Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages

Thomas A. Walsh*, Phil McMinn* and Gregory M. Kapfhammer†
*Department of Computer Science, University of Sheffield, UK
†Department of Computer Science, Allegheny College, USA

***Abstract*—Due to the exponential increase in the number of mobile devices being used to access the World Wide Web, it is crucial that web sites are functional and user-friendly across a wide range of web-enabled devices. This necessity has resulted in the introduction of responsive web design (RWD), which uses complex cascading style sheets (CSS) to fluidly modify a web site's appearance depending on the viewport width of the device in use. Although existing tools may support the testing of responsive web sites, they are time consuming and error-prone to use because they require manual screenshot inspection at specified viewport widths. Addressing these concerns, this paper presents a method that can automatically detect potential layout faults in responsively designed web sites. To experimentally evaluate this approach, we implemented it as a tool, called REDECHECK, and applied it to 5 real-world web sites that vary in both their approach to responsive design and their complexity. The experiments reveal that REDECHECK finds 91% of the inserted layout faults.**

I. INTRODUCTION

When the number of different types of mobile device accessing the Internet were few in number, web developers could create and maintain different versions of a web site intended for optimal viewing on a particular mobile device or a larger laptop or desktop screen. Yet, the recent explosion in the number of mobile devices – OpenSignal reports 18,796 unique devices running Android in the year 2014 [1] – renders this approach to testing infeasible because there are too many devices and associated viewport widths to consider. Since more than 1 in 4 Google searches now originate from a mobile device [2] and 67% of respondents in a recent Google Research survey said they were more likely to make a purchase from a web site if it was "mobile friendly" [3], developers and testers must account for the vast variety of mobile viewport widths.

Responsive web design (RWD) is a recent approach for tackling the proliferation of viewport widths that allows developers to make a single site that is intended to be used and viewed easily across a range of devices, from those with small screen sizes (i.e., mobile devices) to those with larger resolutions usually found on laptop and desktop computers [4]. The key idea behind RWD is that, instead of designing and maintaining separate versions of the web site for particular devices or a set of fixed resolutions [5], the developer creates a scheme to lay out a web page's elements differently depending on the user's particular viewport width. This scheme is referred to as the page's "responsive design". Using "fluid grids" and "flexible images", RWD principles insist that a web site should capably accommodate all possible viewport resolutions [4].

Using media queries, a feature of modern cascading style sheets (CSS) and the hypertext markup language (HTML), a responsively designed web site will detect the current viewport width and adjust element sizes and alignment for it, thus making the best use of the space available and ensuring that a user can easily interact with the page and read its content [4]. This is not the case when a page designed for a desktop is "shrunk" down to fit a smaller screen – something that mobile devices tend to do in the absence of a responsively designed site. While some may argue that RWD only impacts the aesthetics or layout of a web site, it is important to note that it is precisely the presence of these characteristics that increases a site's perceived usability [6] and the loyalty that users feel about it [7].

Even though RWD leads to mobile-friendly sites, recent experiments with web developers suggest that programming HTML and CSS is error-prone and difficult [8]. Additionally, 68,289 questions on the StackOverflow web site have been tagged with labels related to responsive web design [9], suggesting that real-world web developers and testers often struggle with the practice of responsive web design. While existing tools (e.g., WRAITH [10] and RESPONSIVEPX [11]) support some aspects of testing responsively designed web sites, they are time consuming for testers to use because they require manual page inspection at specific viewport widths.

Reacting to the importance of creating responsively designed sites and the challenge of implementing pages that are free of errors in layout, this paper presents a method that models the nuances of a responsive web site and automatically detects changes in it that result from CSS modifications. Implemented as a tool called "REDECHECK" (REsponsive DEsign CHECKer, pronounced "Ready Check"), the presented approach reports potential layout faults in responsively designed web sites.

