SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–43
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Automatically Identifying Potential Regressions in the Layout of Responsive Web Pages

Thomas Walsh[1], Gregory M. Kapfhammer[2], and Phil McMinn[1]*

[1]*University of Sheffield* [2]*Allegheny College*

SUMMARY

Providing a good user experience on the ever-increasing number and variety of devices being used to browse the web is a difficult, yet critical, task. With Responsive Web Design (RWD), front-end web developers design web pages so that they dynamically resize and rearrange content to best fit the dimensions of a device's screen. However, when making code modifications to a responsive page, developers can easily introduce regressions from the correct layout that have detrimental effects at unpredictable screen sizes. For instance, the source code change that a developer makes to improve the layout at one screen size may obscure a page's content at other sizes. Current approaches to testing are often insufficient because they rely on limited tools and error-prone manual inspections of a web page. As such, many unintended regressions in web page layout often go undetected and ultimately manifest in production web sites. To address the challenge of detecting regressions in responsive web pages, this paper presents an automated approach that extracts the responsive layout of two versions of a page and compares them, alerting developers to the differences in layout that they may wish to investigate further. We implemented the approach and empirically evaluated it on 15 real-world responsive web pages. Leveraging code mutations that a tool automatically injected into the pages as a systematic simulation of developer changes, the experiments show that the approach was highly effective. When compared with manual and automated baseline testing techniques, it detected 12.5% and 18.75% more injected changes, respectively. Along with identifying the best parameters for the method that extracts the responsive layout, the experiments show that the approach surpasses the baselines across changes that vary in their impact, but works particularly well for subtle, hard-to-detect mutants, showing the benefits of automatically identifying regressions in web page layout.

Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Responsive web design; Layout failures; Regression testing; Graph differencing

1. INTRODUCTION

Before the advent of smart mobile devices being used to access the web, the main problem facing front-end web developers was the time consuming task of ensuring that their web pages displayed correctly on different browsers [1]. However, the explosion and continuing proliferation of mobile devices has since drastically exacerbated this problem, as developers sought to support the myriad of different screen sizes associated with each unique device. As mobile devices continue to integrate further into the daily lives of many people (notably, more than half of Google searches originate from mobile devices [2], while mobile devices now account for nearly 70% of cumulative digital media consumption [3]), it is critical for developers to ensure that their web sites display correctly and offer a good interaction experience on the significant variety of devices currently in use.

*Correspondence to: Department of Computer Science, The University of Sheffield, 211 Portobello, Sheffield, S1 4DP

To tackle the problem of device proliferation, responsive web design (RWD) is a widely-used design approach that allows developers to build web sites with the aim of providing an optimal user experience regardless of individual device constraints, including screen size [4]. With RWD, a page's design "responds to" the device's viewport (i.e., the viewable portion of a page in a device's browser), such that web elements dynamically resize and rearrange to "fit" to the device's viewport width. The page can extend vertically off the bottom of the viewport by as much as required, meaning that users will still need to scroll up and down to access content that cannot be displayed on a screen all at once, but prevents them having to awkwardly pan and zoom pages originally intended for desktop browsers when rendered on considerably smaller devices, such as mobile phones.

Developers build web pages using Hypertext Markup Language (HTML) elements, which structure text, images, forms, and other content. They then employ Cascading Style Sheets (CSS), which specify rules that visually style and lay out those elements. The HTML and CSS code required to implement a responsive design is often complex, with potentially many hundreds of lines of CSS interacting with a significant number of different HTML elements. Notably, this inherent complexity presents challenges when web developers first learn the skills of their field [5, 6] and continues as they become professionals who must regularly ask their peers questions about topics such as responsive web development [7]. Ultimately, well-trained developers can make mistakes in which web page elements are not responsive and thus fail to layout correctly at a certain viewport widths.

In our previous work, we identified some representative ways in which RWD layouts can "fail". For instance, a web page can exhibit a layout failure when its elements do not fit horizontally alongside one another at certain viewport widths and, as a consequence, protrude off the edge of a page, overlap with one another, or incorrectly wrap to the next line. We presented techniques to identify these issues, and implemented them into a tool called "REDECHECK" (REsponsive DEsign CHECKer, pronounced "Ready Check") [8, 9]. This prior work, however, does not address a different, yet critically important, problem: detecting the regressions from a correct responsive layout that occur when a developer makes modifications to a web page's responsive design.

During the creation of a responsive web site, a developer will make small, seemingly insignificant, incremental changes to a web page's source code that are intended for a particular viewport width (or a particular range of widths), but yet which can introduce unintended side-effects to the page's layout at other, seemingly unrelated, viewport widths. These regressions from the correct layout can sometimes result in serious presentational issues [10]. Since well-designed web sites can have a positive psychological influence on end users, leading to increases in the perceived usability of the site [11] and improved loyalty towards the web site itself [12], identifying these potentially damaging regressions in a web page's layout before it goes live is a critical task for developers.

Regrettably, detecting these layout regressions quickly and accurately is difficult. Existing approaches for tackling this problem, which include "spotchecking" techniques, are insufficient. Spotchecking involves a developer manually checking a web page's layout at a series of individual viewport widths, usually those pertaining to devices in popular use. Even though developer tools such as RESPONSIVEPX [13], RESIZER [14], and WRAITH [15] and browser-based tools such as "Responsive Design Mode" for Firefox and "Device Mode" for Chrome provide a certain amount of limited support when they automatically resize the browser for developers to check pages, unintended layout issues introduced by a change can manifest themselves at unpredictable viewport widths. These regressions may be hard to find since they often only affect a small number of widths out of a large range that accommodates the smaller screens found on mobile phones to larger desktop displays. If a web developer does not check an offending viewport width, then the web page may be deployed with these undetected and unresolved issues. Moreover, as many modern web pages display a significant amount of complex content, developers may fail to spot non-obvious, yet potentially damaging, regressions from the correct layout — even if they do check problematic viewport widths through manual inspections aided by the aforementioned spotchecking tools.

Recognizing the importance of detecting regressions in responsive designs, and the shortcomings of current practices used by developers to detect such issues, this paper presents an automated approach to alerting a developer to the unseen layout regressions arising from changes to the HTML and CSS of responsive web pages. The foundation of the approach is the responsive layout graph

(RLG), a model of a web page's responsive design. The approach extracts "before" and "after" RLGs from versions of a web page and differences the two RLGs, reporting any discrepancies to the web developer. This reduces the testing burden placed on the developer, who now only needs to inspect the list of differences automatically discovered by the technique and verify if any of them were unintentional side-effects. We implemented this technique as a special "regression mode" of our REDECHECK tool, which we refer to throughout this paper as "REDECHECK-*RM*".

This paper is an extension of a short, six-page, "new ideas" conference paper presented at the International Conference on Automated Software Engineering (ASE 2015) [16]. That paper originally introduced the responsive layout graph, which our technique extracted from a web page by sampling its layout at intervals throughout the range of viewport widths that the page might be viewed at by users. It presented two CSS mutation operators, designed to inject mutations capable of inducing regression changes to the layout of responsive web designs, and evaluated whether RLG differencing was capable of detecting these regressions in the layout of five web pages.

This journal version expands and deepens the technical details of our ASE 2015 approach. It furnishes a complete description of how the RLG of a web page is extracted, according to three different methods: the original interval sampling method of the ASE paper and two additional approaches; one that combines interval sampling with additional viewport widths referenced in the CSS code, and a method that exhaustively examines the layout of a web page at each viewport width in a specified range. To more thoroughly evaluate the RLG differencing method in REDECHECK-*RM*, we developed six new HTML and CSS mutation operators to create a larger set of web page modifications with which to evaluate our technique. We adopted two spotchecking procedures for use as a baseline with which to compare the method: a typical manual approach and an automated approach that we developed especially for the purposes of our empirical evaluation.

Correspondingly, this journal version expands the empirical study of the original ASE 2015 "new ideas" paper by extending the set of web pages from five in that paper to 15 subject pages in this journal version, with the new, larger set of 15 pages representing a wide range of domains and complexity levels. We experimentally evaluate REDECHECK-*RM* against the two aforementioned manual and automated spotchecking procedures, finding that it is superior at detecting regressions with higher accuracy. We also introduce a new research question that explores the performance of our approach on layout changes that affect differing ranges of viewport widths. The results show that REDECHECK-*RM* is more effective than both spotchecking methods, particularly for the more "subtle" changes that only occur at a small number of viewport widths. Finally, we conduct an analysis of how the different RLG extraction methods influence the effectiveness and efficiency of the approach, allowing us to recommend a set of configuration parameters that are likely to enable REDECHECK-*RM* to perform well on a wide variety of web pages in future work.

In summary, the contributions made by our original short, six-page conference version of this paper published in the "new ideas" track at ASE 2015 [16], were as follows:

C1. (a) The responsive layout graph (RLG), (b) an outline of the interval sample method for extracting it from a responsive web page, and (c) the original RLG differencing approach;
C2. An experimental evaluation of RLG differencing on five web pages with changes induced by two different mutation operators.

The further contributions made by this extended journal version are as follows:

J1. In support of an automated approach to automatically identifying regression changes in responsive web pages, algorithms that can extract RLGs from web pages and compare them against one another to highlight differences. These algorithms incorporate two new RLG extraction methods: one that can also sample viewport widths from the references found in CSS, and one that exhaustive analyzes web page layout in a pre-defined range (Section 3);
J2. Used to empirically evaluate the presented methods, six new HTML/CSS mutation operators that systematically add potential regressions into responsive web pages (Section 4.2);
J3. An automated spotchecking method that checks the layout of a responsive web page at individual viewport widths, thereby serving as a suitable baseline for comparison with the RLG differencing approach implemented in REDECHECK-*RM*, (Section 4.3);

J4. An extensive empirical evaluation, involving the use of an extended subject set comprising 15 web pages, yielding answers to research questions (Section 4.7) showing that:

(a) When comparing REDECHECK-*RM* with two spotchecking approaches, the RLG differencing technique implemented into REDECHECK-*RM* automatically reveals the most layout regressions, with the greatest number of true positives and true negatives;
(b) When evaluating how REDECHECK-*RM* performs with respect to the number of viewport widths affected by a change, RLG differencing is found to be more effective than spotchecking for "subtle" changes affecting small numbers of viewport widths;
(c) When evaluating the RLG extraction method and the parameters with the best failure detecting–efficiency trade-off, interval sampling with an interval size of 80 pixels combined with viewport widths mined from CSS code is the best overall method, and should therefore enable our technique to perform well on a range of new web pages.

The paper is organized as follows. First, Section 2 begins by reviewing the important background concepts in responsive web design, further highlighting the challenges involved in testing the layout of a responsive web page. Importantly, this section also uses a motivating example to introduce the problem of this paper: automatically detecting regression changes to the layout in responsive web pages. Section 3 introduces the presented approach for modelling responsive web page layout and automatically detecting these changes, while Section 4 presents its empirical evaluation using real-world subjects. The paper then discusses related work in Section 5, while finally Section 6 concludes the paper and suggests important and further novel directions for future work.

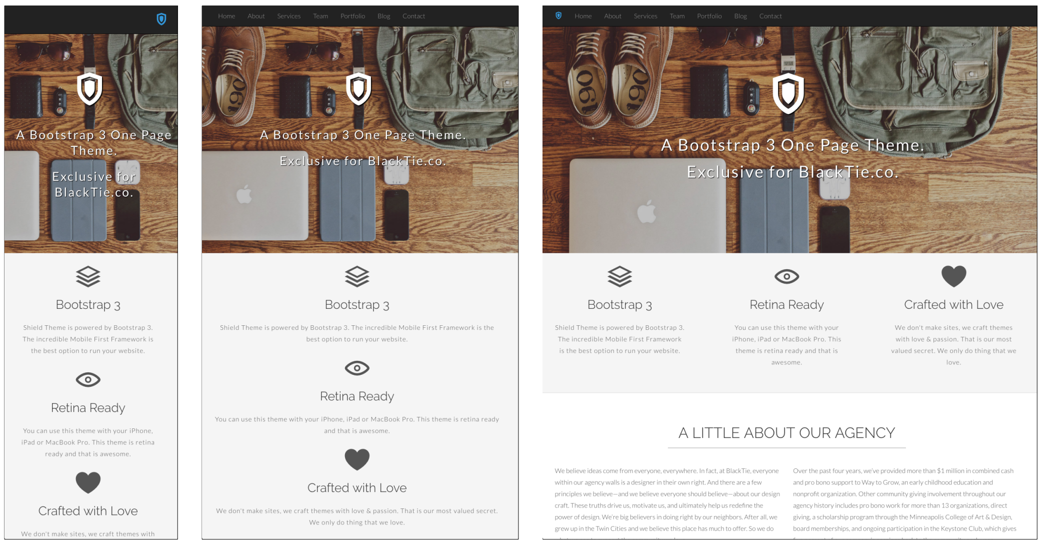
2. BACKGROUND, PROBLEM STATEMENT, AND MOTIVATING EXAMPLE

This section begins by discussing responsive web design in more detail, describing how its various component techniques are implemented. It then introduces the problem addressed by this paper: detecting regression changes in the layout of responsive web pages. Finally the section introduces the alignment graph, a data structure that supports the automated detection of layout issues in cross-browser web page testing. The alignment graph introduces some concepts that are key to this paper, and which inspired the responsive layout graph and our own automated approach for detecting regressions from correct responsive web page layout, which we introduce in Section 3.

2.1. Responsive Web Design

Figure 1 furnishes an example of a responsively designed web page, called "Shield", that is an open-source template available under the Creative Commons Attribution 3.0 License. These three screenshots showcase the dynamic changes to the page's layout at three different viewport widths that are similar to those of the three most common types of devices (i.e., a smartphone, a tablet, and a desktop computer). In the narrow viewport width of part (a), the design stacks all major content in a single, narrow column, while the navigation links are part of a drop-down list that is hidden from the user until they interact with it. Next, part (b) shows the slightly widened resolution where the single column layout for the main content remains, albeit with more horizontal space that allows the design to replace the drop-down list with a navigation bar at the top of the page. As the viewport width further expands to the largest one shown in part (c), the design spreads out the previously stacked content into a three-column layout, making use of the extra space afforded to the page.

Responsive web design consists of three "ingredients": a grid-based layout, flexible images, and media queries [4]. The former two focus on the resizing of various web page elements in order to best fit the viewing environment. In the context of Figure 1(c) as an example, a developer would traditionally use static width declarations such as "`width:380px`" to implement the row of content tiles. In contrast, grid-based layouts and flexible media use "fluid" width declarations, in which an element's width is defined as a percentage of its container (e.g.,"`width:33.3%`"). This allows the web page to handle both narrower and wider viewports more effectively. Applying this methodology


A Bootstrap 3 One Page Theme.
Exclusive for BlackTie.co.
Bootstrap 3
Retina Ready
Crafted with Love
A LITTLE ABOUT OUR AGENCY

(a) 420 pixels wide (b) 768 pixels wide (c) 1200 pixels wide

Figure 1. The “Shield” template for a responsively designed web page (*http://www.blacktie.co/demo/shield*), showing how the layout of a responsive page adjusts to different viewport widths. The Shield site is an open-source template, also used as an experimental subject in this paper, that employs the Bootstrap responsive web design framework available under the Creative Commons Attribution 3.0 License. (This figure originally appeared in the six-page ASE 2015 “new ideas” conference version of this paper [16].)

to all relevant HTML elements builds the foundations for a responsive web page, as each element will be rendered at a size appropriate for the screen size. The final ingredient of RWD, media queries, is a module of CSS3 that allows a developer to activate a set of CSS rules if and only if a specific set of conditions are met. While the media queries specification [17] provides a wide range of options, the most commonly used approach inspects the width of the viewport using either the `min-width` or `max-width` query. For instance, the aforementioned fluid width declaration would be placed into the media query “`@media (min-width:1200px)`”, so that the web page only lays the tiles out in a single row at larger viewport widths where there is enough room to do so.

In the aforementioned media query, the viewport width of 1200 pixels is an instance of a “breakpoint” [4] that, in the context of responsive web design and CSS, is some value whereby different CSS rules will apply to particular element(s) on either side of the pixel boundary. The use of breakpoints allows developers to implement different styles of layout that make the best use of the space afforded by the device in use, and developers often set them based on the viewport widths of common devices or device classes (i.e., mobile phones, tablets, and desktops).

Even with these three ingredients, creating an easy-to-use and aesthetically pleasing responsive web site is a challenging process since both the HTML and the style sheet must be carefully constructed to ensure that the correct rules are applied to the intended elements for all of the various types of devices, thereby producing the intended sizing and layout behavior. Developers frequently choose to leverage one of the many front-end frameworks, such as Twitter’s Bootstrap [18] (used by 21% of the top 1 million sites [19] in 2017) or Zurb’s Foundation [20] (used by organizations such as Dropbox and SlideShare [21]). These frameworks provide CSS rules for an initial set of reusable and common layout components, such as a flexible grid-based layout system and responsive navigation options, thus often making the creation of a responsive web site much easier. Yet, these frameworks for responsive web design are also complex. According to the SCC tool that counts lines of source code [22], a recent version of Bootstrap has about 4,500 lines of HTML, 17,800 lines of CSS, and 41,500 lines of JavaScript — and is therefore potentially difficult for developers to fully understand. Additionally, while these frameworks are useful, they only represent starting points for a mobile-friendly design and thus require customization to ensure a unique, yet cohesive, look and feel.

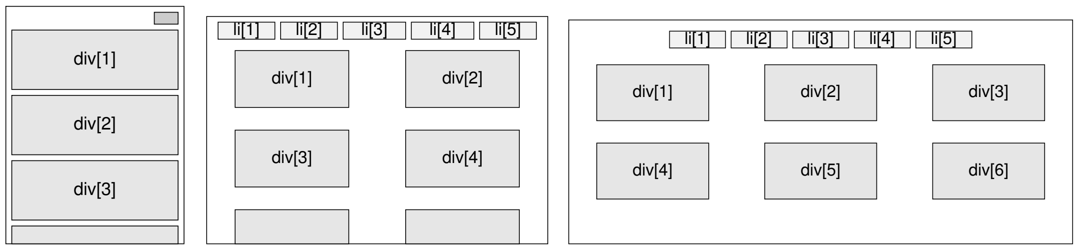
As with all other forms of software, responsive web pages must be tested to ensure their designs are free of issues and that they display correctly. Unfortunately, responsive web pages present several fairly unique and well-documented problems. For instance, one quality assurance team identified three main challenges: the testing environment (i.e., there is a great number of devices and browsers), the difficulty of testing responsive web pages (i.e., testers may not have training in testing this new type of site) and the lack of an effective and usable test automation framework, thereby making testing both time-consuming and error-prone [23]. In the next subsection, we illustrate a particular, but common, problem that we address in this paper with an automated technique, namely that of detecting the layout regressions following changes to responsively designed web pages.

2.2. The Problem Addressed by This Paper: Detecting Regressions in the Layout of Responsive Web Pages

The example in Figure 2 demonstrates how changes to a web page's CSS rules can result in undesirable side effects (i.e., regressions) in the page's layout that are not intended by the developer and often manifest at unexpected viewport widths. If these regressions go unnoticed and more changes are made to the page, then developers may struggle to recall the initial CSS modification(s) that caused the problem, thereby requiring extra effort to diagnose and fix them. This example underscores the challenge of tweaking a responsive design without the support of the automated technique presented in this paper: each time a change is made, all resolutions need to be manually and fully rechecked for unintended side-effects due to the interplay between elements as the viewport size changes, which is a time-consuming and error-prone process. When checking a web page at many viewport widths — and potentially performing a lot of vertical scrolling on web sites designed as single pages — developers can easily miss regression issues during manual inspections, which can have wide-ranging detrimental impacts if they make it onto the live web site.

Suppose that a developer is modifying the web page in Figure 2, focussing on improving the widescreen layout shown in part (a-iii). Noticing the unused space on either side of the navigation links (i.e., `li[1]`–`li[5]`) — and considering a design change involving the addition of icons to the existing text — they modify the `width` declaration for the navigation links so that it increases from 140 pixels to 150 pixels. This CSS change is shown in part (b-i) of Figure 2, with the new appearance of the web page at the widescreen layout shown by part (b-iii). Since the developer's main priority with this code change is the widescreen layout, they observe the widened links, thus confirming the tweak had the intended effect while, perhaps, assuming that it did not introduce inadvertent changes to the page's layout. Depending on the viewport widths at which the developer chooses to test, they may not become aware of regressions caused at other viewport widths. Systematically reducing the viewport width until the navigation links are too wide to fit in a single row will lead to the discovery that the code change had a previously unseen and unintended side effect: now, the last link wraps onto a second row. While the developers did preserve the page's functionality, as `li[5]` can still easily be clicked, the regression from the correct layout of the web page could detrimentally influence a person's ability to easily navigate the site and their perception of the site's quality. Other layout regressions could be more severe, such as unintentionally obscured text or unclickable buttons, impacting the functionality of a web page and having potentially costly repercussions.

Due, at least in part, to the relative dearth of automated testing tools, developers of mobile-friendly web pages are often limited to performing regression tests in the form of a "spotcheck" that involve checking a site at several viewports under the assumption that any layout issues would manifest themselves in at least one of those chosen for inspection. Spotchecking is often used to cover as many devices as possible while minimizing effort, or to target specific popular devices identified through techniques such as web analytics. As such, the widths chosen by a tester often correspond to particular device types, such as the three viewport widths shown in Figure 1 which represent three common device types (e.g., smartphone, tablet, and desktop/laptop). However, as the viewport widths at which unintended side-effects are observable are often unpredictable and not associated with any common devices, spotchecking is highly unlikely to be a reliable means of detecting them.

