not responding adequately as the viewport narrows.

The debugging process found that the left hand form was the offending element. Its static width caused it to collide with the other form as the viewport narrowed. Therefore, the applied patch implemented a fluid width for the left-hand form elements. This ensured they are never too wide and hence never collide with the other form, while remaining usable at all viewport widths. The patch began with a media query which only applies its rules at wider viewports ( $\text{min-width: } 768\text{px}$ ). Then, the CSS property  $\text{max-width: } 100\%$  was added to the form element. This made sure it's width never becomes wider than that of its parent and it scales better as the viewport resizes. Finally, the input box was given a fluid width of 65%. This ensures it resizes in proportion with the whole form, while still leaving enough space to fit alongside the blue form submit button. Figure 7.2 illustrates the web page both before and after the patch.

## PepFeed

When the viewport is wide enough, the page renders the three content panels side by side. At very narrow widths, it stacks them into a single column. However, in between these widths two of the panels overlap, obscuring some of





**Figure 7.2:** Fixing the MidwayMeetup element collision failure.

the content. Prior to the debugging, it appeared likely the root cause of failure would be one or more of the three offending content panels.

Debugging of the failure found the issue to be an incorrect media query. This gave the elements static widths intended for the “desktop” layout when the viewport was as narrow as 415 pixels. Originally, three similar element collision failures had been classified as distinct. However, during debugging it became apparent they were due to incongruous shifts in the three numbered content panels as the viewport expanded and contracted, stemming from the single incorrect media query. The patch was very simple, with a simple change from (max-width: 414px) to (max-width: 767px). This ensured the elements remained in the single column layout until there was sufficient horizontal space to switch to a single row layout. Figure 7.3 illustrates the effects of the patch.

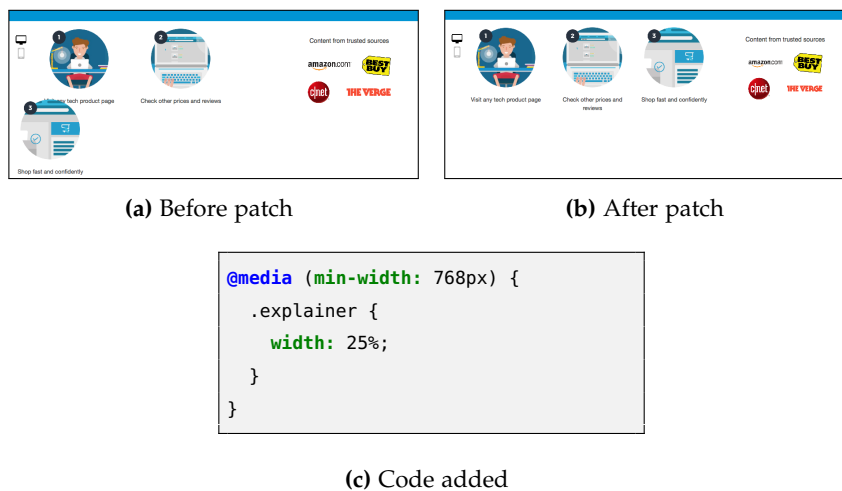


**Figure 7.3:** Fixing the PepFeed element collision failure at narrow widths.

After implementing the patch, another RLF thought to be distinct turned out to have been fixed as well. This was a viewport protrusion failure occurring just below the colliding tiles. Therefore, it was revealed that there were in fact 4 supposedly distinct RLFs that actually all shared the same root cause. However, the wrapping failure in the protruding element still occurred following this patch. This is especially interesting, as these RLFs were present in the human study from Chapter 6. Here, the majority of participants said the viewport protrusion

and the wrapping failure were related as they involved the same element, and the collision shown in Figure 7.3 (a) was the “odd one out”. This root cause analysis reveals that the wrapping failure was actually the odd one out and the other two shared a root cause. Unsurprisingly, not a single participant reported this. It suggests there is potentially a shortcoming in the grouping approach as it does not analyse the underlying HTML and CSS code when grouping the failures.

The web page also exhibited an element collision failure at much wider viewport widths. When the viewport is wide such as on a desktop, the page rendered the four content panels in a row. However, in between these widths one of the panels shifts onto a new row and two of the panels overlap, again obscuring some of the content. Naturally, the root cause is again likely to be in the CSS controlling the offending elements.



**Figure 7.4:** Fixing the PepFeed element collision failure at wide widths.

As expected, the root cause was the static widths applied to the content panels. These meant there was insufficient room for the intended layout. The implemented patch involved creating a media query with the condition (`min-width: 768px`) and applying fluid widths to the three main content panels. This ensured they scale fluidly in response to any changes in the viewport width. Figure 7.4 illustrates the result of this patch.

## StumbleUpon

When the viewport is wide enough, the browser renders the login link in the top right corner and the company logo in the centre of the page, just beneath it. However, as the viewport narrows, the login link wraps slightly and pushes

the logo over to the left hand side of the page. Despite no strict overlap of visual content, the layout shift has clearly caused an incongruous layout, rather than just a DOM level overlap. This is why the failure was classified as a true positive.

Debugging found the root cause to be the two header elements: the company logo and the login link. These had static widths which were too large for the elements to fit side-by-side at narrow viewport widths. There were multiple alternative patches which could have implemented, but the chosen patch reduced the size of the company logo as well as the margins between the offending elements and the edge of the page. When combined, these changes prevented the failure from manifesting, as illustrated by Figure 7.5.

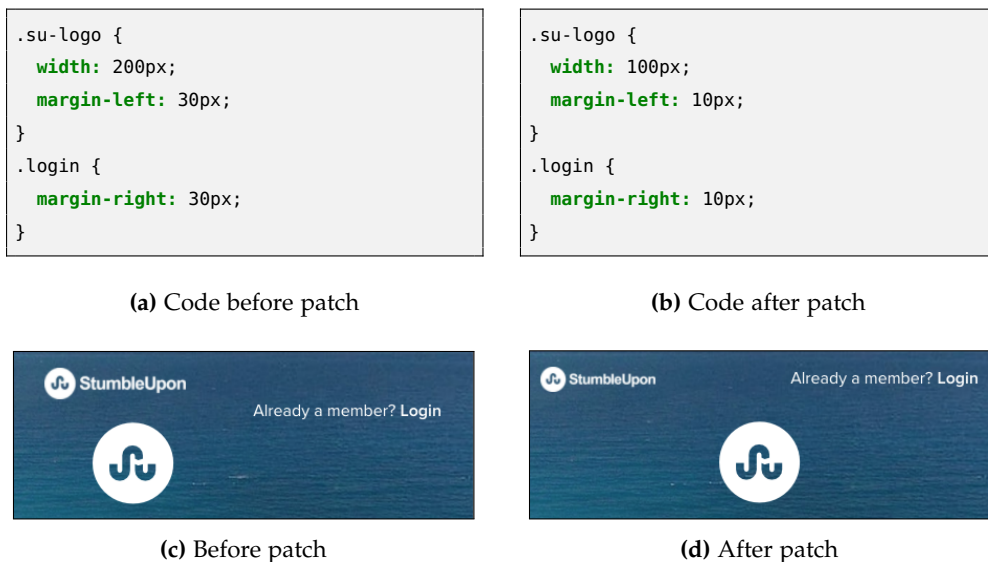


Figure 7.5: Fixing the StumbleUpon element collision failure.

### Will My Phone Work

At the viewport widths of 990px and 991px, a misuse of media queries controlling the switch between the mobile and desktop versions of the site results in the header bar and the main content banner overlapping. This failure is very similar in nature to Cloudconvert. Therefore, the root cause was again highly likely to be the breakpoint of one or more of the media queries controlling the responsive layout.

The root cause was found to be a media query which toggled part of the layout between mobile and desktop layouts at a different breakpoint to the rest of the site. In this case, its breakpoint was 990 pixels rather than 992 pixels. This discrepancy between the media queries led to the element collision observed at

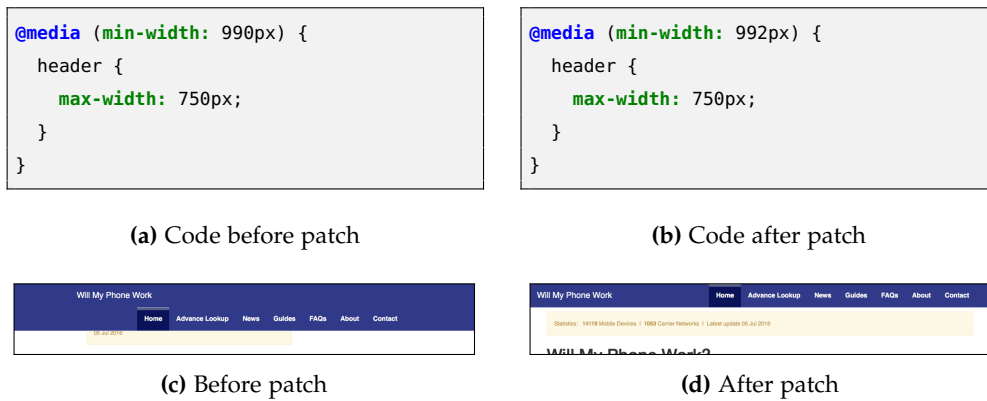


Figure 7.6: Fixing the Will My Phone Work element collision failure.

990-991 pixels. Once the breakpoint was modified, the failure no longer manifested, as shown by Figure 7.6.

7.2.2 Element Protrusion Failures

BugMeNot

At wide viewport widths, the web page’s logo and a search form fit comfortably side-by-side. However, as the viewport narrows, the header becomes too small to accommodate this layout. This causes the search bar to overflow the header. Given the characteristics of the failure, it is likely the root cause is either the protruding form or the header itself.

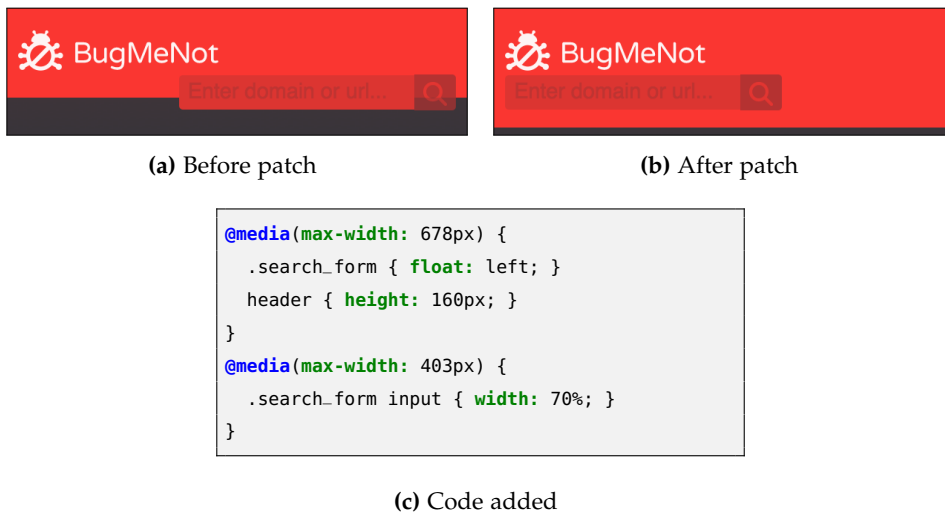


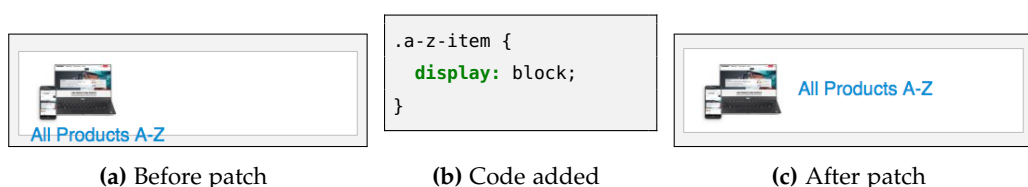
Figure 7.7: Fixing the BugMeNot element protrusion failure.

Debugging the failure showed the root cause to be a combination of two things. Firstly, the static width of the two elements caused the web page to push the form down when the viewport narrowed. Secondly, the fixed height of the header element means it does not expand vertically to accommodate its contents. These two factors combine to cause the failure. There was no feasible way of preventing the page from pushing the form down. Therefore, the implemented patch firstly added the `float: left` property to ensure it appears directly beneath the web page logo. Next, the height of the header was increased to accommodate the new location of the form. These rule declarations were placed inside a media query that only applies them when the viewport is too narrow for the original layout. Figure 7.7 presents the fix and its effect.

At very narrow viewport widths, the protruding text box also begins to protrude outside of the viewport window. Originally, this failure was classified as distinct from the element protrusion failure previously discussed. However, the root cause of both failures was the static width of the text box. Therefore, it is apparent they are instead just two manifestations of the same underlying issue. Because of this, following the aforementioned patch, a fluid width declaration was also applied to the offending element at narrow viewport widths, as shown by part (c) of Figure 7.7. This ensured the viewport protrusion failure no longer occurred.

### ConsumerReports

At wider viewport widths, the text “All Products A-Z” is correctly rendered inside its containing element. However, at narrower widths, a mistake in the CSS causes it to overflow its container. This results in poor aesthetics which could have a detrimental impact on user confidence in the site, as shown by part a) of Figure 7.8. The other tiles in the “Featured Products” section do not evince the same failure. Therefore, the root cause is likely to be a CSS declaration applied only to the “All Products A-Z” element.



**Figure 7.8:** Fixing the ConsumerReports element protrusion failure.

As expected, the root cause was the declaration `display: block;` applied to the `.a-z-item` class. Only the faulty element had this class, rather than all of the

neighbouring tiles. This caused the browser to display the text below the accompanying image, regardless of the width of the tile, text or image. Once the rule had been removed, the failure no longer occurred, with the text rendered to the right of the image.

### PDFescape

At narrow widths, the web page collapses the navigation links in the header to a drop down list which a user can reveal by clicking on a button. At wider widths, the links expand into a horizontal navbar. However, for a small number of widths after the layout changes to the desktop navbar layout, there is insufficient space in the header for the programmed layout. This causes the elements to overflow the header. As the header has the `overflow: hidden` CSS property set, the overflowing links are invisible and unclickable at this small range of widths. This therefore renders large portions of the web site inaccessible at the faulty viewport widths.

```
@media (max-width: 800px) {  
  /* mobile nav layout */  
}
```

(a) Code before patch

```
@media (max-width: 805px) {  
  /* mobile nav layout */  
}
```

(b) Code after patch



(c) Before patch



(d) After patch

**Figure 7.9:** Fixing the PDFescape element protrusion failure.

Debugging revealed the root cause to be a media query with a breakpoint of 800px. This caused the navigation links to display in a single row when there was insufficient room in the header. Therefore, the patch modified the breakpoint to only apply the “desktop” rules when there was enough space to do so. In this case, the query (`max-width: 805px`) controlled when to use the mobile layout. Figure 7.9 illustrates the details of the patch.

### 7.2.3 Viewport Protrusion Failures

#### 3-Minute Journal

At wide viewport widths, the browser displays the main text and the graph side-by-side on the web page. However, as the viewport narrows, parts of the graph start to protrude outside of the viewport window. These become inaccessible without horizontal scrolling. As discussed in Chapter 2, this is something no responsive web page should require a user to do.

Debugging showed the graph had a fixed width applied to it, meaning it did not scale its width in relation to that of the viewport window. The first part of the patch therefore involved replacing the fixed width with a fluid one. In this case 550px became 100%. It was then discovered that the x-axis labels continued to protrude outside of the viewport. Further debugging found each label had the

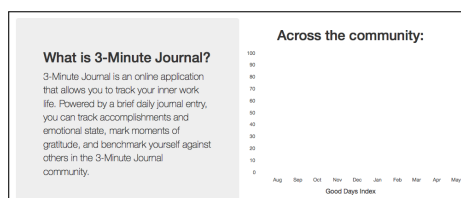
position: absolute attribute. Additional left: ??px declarations individually placed each label in its position. The second part of the patch involved replacing the static left values with fluid ones. These moved the labels closer at narrow viewport widths in order for them all to fit inside the viewport window. Figure 7.10 illustrates the result of the patch.

```
<div style="position: absolute; left: 59px;">Aug</div>
<div style="position: absolute; left: 103px;">Sep</div>
<div style="position: absolute; left: 146px;">Oct</div>
```

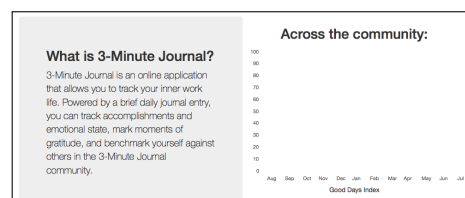
(a) Code before patch

```
<div style="position: absolute; left: 8.5%;">Aug</div>
<div style="position: absolute; left: 17%;">Sep</div>
<div style="position: absolute; left: 25.5%;">Oct</div>
```

(b) Code after patch



(c) Before patch (June and July protruding)



(d) After patch (June and July visible)

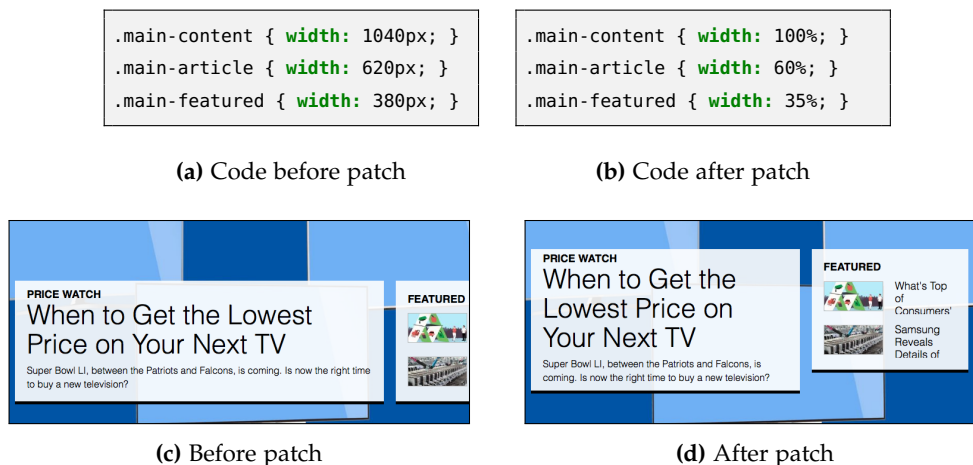
**Figure 7.10:** Fixing the 3-Minute Journal viewport protrusion failure.