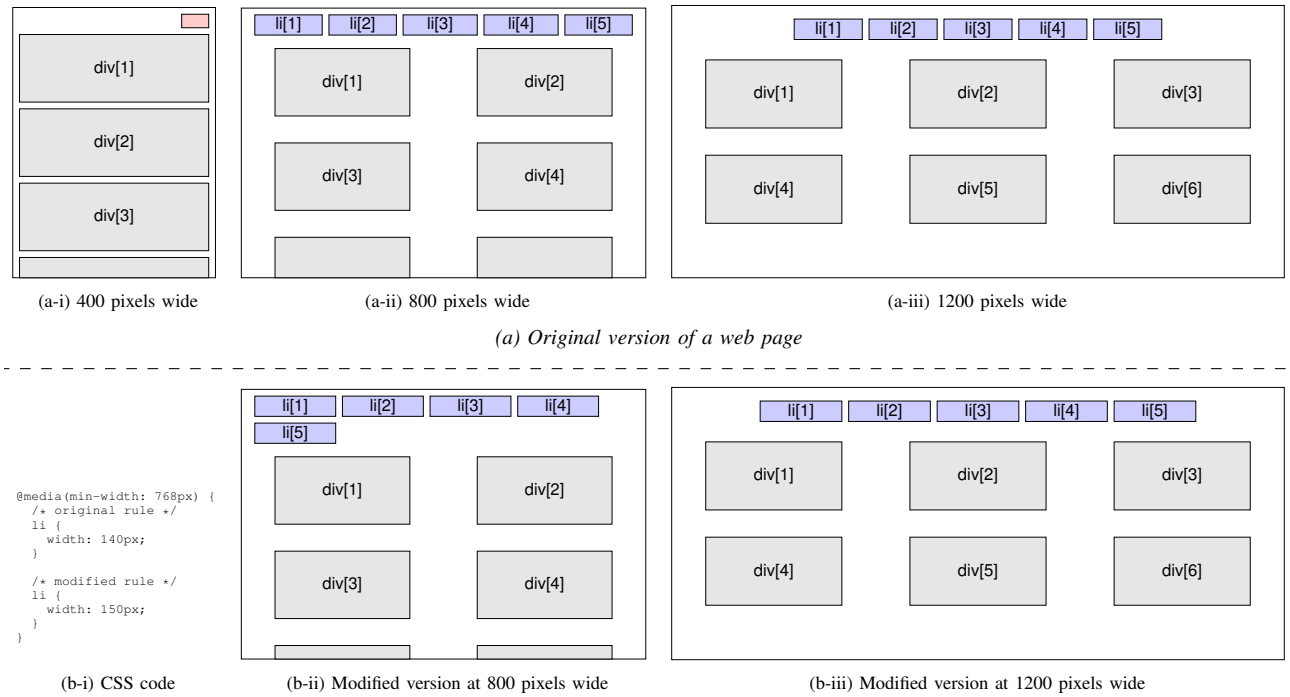
Using 5 web sites that adhere to RWD principles, were functional as of May 2015, and contained between 937 and 8,711 lines of CSS code, we experimentally evaluated the effectiveness of REDECHECK. The experiments demonstrate that our method can automatically find 91% of the inserted faults, while not erroneously reporting faults that do not exist. In summary, the key contributions of this paper are:

1) Implemented as a tool called REDECHECK, a method for automatically detecting layout faults in responsive sites;
2) Using real-world web sites, an initial empirical study demonstrating the benefits of the presented method.

II. BACKGROUND

A. Responsive Web Design

Illustrating some of the complexities inherent in the task of testing a responsive web site, Figure 1a shows a wireframe example of a responsive page at three viewport resolutions similar to those used by a smartphone, tablet, and a desktop computer. The page involves five menu items, implemented with five `li` HTML elements, denoted `li[1]` to `li[5]`; and six content panels, implemented with six `div` HTML elements, denoted `div[1]` to `div[6]`. At the smallest viewport width of 400 pixels, the menu items are hidden and only displayed when the user clicks a button in the top-right corner of the page, while the main content panels are stacked in a single column, requiring the user to scroll down the page to access



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

(a) Original version of a web page

(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 the result of a layout fault

Figure 1. A simple 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 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 incorrectly wraps to the next line.

all of the content. As the viewport widens to 800 pixels, the dropdown button disappears and the menu items are laid out in a single row in the header at the top of the web page, with the content panels rearranged in the main body of the page into a two abreast layout. As the viewport widens further to 1200 pixels the content panels again switch, this time to a three column layout, thus making optimal use of the space available.

Responsive web sites are developed using a combination of HTML and CSS that ultimately creates the document object model (DOM) representing a web site's structure. Using CSS rules, HTML elements can be set to dynamically proportion themselves according to the available space, thus being arranged in containers whose dimensions are fixed by the size of the current viewport. For instance, the following CSS rule ensures that two HTML `div` elements will horizontally take up half of a web page and, in addition, appear side by side:

```
div { width: 50%;}
```

Here, the `div` element is instructed to take up 50% of the width of its container, thereby adjusting to an exact width in pixels that depends on the dimensions of the device or web browser window in which the page is being viewed.