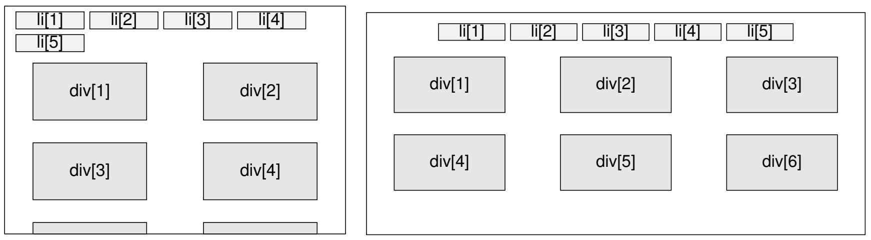


(a-i) 400 pixels wide (a-ii) 800 pixels wide (a-iii) 1200 pixels wide

(a) Original version of a web page

```
@media(min-width: 768px) {
  /* original rule */
  li {
    width: 140px;
  }

  /* modified rule */
  li {
    width: 150px;
  }
}
```



(b-i) CSS code (b-ii) Modified version at 800 pixels wide (b-iii) Modified version at 1200 pixels wide

(b) Modified version of a web page showing a regression from the correct layout

Figure 2. 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. (This figure originally appeared in the six-page ASE 2015 "new ideas" conference version of this paper [16].)

As such, this paper presents and evaluates an automated technique for addressing the problem. The basis of this approach is a model of the dynamic layout of the web page, called the *responsive layout graph* (RLG). This model is inspired, in part, by a prior modeling approach to web pages at single viewport widths, known as the alignment graph, which we discuss in the next subsection.

2.3. The Alignment Graph

As part of a technique for automatically detecting cross-browser issues (XBIs), where two browsers render the same web page differently, Choudhary et al. [24] proposed the *alignment graph*. An alignment graph (AG) models the relative alignments of web page elements with one another. Choudhary et al.'s technique extracted the alignment graphs of the web page rendered in two browsers and compared the two graphs to automatically detect differences in page layout.

In the alignment graph, each page element forms a node. Edges exist between nodes of the graph if their respective web page elements are in one of two relationships — the *parent-child* relationship, where when one element the *parent*, the direct container of another element, the *child*; or the *sibling* relationship, where two elements have the same parent. The alignment graph models the relative alignment of each pair of elements in a parent-child or sibling relationship with a set of *alignment attributes*, which label the edges of the graph. For instance, if a child element may be center-justified within its parent, the set of attributes for the edge in the alignment graph would include the CJ attribute; if it were left or right-justified then the set would include the LJ or RJ attribute, respectively. Meanwhile, a sibling above or below another would be labeled with either the A or B attribute respectively, or L or R if it was situated to the left or right. A browser represents the

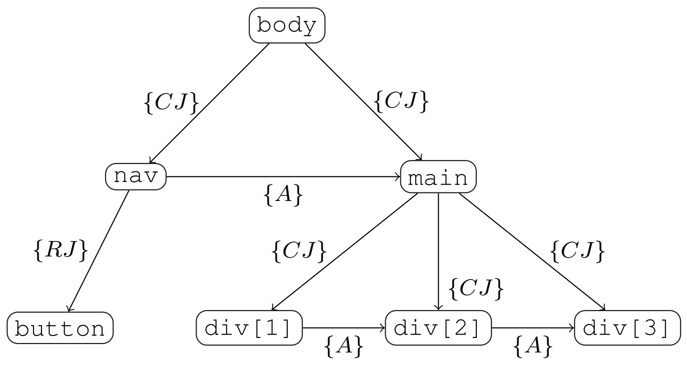


Figure 3. An example alignment graph of the web page's layout featured in Figure 2. This alignment graph is at a viewport width of 400 pixels wide, as shown by the wireframe of part (a-i) of that figure.

structure of each page it renders through its Document Object Model (DOM), a World Wide Web Consortium (W3C) standard [25]. A developer can query or change the structure, style, or content of a web page using JavaScript. The DOM uniquely identifies each HTML element with an *XPath* (XML Path Language), a special syntax for selecting nodes in an XML document [26].

The XBI-identification technique of Choudhary et al. [24] extracts an alignment graph by querying the DOM of a page. It first finds the nodes of the graph by discovering which HTML elements are present on the page. Using the XPaths of each element, it then finds their positions, widths, and heights, discovering their minimum bounding boxes. Using this information, the technique can ascertain layout relationships (i.e., parent-child or sibling) in order to create edges between nodes and the alignment attributes to assign to each edge. Choudhary et al. formally define an alignment graph as a five-tuple $\mathcal{AG} = (\mathcal{E}, \mathcal{R}, \mathcal{T}, \mathcal{Q}, \mathcal{F})$, where $\mathcal{E}$ is the set of nodes of the graph (i.e., web page elements) and $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ is a set of element pairs in one of the relationships in $\mathcal{T} = \{pc, s\}$ (i.e., parent-child or sibling). $\mathcal{Q}$ is the set of alignment attributes featuring in the graph (e.g., A, above and L, left-of), and $\mathcal{F} : \mathcal{R} \rightarrow 2^{\mathcal{Q}}$ is a function that maps edges to their attribute types.

For example, Figure 3 shows the alignment graph of the page of the motivating example shown by Figure 2, and the specific layout of the page at 400 pixels wide. The graph involves nodes corresponding to elements that are not apparent to the end user (i.e., `body`, `nav` and `main`), and so do not appear in the wireframe of the page shown by Figure 2(a-i), but yet form a crucial part of the page's HTML structure and hence its DOM. The root node of the graph is the overall containing element of every HTML page, known as the `body` element. Within the `body` element are the `nav` navigation bar, containing the single `button` element visible at this viewport width, and the `main` element, devoted to the main content of the page, containing the three `div` elements, `div[1]`–`div[3]`. Solid edges between nodes represent parent-child relationships between elements, while dashed edges denote sibling relationships. Each edge is labelled with relative alignment attributes for the elements concerned. For example, the button is right-justified in regard to its parent container, `nav`, so the edge between these elements has the RJ label; while other child elements are center-justified in their containers, and so their respective edges have the CJ attribute. In terms of sibling edges, `nav` is above `main`, and so the dotted edge between the nodes corresponding to these elements has the A attribute, and likewise for `div[1]` and `div[2]`, and `div[2]` and `div[3]`. Note that, for ease of presentation, the alignment graph in this figure does not include reciprocal or transitive sibling edges (e.g., `main` to `nav`, and `div[1]` to `div[2]`, respectively).

As shown by this example, the alignment graph only models the layout of a web page at a single viewport width, which is the 400 pixel width layout of the web page in Figure 1. It cannot simultaneously model the differing layouts at 800 and 1200 pixels wide, making it insufficient, on its own, for accurately representing the responsive layout of a web page. However, in Section 3 of this paper, we build on the concepts of the alignment graph, ultimately presenting an approach that extracts and models the range of layouts possible with a responsively designed web page at different viewport widths, leveraging a representation that we refer to as the *responsive layout graph*.

3. REDECHECK-*RM*: AN AUTOMATED APPROACH TO IDENTIFYING REGRESSIONS FROM CORRECT RESPONSIVE WEB LAYOUTS

As the foundation of our automated approach to identifying web page layout regressions, this paper presents a representation that models the dynamic layout of a responsive web page, called the responsive layout graph (RLG). The RLG takes account of the two critical aspects of responsive web design: the changing visibility and changing relative alignment of web page elements. This graph was inspired, in part, by the modeling of the relative layout of elements in the alignment graph of Choudhary et al. [24], as introduced in the last section. However, while Choudhary et al. showed the alignment graph to be highly effective at both modelling the layout of a web page and detecting differences in that layout across different web browsers at a single resolution, it provides no support for representing the dynamic nature of responsive web pages, such as elements appearing/disappearing and rearranging as the viewport changes. This makes the AG less than suitable for this paper's problem domain, and the core reason why this paper presents the RLG, which comprehensively models a web page's dynamic layout across all viewport widths in a pre-defined range. After formally defining the responsive layout graph in Section 3.1, the remainder of this section explains the process of extracting the RLG of a web page and presents the method that highlights changes to a page's layout — that may be regressions from the correct layout — by comparing "before" and "after" versions of an RLG, a process that we refer to as *RLG differencing*.

Figure 4 illustrates a potential scenario for the use of our approach in a real-world development environment, which we link to the various parts of our motivating example as shown in Figure 2 and described in Section 2.2. In this scenario, the developer is working on the stable version of the web page $\mathcal{W}_{curr}$ (i.e., part (a) of Figure 2). They then make a change to the page's source code (part (b-i) of Figure 2). This change results in the modified web page $\mathcal{W}_{mod}$ (whose layout at two viewport widths is seen in parts (b-ii) and (b-iii) of the same figure). Our approach takes the two versions of the page (i.e., $\mathcal{W}_{curr}$ and $\mathcal{W}_{mod}$) as input. As shown by Figure 4, the *RLG Extraction* component of our approach analyzes $\mathcal{W}_{curr}$ and $\mathcal{W}_{mod}$ to produce models $\mathcal{RLG}_{curr}$ and $\mathcal{RLG}_{mod}$, respectively, before the *RLG Differencing* component compares the two RLG models. Once the approach has generated and output its report, the developer must manually analyze each RLG difference the report highlights and decide whether it represents an unintended side effect of the source code modification, rather than an intentional change to the page's layout. The manual effort required is potentially significantly reduced in comparison to a user resizing their browser window and manually searching for layout issues. If any such issues exist, then the developer must fix them before moving on and repeating the process shown by the figure. If the developer decides that there are no unintended layout changes, then $\mathcal{W}_{curr}$ can be updated to $\mathcal{W}_{mod}$, as any subsequent code changes need to be checked against the most recent version of the web page. Ultimately, this iterative process stops when the web developer no longer needs to change the HTML and CSS source code of $\mathcal{W}_{curr}$ and there are no layout issues that the developer needs to then go on to repair.

3.1. Definitions and an Example

Given a set of nodes, $\mathcal{E}$, representing the elements contained within a web page $\mathcal{W}$, we define two types of constraint that are used to describe the layout of $\mathcal{W}$:

Definition 1. VISIBILITY CONSTRAINT. A visibility constraint vc, for some node $n \in \mathcal{E}$, is a pair (x_1, x_2), where x_1 is a lower viewport width and x_2 is an upper viewport width (i.e., $x_2 > x_1$), 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 next definition of the *alignment constraint* draws on previously described relative layout concepts originally due to the alignment graph, as introduced in Section 2.3. These include the notion of relationship types between web page elements, and the set $\mathcal{T} = \{pc, s\}$, where pc and s signify a *parent-child* and *sibling* relationship, respectively, and the set of attributes $\mathcal{P}$ defining the

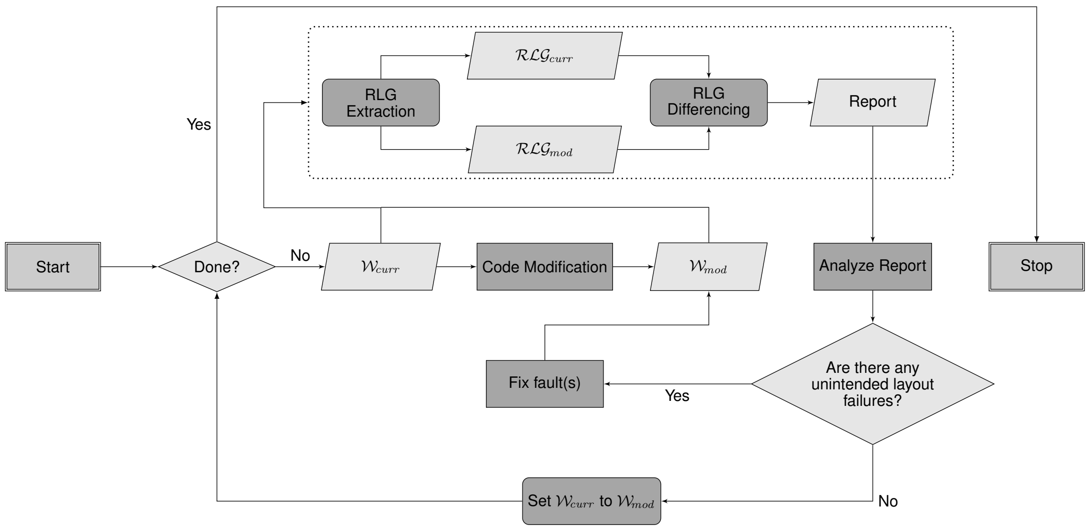


Figure 4. The main usage scenario of this paper's approach to automatically identifying the layout regressions in a responsive web page. In this diagram, the presented approach is contained within the dotted rectangle, inputs and outputs are shown as parallelograms, the decision is depicted as a diamond and automated and manual processes are shown as rectangles with and without rounded corners, respectively (i.e., "RLG Extraction" is automated and "Analyze Report" is manual). This iterative process stops when the developer of a responsively designed web page is done fixing faults, finished making modifications to the page's HTML and CSS, and the report suggests that there are no layout regressions. The dotted section corresponds to the internal working of REDECHECK-*RM*, which we expand in Figure 6.

specifics of the relative layout of two elements with respect to one another. Alignment constraints are used to represent the dynamic nature of responsive web pages, as multiple alignment constraints can exist between a pair of web page elements, describing how their relative layout adapts to the changing constraints induced by different viewport widths.

Definition 2. ALIGNMENT CONSTRAINT. An alignment constraint ac, for a pair of nodes n_1, $n_2 \in \mathcal{E}$, is a 4-tuple $(x_1, x_2, t, \mathcal{P})$, where $t \in \mathcal{T} = \{pc, s\}$ denotes the relationship type (parent-child or sibling) and $\mathcal{P} \in 2^{\mathcal{Q}}$ is a set of alignment attributes that describes the relative layout of n_1 and n_2 between the viewport range of x_1 and x_2; where x_1 and x_2 represent the inclusive lower and upper viewport widths of the range, respectively (i.e., $x_2 > x_1$). (See Section 3.2.2 for a full discussion of attributes that alignment constraints support.)

Web pages are, by nature, hierarchical, with the `<body>` tag representing the "root" of the web page and therefore containing all subsequent elements. Because of this, the RLG is also hierarchical in nature, forming a tree-like structure with visibility constraints added to individual nodes and alignment constraints being added to the edges linking the nodes. Figure 5 shows a snippet of the responsive layout graph for the wireframe example of a web page presented in Figure 2. Each web page element is represented as a node in the graph. The node labels shown in fixed-width font identify each element, as the full unique XPath expressions have been omitted to make the diagram easier to interpret. The root node of the graph is that corresponding to the `body` element, the container for all other elements of the page. This element is the parent of the `nav` element, which contains all navigational elements, including the unordered list element `ul` and its children `li[1]` through `li[5]`, which serve as the menu items at wider viewports. At narrower viewports, this list is replaced by the `button` element. The `body` element is also the parent of the `main` element, which contains all of the content of the web page, represented by the elements `div[1]` through `div[3]`.

The visibility constraints for each element are displayed above the node labels. The variables w_{min} and w_{max} represent the minimum and maximum viewport widths taken into account by the RLG, and therefore elements that are constantly present in the web page have the visibility constraint (w_{min}, w_{max}). Meanwhile, the nodes representing the navigation list (which only appear when the

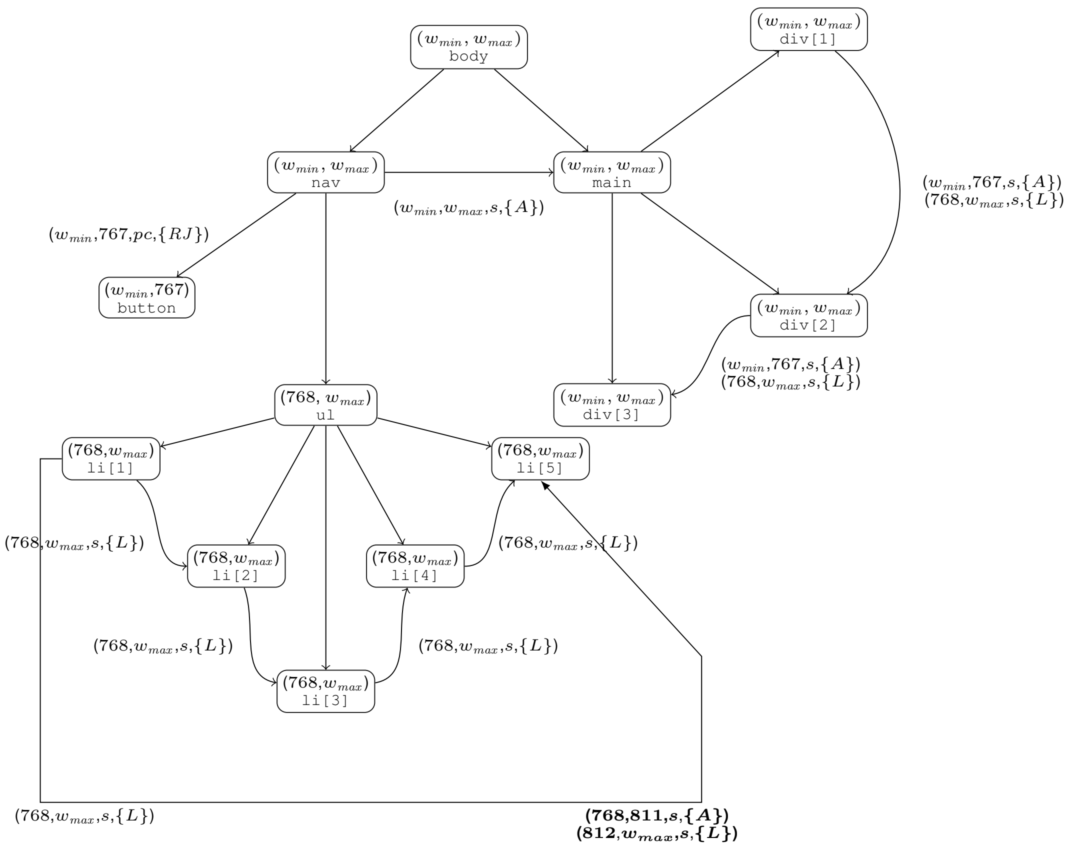

(w_{min}, w_{max}) body
(w_{min}, w_{max}) div[1]
(w_{min}, w_{max}) nav
(w_{min}, w_{max}) main
$(w_{min}, w_{max}, s, \{A\})$
$(w_{min}, 767, s, \{A\})$
$(768, w_{max}, s, \{L\})$
$(w_{min}, 767, pc, \{RJ\})$
$(w_{min}, 767)$ button
(w_{min}, w_{max}) div[2]
$(w_{min}, 767, s, \{A\})$
$(768, w_{max}, s, \{L\})$
$(768, w_{max})$ ul
(w_{min}, w_{max}) div[3]
$(768, w_{max})$ li[1]
$(768, w_{max})$ li[5]
$(768, w_{max}, s, \{L\})$
$(768, w_{max})$ li[2]
$(768, w_{max})$ li[4]
$(768, w_{max}, s, \{L\})$
$(768, w_{max}, s, \{L\})$
$(768, w_{max}, s, \{L\})$
$(768, w_{max})$ li[3]
$(768, w_{max}, s, \{L\})$
$\mathbf{(768, 811, s, \{A\})}$
$\mathbf{(812, w_{max}, s, \{L\})}$

Figure 5. An example of an RLG for the web page, shown in the wireframe example of Figure 2, over a range of viewport widths between w_{min} and w_{max}. For ease of presentation, we omit some edges and constraints. The constraints on the edge `li[1]` to `li[5]` represent the relative layout of the two elements in W_{curr} and W_{mod}, respectively, where the difference is the unseen side effects caused by the code change, in which `li[5]` is forced to wrap onto a new row. The bold-faced annotation is for the code change in Figure 2.

viewport is wide enough) have the visibility constraint $(768,\, w_{max})$, and the drop-down `button` element that disappears at wider viewport widths has a visibility constraint of $(w_{min},\, 767)$.

The directed edges between nodes in the example graph represent the relationships between elements described previously, with each edge containing one or more alignment constraints describing the relationship. For instance, the alignment constraints $(w_{min},\, 767,\, s,\, \{A\})$ and $(768,\, w_{max},\, s,\, \{L\})$ between nodes `div[1]` and `div[2]` represent `div[1]` being above (labelled "A") `div[2]` at narrow viewport widths, before shifting to a side-by-side alignment in which `div[1]` is left of (labelled "L") `div[2]` once the viewport is wide enough to allow it.

The RLGs for parts (a) and (b) of Figure 2 are almost identical, with the only difference being the sibling relationships between `li[5]`, which wraps onto a second row, and its neighbors, which are shown in bold. For simplicity and space reasons, only the edge between `li[1]` and `li[5]` is shown in the figure. For example, in part (a), the edge from `li[1]` to `li[5]` has a single alignment constraint $(768,\, w_{max},\, s,\, \{L\})$, whereas in part (b), the layout failure results in two different alignments being found, corresponding to `li[1]` initially being above `li[5]` as `li[5]` wraps, and then to the left of it, once the viewport is wide enough to fit all five links in a single row.

