ReDeCheck: An Automatic Layout Failure Checking Tool for Responsively Designed Web Pages

Thomas A. Walsh
University of Sheffield, UK

Gregory M. Kapfhammer
Allegheny College, USA

Phil McMinn
University of Sheffield, UK

ABSTRACT

Since people frequently access websites with a wide variety of devices (e.g., mobile phones, laptops, and desktops), developers need frameworks and tools for creating layouts that are useful at many viewport widths. While responsive web design (RWD) principles and frameworks facilitate the development of such sites, there is a lack of tools supporting the detection of failures in their layout. Since the quality assurance process for responsively designed websites is often manual, time-consuming, and error-prone, this paper presents ReDeCheck, an automated layout checking tool that alerts developers to both potential unintended regressions in responsive layout and common types of layout failure. In addition to summarizing ReDeCheck's benefits, this paper explores two different usage scenarios for this tool that is publicly available on GitHub.

CCS CONCEPTS

•**Software and its engineering → Software defect analysis;**

KEYWORDS

Responsive web design, presentation failures, layout failures.

ACM Reference format:
Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 2017. ReDeCheck: An Automatic Layout Failure Checking Tool for Responsively Designed Web Pages. In *Proceedings of International Symposium on Software Testing and Analysis, California, USA, July 10–14, 2017 (ISSTA 2017),* 5 pages.
DOI: 10.1145/3092703.3098221

1 INTRODUCTION

While a recent study by the Pew Research Center found that 95% of Americans currently own a cellphone of some kind, the same report also revealed that 80% of US adults use a desktop or a laptop and 50% have a tablet [1]. Results like these, which mirror those of several other regions of the world [15], mean that web developers must design websites that have aesthetically pleasing and functional layouts on a wide variety of devices. Providing a quality experience to end users can lead to increased revenue [7] and brand loyalty [4], along with enhanced search engine rankings [5]. Conversely, a poor page layout could lead to negative user experiences, particularly on mobile devices, causing the majority of users to simply leave the page [7], leading to the potential for lost revenue [9].

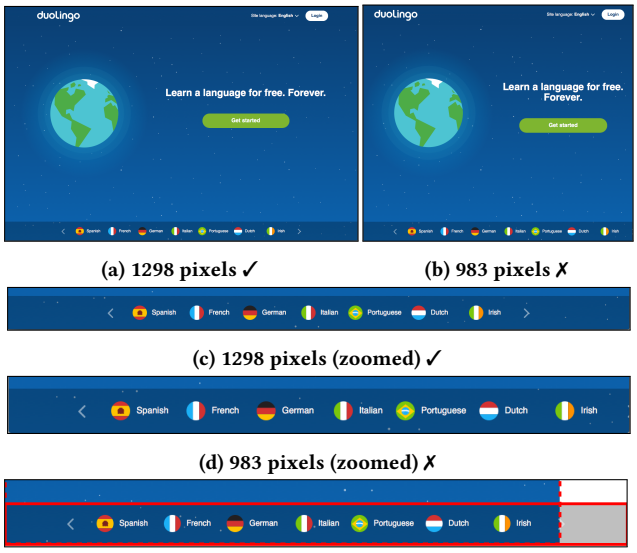
Envisioned Users. Aiming to capitalize on the numerous benefits attributed to supporting users on a wide variety of devices, many web developers have adopted responsive web design (RWD). This helps them in creating web pages that give an enhanced user experience regardless of the device being used to view the page [11]. Properly designed RWD-based sites achieve this by "responding" to the user's device, resizing and rearranging the content to best fit the screen according to three core concepts: fluid grids, flexible media, and media queries [11]. Fluid grids and flexible media use relative sizing to scale web elements to the particular device, thus ensuring that they are rendered within their containers. Media queries allow developers to apply different styling rules depending on the characteristics of the current device — most commonly the width of the device or browser, known as the *viewport width*. Many front-end frameworks, such as Bootstrap [14] and Foundation [13], provide RWD features, thereby giving developers the building blocks with which they can create web pages that are responsive.