Additionally, special CSS rules, known as *CSS3 media queries* [4], can be used to switch on a specific set of CSS rules depending on the viewing device's constraints. In the following code example, an HTML `button` element should only appear at viewport widths below 500 pixels. The button activates a menu that displays a series of navigation links. At larger page widths, such as those on a desktop, there is enough space to always show all these links in a navigation bar, and as such a special button for activating them is not needed.

```
@media(max-width: 499px) {
  button { display: block; }
}
```

```
@media(min-width: 500px) {
  button { display: none; }
}
```

In this code segment, the 500 pixel viewport width is an instance of a *breakpoint*. In the context of responsive web design, a breakpoint is some threshold at which different CSS rules are switched on or off so that a page can be optimally laid out according to the specifics of the particular device being used to view it – in this example, the width of the screen.

B. Layout Faults

Creating a responsive design is a challenging process, since the style sheet must be carefully constructed such that, for different viewing constraints, the right set of CSS rules are switched on and work correctly for the right HTML elements, while other HTML elements correctly adjust in size. While many developers choose to work with a framework, such as Bootstrap [12] or Foundation [13], that provides CSS rules for an initial set of reusable and common layouts, these layouts only represent starting points and thus require customization to ensure a unique, yet cohesive, look and feel for the web site.

Figure 1b demonstrates how small changes to the CSS can result in undesirable effects not intended by the developer and often manifested at different viewport widths. We refer to such mistakes as *layout faults*. Suppose the developer is working with the web page design at a viewport width of 1200 pixels. Seeing the extra width available to the menu item elements, she widens them from 140 pixels to 150 pixels. The CSS change is shown by part b-i of the figure, with the change in appearance of the page at 1200 pixels wide shown by part b-iii. Since her current viewport is set to 1200 pixels wide, she assumes that the tweak had the intended effect and, in addition, introduced no inadvertent layout faults to the web page.

Yet, unless she explicitly checks the page (e.g., by manually adjusting the browser's width), she will be unaware of the impact the change has had at other viewport sizes. At a width of

800 pixels, shown in part b-ii, the menu items are now too wide to fit on one line, with the last element wrapping onto a second. As this issue continues to go unnoticed and more changes are made to the site, the developer may struggle to recall the initial CSS modification that caused the problem, thus requiring more effort to diagnose and fix it. This example underscores the challenge of tweaking a responsive design: each time a change is made, *all* resolutions need to be manually rechecked for unintentional side-effects. This is a time-consuming and error-prone process – in viewing a large number of resolutions and layouts, developers and testers can easily miss faults that ultimately make their way into the live web site.

The previous example shows how a developer is expected to engineer a web page layout that is aesthetically pleasing and easy to use across a range of different screen sizes, managing:

1. Visibility of HTML elements, since not all elements need to be shown at all viewport widths (e.g., the button shown in the top right of the 400 pixel view given in Figure 1a-i).

2. Varying width of HTML elements, since elements change width when the viewport width changes (e.g., the body panels `div[1]`–`div[6]` are 100% of their container's width when the viewport is 400 pixels wide, but then take up a smaller percentage of the width when viewed at larger viewports).

3. Changing relative alignment of HTML elements across different viewport sizes (e.g., the content panels `div[1]`–`div[6]` are stacked at the 400 pixel width view, but then fluidly realign alongside one another at larger screen widths, where they can comfortably fit in such a configuration).

Maintaining all of these aspects at once can be a challenging task. Yet, while the height of an HTML element may also fluidly change, this is usually a result of the browser automatically re-arranging the contained content in response to a change in viewport width. As such, the developer tends not to explicitly control this aspect of the layout through CSS. Unlike its width, the overall height of a web page is not generally fixed, under the reasonable expectation that the user will scroll to access content that does not fit into the height of the viewport. This assumption has led to the emergence of single-page, vertical-scrolling web sites recently becoming a mainstream trend [14].

As far as the authors are aware, there has been no previous work addressing the problem of automatically detecting layout faults following changes to responsive web pages. While various tools have been developed to allow developers to view their web site at the resolutions common to different devices, the viewport sizes tend to be fixed. As such, layout faults *between* the different resolutions presented may be missed. Regardless of whether or not a tester can pick a bespoke viewport, these tools require much manual effort and are thus error-prone.

III. Our Approach: The Responsive Layout Graph

Our approach models the layout of a responsive web page across a series of viewport widths, explicitly taking account of the key aspects of responsive layout detailed in Section II-B, namely the changing *visibility*, *width*, and *relative alignment* of web page elements. We refer to our model as the *Responsive Layout Graph* (RLG), as it builds upon concepts associated with the *alignment graph* that was proposed and developed by Choudhary et al. [15]. A change to a responsive web page causes a change to its RLG, which may either be a legitimate tweak to the page's design, or the indication of a layout fault needing to be brought to the attention of the developer.