Formally, we define the RLG as a 4-tuple $\mathcal{RLG} = (\mathcal{E},\, \mathcal{R},\, \mathcal{F}_{\mathcal{VC}},\, \mathcal{F}_{\mathcal{AC}})$, where $\mathcal{E}$ is the complete set of nodes, one for each web page element, that is displayed for at least one viewport width between w_{min} and w_{max}, and $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ is the set of edges for pairs of nodes for which at least one alignment constraint exists. Next, $\mathcal{F}_{\mathcal{VC}} : \mathcal{E} \rightarrow 2^{\mathcal{VC}}$ is a function mapping an element to a set of visibility constraints where $\forall e \in \mathcal{E},\, |\mathcal{F}_{\mathcal{VC}}(e)| \geq 1$ (i.e., each element is visible for at least one

Algorithm 1 RLG Extraction

```
 1: procedure EXTRACTRESPONSIVELAYOUTGRAPH($\mathcal{W}$, $w_{min}$, $w_{max}$, $w_{step}$, $use\_breakpoints$)
 2:     $\mathcal{S} \leftarrow$ GETSAMPLEWIDTHS($\mathcal{W}$, $w_{min}$, $w_{max}$, $w_{step}$, $use\_breakpoints$)
 3:     $\mathcal{G} \leftarrow$ EXTRACTALIGNMENTGRAPHS($\mathcal{W}, \mathcal{S}$)
 4:     $\mathcal{F_{VC}} \leftarrow$ EXTRACTVISIBILITYCONSTRAINTS($\mathcal{G}$)
 5:     $\mathcal{F_{AC}} \leftarrow$ EXTRACTALIGNMENTCONSTRAINTS($\mathcal{G}, \mathcal{F_{VC}}$)
 6:     $\mathcal{E} \leftarrow$DOM($\mathcal{F_{VC}}$)
 7:     $\mathcal{R} \leftarrow$DOM($\mathcal{F_{AC}}$)
 8:     $\mathcal{RLG} = (\mathcal{E}, \mathcal{R}, \mathcal{F_{VC}}, \mathcal{F_{AC}})$
 9:     return $\mathcal{RLG}$
10: end procedure
```

viewport width); and $\forall e \in \mathcal{E}$ and $\forall vc_a = (x_{1_a}, x_{2_a}) \in \mathcal{F_{VC}}(e)$ and $\forall vc_b = (x_{1_b}, x_{2_b}) \in \mathcal{F_{VC}}(e)$, if $vc_a \neq vc_b$, then $x_{1_a} > x_{2_b} \vee x_{2_a} < x_{1_b}$ (i.e., for a particular viewport width, there is at most one visibility constraint for a web page element). Finally, $\mathcal{F_{AC}} : \mathcal{R} \rightarrow 2^{\mathcal{AC}}$ is a function that maps an edge to a set of alignment constraints, such that $\forall r \in \mathcal{R}$ and $\forall ac_a = (x_{1_a}, x_{2_a}, t_a, \mathcal{P}_a) \in \mathcal{F_{AC}}(r)$ and $\forall ac_b = (x_{1_b}, x_{2_b}, t_b, \mathcal{P}_b) \in \mathcal{F_{AC}}(r)$, if $ac_a \neq ac_b$, then $x_{1_a} > x_{2_b} \vee x_{2_a} < x_{1_b}$. That is, for a particular viewport width, there is at most one alignment constraint for a pair of web page elements.

3.2. Automatically Extracting the Responsive Layout Graph

Algorithm 1 describes the high-level process of extracting an RLG for a given web page, specified as the procedure EXTRACTRESPONSIVELAYOUTGRAPH. The EXTRACTRESPONSIVE-LAYOUTGRAPH procedure takes as input a web page $\mathcal{W}$, and a series of further configuration parameters, which are passed directly into the GETSAMPLEWIDTHS procedure on line 2. The GETSAMPLEWIDTHS procedure, which we elaborate on further in the next section (Section 3.2.1), determines a series of viewport widths $\mathcal{S}$ to use as a sample of the responsive layout of the page.

The algorithm then calls the EXTRACTALIGNMENTGRAPHS procedure (line 3), which we discuss in detail in Section 3.2.2. EXTRACTALIGNMENTGRAPHS extracts the DOM of the web page at each viewport width w in $\mathcal{S}$, captures an alignment graph g of the relative alignment of HTML elements at that viewport width, and places each pair $(w,\ g)$ into the list $\mathcal{G}$. The algorithm then extracts the visibility constraints map $\mathcal{F_{VC}}$ of the RLG through a call to the procedure EXTRACTVISIBILITYCONSTRAINTS (line 4), which we discuss in more detail in Section 3.2.3. The algorithm later uses the domain of the $\mathcal{F_{VC}}$ map to assign the nodes of the RLG, denoted $\mathcal{E}$ (line 6). After extracting visibility constraints, the algorithm then extracts the alignment constraints map of the RLG, $\mathcal{F_{AC}}$, through a call to the EXTRACTALIGNMENTCONSTRAINTS procedure (line 5), which we discuss in more detail in Section 3.2.4. The algorithm later uses the domain of the $\mathcal{F_{AC}}$ map to assign the edges of the RLG, denoted $\mathcal{R}$ (line 7). Finally, the algorithm assembles the RLG from its constituent parts and returns it (lines 8–9).

3.2.1. Sampling the Target Web Page. For responsively designed web pages, the relative alignment of HTML elements changes with the viewport width. In order to build an accurate RLG, the EXTRACTRESPONSIVELAYOUTGRAPH function needs a set of viewport widths that is representative of the different ways in which the page lays out its elements in response to the space available. Obtaining this set of widths from an arbitrary web page is a non-trivial task. This is the job of the GETSAMPLEWIDTHS procedure, called from line 2 of Algorithm 1. Since each web page is different, the set of widths characterizing each of its different layouts will also different.

One possibility is to use *every* width in the range covered by the responsive layout graph. However, this method has the potential to be time consuming, resulting in a technique that is costly for developers to run in the usage scenario depicted by Figure 4. An alternative is to sample widths at intervals throughout the viewport range from w_{min} and w_{max}. However, if the interval is too large, the technique may “skip over” some important changes to the page’s layout. A third possibility is to investigate widths corresponding to breakpoints programmed into the page’s CSS files (for example the 768 pixel breakpoint illustrated in Figure 2). On its own, however, this is likely to be insufficient to correctly extract the full responsive layout, since although breakpoints explicitly define the viewport widths at which the layout of a page changes, the dynamic resizing of elements in accordance with RWD principles can subsequently cause changes in the alignment of elements at viewport widths not explicitly defined by the web page's cascading style sheets.

Therefore, this paper evaluates three different methods, (1) the exhaustive approach, which we refer to as EXTRACT-COMPARE-EXHAUSTIVE; (2) the interval sampling approach, which we refer to as EXTRACT-COMPARE-INTERVAL; and finally (3) interval sampling combined with viewport widths either side of breakpoints defined by the page's CSS code, which we refer to as EXTRACT-COMPARE-INTERVAL-BREAKPOINTS. Each method involves calling the GETSAMPLEWIDTHS function (called from line 2 of Algorithm 1), called for a web page $\mathcal{W}$, but with different parameters. While the w_{min} and w_{max} parameters are set for each method, representing the minimum and maximum widths of a range of viewports to investigate, respectively, w_{step} represents an interval size that is 1 for EXTRACT-COMPARE-EXHAUSTIVE, and a configurable value greater than 1 for EXTRACT-COMPARE-INTERVAL and EXTRACT-COMPARE-INTERVAL-BREAKPOINTS. Finally, $use\_breakpoints$ is a flag that instructs GETSAMPLEWIDTHS to include widths at either side of breakpoint values found in the CSS code of the page. When $use\_breakpoints$ is set to true, the GETSAMPLEWIDTHS function parses the HTML of $\mathcal{W}$ to extract the set of all referenced CSS files. It then parses each CSS file and analyzes each of its media queries to determine breakpoint boundary values. For example, given a media query "`@media(min-width:992px)`", the two boundary values added to $\mathcal{S}$ would be 991 pixels and 992 pixels. This is because 992 pixels is the smallest viewport width to trigger the rule, while 991 pixels is the largest width *not* to trigger it.

The output of any one of the three aforementioned sampling procedures is a series of width values, assigned to $\mathcal{S}$ in the EXTRACTRESPONSIVELAYOUTGRAPH procedure of Algorithm 1. For instance, given the media query "`@media(max-width:768px)`" appearing in a page's CSS, a w_{step} value of 60, and w_{min} and w_{max} values of 400 and 1400, respectively, the set of widths $\mathcal{S}$ returned by GETSAMPLEWIDTHS for the EXTRACT-COMPARE-INTERVAL-BREAKPOINTS method is $\{400, 460, 520, \ldots, 760, 768, 769, 820, \ldots, 1340, 1400\}$, and would therefore be the set of widths assigned to $\mathcal{S}$ by Algorithm 1 on line 2. As we explain in Section 4.5, the values for w_{min} and w_{max} used in this example are the same as those used in our experiments, and the value of w_{step} (i.e., 60) is the step size value used for our first two research questions.

3.2.2. Generating Single Viewport Page Layout Graphs. Called on line 3 by Algorithm 1, the EXTRACTALIGNMENTGRAPHS procedure takes the web page $\mathcal{W}$ and set of viewport widths $\mathcal{S}$, and for each width in $\mathcal{S}$, extracts the DOM of the page to generate an alignment graph of its layout at that particular viewport. It does this using the bounding boxes of each element, which have coordinates (x_1, y_1, x_2, y_2). To illustrate this process, we revisit the motivating example in Figure 2.

The EXTRACTALIGNMENTGRAPHS procedure beings by organizing elements into a graph structure with either "parent-child" or "sibling" edges connecting them. The procedure generates parent-child edges when, through querying the DOM (as described in Section 2.3), an element e_c is contained within the bounding box of some other element e_p. The procedure then generates sibling edges for elements sharing the same parent. Consider the layout shown by part (a-i) of the figure. Suppose the elements `div[1]`, `div[2]`, and `div[3]` are contained within another HTML `div` element, which we will refer to as `main`. The EXTRACTSINGLEVIEWPORTLAYOUTGRAPHS function generates parent-child edges between `main` and `div[1]`, `div[2]`, and `div[3]`, and generates sibling edges between each of `div[1]`, `div[2]`, and `div[3]`.

The procedure then inspects the DOM 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 be aligned on its top (TE) and/or bottom edge (BE). Similarly, if an element is above (A) or below (B) its sibling, it may also be aligned on its left (LE) and/or right edges (RE). As examples of parent-child alignments, `div[1]` through `div[3]` have the same horizontal midpoints

Algorithm 2 Visibility Constraint Extraction

```
1: procedure EXTRACTVISIBILTYCONSTRAINTS($\mathcal{G}$)
2:     $\mathcal{F_{VC}} \leftarrow \perp$
3:     $(w_p, g_p = (\mathcal{E}_p, \ldots)) \leftarrow$FIRST($\mathcal{G}$)
4:     for all $e \in \mathcal{E}_p$
5:         $\mathcal{F_{VC}} \leftarrow$ADDNEWVISIBILITYCONSTRAINT($\mathcal{F_{VC}}, e, w_p$)
6:     while HASMOREELEMENTS($\mathcal{G}$)
7:         $(w_c, g_c = (\mathcal{E}_c, \ldots)) \leftarrow$NEXT($\mathcal{G}$)
8:         for all $e \in \mathcal{E}_c \setminus \mathcal{E}_p$
9:             $w_{first} \leftarrow$ BINARYSEARCHFORVISIBILITYCHANGE($e, w_p, w_c$, appear)
10:            $\mathcal{F_{VC}} \leftarrow$ADDNEWVISIBILITYCONSTRAINT($\mathcal{F_{VC}}, e, w_{first}$)
11:        for all $e \in \mathcal{E}_p \setminus \mathcal{E}_c$
12:            $w_{last} \leftarrow$ BINARYSEARCHFORVISIBILITYCHANGE($e, w_p, w_c$, disappear)
13:            $\mathcal{F_{VC}} \leftarrow$COMPLETEVISIBILITYCONSTRAINT($\mathcal{F_{VC}}, e, w_{last}$)
14:        $w_p \leftarrow w_c$
15:        $\mathcal{E}_p \leftarrow \mathcal{E}_c$
16:    for all $e \in \mathcal{E}_p$
17:        $\mathcal{F_{VC}} \leftarrow$COMPLETEVISIBILITYCONSTRAINT($\mathcal{F_{VC}}, e, w_p$)
18:    return $\mathcal{F_{VC}}$
19: end procedure
```

as their container, resulting in the EXTRACTALIGNMENTGRAPHS procedure applying the alignment attribute CJ applied to the edges between themselves and their parent (`main`). As example of sibling edge labelling, `div[1]` is above `div[2]` (`div[1]`'s y_2 value is greater than `div[2]`'s y_1 value), and so the EXTRACTALIGNMENTGRAPHS procedure annotates the edge with the A attribute. Relationships between web page elements are symmetric in nature, especially those of the sibling type, as two edges representing *X is above Y* and *Y is below X* respectively, are intuitively identical. Thus, for simplicity and efficiency, the procedure only adds one of the two possible edges.

The EXTRACTALIGNMENTGRAPHS procedure returns a list $\mathcal{G}$ pairing each viewport width w in $\mathcal{S}$ with its associated alignment graph g. Algorithm 1 then proceeds to extract visibility and alignment constraints to complete the RLG using $\mathcal{G}$, which we next discuss in detail.

3.2.3. Determining Visibility Constraints. The process for extracting visibility constraints is described by Algorithm 2, and the EXTRACTVISIBILITYCONSTRAINTS procedure, which takes as input the sequence of viewport width-alignment graph pairs, $\mathcal{G}$, arranged in order of ascending viewport width. The algorithm begins by taking the first alignment graph in $\mathcal{G}$ (line 3) using it to initialize visibility constraints for each element e in the graph — which will become part of the eventual RLG — via a call to the procedure ADDNEWVISIBILITYCONSTRAINT. This procedure takes the (initially empty) visibility constraints map, $\mathcal{F_{VC}}$, an element e, and its lower viewport width x_1. It constructs a new visibility constraint in $\mathcal{F_{VC}}$ for e using x_1 and a placeholder for x_2, its upper viewport width, which is unknown at the point at which the constraint is created.

The EXTRACTVISIBILITYCONSTRAINTS procedure updates $\mathcal{F_{VC}}$ as it processes each subsequent alignment graph in $\mathcal{G}$ in the main loop of the algorithm (lines 6–15). The algorithm obtains the next alignment graph g_c (line 7) in $\mathcal{G}$, and then iterates over two sets of elements. The first set (line 8) involves all elements that have are present in the current alignment graph g_c but not the previous graph g_p. These are elements that have "appeared" between the viewport widths of the current and last alignment graph considered, so the procedure performs a binary search between these two widths to establish the exact point the element "appeared". Using the viewport width found by the binary search, the procedure creates a new visibility constraint within $\mathcal{F_{VC}}$ for the element, using the previously described ADDNEWVISIBILITYCONSTRAINT procedure. The second set (line 11) involves all elements present in the previous alignment graph g_p but not the current graph g_c. These are elements that "disappeared" between the viewport widths of the two alignment graphs. Again, the algorithm performs a binary search between the two widths, this time to find the exact point at which the element "disappeared". This width is then used to complete the previously added visibility constraint with an upper bound by the COMPLETEVISIBILITYCONSTRAINT procedure.

Algorithm 3 Alignment Constraint Extraction

```
 1: procedure EXTRACTALIGNMENTCONSTRAINTS($\mathcal{G}$)
 2:     $\mathcal{F}_{\mathcal{AC}} \leftarrow \perp$
 3:     $(w_p, g_p = (\ldots, \mathcal{R}_p, \ldots, \mathcal{F}_p)) \leftarrow$ FIRST($\mathcal{G}$)
 4:     for all $r \in \mathcal{R}_p$
 5:         $\mathcal{F}_{\mathcal{AC}} \leftarrow$ADDNEWALIGNMENTCONSTRAINT($\mathcal{F}_{\mathcal{AC}}, r, w_p, \mathcal{F}_p(r)$)
 6:     while HASMOREELEMENTS($\mathcal{G}$)
 7:         $(w_c, g_c = (\ldots, \mathcal{R}_c, \ldots, \mathcal{F}_c)) \leftarrow$NEXT($\mathcal{G}$)
 8:         for all $r \in \mathcal{R}_c \cap \mathcal{R}_p$
 9:             if $\mathcal{F}_c(r) \neq \mathcal{F}_p(r)$
10:                 $w_{change} \leftarrow$BINARYSEARCHFORLAYOUTCHANGE($r_c, w_p, w_c, \mathcal{F}_c(r), \mathcal{F}_p(r)$)
11:                 $\mathcal{F}_{\mathcal{AC}} \leftarrow$COMPLETEALIGNMENTCONSTRAINT($\mathcal{F}_{\mathcal{AC}}, r, w_{change} - 1$)
12:                 $\mathcal{F}_{\mathcal{AC}} \leftarrow$ADDNEWALIGNMENTCONSTRAINT($\mathcal{F}_{\mathcal{AC}}, r, w_{change}, \mathcal{F}_c(r)$)
13:         for all $r \in \mathcal{R}_c \setminus \mathcal{R}_p$
14:             $w_{first} \leftarrow$BINARYSEARCHFOREDGECHANGE($r_c, w_p, w_c$, appear)
15:             $\mathcal{F}_{\mathcal{AC}} \leftarrow$ADDNEWALIGNMENTCONSTRAINT($\mathcal{F}_{\mathcal{AC}}, r, w_{first}, \mathcal{F}_p(r)$)
16:         for all $r \in \mathcal{R}_p \setminus \mathcal{R}_c$
17:             $w_{last} \leftarrow$BINARYSEARCHFOREDGECHANGE($r, w_p, w_c$, disappear)
18:             $\mathcal{F}_{\mathcal{AC}} \leftarrow$COMPLETEALIGNMENTCONSTRAINT($\mathcal{F}_{\mathcal{AC}}, r, w_{last}$)
19:         $w_p \leftarrow w_c$
20:         $\mathcal{R}_p \leftarrow \mathcal{R}_c$
21:         $\mathcal{F}_p \leftarrow \mathcal{F}_c$
22:     for all $r \in \mathcal{R}_p$
23:         $\mathcal{F}_{\mathcal{AC}} \leftarrow$COMPLETEALIGNMENTCONSTRAINT($\mathcal{F}_{\mathcal{AC}}, r, w_p$)
24:     return $\mathcal{F}_{\mathcal{AC}}$
25: end procedure
```

When the algorithm reaches the last alignment graph in the sequence $\mathcal{G}$ it completes the visibility constraint for each element still visible with an upper viewport width corresponding to the viewport width of that last alignment graph (lines 16–17), thereby signifying their presence at the last viewport width represented by the extracted RLG.

3.2.4. Determining Alignment Constraints. Algorithm 3 extracts alignment constraints for the RLG in a similar style to the visibility constraints extraction algorithm (Algorithm 2), iterating through the sequence of alignment graphs $\mathcal{G}$ at ascending viewport widths. However, instead of being concerned with the visibility of the elements, the algorithm examines how neighboring elements are laid out relative to one another. Throughout the execution of the EXTRACTALIGNMENTCONSTRAINTS procedure, the algorithm builds up the map of alignment constraints, $\mathcal{F}_{\mathcal{AC}}$, which are eventually returned to the main RLG extraction algorithm, and form part of the final RLG. In a similar fashion to Algorithm 2, EXTRACTALIGNMENTCONSTRAINTS begins by taking the alignment graph g_p for the first viewport width in the sequence $\mathcal{G}$ (line 3), and creating partial constraints with a lower viewport width (line 5). The algorithm creates alignment constraints with a lower width for each edge that it observes in the graph. Algorithm 3 then moves through the sequence of layout graphs $\mathcal{G}$ in the loop starting at line 6, taking the next layout g_c.

The algorithm then follows a series of steps depending on whether it can match an edge in the current alignment graph g_c with the previous graph g_p. The algorithm analyzes edges appearing in both graphs (lines 9–12). If the edge has the same parent/child relationship between its nodes, and the same alignment attributes, then the procedure does not need to be take any action. Otherwise, the relative alignment of the nodes of the edge has changed, meaning that the alignment constraints for the edge need updating. First, the algorithm performs a binary search to establish the point between the two viewport widths of the last two graphs that the change first occurred. It uses this value to complete the previously created alignment constraint for the edge with an upper viewport width (through a call to COMPLETEALIGNMENTCONSTRAINT, on line 11), while creating a new constraint, with the new layout attributes, using the value as the lower viewport width for the constraint (through a call to ADDNEWALIGNMENTCONSTRAINT on line 12).

Algorithm 4 Differencing Two RLGs

```
1: procedure DIFFRLGS($\mathcal{RLG}_{curr}$=($\mathcal{E}_{curr}$,$\mathcal{R}_{curr}$,$\mathcal{F}_{\mathcal{VC}_{curr}}$,$\mathcal{F}_{\mathcal{AC}_{curr}}$),$\mathcal{RLG}_{mod}$=($\mathcal{E}_{mod}$,$\mathcal{R}_{mod}$,$\mathcal{F}_{\mathcal{VC}_{mod}}$,$\mathcal{F}_{\mathcal{AC}_{mod}}$))
2:     for all $e \in \mathcal{E}_{curr} \triangle \mathcal{E}_{mod}$
3:         REPORTUNMATCHEDNODE($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, e$)
4:     for all $e \in \mathcal{E}_{curr} \cap \mathcal{E}_{mod}$
5:         for all $vc \in \mathcal{F}_{\mathcal{VC}_{curr}}(e) \triangle \mathcal{F}_{\mathcal{VC}_{mod}}(e)$
6:             REPORTUNMATCHEDVISIBILITYCONSTRAINT($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, e, vc$)
7:     for all $r \in \mathcal{R}_{curr} \triangle \mathcal{R}_{mod}$
8:         REPORTUNMATCHEDEDGE($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, r$)
9:     for all $r \in \mathcal{R}_{curr} \cap \mathcal{R}_{mod}$
10:        for all $ac \in \mathcal{F}_{\mathcal{AC}_{curr}}(r) \setminus \mathcal{F}_{\mathcal{AC}_{mod}}(r)$
11:            COMPAREALIGNMENTCONSTRAINT($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, ac, r$)
12:        for all $ac \in \mathcal{F}_{\mathcal{AC}_{mod}}(r) \setminus \mathcal{F}_{\mathcal{AC}_{curr}}(r)$
13:            COMPAREALIGNMENTCONSTRAINT($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, ac, r$)
14: end procedure
15: procedure COMPAREALIGNMENTCONSTRAINT($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, ac = (x_1, x_2, t, \mathcal{P}), r$)
16:     if $\exists ac' = (x'_1, x'_2, t', \mathcal{P}') \in \mathcal{F}_{\mathcal{AC}_{curr}}(r) : (x_1 \neq x'_1 \vee x_2 \neq x'_2) \wedge (t = t' \wedge \mathcal{P} = \mathcal{P}')$
17:         REPORTBOUNDSDIFFERENCE($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, r, ac, ac'$)
18:     else if $\exists ac' = (x'_1, x'_2, t', \mathcal{P}') \in \mathcal{F}_{\mathcal{AC}_{curr}}(r) : (x_1 = x'_1 \wedge x_2 = x'_2) \wedge (t \neq t' \vee \mathcal{P} \neq \mathcal{P}')$
19:         REPORTATTRIBUTESDIFFERENCE($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, r, ac, ac'$)
20:     else
21:         REPORTCOMPOUNDDIFFERENCE($\mathcal{RLG}_{curr}, \mathcal{RLG}_{mod}, r, ac$)
22: end procedure
```

Edges may "disappear" in g_c that were present in g_p, or appear in g_c when they were not present in g_p, due to the changing visibility of edges. When edges "appear" (lines 13–15), the algorithm conducts a binary search to find the viewport width where the edge are initially present, and creates a new alignment constraint in $\mathcal{F}_{\mathcal{AC}}$ using the width as the lower viewport width of the constraint. Conversely, when edges "disappear" (lines 16–18), the algorithm completes their alignment constraints with the upper viewport width, which it finds through a further binary search.