The graph protrudes outside of the viewport at both wide viewport widths and then again at much narrower ones following a layout shift. Therefore, two dis-

tinct responsive layout failures were originally recorded for this subject. However, after implementing the aforementioned patch for the web page at wide viewport widths, the narrow viewport failure no longer manifested. This indicates the two failures in fact shared a root cause and were not distinct. This proves that the participants in Chapter 6’s human study were in fact correct on Question 6, while the grouping produced by REDECHECK was incorrect.

### ConsumerReports

At wide viewport widths, the “Price Watch” and “Featured” tiles easily fit side-by-side in the main content banner of the page. However, as the viewport starts to narrow, the “Featured” tile starts to protrude outside of the viewport, obscuring most of its content. Although a user could scroll horizontally to view the rest of the content, this seems to be a poor design choice given there are only two elements. Also, many web users nowadays do not expect to scroll horizontally. They may therefore not realise they have the ability to do so and disregard the obscured information.



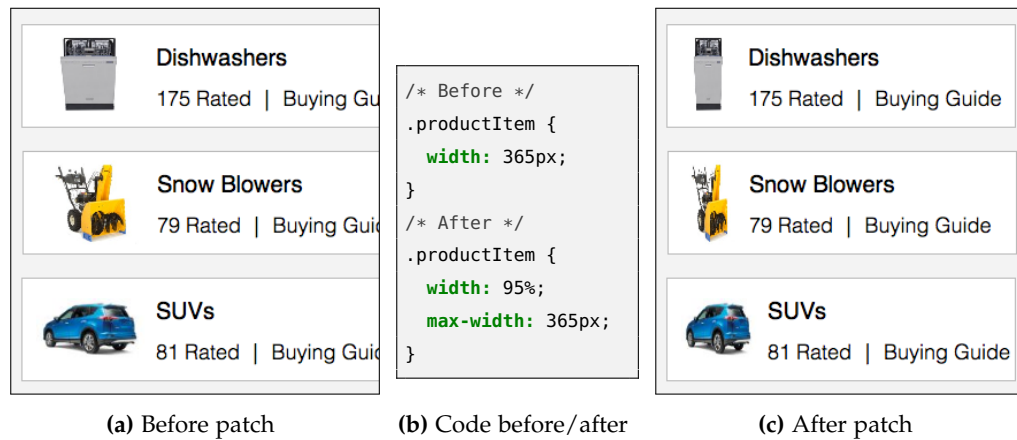
**Figure 7.11:** Fixing the ConsumerReports viewport protrusion failure in the banner.

Debugging found the root cause to be the use of static widths for both the banner itself and its contents. Therefore, the implemented patch modified the width declarations to use fluid values instead. The banner was given a width of 100%, while the “Price Watch” and “Featured” elements were given widths of 60% and 35%, respectively. Figure 7.11 shows the effects of the patch.

Another viewport protrusion occurs further down the page. Here, the “Featured Products” tiles begin to protrude outside the viewport at narrow widths. Initially this means just obscuring the white space on the right hand side of the tiles. However, as the viewport narrows further text begins to become obscured.



Given all six of the tiles protrude outside of the viewport, it is highly likely the root cause lies with a rule applied to all of the tiles.



**Figure 7.12:** Fixing the ConsumerReports viewport protrusion failure in “Featured Products”.

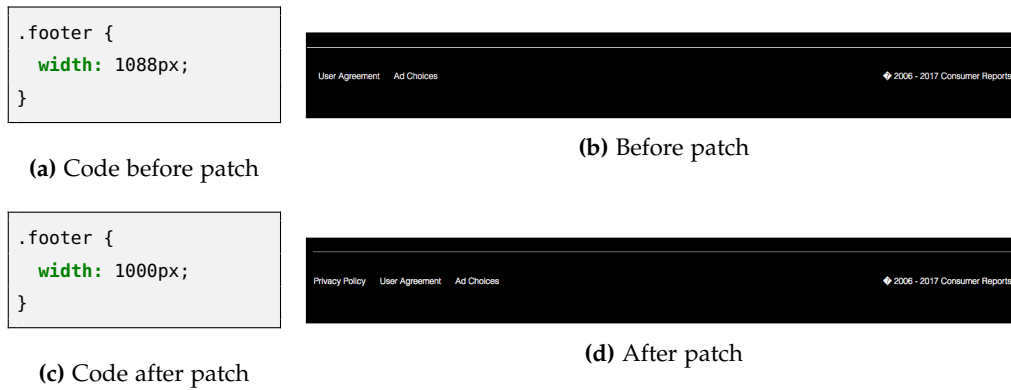
Debugging found the issue to be each of the tiles having a static width of 365px. Intuitively, at very narrow viewport widths (i.e., less than 365 pixels) this causes part of the tile to protrude outside of the viewport. The implemented patch, used a combination of a fluid width declaration (95%) and a `max-width: 365px` declaration. This ensures the elements always fit within the viewport window without becoming too wide for the desired layout. Figure 7.12 illustrates the details and effects of this fix.

There is one final viewport protrusion failure manifesting on ConsumerReports. At wider viewport widths, the navigation links in the footer fit comfortably within the viewport window. However, at narrower viewport widths one of the links (Privacy Policy) is fully obscured as the list protrudes out of the left side of the viewport. Given the nature of the failure, it is likely the failure’s root cause lies with the offending element, rather than with one of its neighbours.

Debugging found the footer had a static width declaration which was often wider than the viewport width. This naturally caused the viewport protrusion. The implemented patch was therefore very straightforward. This width was simply reduced to one smaller than the viewport, as shown by Figure 7.13.

## Duolingo

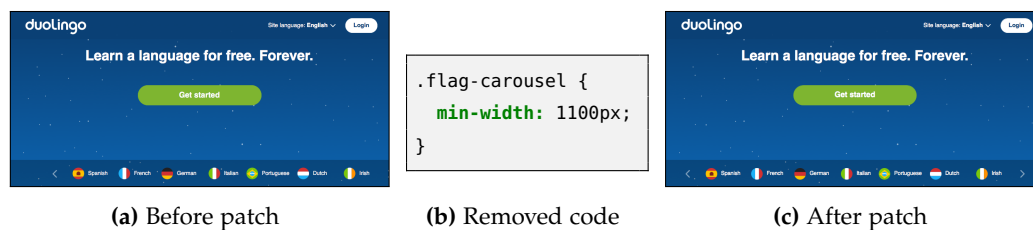
At wide viewport widths, the carousel of different languages easily fits in the main content banner of the page. However, as the viewport starts to narrow, the right navigation arrow starts to protrude outside of the viewport. Eventually, it



**Figure 7.13:** Fixing the ConsumerReports viewport protrusion failure in the footer.

becomes almost unclickable. As the carousel is the only element in the vicinity which is protruding outside of the viewport, the root cause is likely a CSS rule applied to the carousel itself.

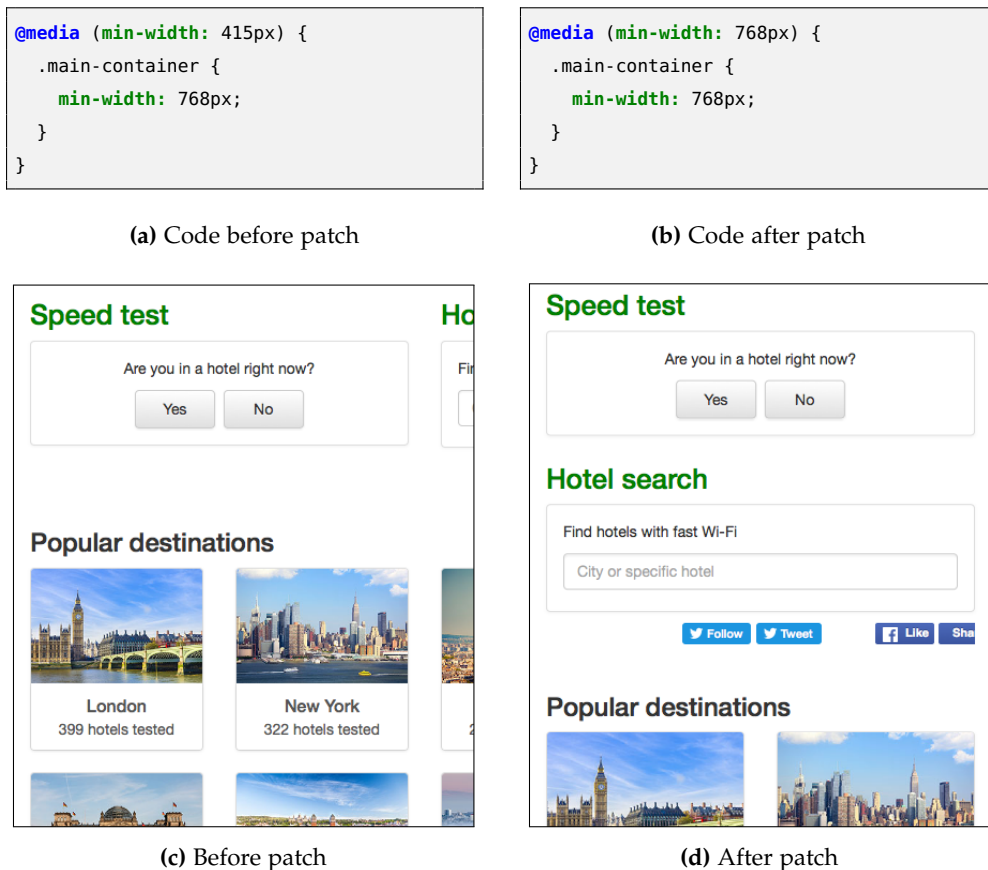
Debugging of the failure showed the carousel already adhered to the fluid grid sizing concept of RWD, with its width set to be 100% of its parent. However, it also had a `min-width: 1100px;` property applied. This caused issues with the web page’s layout when the viewport width was narrower. The patch removed this declaration, allowing the carousel to scale its width up and down in response to a changing viewport width, as shown by Figure 7.14.



**Figure 7.14:** Fixing the Duolingo viewport protrusion failure.

### Hotel WiFi Test

At narrow, mobile-device widths, the main content panels of the web page are in two columns. However, beyond the viewport width of 414px, this shifts to a 4 column layout. Unfortunately, the viewport is not anywhere near wide enough to accommodate the items. This causes almost half of the main content of the web page to protrude outside of the viewport. The nature of the failure indicated the root cause was likely a media query controlling the change from “mobile” to “desktop” layouts.



**Figure 7.15:** Fixing the Hotel WiFi Test viewport protrusion failure.

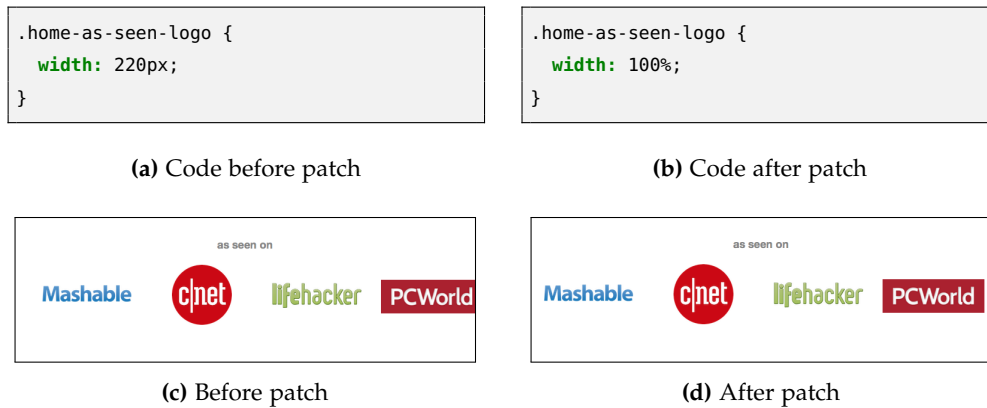
Indeed, debugging found the offending condition, (`min-width: 415px`). This tried to use the desktop layout when the viewport width was still very narrow. The implemented patch modified this condition to (`min-width: 768px`), quickly fixing the problem, as illustrated by Figure 7.15.

### PDFescape

At wide viewport widths, the page displays the row of company logos side-by-side. However, as the viewport starts to narrow, the “PCWorld” logo starts to protrude outside of the viewport. While this does not hamper functionality, the visual appearance is sub-optimal. There is clearly supposed to be white space on the right hand side of the row of logos. Instead, the viewport obscures part of one of the logos.

Debugging found the root cause to be the use of static widths for each of the four logos. Each had the property `width: 220px`. Therefore, at viewport widths such as 768 pixels, the combined width forced the right-most logo outside of the viewport. The patch replaces these static width declarations with fluid ones

(width: 100%). This meant the elements scaled their width correctly in response to different viewport sizes, as shown by Figure 7.16.



**Figure 7.16:** Fixing the PDFescape viewport protrusion failure.

#### 7.2.4 Small-Range Layout Failures

##### Accountkiller

At narrow viewport widths, the main content tiles are in a straightforward two-column layout. Once the viewport width increases to wider than 480 pixels, a media query increases the width of the tiles, while retaining the two-column design. However, for a small range of viewport widths (476 pixels to 480 pixels) the tiles shift into a three-column layout. This has no impact on functionality, but it is likely something the developer will want to address. The preceding and following layouts both implement two columns of tiles, indicating the layout shift is unintentional.

Debugging of the failure found the root cause to be the breakpoint of the media query controlling the change in width of the tiles. At the offending widths, the elements are narrow enough to fit three to a row inside the container. Then, the media query widens them and forces them back to a two to a row layout. Therefore, the implemented patch changes the breakpoint so that the widths increase before they incongruously wrap, as shown by Figure 7.17.

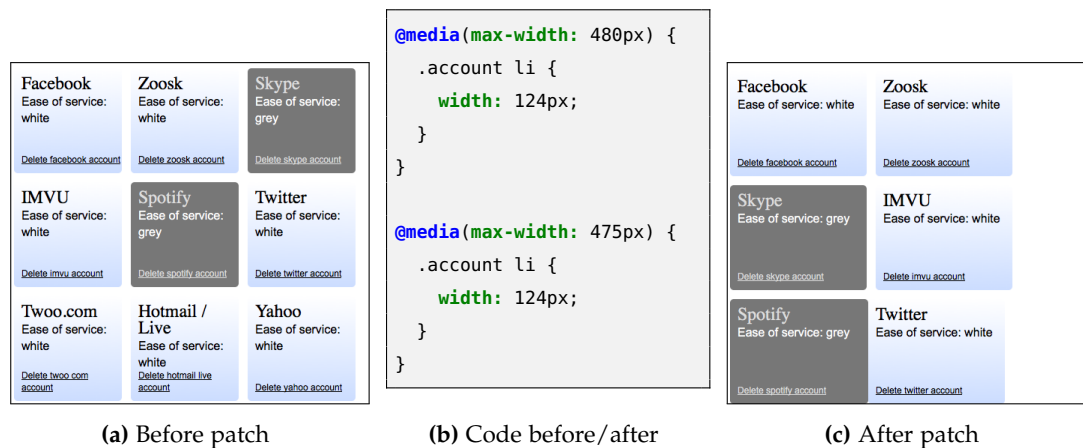


Figure 7.17: Fixing the Accountkiller small-range layout failure.