A. Definitions and Example

Given a set of web page elements E_{RLG} for a web page W, we define three types of constraint (one for each trait of RWD layout) which together form a Responsive Layout Graph:

Definition 1 (Visibility Constraint): A visibility constraint vc, for some web page element $e \in E_{RLG}$, is a pair (x_1, x_2) that represents a range of viewport widths, inclusive of x_1 and x_2, for which e is present in the DOM of a web page.

Definition 2 (Width Constraint): A width constraint wc, for some page element $e \in E_{RLG}$, is a 4-tuple (x_1, x_2, m, c), that describes how the width w_e of the element e (measured in pixels) varies in relation to the width w_p of its direct container (i.e., its parent), using the values m and c derived from the line that fits the element's widths between x_1 and x_2 pixels according to the linear equation $w_e = (\frac{m}{100} \times w_p) + c$.

The final definition draws on concepts from the alignment graph defined by Choudhary et al. [15], in particular the notion of relationship types between web page elements, drawn from the set $\mathcal{T} = \{pc, s\}$. A *parent-child* relationship, denoted by the element pc, exists between a web page element and the element that directly contains it (i.e., its parent). Meanwhile, a *sibling* relationship, denoted by the element s, exists between any pair of elements that share the same parent. These relationships are described with a set of alignment attributes, a subset of $\mathcal{Q} = \{L, R, A, B, \ldots\}$, which characterize the relative alignment of the elements in the layout of the page. L, R, A, and B, for instance, represent an element being "left of", "right of", "above", and "below" its sibling, respectively (we refer the reader to reference [15] for the full definition of $\mathcal{Q}$ and a discussion of all the different alignment attribute types).

Definition 3 (Alignment Constraint): For a pair of web page elements $e_1 \in E_{RLG}$ and $e_2 \in E_{RLG}$, an alignment constraint ac is a 4-tuple $(x_1, x_2, t, \mathcal{P})$, where $t \in \mathcal{T}$ denotes a parent-child or sibling relationship type and $\mathcal{P} \in 2^{\mathcal{Q}}$ is a set of relative alignment attributes, both of which hold for e_1 and e_2 between the viewport widths of x_1 and x_2 pixels.

Note that, in all of these definitions, x_1 represents the lower bound of the constraint, so the condition $x_1 \leq x_2$ holds.

For a web page W it is possible to derive the respective sets of visibility, width, and alignment constraints VC, WC, and AC, which, along with the set of elements E_{RLG}, form an RLG. An example RLG over the range of viewport widths for the web page of Figure 1a is shown in Figure 2. Each web page element $e \in E_{RLG}$ corresponds to a node in the graph, with the node name displayed in bold face and visibility and width constraints shown above and below, respectively. Directed edges in the graph form relationships between HTML elements, labeled with alignment constraints. The variables w_{min} and w_{max} denote the minimum and maximum viewport widths that an RLG considers in the generation process. Many web page elements can be seen over all viewport widths, and so have the visibility constraint (w_{min}, w_{max}). One exception is the button element, which is only visible between w_{min} and 767 pixels. This RLG also shows examples of varying width constraints: for instance, the elements `div[1]`–`div[6]` have the width constraints $(w_{min}, 767, 100, 0)$ and $(768, w_{max}, 0, 300)$. That is, they fill 100% of the parent's width when the viewport is 767 pixels or less wide, appearing at a fixed width of 300 pixels when the viewport is 768 pixels or wider. These elements also exhibit changing alignments over these two ranges, appearing above one another when the viewport width is below 768 pixels wide and otherwise tiled across their containing `main` element.

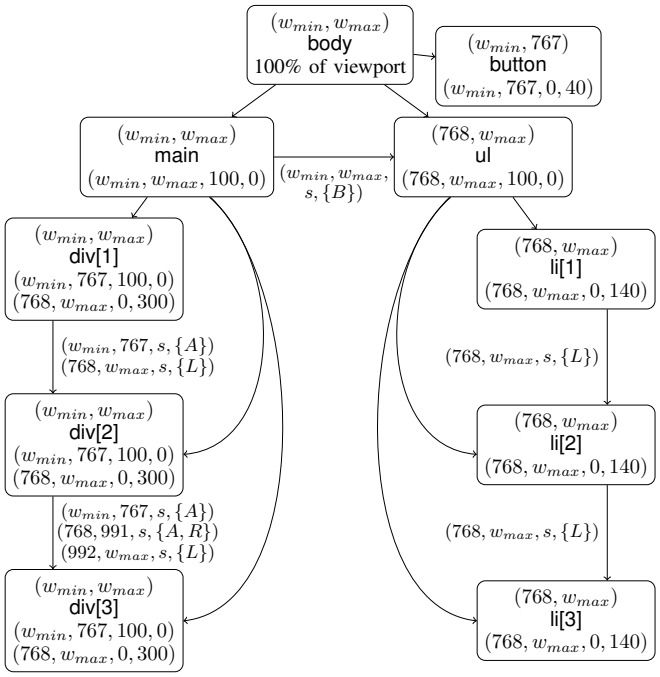


Figure 2. A fragment of the RLG for the web page of Figure 1a over a range of viewport widths between w_{min} and w_{max}. (For space reasons, we have omitted certain nodes, and all alignment constraints on parent-child edges.)