On termination of the main loop, the algorithm sets the upper bound of any outstanding uncompleted alignment constraints in $\mathcal{F}_{\mathcal{AC}}$ with the viewport width of the last layout graph in $\mathcal{G}$ (lines 22 and 23), and returns $\mathcal{F}_{\mathcal{AC}}$ back to the EXTRACTRESPONSIVELAYOUTGRAPH procedure, so that the alignment constraints can form part of the overall RLG.

3.3. *Differencing Two Responsive Layout Graphs*

As previously described in Section 2, the RLG of a responsive web page can change after modifications to either its HTML or CSS code. Algorithm 3.3 therefore takes as input two RLGs, $\mathcal{RLG}_{curr}$ and $\mathcal{RLG}_{mod}$, and produces as output a list of differences between the two, along with the viewport widths at which the differences are observable. Intuitively, if this algorithm determines that $\mathcal{RLG}_{curr}$ and $\mathcal{RLG}_{mod}$ are not the same, then it presents those differences to the developer of the web page since they may be unseen side effects of changes to the web page's source code.

The comparison performed by Algorithm 3.3 begins with a call to the DIFFRLGS procedure with the two RLGs $\mathcal{RLG}_{curr}$ and $\mathcal{RLG}_{mod}$. This procedure starts by finding the set of nodes not present in both RLGs, by taking the asymmetric difference of their node sets (line 2). A node in one RLG matches a node in another RLG if the absolute XPaths of the nodes' elements are identical. Any unmatched nodes are reported using the REPORTUNMATCHEDNODE procedure.

Following this, the algorithm compares visibility constraints for each node present in the two RLGs (lines 4), taking the asymmetric difference of their visibility constraint sets (line 5) and reporting any unmatched constraints as *visibility differences* via a call to the REPORTUNMATCHEDVISIBILITYCONSTRAINT procedure (line 6).

The algorithm then proceeds to find unmatched edges in the two RLGs (lines 7–8). It then continues by comparing alignment constraints in the two graphs, comparing constraints in $\mathcal{RLG}_{curr}$ but not $\mathcal{RLG}_{mod}$ (lines 10–11) and vice versa (lines 12–13). The algorithm outsources the comparison of individual alignment constraints to the COMPAREALIGNMENTCONSTRAINT procedure, which specifically tries to determine whether two non-matching constraints partially

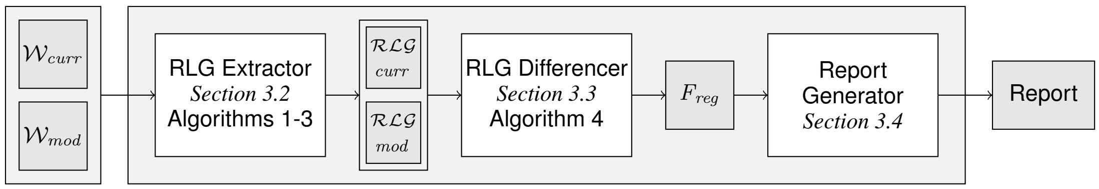


Figure 6. An overview of the high-level components comprising ReDeCheck-*RM*, implemented as a special regression checking mode of our ReDeCheck tool. This figure extends the enclosed dotted section of the usage scenario diagram of Figure 4 that corresponds to the automated operation of the tool.

match, but differ by their viewport widths (a *bounds difference*), their parent/child relationship and/or alignment attributes (an *attributes difference*), or both (a *compound difference*). For a bounds difference (lines 16–17), the set of non-matching viewport widths calculated by the reportBoundsDifference procedure is the symmetric difference of the two sets of viewport widths for the two respective constraints (i.e., the set of widths at which only one of the constraints hold true). For an attributes difference (lines 18-19), the set of non-matching viewport widths calculated by the reportAttributesDifference procedure is an inclusive set of the bounds of the unmatched alignment constraints. For compound differences (lines 20–21), as no partial match has been found, the viewport widths reported by reportCompoundDifference are simply the inclusive range of widths encompassed by the non-matching constraint's viewport range.

To illustrate the whole differencing process, we revisit the motivating example from Figure 2 and the respective RLG example from Figure 5. To begin, all of the nodes and their visibility constraints are correctly matched, before all the alignment constraints are matched, with the exception of the edge between `li[1]` and `li[5]`. For this edge, the "before" RLG, $\mathcal{RLG}_{curr}$, maps to a single alignment constraint, $(768, w_{max}, s, \{L\})$, while the "after" RLG, $\mathcal{RLG}_{mod}$, contains two constraints, $(768, 811, s, \{A\})$ and $(812, w_{max}, s, \{L\})$. Given these constraints, the RLG comparison algorithm will report two model differences to the user as potentially unseen side-effects. The first is a bounds error, as the comparison algorithm will detect a partial match between $(768, w_{max}, s, \{L\})$ from $\mathcal{RLG}_{curr}$ and $(812, w_{max}, s, \{L\})$ from $\mathcal{RLG}_{mod}$, as the two constraints contain identical attribute sets, but differing x_1 values. This difference represents the change in the range of viewport widths for which the two elements are laid out in a row. As the number of alignment constraints mapped to this particular edge differs between the two versions (one in $\mathcal{RLG}_{curr}$ and two in $\mathcal{RLG}_{mod}$), at least one edge will always remain completely unmatched. In this example, it is the constraint $(768, 811, s, \{A\})$ from $\mathcal{RLG}_{mod}$, representing the regression in which `li[5]` wraps onto a new line following the code change made by the web developer.

3.4. *Implementation of the Presented Approach in* ReDeCheck-*RM*

We implemented our approach into our prototype tool, called ReDeCheck [27], which is written in the Java programming language. Since ReDeCheck implements a number of algorithms for testing responsive web pages [8, 27], and to disambiguate this paper's RLG extraction and differencing approach from our prior work, we specifically refer to the implementation of this "regression checking mode" in ReDeCheck as "ReDeCheck-*RM*".

ReDeCheck-*RM* is structured in a similar way to the envisaged tool featured in the usage scenario of Figure 4. It consists of three core modules, the organization of which is shown in Figure 6. The first, the *RLG Extractor*, is responsible for extracting the RLG for a chosen web page, as described by Section 3.2 and Algorithms 1–3. To simplify the process of discovering *parent-child* and *sibling* relationships, and relative alignment attributes, our tool uses an implementation of an R-tree ("rectangle tree") data structure provided by the JSI (Java Spatial Index) RTree Library[†]. Our tool populates the R-tree using the co-ordinates of the minimum bounding boxes of elements,

[†]`http://jsi.sourceforge.net`

```
Differing bounds for BODY/NAV/UL/LI[1] sibling of BODY/NAV/UL/LI[5] {L}
    Original : 768 -> 1400
    Modified : 812 -> 1400
Unmatched in RLG_mod: BODY/NAV/UL/LI[1] sibling of BODY/NAV/UL/LI[5] {A} between 768 -> 811
```

Figure 7. A snippet of a report produced for the example in Figure 2. The "differing bounds" section shows how the range of viewport widths for which `li[1]` is to the left of `li[5]` differs between the two RLGs, while the "`unmatched in RLG_mod`" line shows how for the faulty viewport widths `li[1]` is *above* `li[5]` as `li[5]` wraps. Report content of the form "`768 -> 1400`" denotes a range of viewport widths.

as extracted from the DOM of the page. The second component of REDECHECK-*RM*, the *RLG Differencer*, executes the RLG differencing procedure of Section 3.3 and formalized in Algorithm 4.

Each call that Algorithm 4 makes to a "REPORT..." procedure adds to a running set of differences, denoted F_{reg}, between $\mathcal{RLG}_{curr}$ and $\mathcal{RLG}_{mod}$, and the respective viewport width sets at which they occur. The third module of the REDECHECK-*RM* tool, the *Report Generator*, analyzes the differences and determines the faulty viewport widths, which it then outputs to the web developer in a report. REDECHECK-*RM* uses Selenium version 2.53.1 [28] to drive and interact with the 2.1.1 version of PhantomJS [29], a headless web browser commonly used for automated web page navigation, screen capture, and general web testing. REDECHECK-*RM* also uses Selenium to query the DOM and extract the layout of each web page, as rendered in PhantomJS.

Figure 7 presents a snippet of the report produced by REDECHECK-*RM* for the example in Figure 2. This report aims to help guide and direct the manual analysis performed by the developer, rather than them having to manually scan the page at selected viewport widths for potential problems, a process which is likely to lead to regressions being missed. We evaluate the benefits of REDECHECK-*RM* compared to this traditional manual inspection of a page in the next section.

4. EMPIRICAL EVALUATION

To evaluate the effectiveness and efficiency of Section 3's automated approach to detecting the potential regressions in a web site's layout, we applied our implementation of it, REDECHECK-*RM*, to 15 responsive web pages, with respect to the following three research questions:

RQ1. (a) How accurate is the presented approach at detecting the various types of changes made to the source code (i.e., the HTML and CSS) of a responsive web page? (b) How does the presented approach compare to alternative baseline methods?
This question evaluates the effectiveness of our technique and its use of the responsive layout graph as a model for accurately detecting potential layout regressions in a responsive web page.

RQ2. How does the "subtlety" of a layout change influence the effectiveness of the approach?
This question investigates how effective our technique is at detecting the subtle regressions in a web page's layout that manifest themselves at comparatively few viewport widths.

RQ3. How do different parameter values change the approach's efficiency and effectiveness?
Since the algorithms in REDECHECK-*RM* may be sensitive to their configuration, this question investigates whether changing the parameters used to control the RLG extraction algorithm influences the efficiency and effectiveness of our approach to testing responsive web pages.

4.1. Subject Web Pages

In order to answer the aforementioned research questions, we selected 15 responsive web sites, using the homepages of each site for the experiments. We chose the subjects for our study by following a similar methodology to that of Mahajan et al. [30]; that is, producing a sample of web pages that use a mix of different implementation technologies, involve a range of complexities

Table I. Responsive web pages used in the empirical study of the REDECHECK-*RM* tool. In the column headings, "LOC" refers to lines of code after a standard formatting process, as described in Section 4.1. 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. The numbers in parentheses are the count of the category for the subject's main page, as a CSS file can be applied to multiple pages within a web site. Finally, REDECHECK-*RM* calculated the number of nodes in the DOM of the web page (i.e., "DOM Nodes") by using PhantomJS, a headless web browser commonly used to support the automated testing of web sites.

		HTML		CSS		
WEB SITE NAME	URL	LOC	DOM NODES	LOC	BLOCKS	DECLARATIONS
Aftrnoon	*http://aftrnoon.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 2016	*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)

and coding styles, have a variety of different layout styles, and are collated from a diverse set of different sources. We therefore harvested subjects from the showcases of popular CSS frameworks (Bootstrap [18] and Foundation [20]), web pages linked from sites with aim of exhibiting examples of "good" responsive design to other developers, while also selecting the web pages of well-known companies and organizations. As such, the final collection of web pages is diverse in terms of domain, size, and coding style, with some of them developed with the assistance of a CSS framework and others using bespoke CSS and HTML to support a responsive layout. For instance, the Shield subject uses Bootstrap version 3, while the Treehouse subject uses bespoke CSS styling rules.

The 15 web sites selected were: "Aftrnoon", 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 Section 2.1 and finally, "Treehouse", a platform for technology training. In summary, these web sites come from a wide variety of application domains, thus ensuring the representativeness of this paper's empirical results. To support the replication of this paper's experiments, we have made all of the subjects and the layout regressions available in a GitHub repository at *https://github.com/redecheck/jstvr-webpages*.

Table I shows summary statistics for each of the subject web pages, including the number of lines of code contained within the associated HTML and CSS files. To obtain faster page loading times for their end users, many developers of web sites apply "minification" to the site's CSS code, thereby removing all unnecessary characters and reducing the amount of data that must be transferred from the web server to the client. This process, along with the different developer coding styles for both the HTML and CSS, means that calculating and comparing the amount of code comprising each web page is challenging. To account for these inconsistencies, we used the tools available at *https://www.dirtymarkup.com* to format all the HTML and CSS files in a uniform and consistent manner, thereby supporting a fair comparison of the size and complexity of the sites.

Table I also presents the number of DOM nodes in each web page, an established metric of size. Additionally, we report the number of lines of CSS code, the total number of CSS blocks (i.e., collections of individual CSS declarations applied to certain HTML elements though a CSS selector), and the total number of distinct CSS declarations. Since many responsive web sites are implemented using frameworks (e.g., Bootstrap and Foundation) that contain many lines of CSS code, the table also presents in parentheses the number of blocks and declarations, respectively, that are actually used by the page. In other words, the parenthetic values in Table I only give the count of the CSS blocks and declarations that are applied to at least one HTML element on the page at some viewport width considered in this paper's experiments. The two sets of numbers reflect different aspects of the complexity of the CSS for a particular web page. While the number of "used" blocks reflects a smaller subset of the overall CSS needed to style a particular page, a developer will still need to understand, use, and navigate through all of the CSS rules defined for the web site when modifying a particular page. As seen from these summary values, our selection of web pages represent a range of HTML and CSS style sheet sizes, with the number of DOM nodes ranging from 106 to 472; and the number of CSS blocks ranging from as few as 166 to a maximum of 5,958, of which we found a range of 31 to 158 blocks are explicitly used by our subjects.

4.2. Using Mutation Operators to Create Potential Regressions in Web Page Layout

Figure 4 illustrated the main usage scenario for our approach, in which a web developer modifies the source code of a web page and then uses REDECHECK-*RM* to compare the two versions of the page, thereby discovering any issues introduced by the code changes. As the usage scenario of the presented approach focused on the development stage of the software lifecycle, obtaining real examples of incrementally modified versions of web pages is problematic. Therefore, we designed mutation operators to introduce small changes into the code of the subjects, as the means of obtaining potential regressions from the original layout with which to evaluate our approach.

To generate as wide a variety of potential layout changes as is possible, the set of mutation operators we designed target both the HTML and CSS source code of a web page. Table II describes these operators and furnishes examples of the changes that they make to the page's HTML and CSS code. The first four mutation operators listed by the table apply to a page's HTML. We discuss these first, followed by the second four operators, which apply to a page's CSS.

Developers often choose to use the pre-defined classes from responsive design frameworks (e.g., Bootstrap and Foundation) to enhance the style of the elements on their web pages. These classes provide CSS code for a range of styling aspects, from responsive sizing to typography and colouring, thereby making it a challenge to ensure that the correct classes are applied to the desired elements. When developers do this incorrectly, they can cause regressions in the layout of a page. In observance of this trend, we created three operators focussing on the assignment of these classes to web page elements: *Class Addition*, *Class Deletion*, and *Class Modification*. For instance, as shown by Table II, the Class Addition operator might add Bootstrap's "`col-sm-6`" class to a page element, while Class Deletion could lead to the removal of "`col-xs-12`", and Class Modification may replace the class "`col-xs-12 col-sm-6`" with "`col-xs-12 col-sm-4`".[‡] Additionally, as web developers often need to update the content of a web page rather than just its visual appearance or layout, a fourth operator, *Textual Content*, modifies the text contained within an HTML element. This content modification can change how the browser lays out a web page since the element containing the changed text may expand or contract, potentially causing a chain reaction of follow-on changes to the relative positioning of the elements around it or containing it.

While many web developers might customize the CSS supplied by a responsive framework, others may choose to create and enhance their own CSS. To reflect both of the ways in which a developer could incrementally modify a page's stylesheet code, we also implemented additional mutation operators targeting general CSS code. The operators are split into two categories, each modifying a

[‡]The CSS classes used in these examples, from Bootstrap's CSS, support the layout of elements in a responsive grid [18].

Table II. Descriptions and examples of the mutation operators used in the experiments. 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 mutation operator. Note that the first four operators modify the HTML of a page while the final four change its CSS. Section 4.2 explains how we automatically applied these operators during the experiments.

Operator Name	Description	Before Modification	After Modification
Class Addition	Adds a class to an element	`<div class="col-xs-12">`	`<div class="col-xs-12 col-sm-6">`
Class Deletion	Removes a class from an element	`<div class="col-xs-12 col-sm-6">`	`<div class="col-sm-6">`
Class Modification	Replaces a class with another	`<div class="col-xs-12 col-sm-6">`	`<div class="col-xs-12 col-sm-4">`
Textual Content	Increases/decreases amount of text in an element	`<h1>Welcome</h1>`	`<h1>Welcome to my page</h1>`
Declaration Value	Modifies value of a declaration	`li { width: 75% }`	`li { width: 71% }`
Declaration Unit	Modifies the unit of a declaration's value	`div { width: 50% }`	`div { width: 50px }`
Query Expression	Modifies the media query's expression	`@media (min-width: 640px)`	`@media (max-width: 640px)`
Query Breakpoint	Modifies the media query's numeric value	`@media (min-width: 992px)`	`@media (min-width: 990px)`

different CSS construct. The first modifies individual CSS declarations, either by changing the value of the declaration (which can be numeric or textual depending on the CSS property in question), through the *Declaration Value* operator; or the declaration's unit, through the *Declaration Unit* operator. Our operators adjust numerical values by $\pm$1–10 at uniform random. For instance, as shown by the examples in Table II, a change made by the Declaration Value operator could reduce the width of a list item from 75% to 71%. A change made by the Declaration Unit operator could transform an element's width from a percentage value to one in measured in pixels. The second operator category changes the way in which whole groups of CSS rules are applied by modifying the media queries containing them in the stylesheet. Each operator targets one part of the selected media query. The *Query Expression* operator changes the type of the expression (e.g., from "`min-width`" to "`max-width`"); while the *Query Breakpoint* operator modifies the media query's numeric value, thereby changing the breakpoint at which different CSS rules are applied to the page.

We implemented these operators into a tool that automatically injects mutations into a web page in a four step process, beginning (1) with a static analysis of the web page's HTML to identify the CSS classes that are applied to each element, thereby forming a set of *CSS class modification candidates* for the HTML mutation operators to potentially modify. During this phase, the HTML elements containing text are also collected to form the candidate set for the Textual Content operator. Next, (2), the tool parses and filters the CSS to obtain a set of *CSS modification candidates* for the CSS mutation operators to potentially modify. By checking the block's selectors against the set of selectors identified in the previous phase, the filtering process only considers the CSS blocks that are applied to at least one element on the web page. This prevents the code modification from being injected in a part of the style sheet that is not used by the web page, resulting in a page that is identical to the original (i.e., a potential source of "equivalent mutants" [31]). The set of declarations contained within each block is also pruned to contain only those concerning layout, such as `width`, `padding`, and `margin`, since changing non-layout properties, like `background-color`, may affect the visual appearance of the web page but not its layout. For as many mutant web pages as required, the tool then (3) selects at random one of the eight operators and a suitable modification candidate from the relevant candidate set, also at random; and finally, (4), modifies the chosen candidate, saving the altered web page, including the modified HTML/CSS files, to disk.

Suitable for answering RQ2 of our evaluation, this automated approach to web page mutation results in a wide variety of changes being introduced, from tiny shifts in the position of an element at few viewport widths, to ones with a large visual impact visible at many different viewport widths.

4.3. Baseline Comparison Methods

As discussed in Section 2.1, there has been almost no work in the literature on testing responsive web pages, especially work addressing the problem of automatically detecting regression changes to the layout of a responsive web page. The only exception is our previous work [16], of which this paper is an extension. Our recent work on detecting common layout failures without explicit oracles [8] is not designed to find regressions in layout and instead functions using one RLG rather than two, making it an unsuitable comparative approach. As such, there is a lack of baseline approaches against which we can compare REDECHECK-*RM*. However, as previously mentioned in Section 2.1, testers often perform “spotchecking” to test their web sites, whereby a tester loads a page into a browser, resizes it to series of common viewport widths, and manually checks to ensure that there are no previously unseen layout regressions. We therefore compared REDECHECK-*RM* with two methods based on this spotchecking procedure: one performed manually by the authors, and an automatic method developed by re-purposing a tool from the literature originally used to detect cross-browser issues. For both of these two methods, we used six distinct viewport widths to perform the spotchecking process. These were taken from the preset widths available in two widely used developer tools, Viewport Resizer [32] and Window Resizer [33], which resize a browser to a desired preset viewport width at the click of a button. The preset widths used — 480, 600, 640, 768, 1024, and 1280 pixels — cover widths commonly checked by developers, thereby encompassing a wide range of devices from small mobile phones and tablets to laptops and desktop computers.

Manual Spotchecking Technique. Given the visual nature of the task of testing a web page, it is common for the experimental evaluation of new web testing methods to involve some type of manual checking by humans [1, 24, 34, 35]. This paper’s manual spotchecking method, a process that we call SPOTCHECK-MANUAL, involved humans comparing the original and mutated version of a web page at the aforementioned viewport widths. For the purposes of this paper’s empirical study, manual spotchecking was performed by the authors who used the systematic classification procedure that we detail in Section 4.4 and visualize in Figure 8.