### 7.2.5 Wrapping Failures

#### Accountkiller

When the viewport window is wide enough, all the logos are in a single row. However, when the viewport narrows, the logo “The Atlantic” wraps onto a new line. Eventually the elements all wrap so they are in 2 rows of 3 and then 3 rows of 2. However, this occurs due to incongruous wrapping rather than developer-programmed responsive behaviour. The implemented patch therefore implemented a media query which automatically shifted the layout of the six logos. It changed to a 2 row layout as soon as the viewport became too narrow for a single row layout. It then shifted to a 2 column layout and then finally a single column layout as the viewport continues to narrow. Figure 7.19 illustrates both the added CSS code and its visual effect on the web page.

The web page also exhibits a second wrapping failure, with the “Tweet” icon wrapping onto a new line as the viewport narrows. As the viewport continues to narrow, more icons wrap onto the second row. Therefore, it was important to design a patch which would stop all of the elements from wrapping. The patch began by removing unnecessary padding from around elements such as the Google+ icon, to give the icons a more uniform layout. Then, a media query was introduced for narrow viewport widths which reduced the width of the container holding all the icons. This laid them out in two equal rows, to prevent any further wrapping failures and give the web page a more consistent look. Figure 7.19 illustrates the effects of the patch.

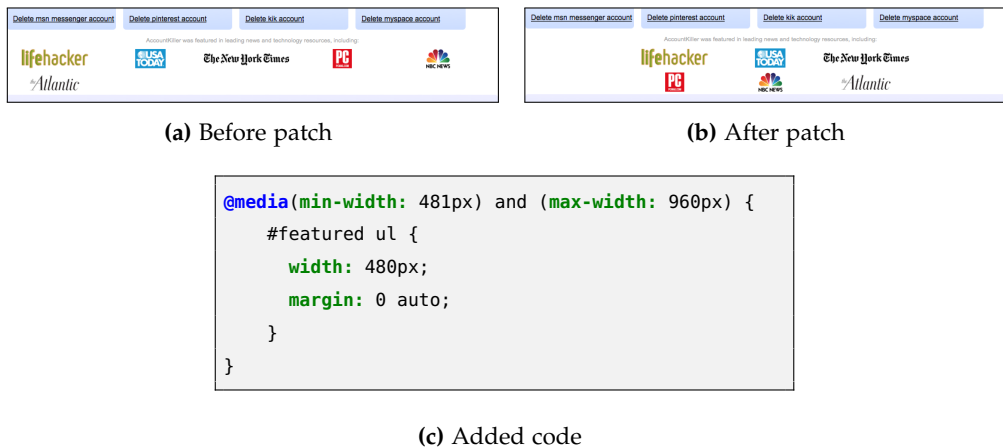


Figure 7.18: Fixing the Accountkiller wrapping failure.

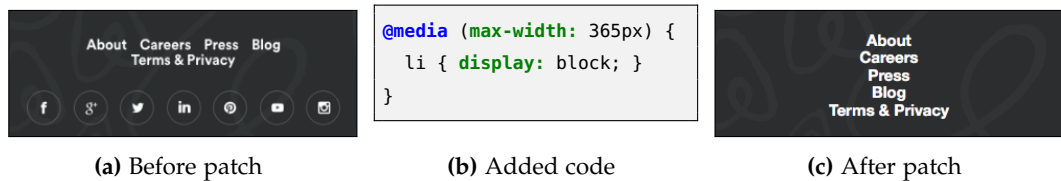


Figure 7.19: Fixing the Accountkiller wrapping failure in the footer.

## Airbnb

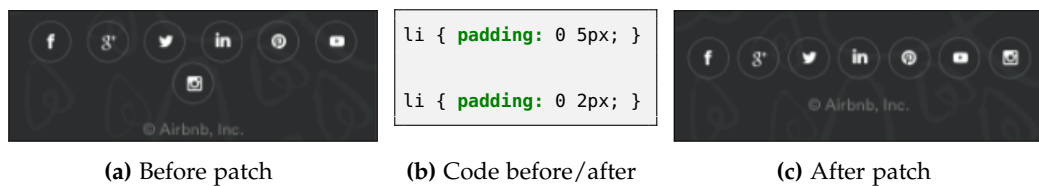
At narrow viewport widths, the “Terms & Privacy” navigation link wraps onto a new row, while the rest of the links are side-by-side. Obviously this is a purely aesthetic issue and has no impact on the functionality of the web page. Even so, it still may be something the developer wants to address. Debugging found the cumulative width of the elements to be considerably wider than their container. The obvious patch would be to reduce the element widths and the spacing between them, to squeeze all five links into a single row. However, this patch was unsuccessful due to the already small spacing between the elements. Therefore, the implemented patch instead added a media query. At narrow widths, it stacked the navigation links vertically. This results in a more elegant

responsive design often implemented by developers when faced with limited horizontal space. Figure 7.20 presents this fix.



**Figure 7.20:** Fixing the Airbnb wrapping failure with “Terms & Privacy”.

At narrow widths, the Instagram logo is also forced onto a new row in a similar manner. In this case, debugging found that reducing the padding applied to the sides of each icon from 5px to 2px achieved an almost identical layout, without causing the final logo to wrap. There is the risk that when browsing on mobile devices elements that are too close to each other are hard to interact with, but the spacing between these elements should be enough for a high level of usability. Figure 7.21 illustrates the patch.

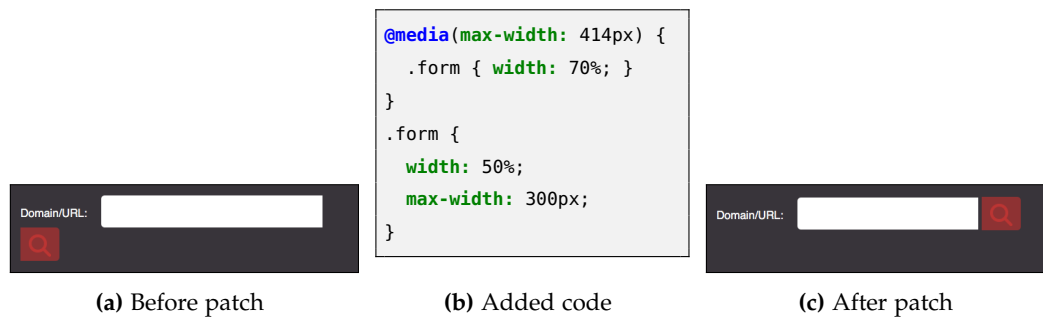


**Figure 7.21:** Fixing the Airbnb wrapping failure with the Instagram logo.

## BugMeNot

The search button wraps onto a new line as the viewport narrows, giving the web page a much less professional look and feel. Intuitively, the root cause is likely to be either the form itself or one of its components. Indeed, the root cause turned out to be the static width of the input box. At narrow viewport widths this caused the cumulative widths of the form components to be too wide to fit side-by-side. The implemented patch therefore introduced a fluid width declaration at both narrow and very narrow viewport widths. To prevent the input box becoming too wide at larger viewport widths and to keep with the initial layout of the page, a `max-width: 300px` declaration was also added to the element. Figure 7.22 illustrates the patched version of the web page.

Originally, the input box of the form also protruded outside of the viewport at very narrow viewport widths. This failure was originally classified as distinct from the aforementioned wrapping failure. However, the patch to stop the

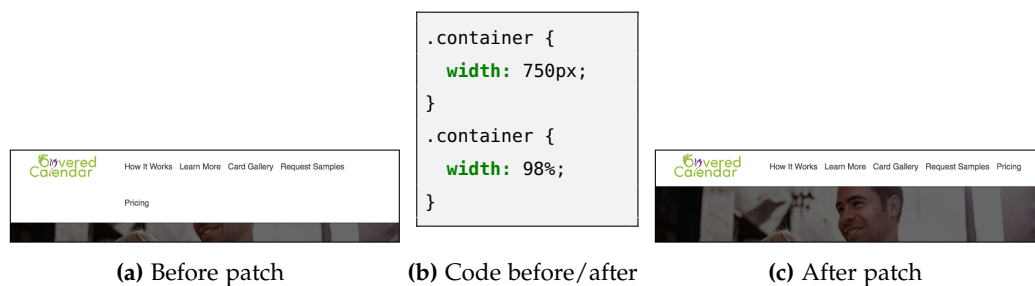


**Figure 7.22:** BugMeNot viewed at a viewport width of 500 pixels.

search button from wrapping also removed the viewport protrusion. This indicates that the two failures in fact shared a root cause and were not distinct. Like the viewport protrusions on 3-Minute Journal, this again shows that the decisions of the participants in the previous chapter’s human study were correct, while REDECHECK was incorrect.

### CoveredCalendar

The “Pricing” navigation link in the header bar wraps onto a new line as the viewport becomes narrow. Given the failure manifests right at the top of the web page, it could negatively influence visitors straight away. Therefore, it should be a high-priority for the developer. The nature of the failure suggests the root cause lies with the element containing the header bar. This is evidently not always wide enough to accommodate the desired single-row layout.

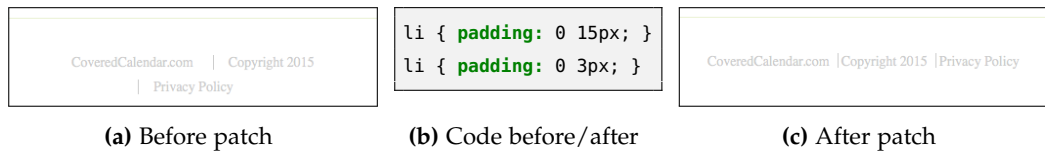


**Figure 7.23:** Fixing the CoveredCalendar viewport protrusion failure in the header.

Debugging of the failure showed that when the header switches from its mobile view to its desktop counterpart, the container element of the logo and navigation links has a fixed width of 750px. This is not wide enough for the intended layout. Therefore, the patch changed this fixed width to a fluid one of 98%. This ensured there was always sufficient space for the links and logo to fit in a single row, as shown by Figure 7.23.



CoveredCalendar also exhibits another wrapping failure in its footer. The “Privacy Policy” link wraps onto a new line as the viewport becomes narrow. As the vertical bar has also wrapped, it indicates the wrapping is not intended layout behaviour, as the “CoveredCalendar.com” link doesn’t have a bar on its left hand side. Intuitively, the root cause is likely to be the list items themselves or their container.



**Figure 7.24:** Fixing the CoveredCalendar wrapping failure in the footer.

Debugging found the root cause to be padding of 15 pixels on both the left and right hand sides of each list element. This significantly increased the width of the elements, causing the wrapping at very narrow viewport widths. The implemented patch, shown by Figure 7.24 involved simply reducing the padding until the elements fit side-by-side. This is fairly crude, as the footer is no longer as aesthetically pleasing as it was with the larger padding. However, it is still better than the alternative of having one of the links wrap onto a new line. Perhaps a better option, although significantly more involved, would be to use a media query to display the elements in a single column at narrow widths. This would likely produce a more professional visual appearance.

### Ninite

The “terms” link wraps onto a new line as the viewport becomes narrow. Debugging showed the root cause to be a combination of the item container not being wide enough and the spacing between the list items being too large. This meant all the elements could not fit in a single row at narrow viewport widths. The patch consisted of two steps. First, the available space in the list container was increased by reducing the left and right padding down from 15px to 1px. Then, the padding on the individual list items was reduced until the items fit side-by-side. Figure 7.25 illustrates the effects of the implemented patch.

### PepFeed

At wide viewport widths, the four source logos are displayed side-by-side, but the “Verge” link wraps onto a new line as the viewport becomes narrow. While clearly not a major issue, it is something the developer may want to address nonetheless as it does have a slightly detrimental impact on the aesthetics of

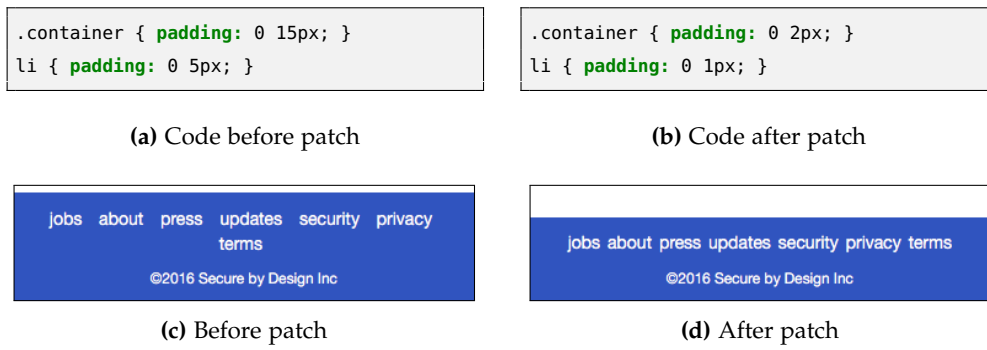


Figure 7.25: Fixing the Ninite wrapping failure.

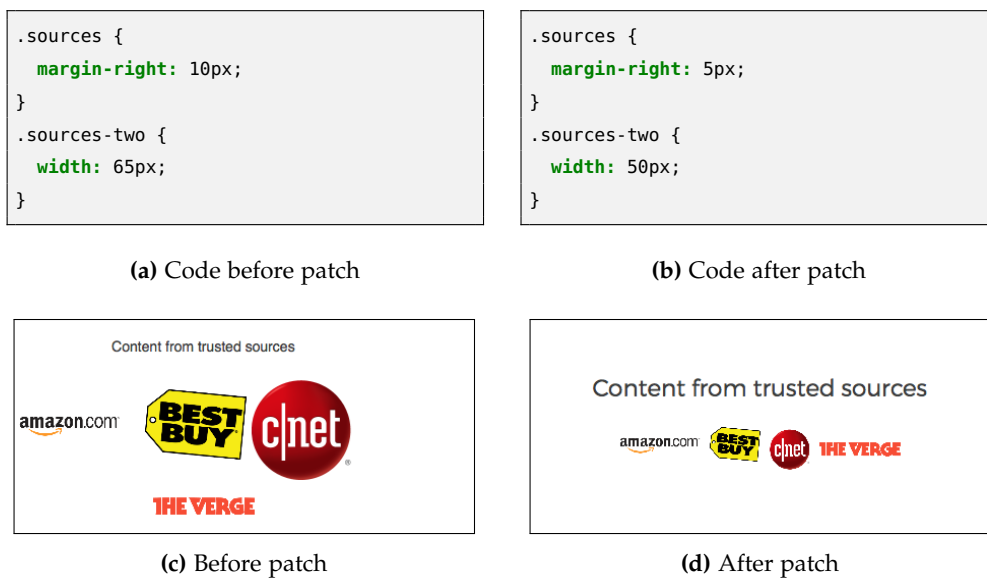
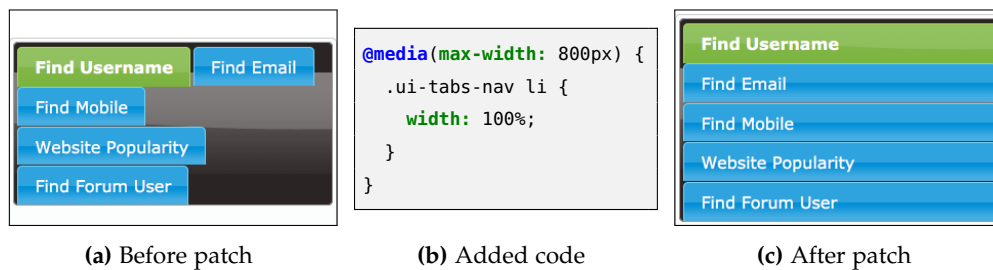


Figure 7.26: Fixing the PepFeed wrapping failure.

the web page. Debugging found the issue to be a combination of the static widths of the four elements and the margins applied to them. Together, this made the cumulative horizontal space required too large for the width of the container of the elements. The implemented fix was therefore simple. It first reduced the width of the elements without making them too small. Then, it reduced the spacing between them without bringing them too close together. Figure 7.26 illustrates the effects of this fix.

### Usersearch

At wide, desktop viewport widths, the five navigation links fit comfortably side-by-side. However, as the viewport narrows, the available space for the links reduces. As such, the navigation links wrap incongruously onto different lines. This creates a significantly less professional looking web page. While it does



**Figure 7.27:** Fixing the Usersearch wrapping failure.

not negatively affect the functionality, many people now expect a mobile-ready “drop-down” menu rather than a desktop menu which has wrapped. Therefore, a developer should consider the failure fairly serious and worthy of fixing.

As one might expect, the debugging process quickly found the root cause to be the navigation elements. These had permanent static widths, meaning at narrower viewport widths they wrapped incongruously. To remedy the issue, the patch introduced a media query with (`max-width: 800px`) as its breakpoint. Inside this, a fluid width declaration `width: 100%;` was applied to the list items. Rather than wrapping, this stacked them in a column, as illustrated by Figure 7.27.

### 7.2.6 Summary of Root Causes

The majority of the RLF root causes presented in this chapter show the majority of both the initial manually created distinct RLF groupings and those automatically generated by REDECHECK were correct. However, there are a couple of instances where RLFs initially thought to be distinct were revealed to share the same root cause.