Formally, an RLG is defined by a 5-tuple $\mathcal{RLG} = (E_{RLG}, \mathcal{R}_{RLG}, \mathcal{F}_{VC}, \mathcal{F}_{WC}, \mathcal{F}_{AC})$, where E_{RLG} is the complete set of web page elements that are displayed for at least one viewport width between w_{min} and w_{max} and $\mathcal{R}_{RLG} \subseteq E_{RLG} \times E_{RLG}$ is the set of edges for pairs of nodes for which at least one alignment constraint exists.

Next, $\mathcal{F}_{VC} : E_{RLG} \rightarrow 2^{\mathcal{VC}}$ is a function mapping an element to a set of visibility constraints where $\forall e \in E_{RLG}, |\mathcal{F}_{VC}(e)| \geq 1$ (i.e., each element is visible for at least one viewport width).

In addition, $\mathcal{F}_{WC} : E_{RLG} \rightarrow 2^{\mathcal{WC}}$ is a function mapping an element to a set of width constraints, such that $\forall e \in E_{RLG}$ and $\forall wc_a = (x_{a1}, x_{a2}, m_a, c_a) \in \mathcal{F}_{WC(e)}$ and $\forall wc_b = (x_{b1}, x_{b2}, m_b, c_b) \in \mathcal{F}_{WC}(e)$, whenever $wc_a \neq wc_b$, the condition $x_{a1} \geq x_{b2} \vee x_{a2} \leq x_{b1}$ must hold. That is, for each page element, any viewport width should fall within the bounds of at most one of that element's set of width constraints.

Finally, $\mathcal{F}_{AC} : \mathcal{R}_{RLG} \rightarrow 2^{\mathcal{AC}}$ is a function that maps an edge to a set of alignment constraints, such that $\forall r \in \mathcal{R}_{RLG}$ and $\forall ac_a = (x_{a1}, x_{a2}, t_a, \mathcal{P}_a) \in \mathcal{F}_{AC}(r)$ and $\forall ac_b = (x_{b1}, x_{b2}, t_b, \mathcal{P}_b) \in \mathcal{F}_{AC}(r)$, if $ac_a \neq ac_b$, then $x_{a1} \geq x_{b2} \vee x_{a2} \leq x_{b1}$. That is, for a particular viewport width, there is at most one alignment constraint for a pair of web page elements.

B. Constructing the Responsive Layout Graph (RLG)

To generate the RLG, the approach first inspects the layout of the web site in question at a series of viewport widths, gaining an overall understanding of the responsive layout and recording its DOM at each viewport width. This represents the starting point for the RLG construction algorithm.

Using the DOMs gathered in the method's initial phase, the second phase of our approach extracts the three categories of constraints, as defined in Section III-A. First, the elements visible at each viewport width are analysed to determine the visibility constraints, paying special attention to elements appearing or disappearing at a particular viewport width. Next, the extraction of alignment constraints begins by creating an element hierarchy at each viewport width where an element's parent is the smallest element to completely contain it, as originally defined by Choudhary et al. [15]. Then, the relative alignment of related elements is generated through the use of DOM coordinates, and subsequently analysed to determine how the elements rearrange themselves to fit within the viewport. Finally, the method examines the width of each element and its parent element across all viewport widths, thereby producing a set of width constraints for that particular element.

To detect layout faults, our approach performs the previous two steps for both the modified and original versions of the web page under test. Next, it performs a simple pairwise comparison on the two generated RLG models, considering all three categories of constraints and reporting any differences.

Table I. RESPONSIVE WEB PAGES USED IN OUR EMPIRICAL STUDY

"LOC" refers to lines of code after a standard formatting process. A CSS block ("Blocks") is a series of individual declarations ("Decls") applied to a class of HTML elements.