Challenges Addressed. Given the complex interplay between web elements and layout rules evident in responsive web pages, implementing them can be difficult, leading to developers introducing undesirable visual effects into their pages. Figure 1 presents an example of a *responsive layout failure* (RLF) in a production web page: at wide viewport widths (i.e., parts (a) and (c)) the carousel of language options fits easily within the page, whereas at narrower widths (i.e., parts (b) and (d)) it protrudes outside of the viewport, making the right scroll arrow unclickable and negatively impacting the page's functionality. Since RLFs often occur intermittently at unpredictable and occasionally small ranges of viewport widths, detecting them is challenging. Moreover, this detection process is often a manual one in which a developer "spotchecks" a web page at certain viewport widths (e.g., 320 pixels for an iPhone, 768 pixels for tablets, and 1024 pixels for laptops), an undertaking that is labor intensive and error prone as certain RLFs can be easy to overlook. Since it is challenging to spotcheck a page at every viewport width, developers may miss layout failures that go on to production.

The Tool. The ReDeCheck tool (Responsive Design Checker, pronounced "Ready Check"), presented in this paper, helps web developers implement responsively designed web pages that exhibit correct layout at different viewport widths. ReDeCheck features two distinct modes for automatically checking a responsive web page's layout: *regression checking* and *common failure detection*. Regression checking compares the responsive layout of the latest and previous versions of a web page and reports to the user a list of potential regression layout issues [19]. Common failure detection analyzes the layout of the latest version of a web page and checks for a number of common types of RLF [18]. ReDeCheck is well documented and currently available on GitHub, under an open-source license, for evaluation and extension [20]. It is platform independent and compatible with a range of frequently used web browsers.

Evaluation Results. We have conducted experiments to evaluate both of ReDeCheck's modes. The results show that the tool can accurately detect the majority of potential layout issues between different versions of a page when in regression checking mode [19]. When targeting specific layout failure types, the tool detected failures in popular websites such as *Duolingo* and *Consumer Reports* —

ISSTA 2017, California, USA
2017. 978-1-4503-5076-1/17/07…$15.00
DOI: 10.1145/3092703.3098221



(e) ReDeCheck failure report

Figure 1: Screenshots of *Duolingo*, where a carousel of languages is correctly centered and with arrows on each side (parts (a) and (c)), before the carousel protrudes outside the viewport as the width narrows, obscuring the right-hand arrow (parts (b) and (d)). Finally, part (e) shows a report, produced by the ReDeCheck tool, that highlights the failure, using the dashed and solid red boxes, to the developer.

in addition to outperforming tool-supported manual spotchecking methods. The tool is also fast, running in less than two minutes on most of the studied pages, making it feasible for developers to integrate it into their responsive web design toolbox [18]. Finally, this paper's case studies show how a developer can use ReDeCheck's failure reports, like the excerpted one in Figure 1(e), to diagnose and repair mistakes in a web page's responsive layout.

2 THE REDECHECK TOOL

ReDeCheck is an automated layout checking tool, providing two different forms of developer support. In *regression checking* mode it compares the responsive layouts of two versions of a web page (i.e., before and after a developer's code modification), reporting a list of layout differences that are potential issues of which the developer may be unaware [19]. The *common failure detection* mode takes the latest version of a web page as input and analyzes its responsive layout, checking for different types of common layout failures that stem from the improper application of RWD principles and were identified through analysis of RWD in practice [18]. These types are, namely, *element collision* (two elements overlapping), *element protrusion* and *viewport protrusion* (elements overflowing their container or the viewport, respectively), *small-range layouts* (layouts only applied for a very small number of viewport widths) and *incorrectly wrapping elements* (elements forced onto a new row due to a lack of horizontal space). Figure 2 shows ReDeCheck's structure: to the left of the dashed vertical line, the modules support regression checking, to the right, common failure checking.

At the core of the ReDeCheck tool is a representation of the dynamic layout of a web page called the *responsive layout graph* (RLG), which models both the changing visibility and alignment

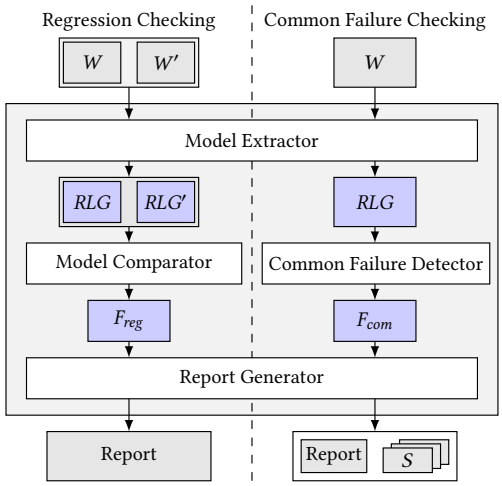


Figure 2: The high-level structure of the ReDeCheck tool. To the left of the dashed vertical line, the modules support regression checking, to the right, common failure checking.

of HTML elements as the viewport expands and contracts. For example, two elements could be rendered one above the other at narrow viewport widths, but side by side on wider viewports with increased horizontal space. Furthermore, some elements may only be visible at certain viewport widths. The *model extractor* module is responsible for extracting the RLG of a specified web page, rendering it in a browser and inspecting its layout at a series of viewport widths via the document object model (DOM).