## 7.3 STUDY RESULTS

**RESEARCH QUESTION ONE:** The previous section showed that the majority of distinct RLF classifications from Chapter 5 were correct. The only incorrect examples were 3-Minute Journal, BugMeNot and PepFeed. For the former, the fault fixing exercise revealed that the 8 TPs reported by REDECHECK actually conflated down to a single distinct RLF. This was despite the failures occurring at very different ranges of viewport widths, with a series of “non-faulty”

widths in between. On BugMeNot, the element protrusion failure and one of the viewport protrusions turned out to share the same root cause. This was also the case for the second viewport protrusion and the wrapping failure. For PepFeed, fault fixing revealed a series of element collision failures and a viewport protrusion that had been classified as distinct were actually all related to the same root cause. These examples will help guide further development in the grouping approach, with the aim of making it more accurate.

Given these incorrect distinct RLF classifications, it is important to re-evaluate the usability of REDECHECK. Chapter 5 explained how the number of distinct viewport ranges was a reliable indicator of the manual effort required to analyse the tool's results. Before the fault fixing experiment, the ratio of viewport ranges to distinct RLFS was 3.4. Given there are actually only 27 distinct RLFS rather than 33, this ratio becomes 4.1. While slightly higher, the benefits of detecting actual layout failures still cancel out the manual effort required by a user.

**Summary of RQ1:** The distinct RLF classifications produced in Chapter 5 were very accurate, with only four incorrect examples. The fault fixing process revealed only 27 distinct RLFS present in the collection of web pages, rather than the initial 33.

**RESEARCH QUESTION TWO:** Following the debugging and subsequent patching of each failure, this question aimed to analyse the complexity of the implemented patches. To do this, the Unix `diff` utility was used to determine the number of *added*, *deleted* and *modified* lines of code by each patch to obtain a metric by which patches could be compared. In some scenarios, using lines of code as a measure for how easy or hard a bug is to fix can be problematic. For instance, in some languages a single line can contain complex boolean logic and therefore identifying the correct modification could be very difficult. However, in HTML and CSS, each line of code generally has a very clearly defined purpose, such as setting the padding for a particular element. This means actually changing the code is much simpler for HTML and CSS than for other languages, allowing lines of code to be used as a proxy for fix difficulty. Another option for measuring relative fix difficulty is the time taken to fix a failure. For instance, more reliable results could be obtained by having a collection of web developers each fix the same RLFS under controlled conditions and analysing the time

required by each one, but unfortunately, time and cost constraints prevented this.

To account for differences in CSS style and to provide a more consistent comparison, the number of individual CSS declarations that were involved in the patch was also calculated, disregarding lines only containing CSS selectors and/or brackets. Then, based on this data and the nature of the patches themselves, each patch was classified into one of four categories; *simple*, *medium*, *complex* and finally *redesign*.

Simple patches are those involving only one or two code changes, such as tweaking the value of a CSS declaration or deleting an unnecessary one. Medium patches are slightly more involved, with 3–5 lines changing such as adding in a new CSS selector block with a couple of declarations inside it. Complex patches are more involved still, with upwards of 5 lines changing. Finally, redesign patches are those where it was deemed there was no feasible fix to the failure and since the developer’s intentions were unknown, the most logical “redesign” of the web page’s layout was implemented. Table 7.1 presents the details and classification of each patch implemented in Section 7.2.

The table shows over half of the patches (14 of 27) were classified as being simple fixes. This highlights the difficulty in implementing reliable and failure-free responsive web pages, as just a single erroneous CSS declaration can have significant effects on the aesthetics of the web page. Interestingly, 11 of these 14 patches involved just changing the value assigned to a property rather than adding further CSS declarations or removing unnecessary ones. This suggests most of the failures in these web pages were due to incorrect application of a CSS property, rather than a failure to apply a particular property. The remaining “simple” patches all involved the deletion of a single CSS declaration which was erroneously being applied to one or more elements. Interestingly, none of the failures were resolved using just the addition of new CSS rules.

Four patches were classified as being of medium complexity. These often involved tweaking the values of several CSS properties rather than just one or two. For instance, the fix for Ninite involved modifying the values of four padding declarations and deleting one margin declaration. In ConsumerReports, the widths of three faulty elements were all modified to resolve a failure.

A further four of the patches were classified as complex, involving upwards of six code modifications. The increased quantity of modifications were caused by the addition of media queries to implement better responsive behaviour or

**Table 7.1:** Summary statistics for the patches implemented for the various failures detected. The labels A, D and C represent additions, deletions and changes respectively. The numbers in parentheses represent the number of individual CSS declarations that were added, deleted or modified.

WEB PAGE	FAULT DESCRIPTION	A	D	C	CLASSIFICATION
Cloudconvert	Banner and header overlapping			2	Simple
MidwayMeetup	Two forms overlapping	6(3)			Medium
PepFeed	Icons overlapping at narrow widths			1	Simple
PepFeed	Icons overlapping at wider widths	19(10)			Complex
StumbleUpon	Login link and logo overlapping			3	Medium
Will My Phone Work	Header and content overlapping			1	Simple
BugMeNot	Form protruding out of header	13(5)			Redesign
ConsumerReports	Text protruding outside of tile		3(1)		Simple
PDFescape	Navigation links protruding out of header			2	Simple
3-Minute Journal	Graph labels protruding out of viewport			14	Complex
ConsumerReports	Banner protruding out of viewport			3	Medium
ConsumerReports	Featured products tiles out of viewport	1		1	Simple
ConsumerReports	Footer link out of viewport			1	Simple
Duolingo	Carousel out of viewport		1		Simple
Hotel WiFi Test	Main content out of viewport			2	Simple
PDFescape	PCWorld logo out of viewport			1	Simple
Accountkiller	Incrongrous layout from 476-480px			1	Simple
Accountkiller	Atlantic wraps	6(3)		1	Redesign
Accountkiller	Tweet wraps	5(2)		1	Redesign
Airbnb	“Terms” wrapping	5(2)			Redesign
Airbnb	Instagram logo wraps			1	Simple
BugMeNot	Search button wraps	7(4)			Complex
CoveredCalendar	Navigation link wraps in header			1	Simple
CoveredCalendar	Link wraps in footer			1	Simple
Ninite	Link wraps in footer		1	4	Medium
PepFeed	The Verge logo wraps			6	Medium
Usersearch	Navigation links all wrap incongruously	5(2)			Redesign

modifications to the CSS rules for several neighbouring elements. For example, the wrapping failures observed on BugMeNot required two additional CSS properties be applied at all viewport widths along with an additional media query containing a specific width declaration. 3-Minute Journal is an example of the other scenario, where the graph canvas and all twelve of the individual x-axis labels had to have a CSS property changed in order to fix the observed viewport protrusion failure.

Finally, five of the patches were placed in the *redesign* class. These failures required more careful planning to patch. Rather than simply modifying code immediately, the most sensible design approach had to be determined before

implementing the fix. An excellent example of this is *Usersearch*, in which the navigation links wrap incongruously as the viewport narrows. Here, the chosen fix implemented a different layout, with the links stacking in a single column. Table 7.1 also showed another interesting result. Redesign patches often required substantially fewer code modifications than medium or high complexity ones. This indicates in certain scenarios it is actually easier to change the responsive layout of the web page, rather than tweak the existing layout until the failure no longer manifests.

The most complex patch implemented consisted of 19 additional lines of CSS code being added to PepFeed, of which 10 were CSS declarations. While this may seem a large amount in comparison to the other patches, it is still not a very large patch size when compared against the overall size of the web page's code base. The size of the patches also shows that actually fixing the observed layout failures is not an overly-complex task. This further highlights the value in reporting them to the developer, as they can often quickly fix them. It also makes it more surprising that the failures manifest on live sites in the first place, given the simplicity with which developers can fix them. It is therefore clear that accurate detection of responsive layout failures is a task web development professionals really do struggle with.

**Summary of RQ2:** Over half of the patches required to fix the failures were simple, with only one or two lines of code modified. This suggests many RLFs arise due to a single programming error from a developer. Of the remaining failures, 5 were classed as "medium", 3 as "complex" and 5 were "redesigns". The largest fix required just 19 lines of additional code, showing that fixing the failures is not necessarily the problem, detecting them in the first place is.

### 7.3.1 Discussion

Following the analysis of both the failures observed and the patches implemented, they were grouped in an attempt to extract various lessons which could be learnt and applied by web developers to help prevent against the wide variety of responsive layout failures observed in our collection of live web pages.

*Using Static Widths:* The first key principle of responsive web design from Marcotte [96] was to use fluid, grid-based layouts (i.e. percentage widths for elements) rather than the more traditional static widths used previously. Failure to do this resulted in a wide array of failures, including elements congruously wrapping, colliding and protruding outside of their container or even the viewport itself. In most cases, the implemented patches involved making the offending elements use fluid widths rather than static ones. In fact, eight of the implemented patches consisted of changing static declarations into fluid declarations to improve the web page's responsiveness. If developers take special care to use fluid widths when building responsive web pages, they stand to substantially mitigate the risk of layout failures. One obvious way of doing this would be to use one of the many front-end frameworks such as Bootstrap [19] or Foundation [161] which come prepackaged with a fluid grid-based layout. However, as some developers may wish to write their CSS from scratch for a more bespoke look, simply ensuring the use of fluid widths whenever possible should be sufficient.

*Lesson One: Always use fluid width declarations when possible to ensure web page element scale their widths to that of the viewport.*

*Media Query Problems* As the number of CSS declarations and media queries used to control the layout of a page increases, it can quickly become difficult for developers to track the breakpoints at which each media query "toggles". As such, throughout this analysis two major pitfalls were observed when it comes to media queries. The first occurs when *two* related media queries equate to *true* at a particular viewport width when only one should, while the second arises when two media queries which should have identical breakpoints don't.

Making sure media queries apply the right rules at the right time can be especially difficult if the developer is using a combination of the `min-width` and `max-width` conditions. Cloudconvert is a good example of this, where the "desktop" layout is introduced with the media query (`min-width: 980px`), meaning the rules for the layout at narrower viewport widths should only be applied at viewport widths up to and including 979px. However, the offending media query contained the condition (`max-width: 980px`), meaning that at 980px both conditions equated to true, resulting in the wrong rules being applied to some elements and thus producing the failure shown by part a) of Figure 7.1. To mitigate the risk of these errors occurring, developers should inspect the layout of the web page at these breakpoints, as any "clashing" media queries would likely be easily detected when viewed at these viewport widths. Many frame-



works also predominantly use the `min-width` condition to try and make it easier for developers to interpret which media queries are applied. However, this also has a downside as a mistake in the order in which the CSS declarations are placed in the file can impact how a web browser parses the CSS and therefore have an effect on the appearance of the web page.

When two media queries controlling the layout of related or neighbouring elements have different breakpoints, undesirable aesthetic effects can often manifest at viewport widths at which only one rather than both of the queries equates to true. A textbook example of this is *Will My Phone Work*, where two queries should both have had the condition (`min-width: 992px`), but one had (`min-width: 990px`) instead. At 990px and 991px, the media query which displayed the navigation links in a row was true, but the one which increased the width of the navigation bar to accommodate the links was not, resulting in the effects shown by part a) of Figure 7.6. Although the implemented patch consisted of removing a `max-width` declaration, the alternative was to change the breakpoints so both media queries “toggle” at the same point. This failure clearly illustrates how careful developers have to be when implementing their web sites to ensure the correct rules are applied at the right viewport widths, as seemingly small errors can cause significantly detrimental issues. One approach to this could be performing “boundary” testing, checking a couple of pixels either side of each planned breakpoint to make sure all the media queries become true or false at the correct viewport widths.

*Lesson Two: Always check the media queries used to control the layout. Make sure that “related” queries have the exact same breakpoint and that only one query controlling a particular element is true for each viewport width.*

Some of the failures observed were due to the developer trying to implement a “desktop” layout on devices with small viewport widths, which can often be a recipe for disaster. For example, *PepFeed* attempted to display four elements side-by-side at viewports as narrow as 415px, whereas the more sensible approach was to keep the elements in the stacked layout, as shown by Figure 7.3. The advice for developers would be to only change to a desktop style layout when they know there will definitely be enough available space to do so. Mobile-friendly layouts often require a reasonably large amount of vertical scrolling due to their narrower nature, but given the ease of scrolling on modern mobile devices, this is unlikely to be seen as a negative by the vast majority of users. To test for these types of issues, it is important for developers to ob-

serve the layout of a web page at a viewport width between those normally associated with mobile and desktop devices.

*Lesson Three: Only attempt to switch a group of elements to a desktop layout when there is guaranteed to be enough room to do so. When in doubt (for instance, at tablet widths), staying with a mobile layout is a perfectly satisfactory choice.*

#### 7.4 CONCLUDING REMARKS

This chapter investigated the root causes of the 33 distinct RLFs identified in Chapter 5, and then detailed the code fixes implemented to fix them. Then, it evaluated the fixes in the context of three main research questions.

The first of these evaluated the correctness of the “distinct” RLF classifications from Chapter 5. The fix results showed that for the vast majority of web pages the classifications were correct. However for three web pages, failures that were originally classified as distinct were revealed to actually stem from the same root cause in the source code. This meant that the number of distinct RLFs identified dropped from 33 to 27.

The second question investigated the complexity of the implemented fixes. Over half of the failures could be fixed by changing just one or two lines of code, indicating that many RLFs stem from just a single mistake on the part of the developer. The most involved fix still required just 19 lines of additional code, showing that actually fixing RLFs is a relatively simple process.

Finally, this chapter analysed the set of fixes as a whole to see if there were any common mistakes that developers regularly make. From this, it then presented three main lessons that developers can use to mitigate the risks of layout failures manifesting in their responsive web pages.

---

## CONCLUSIONS AND FUTURE WORK

---

### 8.1 SUMMARY OF ACHIEVEMENTS

This thesis set out to tackle the important problem of presentation failures in responsive web pages, which have been found to have seriously detrimental impacts on not just the aesthetics of web pages but also the psychology of the users browsing them. This required a number of challenges to be addressed:

1. Develop a method for representing the responsive layout of a web page.
2. Investigate whether this representation can be used to detect changes to a web page's layout.
3. Develop a method for detecting common types of layout failure without an explicit oracle.
4. Further remove the effort burden from the user by developing a technique for automatically grouping related issues together.
5. Investigate the root causes of presentation failures in real-world responsive web pages to determine whether they are complex to fix and if there are common mistakes made by developers.

#### 8.1.1 *The Responsive Layout Graph*

Chapter 3 described and formalised the *responsive layout graph (RLG)*, a model of the dynamic layout behaviour of modern responsive web pages. The model handles the two main aspects of responsive behaviour, namely, the changing *visibility* and *alignment* of elements on the page. The chapter then presented a series of algorithms for obtaining the RLG of a given web page. While others

have proposed models of web page layout, such as the alignment graph [123] and the layout graph [7], they model the web page at a single viewport width and do not take into account any responsive behaviour. The RLG therefore represents an advance over these existing models and forms a novel contribution of this thesis.

### 8.1.2 *Detecting Potentially Unseen Layout Side-Effects*

Following on from the introduction of the RLG in Chapter 3, Chapter 4 proposed an approach for automatically alerting developers to potentially unseen side-effects in the responsive layout of a web page, following a code change. This approach first obtained the RLG for both the “original” and “modified” version of the web page under test, before comparing them to detect any differences. A report of these changes was then output to the user, who could use it to determine whether any unintentional side-effects are present.

The approach was implemented in REDECHECK, an open-source responsive web checking tool. It was then evaluated on a pool of 15 real-world responsive web pages, using mutation analysis to automatically seed code changes. Results showed the approach capable of detecting the vast majority of injected layout changes, outperforming both manual and automated baseline techniques. To the best of my knowledge, this is the first automated approach addressing presentation failures in responsive web pages and therefore represents the second novel contribution of this thesis.

Unfortunately, the approach suffers from a couple of shortcomings. Firstly, there is still quite a significant burden of effort on the developer, as they must manually determine whether any of the layout changes represent real issues. Second, the approach is far less usable if the “original” version of the web page is too far removed from the latest one, as it would likely report an overwhelming number of layout changes. Finally, if a presentation failure is present in both versions of the web page, the approach will be unlikely to bring it to the attention of the developer.

### 8.1.3 *Detecting Common Types of Responsive Layout Failures*

Addressing the shortcomings of the approach in Chapter 4, Chapter 5 presented an approach for identifying five types of *responsive layout failure (RLF)* without the need for an explicit oracle, such as a previous version of the web page. Like the approach in Chapter 4, the foundation of this approach was the RLG. However, rather than comparing two RLGs, this approach analysed a single RLG, searching for patterns that frequently represent common types of responsive layout failure.

The approach was also implemented into `REDECHECK`, which outputs both a text report and a set of highlighted screenshots showing the detected RLFs on the web page. An empirical evaluation using 26 randomly selected real-world web pages found that RLFs were a prevalent problem and that the approach was capable of detecting them. Results also showed the approach outperformed the common industry technique of spot-checking and that `REDECHECK` can perform its analysis in a short amount of time, allowing developers to quickly obtain useful feedback.