Web site name (homepage URL studied)	**HTML** LOC	**HTML** DOM Nodes	**CSS** LOC	**CSS** Blocks	**CSS** Decls
Aftrnoon (*http://aftrnoon.com*)	205	112	2658	458	1000
Briefing (*http://briefi.ng*)	396	196	5814	1147	2173
Bootstrap (*http://getbootstrap.com*)	293	149	8711	1756	3199
Reserve (*http://reserve.com*)	230	125	6792	1375	2539
Responsive Process (*http://responsiveprocess.com*)	267	142	937	165	376

IV. EMPIRICAL EVALUATION

To assess the effectiveness of our approach we applied it to 5 responsively designed web sites that evidenced the following characteristics: (i) they were live and functional as of May 2015, (ii) their developers either used popular RWD frameworks such as Bootstrap [12] and Foundation [13] or hand-coded their own solutions, and (iii) they varied in their size and complexity, as shown by Table I. The primary aim of this experiment was to see how effective the generated RLGs are when used to detect different types of changes made to web pages. To answer this question, we developed and applied mutation operators that insert faults into a web site's CSS files and then checked whether our method could detect these changes.

A. Methodology

We implemented the approach described in Section III as a prototype tool, called "REDECHECK" (REsponsive DEsign CHECKer, pronounced "Ready Check"). It is written in the Java programming language, and where appropriate to avoid duplicating effort, re-uses methods from the alignment graph code of the X-PERT tool [16]. To conduct the experiment, we ran REDECHECK on an iMac running OS X Yosemite with 8GB of RAM and used Selenium (2.43.1) [17] to drive a recent version of Google Chrome (42.0.2311.152).

For each subject web page, we first downloaded all files required to render the page, before automatically generating a set of modified versions of the page (i.e., the mutants). We then ran REDECHECK on both the original version of the web page and each mutant and then classified each comparison result.

Through manipulation of the CSS rules found in the page's style sheet, these small changes affect the page's layout. For this experiment, we created two mutation operators to inject

Table II. MUTATION ANALYSIS RESULTS FOR REDECHECK

Web Page	TP	TN	FP	FN	Recall
Aftrnoon	14	5	0	1	93.3%
Briefing	19	1	0	0	100%
Bootstrap	13	1	0	6	68.4%
Reserve	18	1	0	1	94.7%
Responsive Process	16	4	0	0	100%

the changes. The first, which we refer to as the *breakpoint mutation operator*, targeted the values used in `min-width` and `max-width` declarations in the media queries. This operator adjusts the value by an amount selected at uniform random between ± 10 of the original, as in the following example:

```
@media (min-width: 640px)  →  @media (min-width: 647px)
```

The second operator, referred to as the *rule mutation operator*, targets the `width`, `margin`, and `padding` declarations of the style sheet, again adjusting values by ± 10 of the original at uniform random, as shown in the following example:

```
li { width: 75% }  →  li { width: 73% }
```

To inject mutants, REDECHECK analyzes the HTML of the web page to find all the CSS selectors and media queries linked to each element appearing on the web page. (This is so that the tool does not mutate parts of the style sheet that are irrelevant for the web page concerned.) REDECHECK first parses the web page's CSS files, and then extracts a list of potential mutation locations for each operator. To create a mutant, REDECHECK picks an operator with an even probability, then selects a CSS location at uniform random, and then applies the operator itself.

We used REDECHECK to create 20 mutant versions of each web page. We then used our tool to generate an RLG of the original page and the mutant version, using a sampling step size of 60 pixels in a viewport width range of 400–1300 pixels. We chose the step size of 60 pixels since, during REDECHECK's development and initial experimentation, we observed that it provided a good compromise between the quality of the model generated and the effort required to construct it (i.e., the number of viewport widths that needed to be sampled).

Next, REDECHECK compares the two DOMs of the pages and their respective RLGs and categorizes the result as either a *true positive*, *true negative*, *false positive*, or *false negative*. We categorize the result as "positive" if the RLGs are different, else it is "negative". Whether the positive/negative is true or false depends on whether the DOMs are actually identical or not. (Even though we purposely restrict mutation locations to CSS rules and queries actually linked to elements appearing on the page, equivalent mutants are still possible if, for example, the mutated rule is later overwritten by a subsequent declaration in the style sheet.) If the DOMs are different, a "positive" result is a *true positive*, otherwise it is a *false positive*. Conversely, a "negative" result is a *false negative* when the DOMs are different, otherwise, it is a *true negative*.

B. Threats to Validity

While it is possible that the implementation of REDECHECK may be incorrect, we managed this validity concern through extensive manual inspection – aided by automated methods provided by the Chrome Developer Tools – of responsive layout graphs generated for a wide variety of web sites. Thorough testing of the third-party tools used in REDECHECK, like an HTML and CSS parser and X-PERT [16], also mitigated the risk of defects compromising the empirical results. Moreover, all of the tedious, yet important, analyses – such as the DOM comparisons needed to classify the results into categories such as true positive and false negative – were performed using automated and well-tested procedures.