ReDeCheck picks sample widths by applying *uniform sampling* (e.g., at 60 pixel intervals) and *boundary sampling*, which involves searching the page's stylesheet for the breakpoints at which the layout is intended to change. This two-part combined approach thereby obtains as representative a sample as possible. The layouts at the chosen widths are then analyzed sequentially to model the responsive layout. If difference(s) are found between two consecutive layouts, the tool conducts a binary search to find the exact viewport width at which the layout changes, repeating the process until all layouts are analyzed. ReDeCheck uses Selenium [2] to drive a browser and render the page; Selenium's support of many browsers enables ReDeCheck to integrate into a developer's environment.

In regression checking mode with input web pages W and W', ReDeCheck begins by extracting RLGs for both to produce RLG and RLG'. The *model comparator* module then compares RLG and RLG' using a pairwise matching approach to produce a set of differences that could be regression failures, F_{reg}. Before outputting a report, the *report generator* module analyses F_{reg} to determine the nature of the failures and the viewport widths at which they are visible.

When detecting common layout failures in a single web page W, only one model, RLG, is extracted and analysed by the *common failure detector* module to check for the presence of the different types of responsive layout failures. The report generator then produces a textual report describing the detected failures, F_{com} (i.e., the type of failure, the elements involved, and the range of viewport widths at which the failure manifests) accompanied by a set of screenshots, S, showing each detected failure with the offending HTML elements highlighted. A developer can then inspect each of the individual failure reports. If the report shows an evident RLF

Table 1: Empirical evaluation results for ReDeCheck.

(a) Regression Checking

	TP	TN	FP	FN	Recall
Total	80	12	0	8	90.9%

(b) Common Failure Checking

	TP	FP	NOI	Viewports	RLFs
Total	196	48	83	137	33

(i.e., a *true positive* (TP)) then the developer can debug and fix the issue. If no failure is visible (i.e., a *false positive* (FP)) then no action is required. Finally, if there is no visual failure but — at the level of DOM coordinates — the elements are, for example, overlapping, the developer may want to investigate further as these could be evidence of an underlying structural defect that may manifest visually in the future. We refer to these as *non-observable issues* (NOIs).

ReDeCheck is publicly available as an open-source tool on GitHub [20]; there is extensive documentation on how to install and run the tool and interpret its results. Each of the tool's modules are documented, allowing researchers and developers to understand its design and implementation in Java. The tool also contains a suite of JUnit tests. Finally, ReDeCheck is extensible, supporting the addition of new failure checkers in the common failure detector.

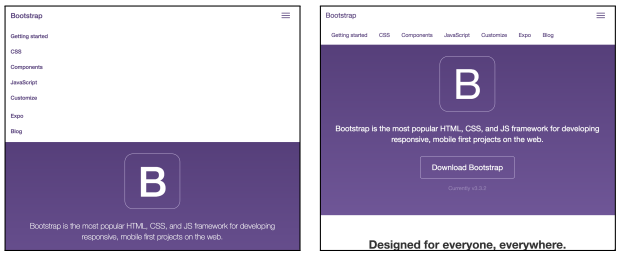
3 APPLYING THE TOOL

Using a variety of pages that were in production use, both of ReDeCheck's modes have been subjected to empirical study to evaluate their effectiveness. This section summarises the methodology and findings of these studies, additionally presenting two case studies showing how ReDeCheck detects and reports layout failures.

3.1 Experimental Studies

Table 1a furnishes the results from the evaluation of ReDeCheck's regression checking mode. Using simple code modification operators that change, for instance, the width, margin or padding of elements, we used a tool to create 20 incrementally modified versions of five responsive pages, for a total of 100 modified web pages. This empirical study revealed that ReDeCheck was capable of detecting a majority of such layout changes, achieving 91% recall [19]. While 15 false negatives were observed, the relevant code modifications minimally changed the web page in a way that did not influence the RLG created by ReDeCheck. As such, it is unlikely that these changes represented failures, limiting any shortcomings of ReDeCheck. Overall, these results indicate that the tool can detect subtle variations in web page layout and notify developers of potential layout issues. For more details, please read reference [19].