The chapter then proposed a couple of small improvements to the approach. The first changed the RLG definition and the second modified the RLF identification approach to reduce the number of false positive failures reported. These changes were then shown to have been very effective, reducing the number of false positives produced while still detecting the same number of true positive RLFs.

Other techniques have aimed to identify different categories of layout issues in web pages (eg. [61, 62]), but this approach is, to the best of my knowledge, the first to do so without the need of an explicit oracle against which the web page's layout is compared. For this reason, it forms the third major contribution of this thesis.

There was one major shortcoming of the approach. Some web pages produced numerous failure reports that were related and likely stemmed from the same root cause. Grouping such failures together is currently a manual task that reduces the overall usability of the approach.

#### 8.1.4 *Grouping Related Failures Together*

Chapter 6 took the approach proposed in the previous chapter and tried to reduce the effort required by a user of the tool by automatically grouping related failures together. This approach used three measures of similarity to work out whether different RLFs were related.

The evaluation found the automated grouping to be highly effective, observing only a few incorrect groupings. Finally, a study of 11 computer science post-graduate and undergraduate students investigating human opinion on both individual RLFs and groupings of them showed the majority of humans agree both with the failures identified and the groupings produced by the approach presented in this chapter. However, there were a few disagreements, in which the participants disagreed with both the manually produced and automatically produced results.

#### 8.1.5 *A Study of Real-World RLFs*

Chapter 7 investigated the root causes of the RLFs identified in Chapter 5. It presented potential fixes for all of the 33 distinct RLFs identified and then presented a discussion of the failures. This allowed further investigation of the manually created distinct RLF groupings, the groupings automatically generated by REDECHECK and the opinions given by the participants in Chapter 6's human study.

The study found several points of interest. Firstly, the manual classifications were very accurate, with only four incorrect examples, including a couple where the human study participants were correct while the gold standard and REDECHECK were in fact, wrong. Secondly, the majority of the RLFs could be fixed with very few lines of code, with only two requiring more than 10 lines of modified code. Finally, developer mistakes generally fall into one of three main categories, including ensuring all elements on the web page adhere to the three ingredients of RWD. This chapter then presented "lessons" web developers could potentially use to mitigate the risks of responsive layout failures occurring in their web pages, which could in turn improve the overall web browsing experience for everyone.

## 8.2 LIMITATIONS AND FUTURE WORK

This section outlines various potential avenues of future work that further explore the problem of responsive web page layout testing. It mainly focusses on work following on from the specific approaches investigated in this thesis: detecting potentially unseen layout side-effects (Chapter 4), and identification of responsive layout failures (Chapters 5-7).

### 8.2.1 *Further Investigation of Web Page Mutants*

Section 2.4.2 introduced mutation analysis, a method of introducing faults or changes into the source code of an application. These changes, or mutants, generally try to model mistakes or changes a real world developer might make. The set of eight operators proposed in Chapter 4 target common CSS and HTML constructs used to control responsive layout, but it is vital they model the typical developer changes. They are based upon personal intuitions regarding the types of changes made in practice, based on my knowledge of responsive web development. Currently, I have not evaluated to what extent the changes introduced by the operators reflect real-world changes.

One possible way to investigate this would be to run a human study simulating a real-world development scenario, such as the usage scenario proposed for the approach in Figure 4.2. The participants could be given a responsive web page and a task to complete, such as changing some aspect of the web page's layout. Then, by tracking the source code changes made by each developer, I could see whether developers frequently made layout changes similar to those produced by the mutation operators. This study could also point out other potential mutation operators that could be implemented, allowing for an even more in-depth empirical evaluation of the overall approach.

### 8.2.2 *Identification of Further Types of RLF*

Chapter 5 introduced five types of common responsive layout failure (RLF) identified through my own experiences of browsing the mobile web. These were element collision, element protrusion, viewport protrusion, small-range layout and element wrapping. As discussed previously in that chapter, it is

possible that other types of RLF are prevalent in real-world responsive web pages. Anecdotally, I observed no other types of failure during the evaluations in Chapter 5 and Chapter 6. However, some participants in the human study in Chapter 6 reported RLFs that did not properly fit in any of the pre-existing categories. For instance, some stated some elements were “misaligned” in relation to others, suggesting that could be one potential failure type not currently supported. As four different algorithms were required to detect the five types of RLF, it is highly likely that a totally new algorithm would be required to detect a new type of RLF, along with changes to the responsive layout information modelled by the RLG.

### 8.2.3 *Automatic Classification of Identified RLFs*

One of the main bottlenecks in the approach presented in Chapter 5 is that a human must manually classify each identified RLF as either a true positive, false positive or non-observable issue. For some web pages, the number of reported failures is small so little manual effort is required. However, for other subjects such as ConsumerReports, Accountkiller, Retail Me Not and Startup Stash, the larger quantities of RLFs means the user has to do considerably more manual analysis in order to reap all the benefits of the approach.

A potentially important avenue of future research therefore lies in the task of automatically classifying RLFs as one of the three categories. Other research has investigated the use of techniques such as machine learning to perform this task ([32, 126, 131]). The manually classified failures from the studies in this thesis could therefore potentially form the beginning of a set of training data for a machine learning classification approach, although many more examples would likely be required to create an effective and accurate classifier. Implementing a technique that accurately classifies failures would likely significantly improve the overall usability of REDECHECK.

### 8.2.4 *Fault Localisation*

This thesis has primarily focussed on identifying failures, which the developer has had to manually classify and fix if necessary. Another way of potentially improving the user experience of REDECHECK and reducing the effort load on the user is to analyse the source code of the web page and “localise” the failure.



In other words, rather than just reporting the presence of an RLF, the approach could point the user in the right direction for fixing the fault by suggesting lines of code that could be responsible for the failure. For example, `WEBSEE`, the tool created by Mahajan et al. [90, 91] outputs a list of web page elements that are most likely to be responsible for the presentation failure in the web page. However, given `REDECHECK` already possesses this information (as it knows which elements are involved in each RLF), it could potentially direct the user to specific lines of faulty code.

### 8.2.5 *Automatic Fault Fixing*

One step further than fault localisation would be the implementation of an automated fault fixing approach. This would not only identify the faulty line or lines of code, but would work out how to modify this code to fix the identified RLF. This could be particularly difficult in scenarios when code has to be added, rather than simply modified.

Recent work by Mahajan et al. has used search-based software engineering and other techniques to develop automated approaches for fixing cross-browser issues [92], internationalization presentation failures [93] and mobile-friendly problems [94]. Similar approaches could be used to provide automatic fixes to identified RLFs, further reducing the effort required by the user.

## 8.3 FINAL REMARKS

Providing an easy and enjoyable web browsing experience for users on all devices is vital in the modern world. Unfortunately, doing so is a labour-intensive and error-prone task, meaning presentation failures are frequently found. Although prior research has targetted orthogonal problems such as cross-browser compatibilities, almost no research has addressed the significant problem of identifying presentation failures in mobile-friendly responsive web pages. Furthermore, the task is very difficult to perform manually, due to the huge variety of devices available and the effort required to thoroughly check a web page's layout.

Therefore, this thesis explored techniques for automatically detecting such issues and reporting them to a developer. It began by proposing a model of a

web page's responsive layout. It then used this model as a way of reporting potentially unseen layout side-effects following changes to the web page's source code. Next, it proposed a series of algorithms that analyse this model to identify five different types of common responsive layout failure (RLF). Following this, it proposed a technique for automatically grouping related RLFs together to increase usability. Finally, it investigated the root causes of a set of 33 real-world RLFs to identify commonalities.

Overall, the work contained within this thesis provides two different approaches for checking the responsive layout of web pages. These approaches are both effective and efficient and should therefore aid real-world developers create better responsive web pages and by extension, improve the web browsing experience for all.

---

APPENDIX

---

9.1 SCREENSHOTS FOR PART ONE OF THE HUMAN STUDY

The screenshot shows a mobile app interface with a light blue background and a white dotted pattern. At the top, there are two blue buttons with white text: "Share On Facebook" and "Share On Twitter". Below these buttons is the text "Marbiru makes cool things; be the first to hear about them." followed by "Email Address:" and a white text input field. At the bottom, there is a blue button with white text that says "Keep Me Up To Date!". Below the button, it says "for Ben, by [Uri](#)".

Figure 9.1: Part One - Question One

Content from trusted sources

amazon.com **BEST BUY** **cnet** **THE VERGE**

Learn how Pep became a  
**Fearless Consumer**

Watch how he uses our mobile and browser apps

Figure 9.2: Part One - Question Two







<p>NUMBER OF USERS</p> <p><b>22 MILLION</b></p> <p>POCKET BY THE NUMBERS</p> <p><b>2 BILLION</b></p> <p>NUMBER OF ITEMS SAVED</p>	<p>“ A great option for those interested in saving video, images, text and other content, all in one place.”</p> <p><b>TNW</b> THE NEXT WEB</p>	<p>INTEGRATED IN 1500+ APPS</p> <p>  </p> <p>  </p> <p>MORE APPS »</p>
---	---	--

Figure 9.3: Part One - Question Three

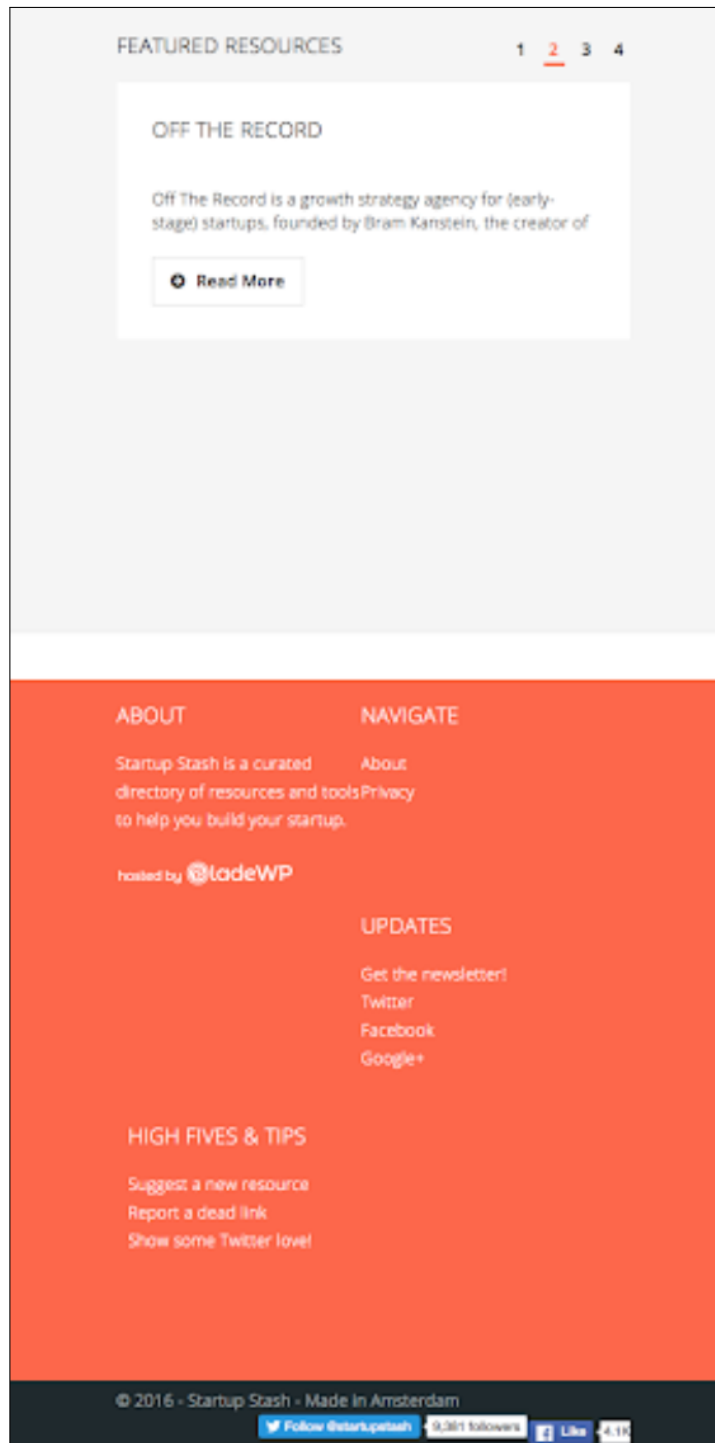


Figure 9.4: Part One - Question Four

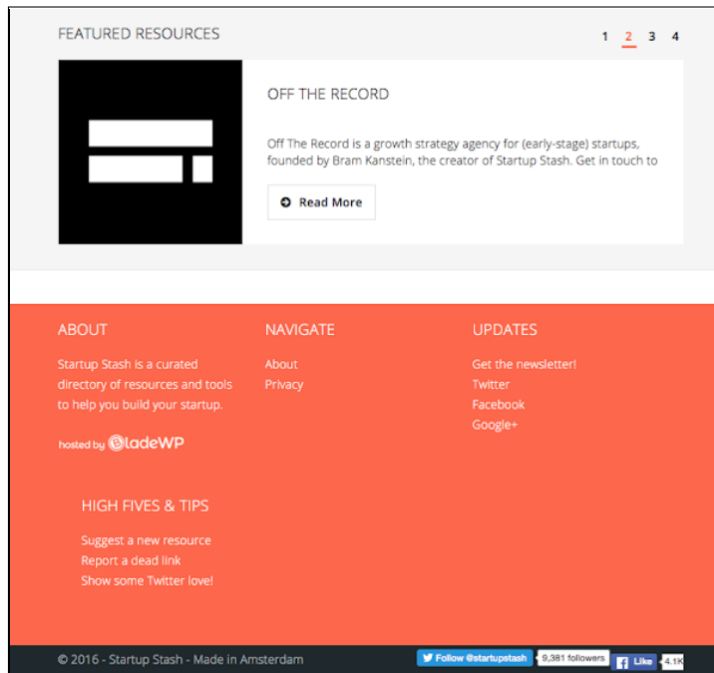


Figure 9.5: Part One - Question Five



Figure 9.6: Part One - Question Six

The screenshot shows a search interface with a dark grey header containing four buttons: 'Find Username' (green), 'Find Email' (blue), 'Find Mobile' (blue), and 'Website Popularity' (blue). Below the header is a 'Find Forum User' button. A 'Search for:' dropdown menu is set to 'Username' with a question mark icon to its right. Below the dropdown is a large text input field with the placeholder text 'Enter username'.

Figure 9.7: Part One - Question Seven

The screenshot shows the homepage of 'Will My Phone Work'. The header is dark blue with the site name 'Will My Phone Work' and a navigation menu with links: 'Home', 'Advance Lookup', 'News', 'Guides', 'FAQs', 'About', and 'Contact'. Below the header is a yellow banner with the date '05 Jul 2016'. The main heading is 'Will My Phone Work?' followed by a paragraph: 'Have you ever asked the question, "Will my mobile phone work with a certain mobile carrier?" Perhaps you're traveling to another country and want to make sure your smartphone works there. You could even be selling a cellular phone and need a website to point people to when they ask if the phone works on a certain carrier. WillMyPhoneWork.net can help you find the answer you're looking for.'

Figure 9.8: Part One - Question Eight

The screenshot shows the footer of the Forvo website. It includes a language selection section: 'Choose your language:' followed by links for 'Deutsch', 'English', 'Español', 'Français', 'Italiano', '日本語', 'Nederlands', 'Polski', 'Português', 'Русский', 'Türkçe', '汉语', and 'and even more languages'. Below this is a navigation menu with links: 'Forvo, the pronunciation dictionary', 'Blog', 'iPhone', 'Tools', 'API', 'Terms and conditions', 'License', 'Privacy', 'About Forvo', 'Contact us', and 'FAQ'. At the bottom left is a blue 'Donate' button with a small icon.

Figure 9.9: Part One - Question Nine

The screenshot shows the homepage of PDFescape. The header is red with the PDFescape logo. The main content area features a laptop displaying a PDF form with fields like 'First Name: Fill Out', 'Last Name: Forms', 'Product Type: Search', and 'Card Type: 100'. To the right of the laptop is the text 'The Original Free PDF Editor & Form Filler Online'. Below this is a dashed box containing the text 'Drop PDF Here (up to 10 MB and 100 pages)'. At the bottom of the dashed box are three buttons: 'Browse...', 'No file selected.', and 'Upload'. Below the laptop are logos for various browsers and operating systems: 'works with Chrome | Firefox 3+ | IE 6+ | Opera 9.5+ | Safari 3+ | Windows'. At the bottom right is the text 'PDFescape Online is Always FREE... No Gotchas!'.