In addition, the empirical results could be compromised if our mutation operators do not introduce layout faults that are often evident in responsively designed web pages. While it is true that we did not insert all of the types of faults that could conceivably be made by web developers (e.g., we do not mutate the `float` property in the CSS files), our mutation operators are both meaningful and varied. By mutating the `min-width`, `max-width`, `width`, `padding`, and `margin` properties, we introduced 20 mutants into each of the 5 web sites, thus creating a total of 100 mutants that represent a wide variety of faults that real-world web developers might create.

C. Results

Table II presents the results of comparing the RLG models generated for the original web pages against the one derived from the mutated versions of those pages. While a number of equivalent mutants (i.e., true negatives) were generated for some of the subjects, in general, most mutants impacted the DOM of each page. The results show that the RLG supports the detection of a large proportion of the injected changes, with no false positives and only a small number of false negatives (i.e., 8 out of the 100 actual mutants). A careful manual analysis of the false negative results reveals that the DOM changes resulting from these CSS mutations were too subtle to cause a change in the generated feature sets in the alignment constraints – typically a very small shift in position of the element – resulting in no change to the final RLG. In conclusion, the results reveal that the RLG model effectively enables the detection of changes to the subject web pages, with no misleading false positives.

D. Discussion

During the implementation and experimental evaluation of REDECHECK, we observed some qualitative benefits associated with this new method for detecting potential layout faults in responsively designed web sites. First, it is important to stress that the automated modeling of a responsive web site obviates the need for a tester to pick viewport widths for inspection, only study the defaults commonly advocated by current testing tools, or accept the random selection of hopefully useful viewports. This feature of REDECHECK is especially beneficial since we noticed that responsive faults are often found at hard-to-predict viewport widths between the breakpoints standardly employed in RWD. We anticipate that testers will also profit from using REDECHECK as it creates a report both highlighting the set of viewport widths that have potential layout faults and giving valuable context in the form of the RLG and the DOMs. This is in contrast to other RWD testing tools that either do not pinpoint differences or only coarsely note a difference in a single viewport width. While it is true that REDECHECK may highlight layout differences that a web developer intended, we argue that the contextualized and informative nature of our tool's report partially mitigates this important concern.

It is worth noting again that REDECHECK could not detect certain subtle faults involving very small changes in the position of a web element. Yet, in our experience, humans viewing these mutated sites were also not able to notice these layout faults, thus suggesting that our tool's deficiency is indeed minimal.

Finally, a tester wishing to find layout failures across an entire responsive site would need to run REDECHECK on a page-by-page basis – a process that is, although potentially cumbersome, fully amenable to automation. With that said, we also found that REDECHECK can effectively test a responsive site implemented according to any viable strategy for introducing responsiveness (i.e., using bespoke CSS declarations or leveraging an existing framework such as Bootstrap [12] or Foundation [13]).

V. RELATED WORK

Since, to the best of our knowledge, this paper is the first to tackle the challenges associated with testing responsively designed web sites, this section briefly reviews the related work in the broader area of web testing. Choudhary et al. have developed several methods for cross-browser testing (XBT), the most recent of which is called X-PERT [15]. Developed by Dallmeier et al., WEBMATE is similar to X-PERT in that it also focuses on cross-browser testing [18]. Since the primary focus of both was single-resolution XBT they are, unlike REDECHECK, not tailored to responsive design.

Perhaps the most similar work to ours is that of Mahajan et al. [19], which aimed to automatically find differences between a graphic of how a web page should look and its final design. However, since this technique only operates at a single, fixed viewport width – and thus it would require a substantial number of graphics to fully test the fluidity of a responsive web page – it is not well suited to the challenges in this paper's problem domain. The WRAITH tool [10] similarly suffers from the same difficulties as Mahajan et al.'s method in that it too differences web page graphics on a per-resolution basis rather than considering the responsive layout of sites.

Currently, there are also several developer tools that address some of the concerns associated with testing responsive web sites. Multi-screenshot tools, such as Kersley's [20], allow a tester to view a site at several common viewport widths while viewport resizers like RESPONSIVEPX [11], support the fine-tuning of the viewport size for testing a specific device. While these may be useful, they are not a substitute for an automated method like the one provided by REDECHECK.

VI. CONCLUSION AND FUTURE WORK

Even though there are clear benefits to making responsive web sites [21], most Fortune 500 companies do not have a mobile-friendly site [22] – an alarming trend due, at least in part we believe, to the inherent difficulties of both developing and satisfactorily testing responsive web sites. Since recent experiments show that programming HTML and CSS is error-prone and difficult [8], we think that developers and testers will welcome tools that automatically test responsive sites.