Automated Spotchecking Technique. As introduced in Section 2.3, the *alignment graph* models a web page at a single viewport width, with the original purpose of checking a web page for cross browser issues. Unlike the RLG, the alignment graph is unsuitable for modeling a web page over a series of viewport widths. However, it can be used to find differences between two versions of a web page at a single viewport width. As such, it may be used as part of an automated version of the spotchecking procedure previously described. We therefore modified Choudhary et al.’s X-PERT tool [24] to create an alignment graph for a web page and its mutated counterpart at the aforementioned viewport widths, and report any differences between the two graphs. We refer to this automated alternative to manual spotchecking as SPOTCHECK-ALIGNMENTGRAPH.

4.4. Manual Classification Procedure

To verify the results of REDECHECK-*RM* and the SPOTCHECK-ALIGNMENTGRAPH automated baseline technique, and as a key component of the SPOTCHECK-MANUAL method, we defined a procedure to manually classify whether a page contains observable layout changes or not. The motivation for us defining such a procedure was to ensure that we performed our manual checking systematically, rigorously, and consistently. Since both automated techniques might report “changes” that are not observable in practice — or miss layout changes that are actually observable — each modified web page used in the experiments needed to be manually inspected against its unmodified counterpart in order to check the outputs of each.

As shown in Figure 8, our manual classification procedure involves a human web developer examining a pair of web pages (i.e., $\mathcal{W}_{curr}$ and $\mathcal{W}_{mod}$, which correspond to the original web page and a mutant in the setting of our evaluation, respectively) rendered by a browser for a particular viewport width, and answering questions in the following four categories:

1. Visibility: Does the visibility of any element differ between the two versions of the page? For instance, at a particular resolution, is an element visible in one version but hidden in the other?

2. Position: Does the positioning of one element in relation to a neighboring element differ between the two versions? For example, are two elements rendered side-by-side in one version of the web page but are stacked one above the other in the other version?

3. Alignment: Is a pair of elements that were in alignment on one or more of their edges no longer aligned? For example, given two elements that were previously aligned on their left edges, are they no longer aligned in this way?

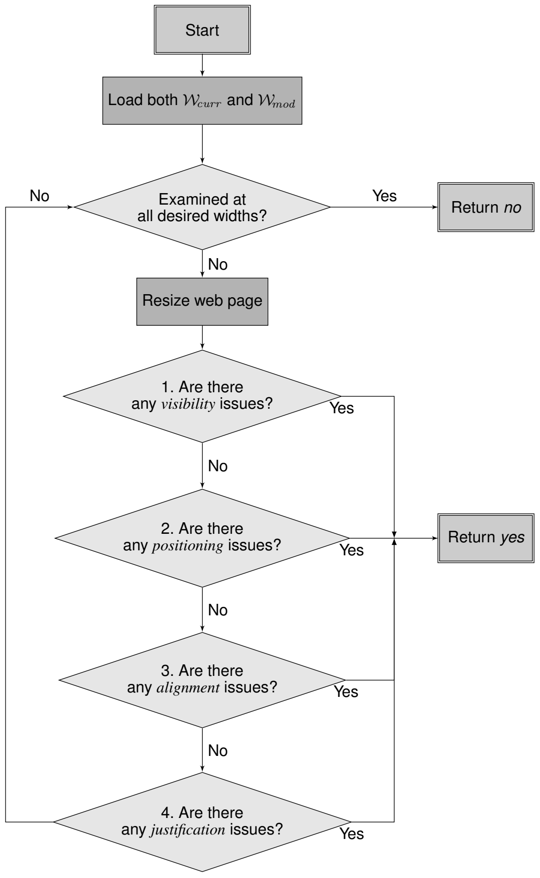

Start
Load both $\mathcal{W}_{curr}$ and $\mathcal{W}_{mod}$
Examined at all desired widths?
Yes
Return *no*
No
No
Resize web page
1. Are there any *visibility* issues?
Yes
No
2. Are there any *positioning* issues?
Yes
Return *yes*
No
3. Are there any *alignment* issues?
Yes
No
4. Are there any *justification* issues?
Yes

Figure 8. The manual procedure for determining if a modified web page contains a layout regression. The individual following this process will ultimately answer *yes* or *no* to the question "does the modified web 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*).

4. Justification: Is an element no longer justified in the same way within its container? For instance, was an element initially right-justified within its container and is now left-justified?

Each author undertook the manual procedure to analyze particular web page mutations with respect to the original version of the page. If, while inspecting the web pages and asking ourselves the questions associated with each of the four categories, we observed a part of the page that convinced us to answer "yes" to a question concerning visibility, position, alignment or justification, we recorded this as a potential layout regression and investigated it further. If we answered "no" to all of the relevant questions for the viewport width currently under inspection, we then repeated the process for each subsequent viewport width, until we answered "yes". If we manually examined all of the viewport widths and concluded that there was not a difference between the original and modified version of the web page, then we classified it as not containing any layout changes. To mitigate any subjectivity or mistakes when following this manual procedure, we reviewed the results of each author together as a committee in order to produce a definitive final classification.

It is worth noting that we cannot exclusively align one of mutation operators from Section 4.2 with one of the four questions that we pose in this section. For instance, it is possible that the mutation of a declaration value or unit could change *either* the visibility of an element (if, for instance, the element is now positioned outside of the viewport) or the positioning of an element in relation to another (if, for example, elements were previously side-by-side and are now stacked). How a mutant manifests itself as a layout failure depends on how the web page uses the modified HTML or CSS.

4.5. *Specific Methodology for Each Research Question*

To answer the research questions stated at the start of this section, we began by using the approach described in Section 4.2 to automatically generate 30 modified versions of each subject web page, thereby producing a set of 450 modifications in total. We then used the implementation of our approach, REDECHECK-*RM* (as introduced in Section 3.4), and this set of modified web pages to answer each of the research questions. We conducted all of the experiments with REDECHECK-*RM* on an iMac with 8GB of RAM running the latest version of macOS, thereby demonstrating that this tool can be run on an "everyday" workstation that a web developer might use.

RQ1: (a) How accurate is the presented approach at detecting the various types of changes made to the source code (i.e., the HTML and CSS) of a responsive web page? (b) How does the presented approach compare to alternative baseline methods?
To answer part (a) of this research question, we ran the REDECHECK-*RM* tool on the input of both the current and the modified version of a web page, configuring it to use a sampling range of 400–1400 pixels, thereby ensuring that the experiments considered viewport widths corresponding to a wide variety of devices, from smartphones to widescreen desktops. We also set the step size to be 60 pixels because it was shown through preliminary experimentation in the conference version of this paper to be effective across many different web pages [16]. Due to the effort needed to manually verify the results of REDECHECK-*RM*, as we described in the last subsection, we used a smaller number of modifications chosen from the overall set of 450 modified pages. We randomly selected four modifications per page, producing a set of 60 mutants in total. We then used REDECHECK-*RM* to compare the RLG of each web page with the RLG of each mutated page. We refer to REDECHECK-*RM*'s result as a *positive* if REDECHECK-*RM* reported a difference. Otherwise, if the tool did not report a difference, then we refer to it as a *negative* result.

To answer part (b) of the research question, we applied both the SPOTCHECK-MANUAL and SPOTCHECK-ALIGNMENTGRAPH methods, as described in Section 4.3, on the same reduced set of web page mutations that we used with REDECHECK-*RM*. As we discussed in Section 4.3, we derived the six viewport widths we chose for both manual and automated spotchecking from the preset widths available in two widely used developer tools, Viewport Resizer [32] and Window Resizer [33]. These widths fall within the sampling range of 400–1400 pixels that we picked for REDECHECK-*RM* to use, thereby allowing us to directly compare the techniques.

For the SPOTCHECK-MANUAL process, each author was provided with screenshots of the two versions of each web page (i.e., the current and modified versions), which we compared in a side-by-side fashion, and applied the manual classification procedure detailed in Section 4.4 and shown visually by Figure 8. We used screenshots taken by the PhantomJS headless browser instead of an actual web browser to ensure that the classification process could be easily replicated, while also accounting for any variability that might be introduced by the use of different operating systems and web browsers. To further remove subjectivity from the process and to mitigate the possibility of mistakes having been made, all of the authors met as a committee to discuss their individual results and assign a final committee decision of *positive* (i.e., the mutant involves at least one observable layout change) or *negative* (i.e., it does not) for each mutant studied.

Similar to REDECHECK-*RM*, for the SPOTCHECK-ALIGNMENTGRAPH technique, we classify a result as *positive* if it reported a difference, else we categorized it as *negative*.

In order to classify each individual result of REDECHECK-*RM*, SPOTCHECK-MANUAL, and SPOTCHECK-ALIGNMENTGRAPH as either a *true positive*, *false positive*, *true negative*, or *false negative*, it was necessary to know whether each modified page actually contained an observable change in its layout. It is worth noting that the results of the SPOTCHECK-MANUAL process cannot form the "gold standard" or the "oracle" in this regard, since changes may exist at viewport widths that were not manually spotchecked during the analysis, as they were not part of the pre-specified set of widths. Because of this, SPOTCHECK-MANUAL is subject to false negatives. We therefore proceeded as follows. Using the results already categorized as *negative* for SPOTCHECK-MANUAL, we continued to perform a full exhaustive examination of the mutant concerned across the full viewport range of 400–1400 pixels. On the basis of whether we were able to observe a layout change (again, following the manual classification procedure) we were able to complete the final true or false designation (i.e., correct or incorrect) of the result for the mutant with each technique. That is, if a method reported a positive result, and a layout change was evident from manually observing the current and modified pages, then we classify the result as a *true positive*. Else, if we did not observe a layout change, we say it is a *false positive*. Conversely, if no change was reported by a method, and yet we observed a change during the manual analysis, we report the result as a *false negative*; else if we could not discern a layout change, we say it is a *true negative*. As with the individual results for SPOTCHECK-MANUAL, the final classifications for each mutant were reviewed as a committee of authors to mitigate against the possibility of the subjectivity of our results.

To summarize our answers to this paper's research questions, we report the higher-is-better *precision*, *recall*, and *accuracy* of each technique. Precision is the number of true positives, TP, divided by the number of true positives and false positives, FP, for a technique, i.e., $TP \div (TP + FP)$; while recall is the number of true positives divided by the sum of true positives and false negatives, FN, i.e., $TP \div (TP + FN)$. Finally, accuracy is the number of true positives and true negatives, TN, divided by the number of all results, i.e., $(TP + TN) \div (TP + TN + FP + FN)$.

Finally, we test for statistical significance of the accuracy of REDECHECK-*RM* compared with the two spotchecking techniques using Fisher's Exact Test, a nonparametric test for categorical data, comparing the numbers of correct $(TP + TN)$ and incorrect $(FP + FN)$ results for each technique, and using those figures to form the contingency matrix required by the test.

RQ2: How does the "subtlety" of a layout change influence the effectiveness of the approach? Depending on the type of modification made to the HTML or CSS source code of a web page, some layout changes can be harder to detect than others, due to the level of the observable impact made by the code modification on the layout of the page, and the number of viewport widths at which these changes are observable. While the former may be problematic for a human developer who is manually inspecting a web page, it should not significantly influence the effectiveness of an automated approach, such as SPOTCHECK-ALIGNMENTGRAPH or REDECHECK-*RM*. However, automated tools may exhibit varying levels of effectiveness depending on the number of viewport widths at which a layout change is observable. To further investigate this issue, we used the number of viewport widths at which the changes are visible as a metric for the *subtlety* of a layout change, where the lower the value, the more "subtle" we deemed the mutation to the web page.

To experimentally study how REDECHECK-*RM* and SPOTCHECK-ALIGNMENTGRAPH compare in terms of detecting failures of varying subtleties, the subtlety of each mutated web page must be determined. To do this in a reliable and consistent manner, we implemented an automated comparison approach that began by extracting the DOM of both the original version and mutated version of each web page at every viewport width in the range used by REDECHECK-*RM* in the previous two research questions (i.e., 400–1400 pixels). Then for each width, our approach automatically compared the pair of DOMs (d_{curr}, d_{mod}), first by attempting to match every element from d_{curr} to an element in d_{mod} using each element's XPath, and then comparing the coordinates of each pair of matched elements. If, at any viewport width, any elements were unmatched or a matched pair of elements possessed different coordinates (i.e., it was either rendered at a different size or a different location), we deem the code change to have an effect on the web page's layout. Since we could fully automated the experiments to answer this research question, we used the full pool of 450 mutants, i.e., each of the 30 mutants for each of the 15 subjects.

After determining the number of distinct viewport widths at which the various layout changes were observable in the web page's DOM, a variable denoted as *VW*, we filtered the pool of changed web pages by removing those for which *VW* = 0, as the code modification had no impact on the original web page and thus the mutated web page can be said to be "equivalent" to the original [31]. Then, we separated the remaining web pages into different groups depending on their *VW* value. That is, we grouped the changed pages according to whether they impacted the DOM for a small number of viewport widths (i.e., 2–3, 4–5, 6–10, or 11–50) or a large number of widths (i.e., 51–100, 101–300, 301– 500, or more than 500). Intuitively, a change in a page's layout is more subtle if it is observable at fewer viewport widths. To investigate effectiveness, we ran REDECHECK-*RM* and SPOTCHECK-ALIGNMENTGRAPH on each modified page according to the same procedure as we used in RQ1. However, due to the aforementioned increase in the number of modified pages for this experiment, we did not manually verify these results. Instead, whenever $VW > 0$ for a particular web page, we regarded any layout differences reported by either the two REDECHECK-*RM* or SPOTCHECK-ALIGNMENTGRAPH methods as a "correct" detection of a layout change to a page.

To judge the effectiveness of the two techniques we calculated its detection rate as the number of layout changes created by the mutation operators, denoted by L, divided by the number of those changes a technique could detect, denoted by D, ultimately performing the calculation $D \div L$.

Finally, we test for statistical significance of the detection results for REDECHECK-*RM* and SPOTCHECK-ALIGNMENTGRAPH by again using the Fisher Exact Test. However, this time we compare the number of detected changes (D) and undetected changes ($L - D$) for each technique, again using those figures to create the contingency matrix required by the test.

RQ3: How do different parameter values change the approach's efficiency and effectiveness? In order to aid users in a development scenario like the one depicted in Figure 4, REDECHECK-*RM* must quickly and accurately provide feedback to developers following their modifications to a web page's source code. As such, it is prudent to identify the most efficient configuration under which REDECHECK-*RM* provides the most effectiveness (i.e., layout change detection capability).

Using the same pool of 450 mutants that we used in RQ2, we answered this research question by running the REDECHECK-*RM* tool on each current and modified page under several different parameter configurations, using different combinations of step sizes and the different RLG extraction techniques that we introduced in Section 3.2, that is, EXTRACT-COMPARE-INTERVAL, EXTRACT-COMPARE-INTERVAL-BREAKPOINTS, and EXTRACT-COMPARE-EXHAUSTIVE.

We investigated the effectiveness and efficiency of REDECHECK-*RM* using combinations of RLG extraction/viewport width sampling techniques and different step sizes, which control the set of viewport widths selected for initial sampling, represented by the set S in Algorithm 1. In the previous research questions, we used 60 pixels as the step size. To assess whether or not a 60 pixel step size appropriately balances the efficiency and effectiveness of the presented method, we experimented with both smaller and larger values, creating a group of nine step sizes: 10, 20, 40, 60, 80, 100, 150, 200, and 500 pixels. For each parameter configuration (i.e., the step size and one of the three aforementioned RLG extraction techniques), we ran the REDECHECK-*RM* tool and recorded three data points: whether the layout modification was correctly detected, the execution time that REDECHECK-*RM* needed to run to completion, and the number of viewport widths sampled by the tool. Since this research question necessitated that we run REDECHECK-*RM* on all 30 versions of the 15 pages using each of the aforementioned step sizes, these results should reveal a step size that works well across a wide variety of pages, while also highlighting the cost-benefit trade-offs associated with the step size and the RLG extraction method.

To further support our response to this research question, we also wanted to determine how much of an improvement in the number of sampled viewport widths could be attributed to the EXTRACT-COMPARE-INTERVAL-BREAKPOINTS method's combined use of both interval and breakpoint sampling. To investigate this issue further, we analyzed the difference in the number of viewport widths required by the two techniques for sampling a specific web page by calculating $\frac{(VW_{IB} - VW_{IB})}{VW_{IB}} \times 100$, where VW_{IB} denotes the number of the sampled viewport widths by the EXTRACT-COMPARE-INTERVAL method and VW_{IB} is the number of viewport widths sampled by EXTRACT-COMPARE-INTERVAL-BREAKPOINTS. When this equation computes a positive value this indicates that EXTRACT-COMPARE-INTERVAL-BREAKPOINTS is more efficient than EXTRACT-COMPARE-INTERVAL because it samples fewer viewport widths. However, when the value computed by this equation is negative, then EXTRACT-COMPARE-INTERVAL is more efficient than EXTRACT-COMPARE-INTERVAL-BREAKPOINTS.

Finally, to analyze the values computed by the aforementioned equation, we tested for statistical significance using the Mann-Whitney U-test and the $\hat{A}_{12}$ effect size metric of Vargha and Delaney [36]. We decided to use these nonparametric tests since we cannot make any assumptions about the normality of our results' distributions. For the $\hat{A}_{12}$ values, a score close to 1 can be interpreted as a high probability that using the EXTRACT-COMPARE-INTERVAL-BREAKPOINTS method would be more efficient (i.e., require fewer viewport widths to be sampled) than the EXTRACT-COMPARE-INTERVAL method, values near 0.5 mean that the two methods are similar in their sampling of viewport widths, and values closer to 0.0 suggest that the EXTRACT-COMPARE-INTERVAL method would be more efficient than EXTRACT-COMPARE-INTERVAL-BREAKPOINTS.

4.6. Threats to Validity

As would be the case with many experiments in the field of software testing, we must consider the threats to the validity of the presented results [37, 38]. One concern is that these results may not generalize to other web pages. We mitigated this threat by selecting the home page of 15 real-world web pages, which varied in both complexity and domain, ranging from small personal pages to much larger ones promoting businesses and mobile applications. The subjects chosen also used, along with two popular front-end RWD frameworks (i.e., Bootstrap [18] and Foundation [20]), bespoke cascading style sheets, thereby ensuring that the chosen web pages represent a wide range of web development strategies. As mentioned in Section 6, we will further demonstrate the generalizability of REDECHECK-*RM* through future experiments with more and more varied web pages.

Since this paper's experiments used the REDECHECK-*RM* tool, threats to validity arising from errors in its implementation must be mitigated as much as possible. We managed the concern that the testing tool was incorrect by using several techniques, including regular unit testing of the tool's individual components. Additionally, during development of REDECHECK-*RM*, for several web pages, we inspected both the individual responsive layout graphs it extracts and the reports it generates, thereby establishing a confidence in their correctness. As REDECHECK-*RM* also uses several third-party tools, such as the JSoup HTML parser [39] and the JStyleParser tool for CSS manipulation [40], we also thoroughly tested each one to mitigate the risk of their implementation defects influencing REDECHECK-*RM* and potentially compromised this paper's empirical results. Finally, it is worth noting that we have released an open-source version of REDECHECK-*RM* [27], thereby allowing researchers to check the correctness of the tools that we used for the experiments.

This paper's experiments evaluated REDECHECK-*RM* in a realistic setting, as visualized by the diagram in Figure 4, in which a developer runs the tool after making an incremental modification to the HTML or CSS of a responsive web page. Since we did not have access to the actual changes that a developer made to a responsive web page, this paper experimentally simulates them through the use of a tool that mutates a page's HTML and CSS code. As such, the experiments could be compromised if the mutation operators did not introduce realistic changes to the page's layout.

While it can be argued that we did not insert every type of code modification that developers make during the development of a responsive web page, our set of operators produce a wide variety of modified pages by targeting different HTML and CSS constructs. By changing both the classes applied to HTML elements and the textual information contained within them — along with many different alignment and layout CSS attributes and the expressions and breakpoints of media queries — we generated 30 changes in each of the 15 web pages used in the evaluation, resulting in a total of 450 modifications representing a wide variety of layout changes that a developer could potentially introduce into their web page. While the design of the code mutation operators was broadly influenced by an investigation into the HTML and CSS programming challenges commonly experienced by web developers [6], it is important to note that the primary purpose of this study was not to see if the tool could find real-world failures but rather to ascertain whether REDECHECK-*RM* accurately detects potential regressions in the layout of a responsive web page.

Some of the results in this paper arise from the authors following a manual procedure to classify some of the web pages according to whether or not they contained a regression from the original layout. As with previous work that addresses the problem of presentation issues in web pages (e.g., Choudhary et al. [24, 35], Mesbah et al. [1], Mahajan et al. [41], Alameer et al. [34]), this classification can only be performed manually. As the manual analysis detailed in this paper was not part of a formal user study and therefore was not conducted under controlled conditions, there is a risk that these results may not be representative of the real-world manual testing performed by web developers. However, while performing manual classification, we followed a procedure, detailed

Table III. The results summarizing how the REDECHECK-*RM* tool, SPOTCHECK-ALIGNMENTGRAPH, and SPOTCHECK-MANUAL detect the mutations in the HTML and CSS of each subject web page.

	REDECHECK-*RM*	SPOTCHECK-ALIGNMENTGRAPH	SPOTCHECK-MANUAL
True Positives	48	39	42
True Negatives	11	8	12
False Positives	1	4	0
False Negatives	0	9	6
Precision	0.98	0.91	1.00
Recall	1.00	0.81	0.88
Accuracy	0.98	0.78	0.90

in Section 4.4 and Figure 8, that mirrors a realistic understanding of how practising developers manually test responsive web pages. Additionally, since the results of the manual classification can be subjective and error-prone, each individually author completed this process before collectively discussing each modified web page to achieve a consensus concerning its final classification. Finally, the results from the manual analysis are primarily a way to anecdotally establish both the benefits of using REDECHECK-*RM* and the challenges and costs of manually testing responsive web pages.