Figure 9.10: Part One - Question Ten

9.2 SCREENSHOTS FOR PART TWO OF THE HUMAN STUDY

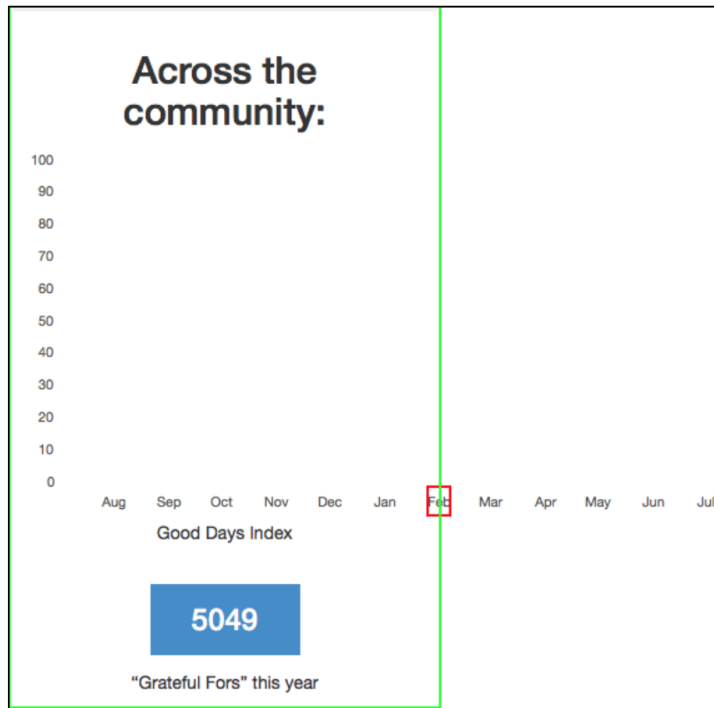


Figure 9.11: Part Two - Group One - Failure One





Figure 9.12: Part Two - Group One - Failure Two



Figure 9.13: Part Two - Group One - Failure Three



Figure 9.14: Part Two - Group One - Failure Four



Figure 9.15: Part Two - Group One - Failure Five



Figure 9.16: Part Two - Group One - Failure Six



Figure 9.17: Part Two - Group Two - Failure One

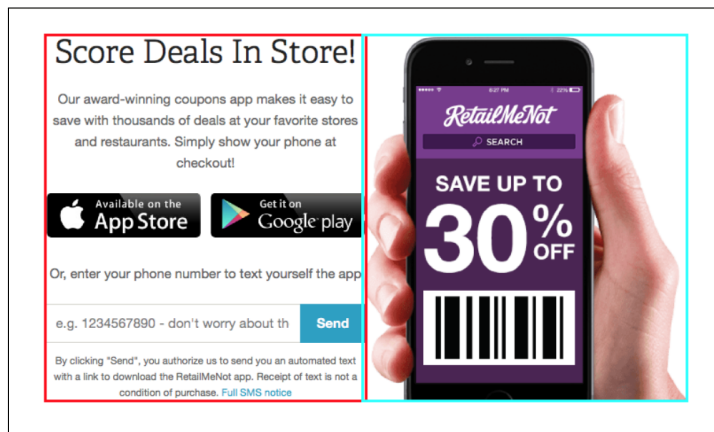


Figure 9.18: Part Two - Group Two - Failure Two



Figure 9.19: Part Two - Group Two - Failure Three



Figure 9.20: Part Two - Group Three - Failure One

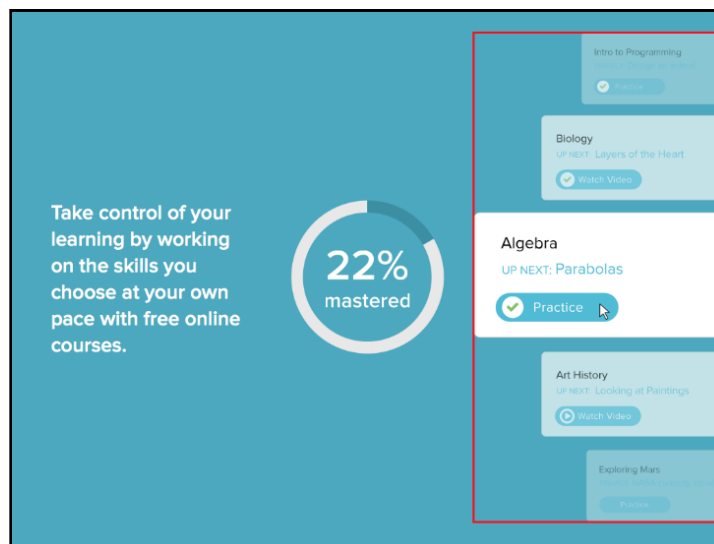


Figure 9.21: Part Two - Group Three - Failure Two

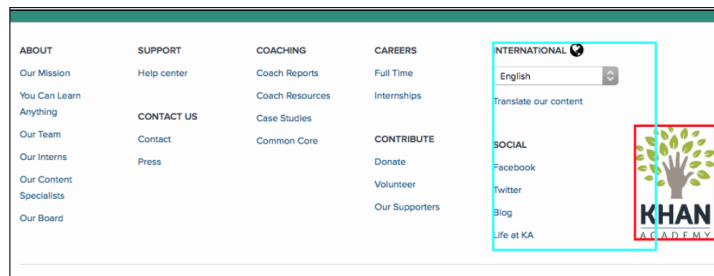


Figure 9.22: Part Two - Group Three - Failure Three

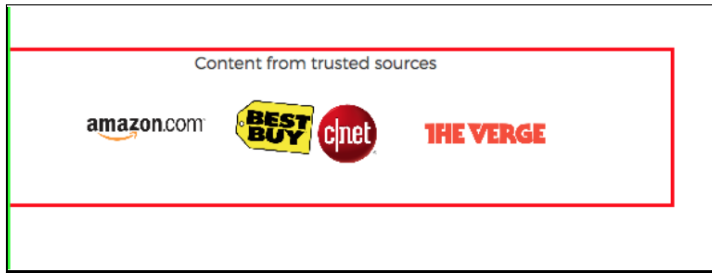


Figure 9.23: Part Two - Group Four - Failure One



Figure 9.24: Part Two - Group Four - Failure Two

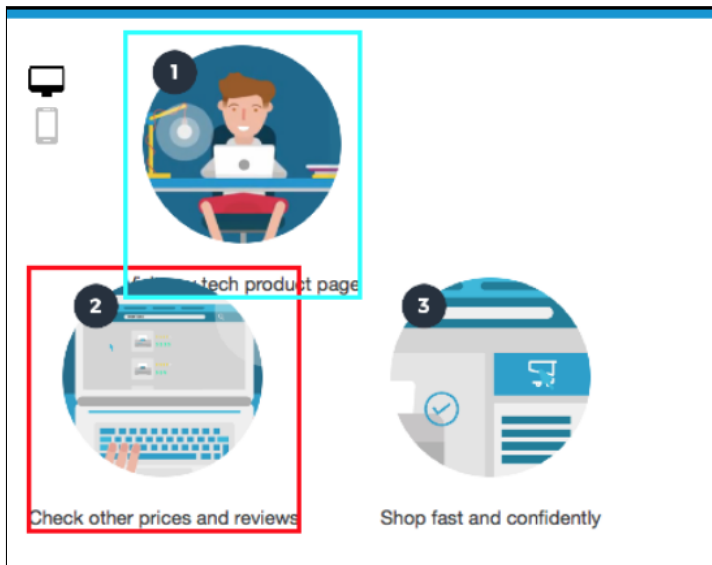


Figure 9.25: Part Two - Group Four - Failure Three



Figure 9.26: Part Two - Group Five - Failure One

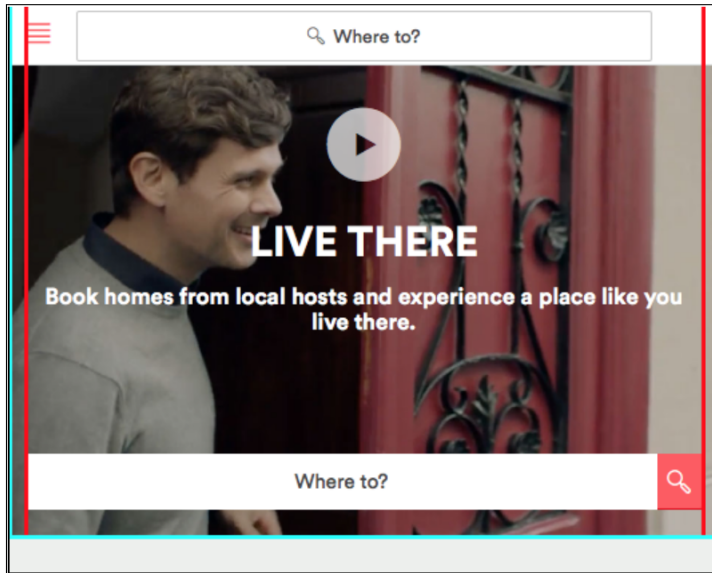


Figure 9.27: Part Two - Group Five - Failure Two

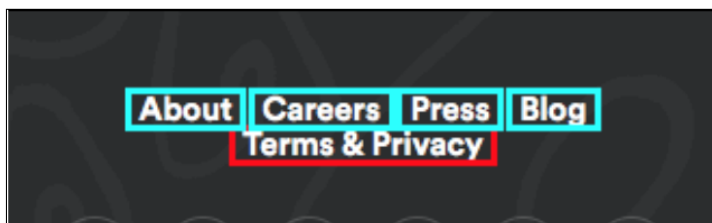


Figure 9.28: Part Two - Group Five - Failure Three



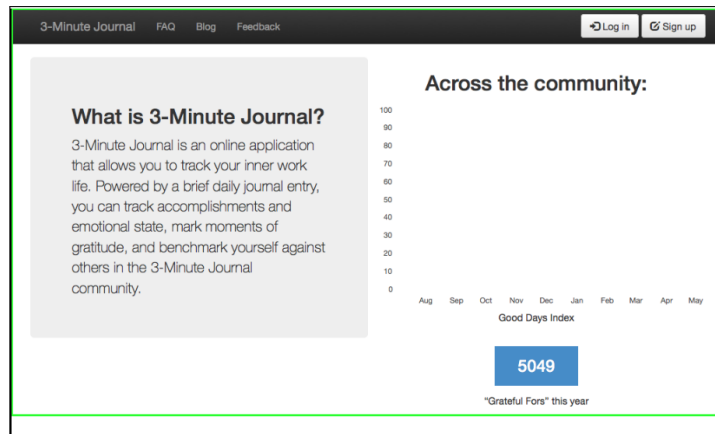


Figure 9.29: Part Two - Group Six - Failure One



Figure 9.30: Part Two - Group Six - Failure Two

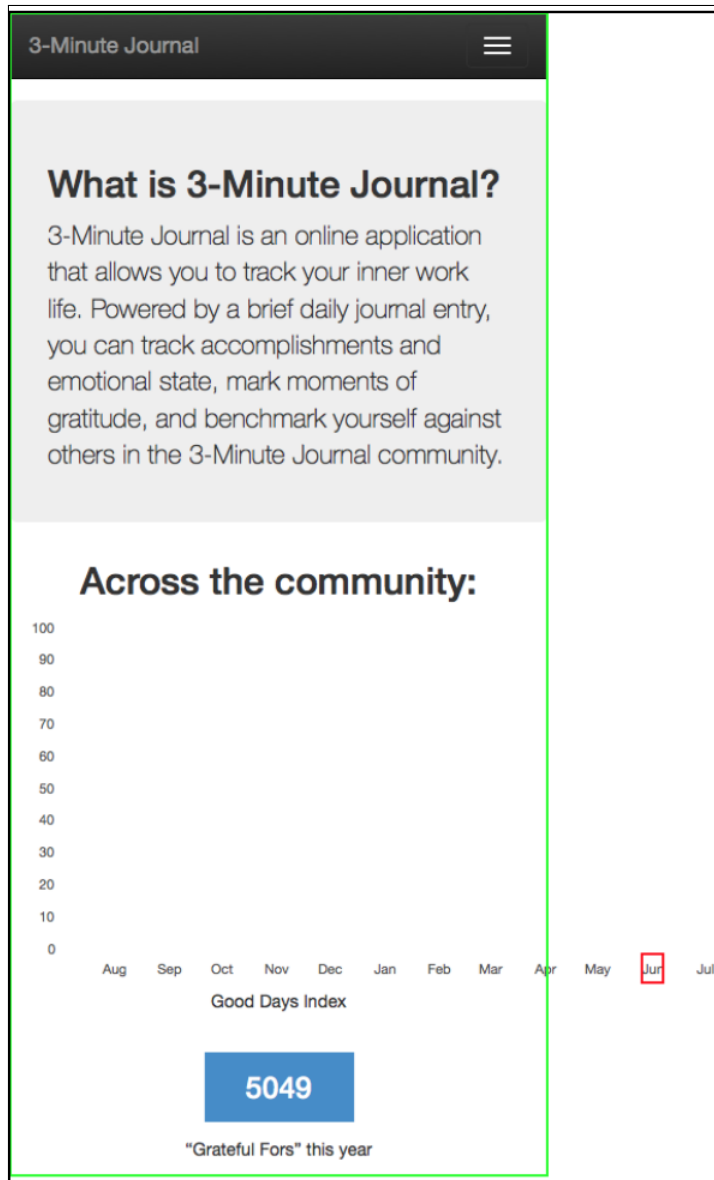


Figure 9.31: Part Two - Group Six - Failure Three

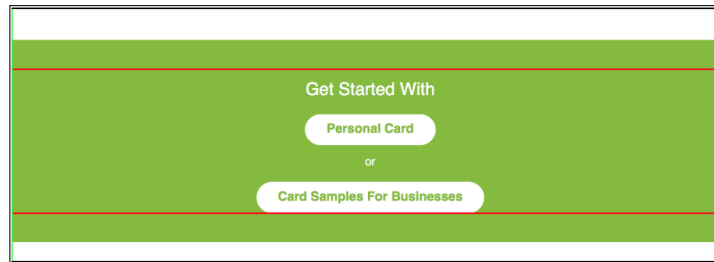


Figure 9.32: Part Two - Group Seven - Failure One



Figure 9.33: Part Two - Group Seven - Failure Two

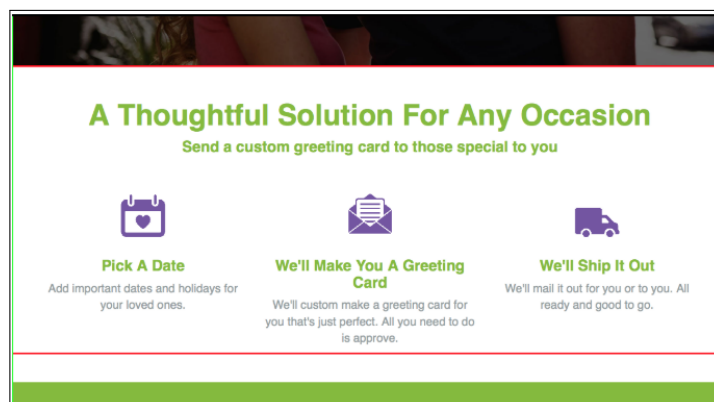


Figure 9.34: Part Two - Group Seven - Failure Three

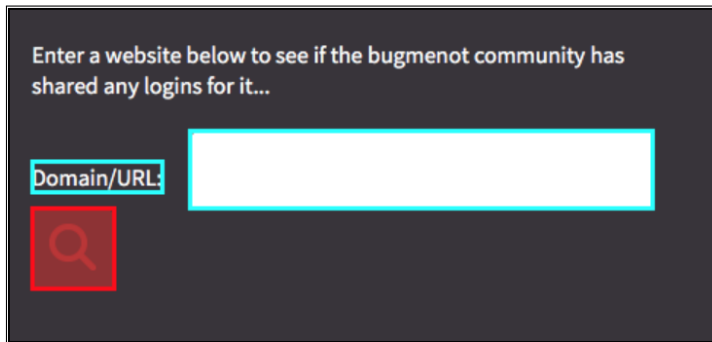


Figure 9.35: Part Two - Group Eight - Failure One

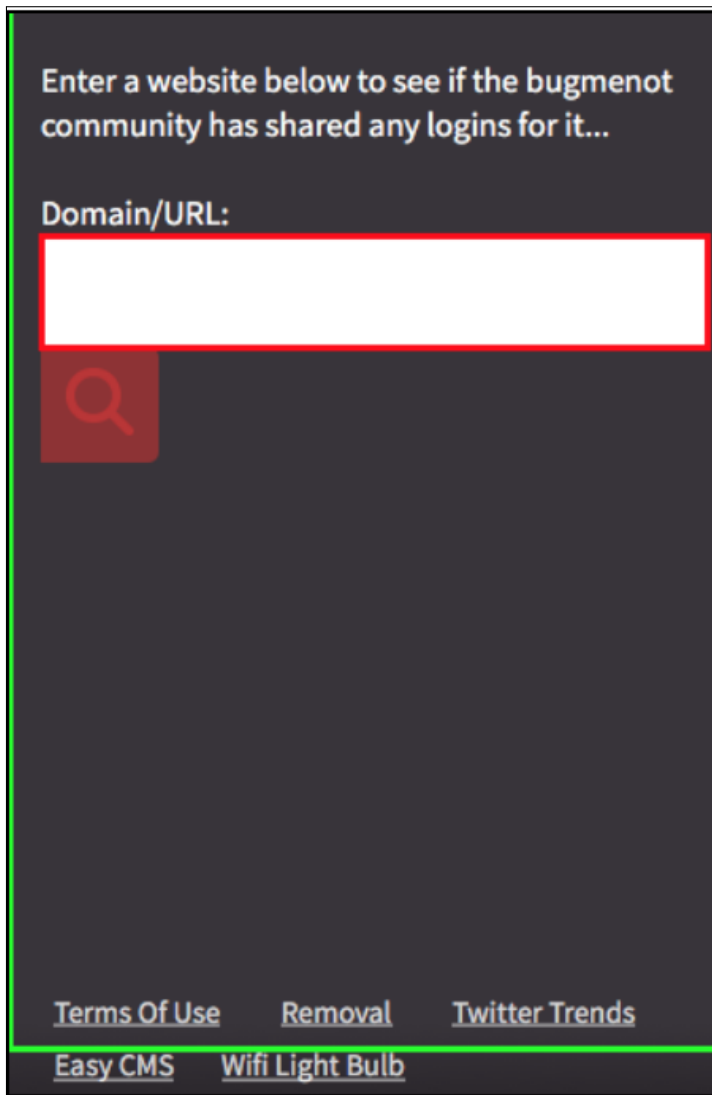


Figure 9.36: Part Two - Group Eight - Failure Two



**Figure 9.37:** Part Two - Group Eight - Failure Three

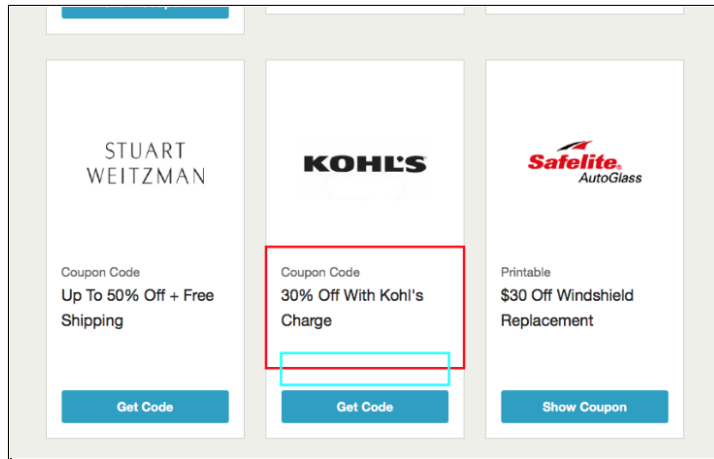


Figure 9.38: Part Two - Group Nine - Failure One

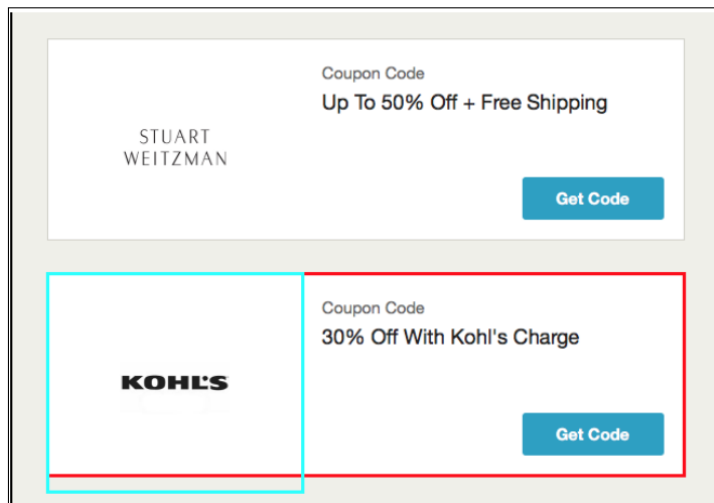



Figure 9.39: Part Two - Group Nine - Failure Two




Figure 9.40: Part Two - Group Nine - Failure Three

## Featured Products




### Dishwashers

175 Rated | [Buying Guide](#)




### Snow Blowers

79 Rated | [Buying Guide](#)




### SUVs

81 Rated | [Buying Guide](#)




### TVs

186 Rated | [Buying Guide](#)



### Winter/Snow Tires

22 Rated | [Buying Guide](#)



[All Products A-Z](#)

Figure 9.41: Part Two - Group Ten - Failure One

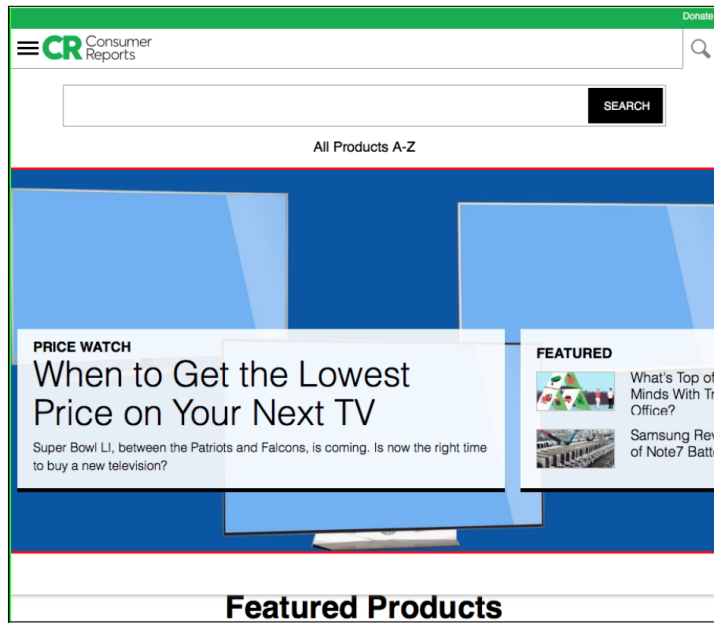




Figure 9.42: Part Two - Group Ten - Failure Two




## Featured Products




**Dishwashers**  
175 Rated | [Buying Guide](#)




**Snow Blowers**  
79 Rated | [Buying Guide](#)




**SUVs**  
81 Rated | [Buying Guide](#)



**TVs**  
186 Rated | [Buying Guide](#)



**Winter/Snow Tires**  
22 Rated | [Buying Guide](#)



[All Products A-Z](#)

Figure 9.43: Part Two - Group Ten - Failure Three

## Featured Products









	<b>Dishwashers</b> 175 Rated   <a href="#">Buying Guide</a>
	<b>Snow Blowers</b> 79 Rated   <a href="#">Buying Guide</a>
	<b>SUVs</b> 81 Rated   <a href="#">Buying Guide</a>
	<b>TVs</b> 186 Rated   <a href="#">Buying Guide</a>
	<b>Winter/Snow Tires</b> 22 Rated   <a href="#">Buying Guide</a>
	<a href="#">All Products A Z</a>

Figure 9.44: Part Two - Group Ten - Failure Four


## Featured Products




**Dishwashers**  
175 Rated | [Buying Guide](#)




**Snow Blowers**  
79 Rated | [Buying Guide](#)




**SUVs**  
81 Rated | [Buying Guide](#)



**TVs**  
186 Rated | [Buying Guide](#)



**Winter/Snow Tires**  
22 Rated | [Buying Guide](#)



[All Products A-Z](#)

Figure 9.45: Part Two - Group Ten - Failure Five

## Featured Products







	<b>Dishwashers</b> 175 Rated   <a href="#">Buying Guide</a>
	<b>Snow Blowers</b> 79 Rated   <a href="#">Buying Guide</a>
	<b>SUVs</b> 81 Rated   <a href="#">Buying Guide</a>
	<b>TVs</b> 186 Rated   <a href="#">Buying Guide</a>
	<b>Winter/Snow Tires</b> 22 Rated   <a href="#">Buying Guide</a>
	<a href="#">All Products A-Z</a>

Figure 9.46: Part Two - Group Ten - Failure Six

## Featured Products







	<b>Dishwashers</b> 175 Rated   <a href="#">Buying Guide</a>
	<b>Snow Blowers</b> 79 Rated   <a href="#">Buying Guide</a>
	<b>SUVs</b> 81 Rated   <a href="#">Buying Guide</a>
	<b>TVs</b> 186 Rated   <a href="#">Buying Guide</a>
	<b>Winter/Snow Tires</b> 22 Rated   <a href="#">Buying Guide</a>
	<a href="#">All Products A-Z</a>

Figure 9.47: Part Two - Group Ten - Failure Seven



---

## BIBLIOGRAPHY

---

- [1] Responsive web design — The viewport. URL: [https://www.w3schools.com/css/css\\\_rwd\\\_viewport.asp](https://www.w3schools.com/css/css\_rwd\_viewport.asp).
- [2] Android device fragmentation. URL: <http://opensignal.com/reports/2015/08/android-fragmentation/>.
- [3] Firefox developer tools: Responsive design mode. URL: [https://developer.mozilla.org/en-US/docs/Tools/Responsive\\\_Design\\\_Mode](https://developer.mozilla.org/en-US/docs/Tools/Responsive\_Design\_Mode).
- [4] REDECHECK tool and ISSTA results archive. URL: <http://redcheck.org/issta17/>.
- [5] RLF fixing think-aloud archive. URL: <http://redcheck.org/rlffixing/>.
- [6] Hamed Ahmadi and Jun Kong. “Efficient Web Browsing on Small Screens.” In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. 2008.
- [7] 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.
- [8] Michael Albers and Loel Kim. “Information Design for the Small-screen Interface: An Overview of Web Design Issues for Personal Digital Assistants.” In: *Technical Communication* 49.1 (2002), pp. 45–60.
- [9] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. “Using GUI Ripping for Automated Testing of Android Applications.” In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. 2012.
- [10] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. “Why does the orientation change mess up my Android application? From GUI failures to code faults.” In: *Software Testing, Verification and Reliability* 28.1 (2018).
- [11] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 1st ed. New York, NY, USA: Cambridge University Press, 2008.

- [12] Android. *User Interface Guidelines*. URL: [https://developer.android.com/guide/practices/ui\\_guidelines/index.html](https://developer.android.com/guide/practices/ui_guidelines/index.html).
- [13] Apple. *Human Interface Guidelines*. URL: <https://developer.apple.com/ios/human-interface-guidelines/overview/themes/>.
- [14] BBC Wraith. *Wraith*. 2015. URL: <https://github.com/BBC-News/wraith>.
- [15] Paul Bakaus. *Simulate Mobile Devices with Device Mode*. URL: <https://developers.google.com/web/tools/chrome-devtools/device-mode/>.
- [16] Shumeet Baluja. "Browsing on Small Screens: Recasting Web-page Segmentation into an Efficient Machine Learning Framework." In: *15th International Conference on World Wide Web*. 2006.
- [17] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey." In: *Transactions on Software Engineering* 41.5 (2015).