As such, this paper presents an automated method for detecting potential faults in the layout of responsively designed pages, thereby tackling the challenges of implementing and testing web sites that adhere to RWD principles, To experimentally evaluate this new approach, we implemented it as a tool called REDECHECK and applied it to 5 responsive web sites that were live as of May 2015. In support of assessing whether or not REDECHECK could detect small changes in a web page's layout, we designed breakpoint and rule mutation operators to manipulate the declarations found in a site's CSS style sheets.

The core focus of our future work is to filter the results generated by REDECHECK so that only unintentional changes, such as the one presented in Section II-B, are reported. In addition, to further evaluate the effectiveness of our approach we will conduct experiments with both a greater variety of responsively designed sites and more mutation operators. Finally, we plan to extend REDECHECK with new features to support more of the complexities of real-world pages. For instance, we will add the capability to handle JavaScript code that responds to user interaction and enhance our model of a site's responsive behavior so that it incorporates characteristics of web elements (e.g., the font and color of headers). While the second of these two features will expand the scope of REDECHECK beyond of the important domain of responsive layout faults, we judge that the combination of the suggested future work with the sophisticated method presented in this paper will yield a useful tool that effectively aids developers and testers as they create high-quality responsive web sites.

REFERENCES

[1] A. Van't Hof, H. Jamjoom, J. Nieh, and D. Williams, "Flux: Multi-surface computing in Android," in *Proc. of the 10th EuroSys*, 2015.

[2] Statista, "Statistics portal," accessed: 2015-27-02. [Online]. Available: http://goo.gl/19fZcq

[3] R. Hof, "Google research: No mobile site = lost customers," *Forbes*, 2012. [Online]. Available: http://goo.gl/khJ09v

[4] E. Marcotte, *Responsive Web Design*. A Book Apart, 2014.

[5] A. Gustafson, *Adaptive Web Design*. Easy Readers LLC, 2011.

[6] J. Hartmann, A. Sutcliffe, and A. De Angeli, "Investigating attractiveness in Web user interfaces," in *Proc. of the 25th SIGCHI*, 2007.

[7] D. Cyr, M. Head, and A. Ivanov, "Design aesthetics leading to m-loyalty in mobile commerce," *Inf. Manag.*, vol. 43, no. 8, 2006.

[8] T. H. Park, B. Dorn, and A. Forte, "An analysis of HTML and CSS syntax errors in a Web development course," *Trans. Comput. Educ.*, vol. 15, no. 1, 2015.

[9] StackExchange Data Explorer, "Prevalence of Bootstrap, Foundation, and Responsive as tags," accessed: 2015-06-05. [Online]. Available: http://goo.gl/sEknaT

[10] BBC News, "Wraith," accessed: 2015-02-02. [Online]. Available: https://github.com/BBC-News/wraith

[11] R. Sharp, "ResponsivePX: Find that tricky breakpoint," accessed: 2015-02-02. [Online]. Available: http://responsivepx.com/

[12] M. Otto and J. Thornton, "Bootstrap: Mobile-first and responsive front-end framework," accessed: 2015-04-05. [Online]. Available: http://getbootstrap.com/

[13] ZURB Corporation, "Foundation: Responsive front-end framework," accessed: 2015-04-05. [Online]. Available: http://foundation.zurb.com/

[14] C. Bloq, "5 of this year's vertical scrolling trends," accessed: 2015-31-07. [Online]. Available: http://www.creativebloq.com/web-design/vertical-scrolling-trends-2015-121413698

[15] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-PERT: Accurate identification of cross-browser issues in Web applications," in *Proc. of the 35th ICSE*, 2013.

[16] ——, "X-PERT: A web application testing tool for cross-browser inconsistency detection," in *Proc. of ISSTA*, 2014.

[17] "Selenium: Web browser automation," accessed: 2015-02-02. [Online]. Available: http://www.seleniumhq.org/

[18] V. Dallmeier, B. Pohl, M. Burger, M. Mirold, and A. Zeller, "WebMate: Web application test generation in the real world," in *Proc. of ICSTW*, 2014.

[19] S. Mahajan and W. G. J. Halfond, "Finding HTML presentation failures using image comparison techniques," in *Proc. of the 29th ASE*, 2014.

[20] M. Kersley, "Responsive design testing," accessed: 2015-02-02. [Online]. Available: http://mattkersley.com/responsive/

[21] C. Dougherty, "Google adds 'mobile friendliness' to its search criteria," *The New York Times*, 2015.

[22] T. McCorkindale and M. Morgoch, "An analysis of the mobile readiness and dialogic principles on Fortune 500 mobile websites," *Pub. Rel. Rev.*, vol. 39, no. 3, 2013.