To support the replication of this paper's experiments and to enable the inspection of its results by researchers and web developers, we have made all of the subjects and the mutants generated by our tool publicly available at *https://github.com/redecheck/jstvr-webpages*. The REDECHECK tool that features REDECHECK-*RM* is also available at *https://github.com/redecheck/redecheck*.

4.7. Answers to the Research Questions

RQ1: (a) How accurate is the presented approach at detecting the various types of changes made to the source code (i.e., the HTML and CSS) of a responsive web page? (b) How does the presented approach compare to alternative baseline methods?
Our manual classification procedure revealed that out of the 60 HTML/CSS code changes made by our mutation operators, 48 resulted in manually observable layout changes (i.e., simulated regressions), while 12 did not (i.e., could be considered to be equivalent to the original versions of their respective pages). Table III presents the results of each technique.

In response to part (a) of the research question, the table shows that REDECHECK-*RM* tool was able to correctly detect all 48 layout changes as true positives, with 11 correct true negatives, and only one false positive. Careful manual analysis of this false positive result revealed that the change applied by the mutation operator caused no observable difference to the web page, while the underlying DOM structures were different enough to cause a difference in the generated RLGs. As shown by Table III, REDECHECK-*RM* accurately reported layout changes 98% of the time.

In response to part (b) of the research question, Table III also show that REDECHECK-*RM* outperformed both SPOTCHECK-ALIGNMENTGRAPH and SPOTCHECK-MANUAL across the 60 pages with changed layout. SPOTCHECK-ALIGNMENTGRAPH only correctly classified 39 of the 48 layout changes as true positives, while SPOTCHECK-MANUAL achieved 42.

This result demonstrates that, for true positives, REDECHECK-*RM* is 12.5% more effective than the manual approach and 18.75% more effective than the automated spotchecker that uses the alignment graph. REDECHECK-*RM* also produced fewer false positives that SPOTCHECK-ALIGNMENTGRAPH and fewer false negative results than both spotchecking approaches, with SPOTCHECK-ALIGNMENTGRAPH and SPOTCHECK-MANUAL producing 9 and 6 false negatives, respectively, while REDECHECK-*RM* produced no false negatives. Notably, the false positives for the spotchecking techniques were caused by the layout change only being visible at a width not examined by either approach. In contrast, REDECHECK-*RM*'s sampling approach correctly identifies the layout changes and reports them back to the developer, demonstrating its superiority at finding regressions in a page's layout. Overall, REDECHECK-*RM* achieves the highest recall and is the most accurate technique, although because of the one false positive result, has a 0.02 lower score for precision than the careful manual checking of the authors for SPOTCHECK-MANUAL.

Table IV. The results summarizing how effective REDECHECK-*RM* and SPOTCHECK-ALIGNMENTGRAPH are at detecting layout changes that vary according to how "subtle" they are. In this table, *VW* stands for the set of distinct viewport widths at which the layout change is evident, meaning that a layout change is less subtle as the magnitude of *VW* increases as it is visible at a greater number of viewport widths. L represents the total number of (non-equivalent) layout changes created by the mutation operators, while D is the number of those changes a technique could detect. We calculated the values in the "Detection" column (i.e., the proportion of layout changes detected) according to the equation $D \div L$.

		REDECHECK-*RM*		SPOTCHECK-ALIGNMENTGRAPH	
VW	L	D	Detection	D	Detection
1	5	5	1.00	1	0.20
2–3	9	9	1.00	4	0.44
4–5	11	8	0.73	2	0.18
6–10	11	10	0.91	9	0.82
11–50	4	4	1.00	3	0.75
51–100	5	5	1.00	3	0.60
101–300	26	22	0.85	20	0.77
301–500	29	26	0.90	24	0.83
501+	198	176	0.89	176	0.89
Total	298	264	0.89	242	0.81

Finally, the results of statistical significance testing using Fisher's Exact Test were as follows. When comparing REDECHECK-*RM* with SPOTCHECK-ALIGNMENTGRAPH the *p*-value is 0.000978, and when comparing REDECHECK-*RM* with SPOTCHECK-MANUAL the *p*-value is 0.114. These *p*-values indicate that the probability of the spotchecking techniques actually outperforming REDECHECK-*RM* in the general case is low, but only significant for the results of SPOTCHECK-ALIGNMENTGRAPH at a typically used alpha level such as $\alpha = 0.05$. Nevertheless, the practical advantage of REDECHECK-*RM* over SPOTCHECK-MANUAL is clear, since the former is a fully automated technique, whereas SPOTCHECK-MANUAL involves manual effort.

Conclusion for RQ1: Part (a): REDECHECK-*RM* has an accuracy of 0.98, meaning that it correctly reported layout changes 98% of the time, with only one misleading false positive result. Part (b): REDECHECK-*RM*, with the greatest number of true positives and true negatives, is a superior checking option compared to both SPOTCHECK-ALIGNMENTGRAPH and SPOTCHECK-MANUAL, classifying 12.5% and 18.75% more true positives respectively. Overall, REDECHECK-*RM* has the highest accuracy of the three techniques at 0.98, compared to an accuracy of 0.78 and 0.90 for SPOTCHECK-ALIGNMENTGRAPH and SPOTCHECK-MANUAL, respectively.

RQ2: How does the "subtlety" of a layout change influence the effectiveness of the approach? Table IV presents the layout change detection results for both REDECHECK-*RM* and SPOTCHECK-ALIGNMENTGRAPH when executed on the larger pool of modified pages. The results are grouped into "buckets" based on the number of modified viewport widths, with the different buckets listed in the first column, while the second column, with the label L, shows the number of changed pages for each bucket. Buckets for which *VW* is small contain changes considered "subtle", while larger values represent more "obvious" changes. The table's results show that REDECHECK-*RM* outperforms SPOTCHECK-ALIGNMENTGRAPH for all buckets, often by a considerable margin. The number of viewport widths at which there is a layout change is shown to have negligible impact on the performance of REDECHECK-*RM*, which achieves a detection rate of 0.72 or above for all buckets. In contrast, the detection rates achieved by SPOTCHECK-ALIGNMENTGRAPH are inconsistent. Critically, the value of *VW* appears to influence the change detection capability of SPOTCHECK-ALIGNMENTGRAPH, with the number of changes successfully detected generally trending upwards as the value of *VW* increases. This trend is to be expected, since, intuitively, the more viewport widths at which a layout change is observable, the more likely is SPOTCHECK-ALIGNMENTGRAPH to observe and report it.

For pages in which the modification had little subtlety, such as those that manifest themselves at more than 300 viewport widths, the relative performances of REDECHECK-*RM* and SPOTCHECK-ALIGNMENTGRAPH are comparable, with high (> 0.80) detection achieved by both. However, due to the large number of viewport widths at which the layout change is visible, it is likely many of these differences would be easily detectable by a human developer. Therefore, "subtle" changes to the web page that are visible at fewer viewport widths are arguably the more important ones for evaluating the effectiveness of these two tools — as these are more likely to go undetected and end up manifesting in a production web page. For modified pages with 10 or fewer modified viewport widths, the detection percentage of REDECHECK-*RM* is still very high, with only four out of 36 layout changes going undetected, and an overall detection rate of 0.89. In contrast, for the same set of subtle mutants, SPOTCHECK-ALIGNMENTGRAPH failed to detect 20 of the 36 changes in a page's layout, resulting in a much lower detection rate of 0.44.

We investigated the changes not detected by REDECHECK-*RM*, and found that, while they introduced changes at the DOM level, three were not significant enough to change the relative alignment of the elements. Our manual analysis of these changes, as per the procedure of Section 4.4, concluded that they had no observable impact on the web page. The fourth change did have a detectable impact on the page, but was not reported by REDECHECK-*RM*. This was because it affected an anchor "`<a>`" HTML element that is not accounted for by the RLG because it considers it an "inline" element (similar to text markup elements such as "`<strong>`" and "`<em>`", which make characters appear in boldface and italic, respectively, unless programmed to do otherwise in the CSS of a page). However, in this instance, the change to the element's properties in the DOM produced an observable impact on the page. We intend to investigate how we can account for such "corner case" issues as part of future work, as explained in Section 6.

Overall, however, these results illustrate how important the choice of viewport widths is when checking the responsive design of a web page, lending strong empirical support for the benefits of employing the RLG to represent the whole range of viewport widths instead of the static viewport modelling used by SPOTCHECK-ALIGNMENTGRAPH and other web testing tools.

Finally, we tested for statistical significance with Fisher's Exact Test, using the total numbers of layout changes detected and not detected by each of the two techniques compared in this research question. The p-value is 0.01594, indicating the probability of SPOTCHECK-ALIGNMENTGRAPH outperforming REDECHECK-*RM* is low, giving a result that is statistically significant at $\alpha = 0.05$.

Conclusion for RQ2: The results show that the performance of REDECHECK-*RM* is consistent regardless of the subtlety of a layout change, with good levels of detection observed across all nine buckets. The results also suggest that the effectiveness of REDECHECK-*RM* is not correlated to the value of *VW*, while also revealing a link between the number of modified viewport widths and SPOTCHECK-ALIGNMENTGRAPH's ability to detect the layout changes. Ultimately, this result shows that the choice of the viewport widths to inspect is of great significance to any single-viewport approach, like SPOTCHECK-ALIGNMENTGRAPH, and thus lends empirical support to the benefits of using an RLG to model a web page after performing a thorough sampling of its responsive layout.

RQ3: How do different parameter values change the approach's efficiency and effectiveness? Table V shows the number of layout changes detected for each of the subject web pages under each combination of sample technique and step size, along with the number of layout changes introduced into each subject. The change detection results reveal that, for the pool of changes considered, the step size used in the initial sampling process has only a very minor impact on the capabilities of REDECHECK-*RM* when using either the EXTRACT-COMPARE-INTERVAL-BREAKPOINTS or EXTRACT-COMPARE-INTERVAL sampling approach, and that the two approaches are almost identical in terms of their change detection ability. There are, however, a couple of interesting results to note. First, for Ashton Snook and Pay Demand, EXTRACT-COMPARE-INTERVAL-BREAKPOINTS outperforms EXTRACT-COMPARE-INTERVAL across all step sizes. This illustrates the benefits of combining the basic interval sampling approach with breakpoint sampling, as only sampling "blindly" means the approach can fail to model layout changes occurring around breakpoints coded in the page's CSS files. Secondly, in five of the subjects, using a larger step size resulted in a small reduction in fault detection ability when using EXTRACT-COMPARE-INTERVAL, while EXTRACT-COMPARE-INTERVAL-BREAKPOINTS demonstrates no reduction in detection ability as the step size increases, furnishing further empirical support for the inclusion of the extracted breakpoints in the initial sample of viewport widths.

Table V. REDECHECK-*RM*'s effectiveness at detecting layout changes at step sizes that vary from 10px to 500px. The label "I" stands for the EXTRACT-COMPARE-INTERVAL method and "IB" denotes the use of the EXTRACT-COMPARE-INTERVAL-BREAKPOINTS method that combines interval sampling with the use of breakpoints. This table also uses the "EX" label for the EXTRACT-COMPARE-EXHAUSTIVE method.

WEB PAGE	NUM.	10px		20px		40px		60px		80px		100px		150px		200px		500px		EX
	CHANGES	I	IB	I	IB	I	IB	I	IB	I	IB	I	IB	I	IB	I	IB	I	IB	
Aftrnoon	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
Coursera	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
Denon	10	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
BitTorrent	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	28	27	28	28	28	27	28	27	28	27	28	26	28	28
ISSTA 2016	19	12	12	12	12	12	12	12	12	12	12	12	12	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	19	18	19	18	19	17	19	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

The results show that EXTRACT-COMPARE-INTERVAL-BREAKPOINTS is better than EXTRACT-COMPARE-INTERVAL for six of the subjects and that, for all other web pages, the techniques are equally effective, leading us to recommend the use of EXTRACT-COMPARE-INTERVAL-BREAKPOINTS over EXTRACT-COMPARE-INTERVAL. Finally, the results show that employing the EXTRACT-COMPARE-EXHAUSTIVE sampling technique offers almost negligible additional detection capability, detecting the same number of responsive layout changes as at least one, and often several, configurations of the other two extraction techniques. Ultimately, the results in Table V provide empirical support for the benefits of EXTRACT-COMPARE-INTERVAL-BREAKPOINTS's combined approach to sampling the layout of a responsive web page.

These detection results make choosing a step size to recommend easier as it is no longer based on a compromise between the capability to detect layout changes and execution time and is instead a decision based primarily on efficiency. Given EXTRACT-COMPARE-EXHAUSTIVE provides minimal effectiveness benefit and is significantly less efficient than either EXTRACT-COMPARE-INTERVAL-BREAKPOINTS or EXTRACT-COMPARE-INTERVAL, we do not recommend it for use in the iterative development scenario we envisioned for REDECHECK-*RM*, as shown in Figure 4. For instance, the execution times for EXTRACT-COMPARE-EXHAUSTIVE ranged from 89 seconds to 148 seconds, with a mean of 108 seconds. In comparison, EXTRACT-COMPARE-INTERVAL-BREAKPOINTS and EXTRACT-COMPARE-INTERVAL demonstrated average execution times of less than 30 seconds for all step sizes. The main reason for this result is that EXTRACT-COMPARE-EXHAUSTIVE samples every web page at all 1001 viewport widths in the sample range, regardless of page complexity or how frequently the relative layout of elements change, resulting in wasted computation and a delay in returning results to the end-user. In comparison, using the other two approaches, RLGs for several subjects could be extracted by sampling less than 100 viewport widths — a reduction in sampling effort of over 90% — due to the more "intelligent" sampling that, only when the situation requires it, operates at a finer granularity and samples at a CSS breakpoint.

It is also important to note that, in responsive web design, there is an expectation of consistency in layout across similar viewport widths. Since the results in Table V indicate that REDECHECK-*RM* and EXTRACT-COMPARE-INTERVAL can capture the majority of responsive layout changes, this means that EXTRACT-COMPARE-EXHAUSTIVE samples a web page at an unnecessarily large number of viewport widths. Finally, for the usage scenario presented in Figure 4, a mean execution time of nearly two minutes, as observed for EXTRACT-COMPARE-EXHAUSTIVE, is likely to be too high by many web developers, as it would likely disrupt their workflow and force them

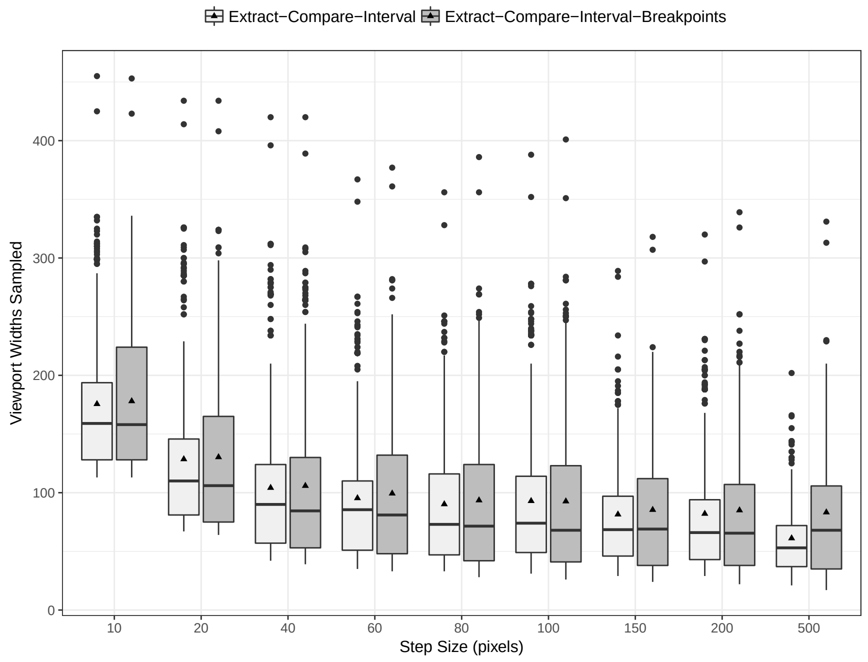


Figure 9. Across all 15 subject web pages and for the six studied step sizes, the number of viewport widths sampled by EXTRACT-COMPARE-INTERVAL (i.e., light grey boxes) and EXTRACT-COMPARE-INTERVAL-BREAKPOINTS (i.e., dark grey boxes) during the construction of the responsive layout graph. In this plot, each box represents the inter-quartile range (IQR). The whiskers in this plot extend up to 1.5 times the IQR and the line across the middle of the box marks the median value. Additionally, the triangle in the boxes denotes the mean and the filled circles extending beyond the whiskers correspond to outliers.

to pause their work while REDECHECK-*RM* runs, something that we deem unacceptable even though Table V shows that EXTRACT-COMPARE-EXHAUSTIVE detects one more layout change than EXTRACT-COMPARE-INTERVAL-BREAKPOINTS. With that said, we allow that EXTRACT-COMPARE-EXHAUSTIVE is a sensible choice when testing time is not constrained or when a web developer decides that they cannot risk overlooking any responsive layout changes — for example, they are making final changes to a web page, or making changes to a web page that is already live.

After discarding EXTRACT-COMPARE-EXHAUSTIVE as a candidate for recommendation, it is important to compare EXTRACT-COMPARE-INTERVAL-BREAKPOINTS and EXTRACT-COMPARE-INTERVAL. The efficiency results shown by the boxplot in Figure 9 reveal that for both sampling techniques, using a very small step size such as 10 pixels results in, as might be expected, a much larger number of viewport widths being sampled during the building of the RLGs in comparison to other larger step sizes. For instance, the mean numbers of required widths for 10 pixels were 176 for EXTRACT-COMPARE-INTERVAL and 180 for EXTRACT-COMPARE-INTERVAL-BREAKPOINTS, while for 20 pixels it was 129 and 130, respectively. The number of viewport widths required continues to trend to lower values as the step size in use increases, with a noticeable difference between 20 pixels and 40 pixels, before reaching a plateau at around 60 pixels.

Interestingly, there was almost no difference between the values for EXTRACT-COMPARE-INTERVAL-BREAKPOINTS and those of EXTRACT-COMPARE-INTERVAL. We expected EXTRACT-COMPARE-INTERVAL-BREAKPOINTS to result in a reasonable reduction in the number of widths sampled overall by removing the need for the binary searches that may require sampling at a costly number of viewport widths. We hypothesized two main reasons for this. The first was that many layout changes observed during the extraction of the RLG could be at viewport widths not programmed as breakpoints in the site's CSS files, meaning that identical searches were performed using both sampling techniques. The second was that, for most web pages, adding a set of breakpoints into the initial sample set would result in a considerable initial overhead for the approach, making it more difficult to obtain an overall improvement in efficiency.

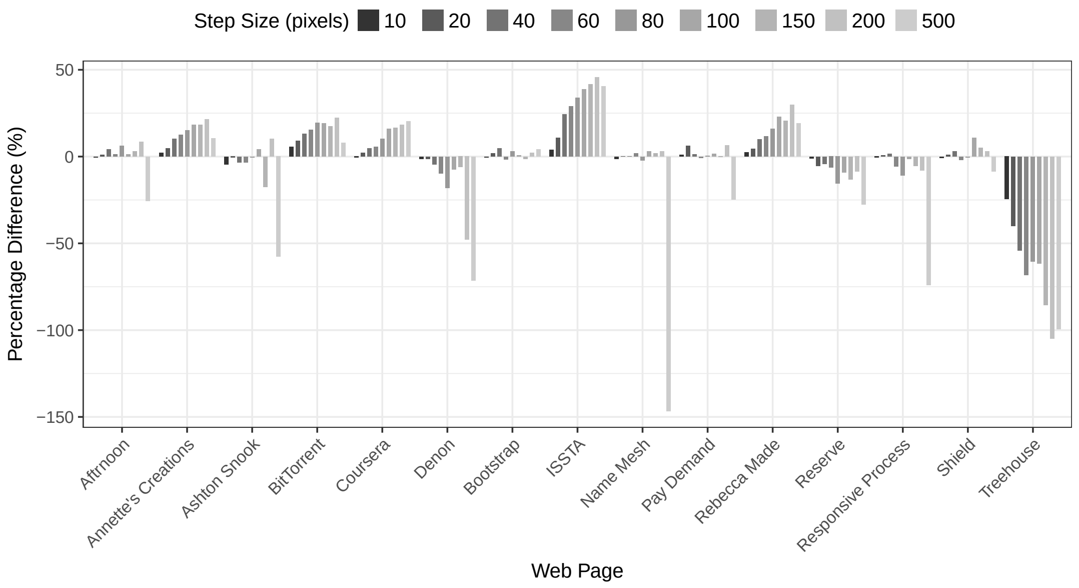


Figure 10. Following the equation furnished in Section 4.5, the percentage difference between the number of sampled viewport widths for EXTRACT-COMPARE-INTERVAL-BREAKPOINTS and EXTRACT-COMPARE-INTERVAL. In this graph, a positive value indicates that EXTRACT-COMPARE-INTERVAL-BREAKPOINTS sampled fewer viewport widths than the EXTRACT-COMPARE-INTERVAL method. Alternatively, a negative value reveals that EXTRACT-COMPARE-INTERVAL-BREAKPOINTS sampled more viewport widths than EXTRACT-COMPARE-INTERVAL. For each subject web page, the graph plots the percentage difference for each of the nine step sizes (i.e., 10, 20, 40, 60, 80, 100, 150, 200, and 500 pixels).

To further investigate this issue, we analyzed the difference in the number of viewport widths sampled on a web page by web page basis. Figure 10 shows the mean percentage difference for each web page using each step size, using the equation defined in Section 4.5's description of the methodology for RQ3. The most striking results are clearly those of ISSTA 2016, for which EXTRACT-COMPARE-INTERVAL-BREAKPOINTS is substantially more efficient than EXTRACT-COMPARE-INTERVAL and for Treehouse, where the opposite result is observed and EXTRACT-COMPARE-INTERVAL-BREAKPOINTS sampled significantly more viewport widths than EXTRACT-COMPARE-INTERVAL. Further analysis revealed this to be due to ISSTA 2016 exhibiting layout changes almost entirely at widths programmed as breakpoints in its CSS, allowing for the added breakpoints to provide a large boost in efficiency, while Treehouse had so many breakpoints in its style sheet and therefore in the initial sample set, that it was challenging for EXTRACT-COMPARE-INTERVAL-BREAKPOINTS to offer efficiency savings over EXTRACT-COMPARE-INTERVAL.