- [18] Patrick Baudisch, Xing Xie, Chong Wang, and Wei-Ying Ma. "Collapse-to-zoom: Viewing Web Pages on Small Screen Devices by Interactively Removing Irrelevant Content." In: *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*. UIST '04. 2004.
- [19] Twitter Bootstrap. *Bootstrap: Responsive front-end framework*. 2015. URL: <http://getbootstrap.com/>.
- [20] Penelope A Brooks and Atif M Memon. "Automated GUI Testing Guided by Usage Profiles." In: *22nd International Conference on Automated Software Engineering*. 2007.
- [21] BrowserStack. *BrowserStack*. 2017. URL: <https://www.browserstack.com>.
- [22] Ilene Burnstein. *Practical Software Testing: A Process-Oriented Approach*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1441928855, 9781441928856.
- [23] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN: 0471958697, 9780471958697.
- [24] Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. "Seeing the Whole in Parts: Text Summarization for Web Browsing on Hand-held Devices." In: *Proceedings of the 10th International Conference on World Wide Web*. WWW '01. 2001.



- [25] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. "Issues in Using Students in Empirical Studies in Software Engineering Education." In: *Proceedings of the 9th International Symposium on Software Metrics*. METRICS '03. 2003.
- [26] Anthony Casalena. *Squarespace*. URL: <https://www.squarespace.com/>.
- [27] Rory Cellan-Jones. *Google's 'mobilegeddon'*. 2015. URL: <http://www.bbc.co.uk/news/technology-32393050>.
- [28] Google Webmaster Central. *Rolling out the mobile-friendly update*. 2015. URL: <https://webmasters.googleblog.com/2015/04/rolling-out-mobile-friendly-update.html>.
- [29] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. "GUI Testing Using Computer Vision." In: *SIGCHI Conference on Human Factors in Computing Systems*. 2010.
- [30] Yu Chen, Wei-Ying Ma, and Hong-Jiang Zhang. "Detecting Web Page Structure for Adaptive Viewing on Small Form Factor Devices." In: *Proceedings of the 12th International Conference on World Wide Web*. WWW '03. 2003.
- [31] Luca Chittaro. "Visualizing Information on Mobile Devices." In: *Computer* 39.3 (2006), pp. 40–45. ISSN: 0018-9162.
- [32] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. "Cross-Check: 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.
- [33] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. "Web-Diff: Automated Identification of Cross-browser Issues in Web Applications." In: *Proceedings of the 26th International Conference on Software Maintenance*. 2010.
- [34] Google Corporation. *Google Analytics Solutions*. URL: <https://www.google.co.uk/analytics/>.
- [35] Google Corporation. *Mobile-Friendly Test*. 2015. URL: <https://www.google.com/webmasters/tools/mobile-friendly/>.
- [36] ZURB Corporation. *Mobile-First*. Accessed: 2015-06-29. URL: <http://zurb.com/word/mobile-first>.
- [37] Creative Bloq. *Creative Bloq: 5 of this year's Vertical Scrolling Trends*. <http://www.creativebloq.com/design/vertical-scrolling-trends-2015-121413698>. 2015.

- [38] John Cristy. *ImageMagick*. Accessed: 2015-06-19. URL: <http://www.imagemagick.org/>.
- [39] Sophie Curtis. *Google search overhaul could spark 'Mobilegeddon'*. 2015. URL: <http://www.telegraph.co.uk/technology/google/11549615/Google-search-overhaul-could-cause-Mobilegeddon.html>.
- [40] Dianne Cyr, Milena Head, and Alex Ivanov. "Design Aesthetics Leading to M-loyalty in Mobile Commerce." In: *Information & Management* 43.8 (2006).
- [41] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. "WebMate: A Tool for Testing Web 2.0 Applications." In: *Proceedings of the Workshop on JavaScript Tools*. 2012.
- [42] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. "WebMate: Generating Test Cases for Web 2.0." In: *Software Quality: Increasing Value in Software and Systems Development*. Vol. 133. 2013.
- [43] Valentin Dallmeier, Bernd Pohl, Martin Burger, Michael Miroid, and Andreas Zeller. "WebMate: Web Application Test Generation in the Real World." In: *Software Testing, Verification and Validation Workshops (ICSTW), 2014 Seventh International Conference on*. 2014.
- [44] Cambridge English Dictionary. *Spot Check*. URL: <https://dictionary.cambridge.org/dictionary/english/spot-check>.
- [45] Monica Dinculescu. *Automatic visual diffing with Puppeteer*. URL: <https://meowni.ca/posts/2017-puppeteer-tests/>.
- [46] *Duda One*. URL: <https://www.duda.co/responsive-website-builder>.
- [47] Cyntrica Eaton and Atif M Memon. "An Empirical Approach to Evaluating Web Application Compliance Across Diverse Client Platform Configurations." In: *International Journal of Web Engineering and Technology* 3.3 (2007).
- [48] Wayne W. Eckerson. "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications." In: *Open Information Systems* 10.1 (1995).
- [49] Website Builder Expert. *3 Amazingly Easy Responsive Website Builders To Use in 2018*. URL: <https://www.websitebuilderexpert.com/how-to-create-a-responsive-website/>.

- [50] Mattia Fazzini and Alessandro Orso. “Automated Cross-platform Inconsistency Detection for Mobile Apps.” In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. 2017, pp. 308–318.
- [51] *Fighting Layout Bugs*. URL: <https://code.google.com/archive/p/fighting-layout-bugs/>.
- [52] Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. “A systematic mapping study of web application testing.” In: *Information and Software Technology* 55.8 (2013), pp. 1374–1396. ISSN: 0950-5849. DOI: <http://dx.doi.org/10.1016/j.infsof.2013.02.006>. URL: <http://www.sciencedirect.com/science/article/pii/S0950584913000396>.
- [53] Robert L. Glass. “Frequently Forgotten Fundamental Facts About Software Engineering.” In: *IEEE Softw.* 18.3 (May 2001), pp. 112–111. ISSN: 0740-7459. DOI: [10.1109/MS.2001.922739](https://doi.org/10.1109/MS.2001.922739). URL: <http://dx.doi.org/10.1109/MS.2001.922739>.
- [54] Craig Grannell. *15 top web design and development trends for 2012*. 2012. URL: <http://www.creativebloq.com/industry-trends/15-top-web-design-and-development-trends-2012-1123018>.
- [55] André MP Grilo, Ana CR Paiva, and João Pascoal Faria. “Reverse Engineering of GUI Models for Testing.” In: *5th Iberian Conference on Information Systems and Technologies*. 2010.
- [56] Yuepu Guo and Sreedevi Sampath. “Web Application Fault Classification — An Exploratory Study.” In: *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement*. 2008.
- [57] Aaron Gustafson. *Adaptive Web Design. Crafting Rich Experiences with Progressive Enhancement*. 2011.
- [58] Antonin Guttman. “R-trees: A Dynamic Index Structure for Spatial Searching.” In: *International Conference on Management of Data*. ACM, 1984, pp. 47–57.
- [59] William G. J. Halfond. “Automated Checking of Web Application Invocations.” In: *Proceedings of the 23rd International Symposium on Software Reliability Engineering*. 2012.
- [60] William G. J. Halfond and Alessandro Orso. “Automated Identification of Parameter Mismatches in Web Applications.” In: *Proceedings of the 16th International Symposium on Foundations of Software Engineering*. 2008.