The remaining web pages exhibit much less severe benefits or costs, with the majority indicating that REDECHECK-*RM*'s combined sampling with EXTRACT-COMPARE-INTERVAL-BREAKPOINTS offered either equivalent or slightly increased efficiency. Further analysis of these scenarios showed that sometimes the savings captured by reducing the quantity of binary searches is only sufficient to cause EXTRACT-COMPARE-INTERVAL-BREAKPOINTS to be equally as efficient as EXTRACT-COMPARE-INTERVAL. In these scenarios, using EXTRACT-COMPARE-INTERVAL-BREAKPOINTS, and the larger initial sample associated with it, would likely result in a more accurate RLG which better represents the overall responsive layout of the page, as the initial sample would be less likely to overlook changes in layout. In contrast, using the smaller sample size from EXTRACT-COMPARE-INTERVAL could potentially fail to sample certain layout behaviors which occur between two consecutive sample widths, producing an RLG which does not accurately model the web page's layout. For this reason, if their efficiencies are equivalent, then we support the use of EXTRACT-COMPARE-INTERVAL-BREAKPOINTS over EXTRACT-COMPARE-INTERVAL.

To further investigate the trends illustrated in Figure 10, we performed statistical hypothesis testing on the data for the various step sizes using two different statistical calculations, using

Table VI. The left-hand table shows the results of the statistical hypothesis tests and effect size computations that compare the EXTRACT-COMPARE-INTERVAL-BREAKPOINTS and EXTRACT-COMPARE-INTERVAL techniques at each step size. The right-hand table shows the comparison of consecutive step sizes using the EXTRACT-COMPARE-INTERVAL-BREAKPOINTS sampling approach. Section 4.5 explains the statistical methods used to compute the p-value and the $\hat{A}_{12}$ value shown in both of these tables.

STEP SIZE (pixels)	p-value	$\hat{A}_{12}$
10	0.820	0.496
20	0.384	0.517
40	0.299	0.520
60	0.611	0.510
80	0.257	0.522
100	0.063	0.536
150	0.399	0.516
200	0.129	0.529
500	0.000	0.426

STEP SIZE (pixels)	p-value	$\hat{A}_{12}$
10 vs 20	0.000	0.751
20 vs 40	0.000	0.644
40 vs 60	0.031	0.542
60 vs 80	0.010	0.549
80 vs 100	0.335	0.519
100 vs 150	0.115	0.530
150 vs 200	0.419	0.516
200 vs 500	0.634	0.509

the approach described in Section 4.5. First, we compared the distributions of the two sampling techniques for each of the nine step sizes, with the results shown in Table VI. The Mann–Whitney U-test values for eight of the nine step sizes show that there is no evidence to support the statement that the two techniques demonstrate significantly different efficiencies for the subjects considered in this study, as their values were higher than the standard threshold value of $\alpha = 0.05$, with the only exception being 500 pixels. The observed values provide further evidence of the small efficiency benefit of using EXTRACT-COMPARE-INTERVAL-BREAKPOINTS, as the majority of values are above 0.05. Yes, none of these values are large enough to be classified as a significant effect size.

Following on from our previous statistical tests which showed EXTRACT-COMPARE-INTERVAL-BREAKPOINTS to be slightly, although not significantly, more efficient than EXTRACT-COMPARE-INTERVAL, we conducted further statistical tests to determine a suitable step size with which to run EXTRACT-COMPARE-INTERVAL-BREAKPOINTS, the results of which are shown in Table VI. The results paint a similar picture to the boxplots shown in Figure 9. The negligible p-value in Table VI resulting from the Mann–Whitney U-test values for the pairs 10 pixels–20 pixels, 20 pixels–40 pixels, 40 pixels–60 pixels, and 60 pixels–80 pixels are indicative of the number of sampled widths required decreasing as the step size increases, while the values of 0.335 for 80 pixels–100 pixels and 0.115 for 100 pixels–150 pixels correspond with the plateau at the 80 pixels step size.

Finally, if the values for $\hat{A}_{12}$ are interpreted as probabilities, then we can see that 20 pixels is highly likely, at 75.1%, to outperform (i.e., require fewer sampled viewport widths) 10 pixels while 40 pixels will outperform 20 pixels with a probability of 64.4%. The comparison between 40 pixels and 60 pixels produced an effect size of 0.542, suggesting our decision to use 60 pixels rather than 40 pixels was sensible, albeit not significantly. However, when comparing 60 pixels and 80 pixels, the effect size of 0.549, which when using the thresholds suggested by Vargha and Delaney [36], indicates the benefit of using 80 pixels instead of 60 pixels is nearly classified as having a "small" effect size, indicating that we could have potentially used 80 pixels rather than 60 pixels. Finally, the values corresponding to the 80 pixels–100 pixels comparisons and onwards are close to 0.5, indicating that the performance achieved by each technique in the pair of step sizes is nearly equivalent. When we combine this insight with the plateaus observed in Figure 9, we conclude that there is little benefit from using a step size of 100 pixels rather than the 60 pixels we used — or the 80 pixels step size that the statistical hypothesis tests suggest we could also adopt.

Conclusion for RQ3: For the pool of mutated subjects used in this evaluation, neither the sampling technique nor the step size used had a significant impact on the change detection capability of REDECHECK-*RM*. With the exclusion of a minimal number of outliers, EXTRACT-COMPARE-INTERVAL-BREAKPOINTS was shown to provide an efficiency benefit over EXTRACT-COMPARE-INTERVAL for all but two of the step sizes investigated. The results also support our use of 60 pixels as the step size in the previous research questions and we therefore recommend it or 80 pixels as a global step size, which should perform well across a wide variety of responsive web pages.

4.8. Discussion

Since the results for the SPOTCHECK-MANUAL technique required this paper's authors to systematically classify the mutants, following the procedure described in Section 4.4 and illustrated in Figure 8, this process also affords the opportunity to characterize the mutated pages. Our review of those mutants that contained layout changes suggests that the mutation operators in Table II produced mutants with regressions that did, in fact, vary in terms of their subtlety. For instance, while some mutants exhibited layout changes in a navigation bar at many viewport widths, other mutants had elusive layout regressions that only appeared at a few viewport widths. We also noticed that many of the mutants that did not have layout regressions were still challenging to confirm as correct because their designs were nuanced, placing web elements in a way that was not obviously proper until we inspected multiple viewport widths. In summary, after studying the responsive layouts of the mutated subjects we were convinced that these mutants were representative in nature and thus form a suitable foundation for this paper's experiments, an insight that is open to external confirmation since additional details about the subjects and their mutants are available in a GitHub repository that we have made publicly available at *https://github.com/redecheck/jstvr-webpages*.

Although not performed in a controlled environment, our experiments did highlight one of the major problems with manual web page testing: its labour intensiveness. For example, when the authors operated as web developers performing manual testing for RQ1, we recorded and averaged the times for taken by each of the three authors for each web page. We found the shortest mean time observed was 37 seconds for Annette's Creations, while the longest was observed while analyzing Shield, where one author took on average more than 10 minutes to perform their manual checks. We observed a general trend in which checking time increased as the size and complexity of the web page increased; this is to be expected, as larger web pages not only contain more elements to check, but can in many cases require considerable manual scrolling up and down the page. The aforementioned subjects are excellent examples of this, with Annette's Creations requiring very little scrolling to view the entire page, even at small viewport widths, while Shield requires a large amount of scrolling at all but the very widest viewport widths. We also observed considerable differences in the time required by the different authors, suggesting that in practice, some web developers may require considerably more time to perform a manual checking procedure than others.

It is worth noting, however, that REDECHECK-*RM* was faster than even the fastest human author/developer during our experiments. As described in Section 4.5, each of the three authors followed the manual procedure outlined in Section 4.4 before discussing each result as a committee to produce a final classification. During this process, we discovered our initial classifications differed for 21 of the 60 modified pages, highlighting not only the subjectiveness of manual testing, as different developers had varying concepts of what constituted a layout change, but also its error-prone nature. Notably, in some instances at least one developer failed to identify the injected layout change. This further motivates the technique presented in this paper as it is clear that a reliable, quick, and automated approach to responsive testing is practically beneficial.

In the remainder of this subsection, we discuss additional insights that emerged during the implementation, use, and experimental evaluation of the REDECHECK-*RM* tool.

While implementing and empirically studying REDECHECK-*RM*, we observed several qualitative benefits to using our new method for automatically detecting layout changes in responsive web pages. First and foremost, it is critical to stress the automated modelling of a web page obviates the burden upon the web developer to manually select viewport widths for inspection, or to only study the widths advocated by current testing tools, which was shown in RQ1 and RQ2 of our evaluation to be problematic. In theory, the web developer could also simply select a set of random widths for inspection in the hope of detecting any layout issues that may have occurred, but this approach would likely be highly unreliable as there is no guarantee the change will be observable in the browser at any of the selected widths, thus meaning many changes could go undetected.

For instance, in one of our previous empirical studies [8], selecting random widths or those widths advocated by testing tools was shown to miss between 19% and 34% of the verified layout failures investigated, providing further empirical evidence for the shortcomings of either manual or automated spotchecking. The modelling process of REDECHECK-*RM* is particularly beneficial since we observed that changes to a page's responsive layout frequently manifest themselves at hard-to-predict viewport widths that are often between the regular breakpoints employed by developers and RWD frameworks and well away from the default widths suggested by most testing tools.

Our experiences also suggest that REDECHECK-*RM*'s automatically generated reports will prove useful to developers as a guide for their in-depth manual inspection of a page. Given its modelling of dynamic layout changes across a full range of viewport widths, the RLG and the accompanying DOM-based context presented in the report show not only if a change has been detected, but also where (which elements/relationships), how (what has changed in the responsive layout) and when (at which viewport widths is the change visible). We anticipate that this detailed information will usefully accompany any verdict as to whether the tool detected any changes in a web page's layout. We also foresee these benefits becoming even more pronounced with further enhancements to REDECHECK-*RM*, such as an interactive RLG model with the differing parts of the graph highlighted for the developer, or a screenshot-based approach showing the modified elements and relationships. Additional plans for improvements to REDECHECK-*RM* can be found in Section 6.

While the results of our experiments show REDECHECK-*RM* failed to detect a small proportion of the injected layout changes, manual analysis of some of these false-negative results revealed the vast majority of them were due to the changes in the underlying DOM representation of the web page being too small to produce a difference in the extracted attribute sets. A common example of this would be a small shift in the position of an element, something which is highly unlikely to be observed by a human viewing the web page, suggesting the shortcoming in our approach is minimal. In summary, our experiences suggest that the REDECHECK-*RM* tool automatically detects regressions in responsive layout that web developers will deem worthy of further investigation.

Finally, since many modern web sites consist of multiple pages, a tester wishing to check for layout changes across an entire site would need to run REDECHECK-*RM* on a page-by-page basis. Since CSS files are almost always shared between pages to preserve a consistent look-and-feel on a site, it is important to check for layout regressions on all pages, as a CSS modification which causes no changes on one page could have a dramatic influence on another page. While this may be cumbersome, the process is fully amenable to automation with the current implementation of the tool. With that said, we plan, as part of future work, to enhance REDECHECK-*RM* so that it offers a more streamlined experience to developers who test sites with many pages. However, given the increasing prominence of single-page web applications [42], we anticipate that the current version of the REDECHECK-*RM* tool will effectively support the testing of many popular web sites.

5. RELATED WORK

Along with reviewing the related work in responsive web testing, this section explains relevant work in areas such as graphical user interface (GUI) testing, mobile app testing, and regression testing. This section also overviews some developer tools for responsive web testing, noting their limitations compared to REDECHECK-*RM*. Since we used mutation operators to create potential layout regressions, this section also reviews the related work in the area of web mutation testing. It is worth noting at the outset of this section that this paper's authors also described the REDECHECK tool that automatically determines if a web page exhibits one of several representative layout failures [8, 9]. In collaboration with Althomali, a subset of this paper's author list also developed an automated web testing technique, called VISER, that confirms whether or not a layout failure detected by REDECHECK is visible to humans [43]. Yet, this work on REDECHECK and VISER does not address the problem that REDECHECK-*RM* handles: detecting regressions from a correct responsive layout that occur when a developer modifies a web page's responsive design.

GUI Testing. In the same way that the REDECHECK-*RM* tool uses the RLG to model a web page's responsive layout, several prior approaches have used a model for a graphical user interface. For instance, Memon et al. developed a technique called GUI ripping, which automatically explored the interface of a GUI and created a "GUI forest" to represent its window and elements [44]. Other models of GUIs, like the one presented by Yang et al., also support various automated testing activities [45]. While representations like these support the testing of GUI functions, they do not specifically enable the testing of GUI layout, as the RLG presented in this paper does for web pages. In the same way that the RLG supports the detection of regressions in the layout of a responsive web page, these representations of GUIs can also support regression testing [46]. Finally, much like this paper's automated approach for differencing two RLGs, prior work by Xie et al. showed how to use an extracted model of a GUI to detect differences in versions of a graphical application [47].

Mobile App Testing. Since mobile apps also run on devices with a variety of viewport widths, there are some similarities between this paper's web-based technique and those that test mobile apps. Leveraging Memon et al.'s GUI ripping technique, Amalfitano et al. presented a tool that systematically extracts and explores the interface of an Android app [48]. In addition to testing a mobile app's behavior when the device screen orientation changes [49], other approaches automate testing tasks for those mobile apps that have certain interface design characteristics [50, 51]. Much like this paper, several prior articles presented tools that automatically report either design violations or potential defects in a mobile app. Although their focus is not app layout, Hu et al. presented a technique that leverages many physical or emulated devices to explore an Android app and suggest potential defects [52]. Finally, Moran et al. respectively proposed two approaches that detect and summarize the changes in the interface to a mobile app [53] and report violations of design guidelines for apps [54], much in the same way the presented tool works for responsive web pages.

Need for Web Testing. Motivating the development of REDECHECK-*RM*, there is an extensive literature on both the necessity of a suitable web site and the challenges of web development. For instance, prior work has revealed the benefits of having a web page with an aesthetically pleasing layout, thereby motivating the need for the technique presented in this paper. Lee and Koubek also discovered that, while many aspects of a web page's design influence perceived usability, its layout was the determining factor for many people [55]. Moreover, Robins and Holmes showed that people judge a web page as highly credible if it has polished aesthetics [56]. While Mbipom and Harper found that an aesthetically pleasing web page was more accessible for individuals with a visual impairment [57], Cyr et al. reported that a web page with a good layout engenders greater customer loyalty [12]. Finally, Li et al. noted that users will stop purchasing products from a web site if it contains visible failures [58]. Although these papers show that an organization benefits from having a suitable web site, there are inherent challenges associated both with learning web development fundamentals [5, 6] and keeping current with the emerging responsive web design frameworks [7].

Traditional Web Testing. Since previous studies of web applications demonstrated that failures in a page's appearance make up one of the largest categories of defects in a deployed site [59], many researchers have created techniques to test web pages. For instance, Wang et al. designed a technique for enhancing the presentation layer of a web application [60]. Yet, since presentation failures often appear in production sites — even with use of methods like the aforementioned one — it is important to subject them to further testing, leveraging approaches that target dynamic web applications, such as the ones presented by Dallmeier et al. [61], Marchetto et al. [62], Milani et al. [63], and Mesbah et al. [64]. Several prior papers, like that of Halfond and Orso [65] and Halfond [66], emphasize testing the function calls to the server code in a dynamic web application. Other work by Sampath et al. [67, 68] and Sprenkle et al. [69, 70] enhanced web testing tools by modelling the behavior of the people who browse a web site. Unlike this paper's method, those presented by Halfond and Orso, Wassermann and Su, Sampath et al., and Sprenkle et al. are not tailored to check the responsive layout of a web page. Finally, while the approach presented by Wang et al. [71] uses a graph to test a traditional web page, it also does not focus on detecting regressions in responsive layout.

Graph-Based and Cross-Browser Web Testing. Although the REDECHECK-*RM* tool presented in this paper extracts and differences a responsive layout graph for a web page, it is worth noting that that are many program-based techniques that extract and difference graphs, such as those developed by Apiwattanapong et al. [72] and Raghavan et al. [73]. With that said, since this paper focuses on automatically checking web pages, the remainder of the discussion only reviews prior work in the domain of web applications. As first discussed in Section 2.3, Choudhary et al. [24] proposed the alignment graph, as part of the X-PERT tool, which uses a combination of techniques for the automatic detection of cross-browser incompatibilities [24]. A tool developed by Dallmeier et al., called WEBMATE, also focused on cross-browser testing as it searched for visual differences and missing functionality, among other issues [61]. Along with the work by Mesbah et al. that drew attention to the need for cross-browser testing [1], the WEBDIFF tool, created by Choudhary et al. [35] was one of the first approaches in this area. Finally, Alameer et al. [34] investigated the detection of presentation failures introduced after the translation of a web page into a different language, using relative layout as the basis for their model and a layout graph which formed the foundation of their differencing tool, called GWALI. It is important to note that, unlike the focus of this paper, all of the aforementioned methods do not consider a web page's responsive design. With that said, these examples of related work share one point in common with this paper's experiments: the use of manual inspection. For instance, Alameer et al. [34], Choudhary et al. [24, 35], and Mesbah et al. [1] all present studies of web testing techniques that incorporate manual inspection.

Image-Based Web Testing. Leveraging either screenshots or graphical mockups of a web page, other testing techniques surface presentation failures by detecting the differences between the provided images. For instance, tools like WEBSEE [30, 74] and FIERYEYE [75] use image differencing to compare, for instance, the before and after screenshots of a web page, reporting any visual differences to a developer who would then attempt to localize the problem in the page's HTML and CSS source code. Other tools, like BROWSERBITE [76], combine image comparison methods with machine learning techniques to detect cross-browser differences in a web page. Developer testing tools, like WRAITH [15], also use image comparisons to highlight the differences between two versions of a web page and therefore suffer from the same limitations as WEBSEE and FIERYEYE in the domain of responsive web page testing. Although not in the domain of web testing, other prior work, like that of Chang et al., showed how to use the image comparison methods employed by tools like WEBSEE to test applications with a GUI [77]. In contrast to REDECHECK-*RM*, all of the aforementioned tools operate at a single viewport width and are therefore not well-suited to testing responsive web pages since they would require a substantial number of screenshots. It may also be difficult for developers to apply the tools like WEBSEE, FIERYEYE, and WRAITH to the task of responsive web page testing since the results in Section 4 highlight the difficulties that arise when selecting specific viewport widths at which to inspect a web page's responsive layout. Finally, it is worth noting that the approach presented by Mahajan and Halfond also used image comparison, in conjunction with the same R-tree adopted by REDECHECK-*RM*, to both detect and automatically localize the HTML elements that are most likely to cause web presentation failures [41].

Specification-Based Web Testing. Prior work in web testing also includes specification-based approaches to detecting web presentation issues. For instance, the CORNIPICKLE [78, 79] tool by Hallé et al. requires a tester to define the intended layout of a web page through the use of a layout specification language. This tool then notifies the user if any of the layout constraints have been violated during testing. It is important to note that CORNIPICKLE targets static layout failures and provides no support for specifying a page's responsive layout, thus limiting its applicability in this paper's domain. Moreover, CORNIPICKLE's usefulness depends on the quality of the specification written by the web developer. Since the creation and maintenance of a layout specification for input to CORNIPICKLE may be a time-consuming and error-prone process, the REDECHECK-*RM* tool presented in this paper will continue to prove useful to many practicing web developers.

Developer Tools. While many web developers support testing a page on actual devices [80], this is not always a feasible option given the many available devices. However, there are a wide range of software tools available as alternatives. For instance, services such as BrowserStack [81] give developers the ability to remotely check a page on a wide array of devices. Multiple screenshot tools, such as the one developed by Kersley [82], display a web page at a series of common viewport widths, allowing a tester to see a page's responsive behavior and detect some obvious layout regressions. Viewport resizing tools, such as RESPONSIVEPX [13], RESIZER [14], and WRAITH [15] and browser-based tools such as "Responsive Design Mode" for Firefox and "Device Mode" for Chrome support the automated resizing of a browser, thereby allowing developers to check pages for layout regressions. While these spotchecking tools can be useful to web developers, they may, unlike the REDECHECK-*RM* tool presented in this paper, cause a developer to overlook subtle layout regressions. Finally, Google's mobile-friendly testing tool [83] checks for common usability issues, such as too-small fonts and the use of Flash, and returns a verdict on the mobile-friendliness of a web page. Unlike REDECHECK-*RM*, it does not provide support for checking a web page's responsive layout, and is therefore limited for the problem presented in this paper.

Mutation Testing. Frequently used to assess the adequacy of a test suite and to obtain subjects for use in empirical studies, mutation testing techniques make syntactic changes to a program [31, 84]. The mutation testing techniques for web applications proposed by Praphamontripong et al. [85, 86] implemented operators for HTML, targeting the functional aspects of a page by changing link destinations, rather than the properties that control the layout of a web page. Instead of focusing on a page's HTML, both Mirshokraie et al. [87] and Rodríguez-Baquero and Linares-Vásquez [88] developed mutation operators for web applications, presenting tools that create mutants for the JavaScript programs used by a web page. Since none of these mutation methods were suitable for automatically inserting potential regressions into a web page's layout, to evaluate REDECHECK-*RM* this paper introduced an automated mutation technique that alters the HTML and CSS source code of a web page, thereby inserting a potential regression in its responsive layout.