- [61] 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.
- [62] Sylvain Hallé, Nicolas Bergeron, Francis Guérin, Gabriel Le Breton, and Oussama Beroual. “Declarative Layout Constraints for Testing Web Applications.” In: *Journal of Logical and Algebraic Methods in Programming* 85 (2016).
- [63] Mark Harman and Bryan F Jones. “Search-based software engineering.” In: *Information and Software Technology* 43.14 (2001), pp. 833–839. ISSN: 0950-5849.
- [64] Matthew Harris. *Responsive vs Adaptive Design — Which is Best for Mobile Viewing of Your Website?* 2015. URL: <http://mediumwell.com/responsive-adaptive-mobile/>.
- [65] Matthew Harris. *Responsive vs Adaptive Design – Which is Best for Mobile Viewing of Your Website?* 2015. URL: <http://mediumwell.com/responsive-adaptive-mobile/>.
- [66] Jan Hartmann, Alistair Sutcliffe, and Antonella De Angeli. “Investigating Attractiveness in Web User Interfaces.” In: *Proceedings of the International Conference on Human Factors in Computing Systems*. 2007.
- [67] Robert Hof. *Google Research: No Mobile Site = Lost Customers*. 2012. URL: <http://www.forbes.com/sites/roberthof/2012/09/25/google-research-no-mobile-site-lost-customers/>.
- [68] Robert Hof. *Why Google’s Mobilegeddon Isn’t The End Of The World For Most Websites*. 2015. URL: <https://www.forbes.com/sites/roberthof/2015/04/21/why-googles-mobilegeddon-isnt-the-end-of-the-world-for-most-websites/#5127cd2f4532>.
- [69] Destiny Montague & Lara Hogan. *Building a Device Lab*. Five Simple Steps, 2015.
- [70] Martin Höst, Björn Regnell, and Claes Wohlin. “Using Students As Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment.” In: *Empirical Softw. Engg.* 5.3 (Nov. 2000), pp. 201–214. ISSN: 1382-3256.
- [71] Cuixiong Hu and Iulian Neamtii. “Automating GUI Testing for Android Applications.” In: *Proceedings of the 6th International Workshop on Automation of Software Test*. AST ’11. 2011.

- [72] Google Inc. *Android Developers*. Accessed: 2015-26-05. URL: <http://developer.android.com/sdk/index.html>.
- [73] Jeremy Durant. *All the responsive Web design statistics you need*. URL: <https://www.bopdesign.com/bop-blog/2015/02/responsive-web-design-statistics/>.
- [74] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing." In: *IEEE Trans. Softw. Eng.* 37.5 (2011), pp. 649–678. ISSN: 0098-5589.
- [75] S. C. Johnson. "Lint, a C Program Checker." In: *COMP. SCI. TECH. REP.* 1978, pp. 78–1273.
- [76] Matt Jones, Gary Marsden, Norliza Mohd-Nasir, Kevin Boone, and George Buchanan. "Improving Web Interaction on Small Displays." In: *Proceedings of the Eighth International Conference on World Wide Web. WWW '99*. Toronto, Canada, 1999, pp. 1129–1137.
- [77] M. E. Joorabchi and A. Mesbah. "Reverse Engineering iOS Mobile Applications." In: *2012 19th Working Conference on Reverse Engineering*. 2012.
- [78] Matt Kersley. *Responsive Design Testing*. 2017. URL: <http://mattkersley.com/responsive/>.
- [79] Carlos Bernal-Cárdenas Dan Jelf Denys Poshyvanyk Kevin Moran Boyang Li. "Automated Reporting of GUI Design Violations for Mobile Apps." In: *40th International Conference on Software Engineering (ICSE'18)*. 2018.
- [80] Hammad Khalid, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. "Prioritizing The Devices To Test Your App On: A Case Study Of Android Game Apps." In: *22nd International Symposium on Foundations of Software Engineering*. 2014.
- [81] Cody Kinneer, Gregory M. Kapfhammer, Chris J. Wright, and Phil McMinn. "Automatically Evaluating the Efficiency of Search-Based Test Data Generation for Relational Database Schemas." In: *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*. 2015.
- [82] Labcase. *Open Device Lab in a Case*. URL: <http://labcase.org/>.
- [83] Eunshil Lee, Jinbeom Kang, Joongmin Choi, and Jaeyoung Yang. "Topic-Specific Web Content Adaptation to Mobile Devices." In: *2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006 Main Conference Proceedings)(WI'06)* (2006), pp. 845–848.

- [84] Sangwon Lee and Richard J. Koubek. "The Effects of Usability and Web Design Attributes on User Preference for e-Commerce Web Sites." In: *Computers in Industry* 61.4 (2010).
- [85] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals." In:
- [86] Wanchun Li, Mary Jean Harrold, and Carsten Görg. "Detecting User-visible Failures in AJAX Web Applications by Analyzing Users' Interaction Behaviors." In: *Proceedings of the 25th International Conference on Automated Software Engineering*. 2010.
- [87] Hao Liu, Xing Xie, Wei-Ying Ma, and Hong-Jiang Zhang. *Automatic Browsing of Large Pictures on Mobile Devices*. Tech. rep. MSR-TR-2003-49. Microsoft Research, 2003, p. 8.
- [88] 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.
- [89] 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.
- [90] 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.
- [91] 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.
- [92] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. "Automated Repair of Layout Cross Browser Issues Using Search-Based Techniques." In: *International Conference on Software Testing and Analysis (ISSTA 2017)*. 2017, pp. 249–260.
- [93] 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: *International Conference on Software Testing, Validation and Verification (ICST 2018) (To Appear)*. 2018.

- [94] Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G.J. Halfond. "Automated Repair of Mobile Friendly Problems in Web Pages." In: *International Conference on Software Engineering (ICSE 2018) (To Appear)*. 2018.
- [95] Ethan Marcotte. *Responsive Web Design*. 2010. URL: <https://alistapart.com/article/responsive-web-design>.
- [96] Ethan Marcotte. *Responsive Web Design*. A Book Apart, 2014.
- [97] Grace Mbipom and Simon Harper. "The Interplay Between Web Aesthetics and Accessibility." In: *Proceedings of the 13th International Conference on Computers and Accessibility*. 2011.
- [98] Tina McCorkindale and Meredith Morgoch. "An Analysis of the Mobile Readiness and Dialogic Principles on Fortune 500 Mobile Websites." In: *Public Relations Review* 39.3 (2013).
- [99] William M. McKeeman. "Differential testing for software." In: *Digital Technical Journal* 10.1 (1998).
- [100] Phil McMinn. "Search-Based Software Testing: Past, Present and Future." In: *International Workshop on Search-Based Software Testing (SBST 2011)*. IEEE, 2011, pp. 153–163.
- [101] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. "DART: A Framework for Regression Testing Nightly/Daily Builds of GUI Applications." In: *International Conference on Software Maintenance 2003*. 2003.
- [102] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing." In: *Proceedings of the 10th Working Conference on Reverse Engineering*. 2003.
- [103] Chafik Meniar, Florence Opalvens, and Sylvain Hallé. "Runtime Verification of User Interface Guidelines in Mobile Devices." In: *Runtime Verification*. Springer International Publishing, 2017, pp. 410–415.
- [104] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes." In: *ACM Transactions on the Web (TWEB)* 6.1 (2012).
- [105] Ali Mesbah and Mukul R Prasad. "Automated Cross-browser Compatibility Testing." In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011.
- [106] Ali Mesbah, Arie Van Deursen, and Danny Roest. "Invariant-Based Automatic Testing of Modern Web Applications." In: *Transactions on Software Engineering* 38.1 (2012).

- [107] Eleni Michailidou, Simon Harper, and Sean Bechhofer. "Visual Complexity and Aesthetic Perception of Web Pages." In: *Proceedings of the 26th Annual ACM International Conference on Design of Communication*. 2008.
- [108] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. "Leveraging Existing Tests in Automated Test Generation for Web Applications." In: *Proceedings of the 29th International Conference on Automated Software Engineering*. 2014.
- [109] E. F. Miller and W. E. Howden. *Software Testing and Validation Techniques*. 1978.
- [110] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. "Testing Android Apps Through Symbolic Execution." In: *SIGSOFT Softw. Eng. Notes* 37.6 (2012).
- [111] Mobify. *70 Stunning Responsive Sites for Your Inspiration*. URL: <https://www.mobify.com/insights/70-stunning-responsive-websites-for-your-inspiration/>.
- [112] Rodrigo M.L.M. Moreira and Ana C.R. Paiva. "PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-based GUI Testing." In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. 2014, pp. 863–866.
- [113] Rodrigo M.L.M. Moreira, Ana C.R. Paiva, Miguel Nabuco, and Atif Memon. "Pattern based GUI testing: Bridging the gap between design and quality assurance." In: *Software Testing, Verification and Reliability* 27.3 (2017).
- [114] Inês Coimbra Morgado and Ana C. R. Paiva. "Mobile GUI testing." In: *Software Quality Journal* (2017).
- [115] Mozilla Firebug. URL: <http://getfirebug.com>.
- [116] Gustav Öquist and Mikael Goldstein. "Towards an Improved Readability on Mobile Devices: Evaluating Adaptive Rapid Serial Visual Presentation." In: *Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction*. Mobile HCI '02. London, UK, UK: Springer-Verlag, 2002, pp. 225–240. ISBN: 3-540-44189-1. URL: <http://dl.acm.org/citation.cfm?id=645739.666578>.
- [117] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. "Mutation testing advances: an analysis and survey." In: *Advances in Computers* (2017).



- [118] Thomas H Park, Brian Dorn, and Andrea Forte. "An Analysis of HTML and CSS Syntax Errors in a Web Development Course." In: *Transactions on Computing Education* 15.1 (2015).
- [119] Upsorn Praphamontripong and Jeff Offutt. "Applying Mutation Testing to Web Applications." In: *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. ICSTW '10. 2010.
- [120] *Responsinator*. 2017. URL: <https://www.responsinator.com/>.
- [121] *Responsive Design Checker*. 2017. URL: <http://responsivedesignchecker.com>.
- [122] David Robins and Jason Holmes. "Aesthetics and Credibility in Web Site Design." In: *Information Processing & Management* 44.1 (2008).
- [123] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. "X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications." In: *Proceedings of the 35th International Conference on Software Engineering*. 2013.
- [124] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. "X-PERT: A Web Application Testing Tool for Cross-Browser Inconsistency Detection." In: *2014 International Symposium on Software Testing and Analysis*. 2014.
- [125] Yeonhee Ryou and Sukyoung Ryu. "Automatic Detection of Visibility Faults by Layout Changes in HTML5 Web Pages." In: *Proceedings of the 11th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2018.
- [126] Tõnis Saar, Marlon Dumas, Marti Kaljuve, and Nataliia Semenenko. "Browserbite: Cross-browser Testing via Image Processing." In: *Softw. Pract. Exper.* 46.11 (2016).
- [127] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. "Are Students Representatives of Professionals in Software Engineering Experiments?" In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 666–676. ISBN: 978-1-4799-1934-5. URL: <http://dl.acm.org/citation.cfm?id=2818754.2818836>.

- [128] E. Selay, Z. Q. Zhou, and J. Zou. "Adaptive Random Testing for Image Comparison in Regression Web Testing." In: *2014 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. 2014, pp. 1–7.
- [129] E. Selay, Z. Q. Zhou, T. Y. Chen, and F. C. Kuo. "Adaptive Random Testing in Detecting Layout Faults of Web Applications." In: (2018).
- [130] Selenium: Web Browser Automation. 2015. URL: <http://www.seleniumhq.org/>.
- [131] N. Semenenko, M. Dumas, and T. Saar. "Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features." In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 528–531.
- [132] Remy Sharp. *ResponsivePX*. 2015. URL: <http://responsivepx.com/>.
- [133] Ivan Shubin. *Galen Framework*. URL: <http://galenframework.com/>.
- [134] Somo. *Google Mobilegeddon is almost here - and brands aren't paying attention*. 2015. URL: <https://blog.somoglobal.com/blog/google-mobilegeddon-is-almost-here-and-brands-arent-paying-attention>.
- [135] Stackify. *What is N-Tier Architecture? How It Works, Examples, Tutorials, and More*. URL: <https://stackify.com/n-tier-architecture/>.
- [136] Techopedia. *Mobile Device*. URL: <https://www.techopedia.com/definition/23586/mobile-device>.
- [137] Techopedia. *Multitouch*. URL: <https://www.techopedia.com/definition/24263/multitouch>.
- [138] András Vargha and Harold D Delaney. "A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong." In: *Journal of Educational and Behavioral Statistics* 25.2 (2000).
- [139] *Viewport Resizer*. 2015. URL: <http://lab.maltewassermann.com/viewport-resizer/>.
- [140] S. Vilkomir and B. Amstutz. "Using Combinatorial Approaches for Testing Mobile Applications." In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 2014, pp. 78–83.
- [141] M. A. Vouk. "Back-to-back Testing." In: *Inf. Softw. Technol.* 32.1 (Jan. 1990), pp. 34–45. ISSN: 0950-5849. DOI: 10.1016/0950-5849(90)90044-R. URL: [http://dx.doi.org/10.1016/0950-5849\(90\)90044-R](http://dx.doi.org/10.1016/0950-5849(90)90044-R).
- [142] W3C. *HTML & CSS*. URL: <https://www.w3.org/standards/webdesign/htmlcss>.

- [143] W3C. *JavaScript Web APIs*. URL: <https://www.w3.org/standards/webdesign/script>.
- [144] Thomas A Walsh. *REDECHECK Chapter 4 Results Archive*. 2018. URL: <https://github.com/redecheck/jstvr-data>.
- [145] Thomas A Walsh. *REDECHECK mutated web pages*. 2018. URL: <https://github.com/redecheck/jstvr-webpages>.
- [146] Thomas A Walsh. *REDECHECK web page examples*. 2018. URL: <https://github.com/redecheck/example-webpages>.
- [147] 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 26th International Conference on Software Testing and Analysis*. 2017.
- [148] Thomas A Walsh, Gregory M Kapfhammer, and Phil McMinn. "Automatically Alerting Developers to the Unseen Side Effects of Incremental Changes to Responsive Web Pages." In: *Software Testing, Verification and Reliability (Under Review)* (2017).
- [149] 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.
- [150] 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).
- [151] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. "Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis." In: *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*. 2012.
- [152] Gary Wassermann and Zhendong Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities." In: *Proceedings of the 28th International Conference on Programming Language Design and Implementation*. 2007.
- [153] Gary Wassermann and Zhendong Su. "Static Detection of Cross-site Scripting Vulnerabilities." In: *Proceedings of the 30th International Conference on Software Engineering*. 2008.
- [154] Weebly. URL: <https://www.weebly.com>.

- [155] *Window Resizer*. 2015. URL: <http://ionut-botizan.net/window-resizer/>.
- [156] Ou Wu, Yunfei Chen, Bing Li, and Weiming Hu. "Evaluating the Visual Quality of Web Pages Using a Computational Aesthetic Approach." In: *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*. 2011.
- [157] Xiangye Xiao, Qiong Luo, Dan Hong, and Hongbo Fu. "Slicing\*-tree based web page transformation for small displays." In: *CIKM*. 2005.
- [158] Yunpeng Xiao, Yang Tao, and Qian Li. "Web Page Adaptation for Mobile Device." In: *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing (2008)*, pp. 1–5.
- [159] Xing Xie, Chong Wang, Li-Qun Chen, and Wei-Ying Ma. "An adaptive web page layout structure for small devices." In: *Multimedia Systems 11* (2005), pp. 34–44.
- [160] Xinyi Yin and Wee Sun Lee. "Using link analysis to improve layout on mobile devices." In: *WWW*. 2004.
- [161] ZURB Corporation. *Foundation: Responsive front-end framework*. 2015. URL: <http://foundation.zurb.com/>.
- [162] StackExchange Data Explorer. *Prevalence of Tags Related to Responsive Web Design*. 2017. URL: <http://data.stackexchange.com/stackoverflow/query/edit/541879>.
- [163] Segue Technologies. *Challenges When Testing a Responsive Website*. 2014. URL: <https://goo.gl/cwPlsH>.
- [164] Statista. *Statistics Portal*. 2017. URL: <http://goo.gl/19fZcq>.