Automated Repair of Presentation Failures. There have also been several recent works devoted to automatically repairing the presentation failures surfaced by web testing tools. For instance, Mahajan et al. present the $\mathcal{X}$FIX tool [89] and a search-based technique for automatically fixing cross-browser issues [90]. They also explain how to fix internationalization presentation failures using a combination of search-based techniques and style similarity clustering in the $\mathcal{I}$FIX [91] and $\mathcal{I}$FIX^{++} tools [92]. Alameer et al. [93] improve the efficiency of these techniques by incorporating constraint solving. Finally, Mahajan et al. [94] present a search-based approach to repairing mobile-friendly issues, including illegible font sizes, inadequate tap target spacing, and content sizing.

6. CONCLUSIONS AND FUTURE WORK

Before people commonly browsed the web with a mobile device, the main problem facing front-end web developers was the time consuming task of ensuring that their web pages displayed correctly on different desktop browsers [1]. Yet, the recent proliferation of web-enabled mobile devices has made it critical for developers to ensure that web pages provide a good user experience across a wide variety of viewport widths [95, 96]. Through the use of concepts such as grid-based layouts, flexible images, and CSS media queries, responsive web design (RWD) is an approach to web development that helps developers to create sites that accommodate the wide variety of viewport widths associated with the commonly used desktops, laptops, tablets, and smartphones [4]. Even though there are clear benefits to making responsive web sites [97], creating a correct web page is inherently challenging [6, 7]. In fact, there is evidence to suggest that practising software developers have many questions about developing mobile-friendly web pages: as of March 2020, the StackOverflow site hosts 1,416,648 questions tagged with labels connected to responsive web design [98]. Since it is often difficult for a developer to ascertain how modification's to a web page at one viewport width will influence its layout at other viewports, this paper presents REDECHECK-*RM*, a tool that automatically detects regressions in a page's responsive layout.

The foundation of REDECHECK-*RM* is the responsive layout graph (RLG) that represents the two critical aspects of responsive web design: the changing visibility and changing relative alignment of web page's elements [4]. After a web developer modifies a web page's HTML and/or CSS source code, REDECHECK-*RM* extracts "before" and "after" RLGs from versions of a web page and differences them, reporting any discrepancies. The use of REDECHECK-*RM* reduces the testing burden placed on the web developer, who now only needs to inspect the list of differences automatically discovered by the technique and determine if any of them were unintended regressions in the page's layout. Without REDECHECK-*RM*, a web developer would have to look for layout regressions by either manually spotchecking the web page or using a tool that inspects the page at a minimal number of viewport widths — both processes that are time-consuming and error-prone and could ultimately allow incorrect layouts to appear in the production version of a page.

Using 15 real-world web pages from a variety of application domains, we conducted experiments to study REDECHECK-*RM*'s efficiency and effectiveness. To determine if REDECHECK-*RM* can detect layout regressions in these subjects, we created and applied mutation operators that systematically modified both the HTML and CSS source code of a web page, thereby enabling us to answer three research questions. The experiments reveal that the answer to part (a) of the first research question is that REDECHECK-*RM* is effective at detecting the layout changes in the mutants with no false negative results and just one false positive result. Importantly, an investigation of the single false positive revealed that it had subtle shifts in the position of elements that would not be readily apparent to either a developer or a page visitor. Answering part (b) of the first question, the results show that REDECHECK-*RM*, with the greatest number of true positives and true negatives, is the best checking technique when compared to either manual spotchecking or a differencing tool that uses an alignment graph, classifying 12.5% and 18.75% more true positives, respectively.

The second research question determined how different methods for detecting responsive layout regressions varied in their effectiveness as the regression itself differed in its subtlety. The results show that the effectiveness of REDECHECK-*RM* is consistent regardless of the subtlety of a layout change, with good levels of detection observed across all studied subtleties. In contrast, a spotchecking technique based on the alignment graph is only effective at detecting layout regressions that are not subtle, providing inaccurate guidance to a web developer when the layout change is only visible at a minimal number of viewport widths. Finally, since a developer can configure how REDECHECK-*RM* extracts a page's responsive layout graph, the third research question studied how the tool's efficiency and effectiveness varied when paired with one of three RLG extraction methods. Along with revealing that exhaustive sampling of a page is not necessary, the results show that the most cost-effective approach extracts the RLG by sampling the page at both regular intervals and at breakpoints found in the page's style sheets. Finally, the results also support the use of either 60 or 80 pixels as the step size at which REDECHECK-*RM* should sample the page.

Since this paper's experimental results demonstrate the benefits of using REDECHECK-*RM*, we plan to further develop and evaluate the tool in future work. One focus of our future work is integrating into REDECHECK-*RM* approaches from other tools that leverage image differencing, such as WEBSEE [99] and VISER [43]. In addition to adding support for dynamic web pages and multiple-page web sites, as in tools like CRAWLJAX [100] and VFDETECTOR [101], we also plan for REDECHECK-*RM* to detect page changes that involve, for instance, fonts, colors, and the content of inline elements. Finally, in light of Section 4.7's results, we plan to enhance REDECHECK-*RM* to better ensure that it can even detect the subtle layout faults that a human might initially overlook.

Since this paper's experimental results depend on the representativeness of the mutants that serve as potential responsive layout regressions, we intend to add more and more varied mutation operators. Along with supporting the mutation of a web page's JavaScript, as done by the MUTANDIS tool [87], we intend for our mutation testing method to support the focused manipulation of a page's DOM, thereby better enabling the systematic creation of subtle mutants. After extending our mutation testing tool, we will follow the procedure proposed by Just et al. [102] to experimentally determine if these mutants are representative of real-world regressions in a web page's responsive layout. Before conducting this experiment and the others that will compare REDECHECK-*RM* to the new approaches mentioned in the previous paragraph, we will also collect larger and more complex subjects, thereby mitigating the validity threats of these future studies.

After experimentally enhancing our understanding of the trade-offs in using REDECHECK-*RM* to automatically identify layout regressions in complex responsive web pages, we intend to conduct a number of studies involving human subjects. Since this paper furnishes anecdotal evidence of the challenges that web developers face when checking a responsive page, we intend to conduct follow-on experiments with humans who differ in their level of expertise in responsive web development. Varying the type of layout regression and following a procedure similar to the one proposed by Fraser et al. [103], we will also experimentally determine the ways in which REDECHECK-*RM*, in comparison to baselines involving manual and automated spotchecking, helps web developers to detect layout regressions. Building on this paper's study, the goal of these follow-on experiments is to more accurately characterize the cost-benefit trade-offs associated with using REDECHECK-*RM*. The combination of this paper's implementation and evaluation of REDECHECK-*RM* with both the suggested enhancements to the tool and the new experimental studies will result in a cost-effective and well-understood tool that supports the creation of high-quality responsive web applications.

REFERENCES

1. Mesbah A, Prasad MR. Automated cross-browser compatibility testing. *Proceedings of the International Conference on Software Engineering*, 2011.
2. Statista. "Statistics portal: Mobile share of organic search", 2017. URL `https://www.statista.com/statistics/297137/mobile-share-of-us-organic-search-engine-visits/`.
3. Comscore. Comscore digital media blog: "Smartphone apps are now 50% of all U.S. digital media time spent", 2016. URL `http://www.comscore.com/Insights/Blog/Smartphone-Apps-Are-Now-50-of-All-US-Digital-Media-Time-Spent`.
4. Marcotte E. *Responsive Web Design*. A Book Apart, 2014.
5. Alston P, Walsh D, Westhead G. Uncovering "threshold concepts" in web development: An instructor perspective. *Transactions on Computing Education* 2015; **15**(1).
6. Park TH, Dorn B, Forte A. An analysis of HTML and CSS syntax errors in a web development course. *Transactions on Computing Education* 2015; **15**(1).
7. Bajaj K, Pattabiraman K, Mesbah A. Mining questions asked by web developers. *Proceedings of the Working Conference on Mining Software Repositories*, 2014.
8. Walsh TA, Kapfhammer GM, McMinn P. Automated layout failure detection for responsive web pages without an explicit oracle. *Proceedings of the International Symposium on Software Testing and Analysis*, 2017.
9. Walsh TA, Kapfhammer GM, McMinn P. REDECHECK: An automatic layout failure checking tool for responsively designed web pages. *Proceedings of the International Symposium on Software Testing and Analysis – Demonstration Papers*, 2017.
10. Frost B. "Mobile web problems and how to avoid them", 2020. URL `https://bradfrost.com/blog/post/mobile-web-problems/`.
11. Hartmann J, Sutcliffe A, De Angeli A. Investigating attractiveness in web user interfaces. *Proceedings of the International Conference on Human Factors in Computing Systems*, 2007.
12. Cyr D, Head M, Ivanov A. Design aesthetics leading to M-loyalty in mobile commerce. *Information & Management* 2006; **43**(8).
13. ResponsivePX. ResponsivePX tricky breakpoint finder, 2020. URL `http://responsivepx.com/`.
14. Google. "Resizer: An interactive viewer that helps designers test material design breakpoints across desktop, mobile, and tablet", 2020. URL `https://material.io/resources/resizer/`.
15. BBC News. Wraith: Responsive screenshot comparison tool, 2020. URL `https://github.com/BBC-News/wraith`.
16. Walsh TA, McMinn P, Kapfhammer GM. Automatic detection of potential layout faults following changes to responsive web pages. *Proceedings of the International Conference on Automated Software Engineering*, 2015.
17. World Wide Web Consortium. "CSS3 @media rule", 2016. URL `http://www.w3schools.com/cssref/css3_pr_mediaquery.asp`.
18. Bootstrap. Bootstrap: Responsive front-end framework, 2020. URL `http://getbootstrap.com/`.
19. BuiltWith. "Bootstrap usage statistics", 2017. URL `https://trends.builtwith.com/docinfo/Twitter-Bootstrap`.
20. ZURB. Foundation: Responsive front-end framework, 2020. URL `https://get.foundation/`.
21. ZURB. "Websites using ZURB Foundation", 2020. URL `https://zurb.com/responsive`.
22. Boyter B. "SCC: A very fast accurate code counter with complexity calculations and COCOMO estimates written in pure Go", 2018. URL `https://github.com/boyter/scc`.
23. Creative Bloq. "8 RWD problems (and how to avoid them)", 2014. URL `http://www.creativebloq.com/rwd/responsive-design-problems-12142790`.
24. Choudhary SR, Prasad MR, Orso A. X-PERT: Accurate identification of cross-browser issues in web applications. *Proceedings of the International Conference on Software Engineering*, 2013.
25. (W3C) WWWC. "JavaScript HTML DOM", 2020. URL `http://www.w3schools.com/js/js_htmldom.asp`.
26. World Wide Web Consortium. "XPath syntax", 2016. URL `http://www.w3schools.com/xsl/xpath_syntax.asp`.
27. Walsh TA, McMinn P, Kapfhammer GM. REDECHECK: Automatically detecting layout failures in responsive web pages, 2018. URL `https://github.com/redecheck/redecheck`.
28. SeleniumHQ. Selenium: Web browser automation, 2020. URL `https://www.selenium.dev/`.
29. PhantomJS. PhantomJS: Scriptable headless browser, 2020. URL `http://phantomjs.org`.
30. Mahajan S, Halfond WGJ. Detection and localization of HTML presentation failures using computer vision-based techniques. *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2015.
31. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering* 2011; **37**(5).
32. Viewport Resizer. Viewport resizer: Developer device testing toolbar for emulating screen resolutions, 2017. URL `http://lab.maltewassermann.com/viewport-resizer/`.
33. Window resizer 2015. URL `http://ionut-botizan.net/window-resizer/`.

34. Alameer A, Mahajan S, Halfond WGJ. Detecting and localizing internationalization presentation failures in web applications. *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2016.
35. Choudhary SR, Versee H, Orso A. WebDiff: Automated identification of cross-browser issues in web applications. *Proceedings of the International Conference on Software Maintenance*, 2010.
36. Vargha A, Delaney HD. A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 2000; **25**(2).
37. Ampatzoglou A, Bibi S, Avgeriou P, Verbeek M, Chatzigeorgiou A. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. *Information and Software Technology* 2019; **106**.
38. Mustafa N, Labiche Y, Towey D. Mitigating threats to validity in empirical software engineering: A traceability case study. *Proceedings of the Annual Computer Software and Applications Conference*, 2019.
39. JSoup. JSoup: Java HTML Parser, 2020. URL `http://jsoup.org`.
40. JStyleParser. JStyleParser: A CSS parser written in Java, 2020. URL `http://cssbox.sourceforge.net/jstyleparser/`.
41. Mahajan S, Halfond WGJ. Finding HTML presentation failures using image comparison techniques. *Proceedings of the International Conference on Automated Software Engineering*, 2014.
42. Gerstaecker H. "Single page applications: The rise of web apps in 2020", 2020. URL `https://hackernoon.com/single-page-applications-the-rise-of-web-apps-in-2020-un6c32gm`.
43. Althomali I, Kapfhammer GM, McMinn P. Automatic visual verification of layout failures in responsively designed web pages. *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2019.
44. Memon AM, Banerjee I, Nagarajan A. GUI ripping: Reverse engineering of graphical user interfaces for testing. *Proceedings of the Working Conference on Reverse Engineering*, 2003.
45. Yang W, Prasad MR, Xie T. A grey-box approach for automated GUI-model generation of mobile applications. *International Conference on Fundamental Approaches to Software Engineering*, 2013.
46. Memon AM, Soffa ML. Regression testing of GUIs. *Proceedings of the 11th International Symposium on the Foundations of Software Engineering*, 2003.
47. Xie Q, Grechanik M, Fu C, Cumby C. Guide: A GUI differentiator. *Proceedings of the International Conference on Software Maintenance*, 2009.
48. Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM. Using GUI ripping for automated testing of Android applications. *Proceedings of the International Conference on Automated Software Engineering*, 2012.
49. Amalfitano D, Riccio V, Paiva ACR, Fasolino AR. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Software Testing, Verification and Reliability* 2018; **28**(1).
50. Moreira RM, Paiva AC. PBGT tool: An integrated modeling and testing environment for pattern-based GUI testing. *Proceedings of the International Conference on Automated Software Engineering*, 2014.
51. Moreira RM, Paiva AC, Nabuco M, Memon A. Pattern based GUI testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability* 2017; **27**(3).
52. Hu G, Yuan X, Tang Y, Yang J. Efficiently, effectively detecting mobile app bugs with AppDoctor. *Proceedings of the European Conference on Computer Systems*, 2014.
53. Moran K, Watson C, Hoskins J, Purnell G, Poshyvanyk D. Detecting and summarizing GUI changes in evolving mobile apps. *Proceedings of the International Conference on Automated Software Engineering*, 2018.
54. Moran K, Li B, Bernal-Cárdenas C, Jelf D, Denys P. Automated reporting of GUI design violations for mobile apps. *Proceedings of the International Conference on Software Engineering*, 2018.
55. Lee S, Koubek RJ. The effects of usability and web design attributes on user preference for e-commerce web sites. *Computers in Industry* 2010; **61**(4).
56. Robins D, Holmes J. Aesthetics and credibility in web site design. *Information Processing & Management* 2008; **44**(1).
57. Mbipom G, Harper S. The interplay between web aesthetics and accessibility. *Proceedings of the International Conference on Computers and Accessibility*, 2011.
58. Li W, Harrold MJ, Görg C. Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors. *Proceedings of the International Conference on Automated Software Engineering*, 2010.
59. Guo Y, Sampath S. Web application fault classification: An exploratory study. *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2008.
60. Wang X, Zhang L, Xie T, Xiong Y, Mei H. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2012.
61. Dallmeier V, Burger M, Orth T, Zeller A. WebMate: A tool for testing Web 2.0 applications. *Proceedings of the Workshop on JavaScript Tools*, 2012.
62. Marchetto A, Tonella P, Ricca F. State-based testing of AJAX Web applications. *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008.
63. Milani Fard A, Mirzaaghaei M, Mesbah A. Leveraging existing tests in automated test generation for web applications. *Proceedings of the International Conference on Automated Software Engineering*, 2014.
64. Mesbah A, Van Deursen A, Roest D. Invariant-based automatic testing of modern web applications. *Transactions on Software Engineering* 2012; **38**(1).
65. Halfond WGJ, Orso A. Automated identification of parameter mismatches in web applications. *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008.
66. Halfond WGJ. Automated checking of web application invocations. *Proceedings of the International Symposium on Software Reliability Engineering*, 2012.
67. Sampath S, Sprenkle S, Gibson E, Pollock L, Souter Greenwald A. Applying concept analysis to user-session-based testing of web applications. *Transactions on Software Engineering* 2007; **33**(10).
68. Sampath S, Bryce RC, Viswanath G, Kandimalla V, Koru AG. Prioritizing user-session-based test cases for web applications testing. *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008.

69. Sprenkle S, Gibson E, Sampath S, Pollock L. Automated replay and failure detection for web applications. *Proceedings of the International Conference on Automated Software Engineering*, 2005.
70. Sprenkle SE, Pollock LL, Simko LM. Configuring effective navigation models and abstract test cases for web applications by analyzing user behaviour. *Software Testing, Verification and Reliability* 2013; **23**(6).
71. Wang W, Sampath S, Lei Y, Kacker R, Kuhn R, Lawrence J. Using combinatorial testing to build navigation graphs for dynamic web applications. *Software Testing, Verification and Reliability* 2016; **26**(4).
72. Apiwattanapong T, Orso A, Harrold MJ. A differencing algorithm for object-oriented programs. *Proceedings of the International Conference on Automated Software Engineering*, 2004.
73. Raghavan S, Rohana R, Leon D, Podgurski A, Augustine V. Dex: A semantic-graph differencing tool for studying changes in large code bases. *Proceedings of the 20th International Conference on Software Maintenance*, 2004.
74. Mahajan S, Halfond WG. WebSee: A tool for debugging HTML presentation failures. *Proceedings of the International Conference on Software Testing, Validation and Verification*, 2015.
75. Mahajan S, Li B, Behnamghader P, Halfond WGJ. Using visual symptoms for debugging presentation failures in web applications. *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2016.
76. Saar To, Dumas M, Kaljuve M, Semenenko N. Browserbite: Cross-browser testing via image processing. *Software: Practice and Experience* 2016; **46**(11).
77. Chang TH, Yeh T, Miller RC. GUI testing using computer vision. *Proceedings of the International Conference on Human Factors in Computing Systems*, 2010.
78. Hallé S, Bergeron N, Guerin F, Le Breton G. Testing web applications through layout constraints. *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2015.
79. Hallé S, Bergeron N, Guérin F, Le Breton G, Beroual O. Declarative layout constraints for testing web applications. *Journal of Logical and Algebraic Methods in Programming* 2016; **85**.
80. Montague D, Hogan L. *Building a Device Lab*. Five Simple Steps, 2015.
81. BrowserStack. Browserstack, 2017. URL `https://www.browserstack.com`.
82. Kersley M. "Responsive design testing", 2017. URL `http://mattkersley.com/responsive/`.
83. Google. Mobile-friendly test, 2015. URL `https://www.google.com/webmasters/tools/mobile-friendly/`.
84. Kapfhammer GM. Software testing. *The Computer Science Handbook*. 2004.
85. Praphamontripong U, Offutt J. Applying mutation testing to web applications. *Workshop Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2010.
86. Praphamontripong U, Offutt J, Deng L, Gu J. An experimental evaluation of web mutation operators. *Proceedings of the Workshop on Mutation Analysis*, 2016.
87. Mirshokraie S, Mesbah A, Pattabiraman K. Guided mutation testing for JavaScript web applications. *Transactions on Software Engineering* 2015; **41**(5).
88. Rodríguez-Baquero D, Linares-Vásquez M. Mutode: Generic JavaScript and Node.js mutation testing tool. *Proceedings of the International Symposium on Software Testing and Analysis*, 2018.
89. Mahajan S, Alameer A, McMinn P, Halfond WGJ. XFix: Automated tool for repair of layout cross browser issues. *International Conference on Software Testing and Analysis*, 2017.
90. Mahajan S, Alameer A, McMinn P, Halfond WGJ. Automated repair of layout cross browser issues using search-based techniques. *Proceedings of the International Conference on Software Testing and Analysis*, 2017.
91. Mahajan S, Alameer A, McMinn P, Halfond WGJ. Automated repair of internationalization failures using style similarity clustering and search-based techniques. *Proceedings of the International Conference on Software Testing, Validation and Verification*, 2018.
92. Mahajan S, Alameer A, McMinn P, Halfond WGJ. Effective automated repair of internationalization presentation failures in web applications using style similarity clustering and search-based techniques. *Software Testing, Verification and Reliability*, To Appear.
93. Alameer A, Chiou PT, Halfond WGJ. Efficiently repairing internationalization presentation failures by solving layout constraints. *Proceedings of the International Conference on Software Testing, Validation and Verification*, 2019.
94. Mahajan S, Abolhassani N, McMinn P, Halfond WGJ. Automated repair of mobile friendly problems in web pages. *Proceedings of the International Conference on Software Engineering*, 2018.
95. Sterling G. "No, apps aren't winning. The mobile browser is.", 2015. URL `https://marketingland.com/morgan-stanley-no-apps-arent-winning-the-mobile-browser-is-144303`.
96. Van't Hof A, Jamjoom H, Nieh J, Williams D. Flux: Multi-surface computing in Android. *Proceedings of the European Conference on Computer Systems*, 2015.
97. Dougherty C. Google adds 'mobile friendliness' to its search criteria. *The New York Times*, 2015.
98. StackExchange Data Explorer. "Prevalence of tags related to responsive web design", 2017. URL `http://data.stackexchange.com/stackoverflow/query/edit/541879`.
99. Mahajan S, Halfond WGJ. WebSee: A tool for debugging HTML presentation failures. *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2015.
100. Mesbah A, van Deursen A, Lenselink S. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. *Transactions on the Web* 2012; **6**(1).
101. Ryou Y, Ryu S. Automatic detection of visibility faults by layout changes in HTML5 web pages. *Proceedings of the International Conference on Software Testing, Validation and Verification*, 2018.
102. Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing? *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014.
103. Fraser G, Staats M, McMinn P, Arcuri A, Padberg F. Does automated unit test generation really help software testers? A controlled empirical study. *Transactions on Software Engineering and Methodology* 2015; **24**(4).