To investigate the prevalence of the five common types of RLFs in real-world web pages and evaluate ReDeCheck's ability to detect them, we ran the tool on a corpus of 26 randomly collected web pages of varying complexity and domain. ReDeCheck found RLFs of all five common types. Well over half of the web pages, including well-known ones such as *Duolingo* and *Consumer Reports*, contained RLFs [18]. Since these pages were all in production use and, we surmise, already subject to extensive testing, this result emphasises the importance and real-world relevance of the presented tool, further underscoring the benefits of using ReDeCheck's automated layout checkers. This empirical study also revealed that ReDeCheck outperformed several spotchecking approaches. For instance, a manual spotcheck missed *Duolingo*'s layout failure, as highlighted in Figure 1, illustrating the error-prone nature of current industrial quality assurance practices for responsive pages.




```
Differing bounds for /NAV/UL/LI[1] - /NAV/UL/LI[2] (leftOf, topAlign, bottomAlign):
Original : 768 -> 1300
Modified : 766 -> 1300
```

Figure 3: The original version of getbootstrap.com (top-left), the incrementally modified version (top-right), and a snippet of the report produced by ReDeCheck (bottom).

While the majority of the generated reports were true positives, Table 1b reveals that ReDeCheck produced some FP and NOI reports. For instance, when the tool incorrectly classified three elements in one page as a "row", it identified alignment shifts in these elements as a wrapping failure when they were not. ReDeCheck reported NOIs when it detected significant changes in the DOM that were not visible in the page's layout. For example, the tool reported a layout failure for elements that protruded invisibly out of their container. While NOIs are not failures per-se, it is appropriate for ReDeCheck to report them as they may manifest as layout failures in future versions of the web page. Moreover, instead of studying each of the generated reports, a developer using ReDeCheck could visit every distinct viewport range highlighted by the tool. Since Table 1b shows that the tool reported 137 different viewport ranges for the 33 distinct RLFs, a developer would only have to study, on average, 4.2 viewport widths to find each distinct RLF. Full evaluation details and further RLF examples can be found in reference [18].

3.2 Case Studies

Regression checking. Figure 3 presents an example from one of the web pages in our first empirical study, getbootstrap.com. In the original version shown on the top-left, the navigation links are rendered in a vertical dropdown list at narrow viewport widths before being displayed as a horizontal navigation bar at viewport widths of 768 pixels and wider. However, following a small change to the style rules of the web page, for a couple of viewport widths (i.e., 766–767 pixels) the web page renders the navigation links in a row rather than a column, defeating the point of implementing a drop-down navigation list and producing a far less professional aesthetic, as shown in the top-right screenshot of this figure.

Since the failure is only observable at a couple of viewport widths, it could lie dormant. For example, if developers only checked the layout at 768 pixels — as many do since this width is advocated by numerous RWD tools (e.g., [8, 21]) — they would be unaware that the web page now contained this failure. The report snippet shown at the bottom of this figure inducates that ReDeCheck detects this potential issue and alerts the developer to its presence. It also provides useful diagnostic information, such as the viewport widths at which the page's layout has changed and the elements involved, allowing the developer to ascertain if a failure is actually evident.

In the case of the web page in Figure 3, the textual report would alert the developer to a change in the alignment constraint between

`LI[1]` and `LI[2]` (i.e., the "Getting Started" and "CSS" links in the header): `LI[1]` is aligned to the left of `LI[2]` and they are both top and bottom aligned for a different range of viewport widths. Originally, the layout is evident for viewport widths of 768 pixels and higher (denoted by "`768 -> 1300`"). Following the code modification, the layout instead begins at 766 pixels, motivating a developer to change the page's style rules to ensure that the navigation links are correctly rendered in a single column as was intended, restoring a professional look and feel to the web site. Since the current failure reports include DOM references, and therefore may be hard to interpret, further engineering is required to make them more useful for developers, which we plan to undertake as part of future work.

Common failure detection. Figure 1(e) gives a failure report screenshot of *Duolingo* produced by ReDeCheck during our second evaluation of the tool. The solid red box highlights the faulty carousel as it protrudes outside of the viewport window (denoted by a dashed red line), making the carousel difficult to use as the right-hand arrow is obscured from view. This is an example of a web page providing a good browsing experience on smartphones and laptops, but neglecting devices with viewport widths in between, such as tablets. Violating a core principle of RWD — that a page should provide a suitable experience regardless of the viewport width at which it is viewed — this failure reduces functionality.

By detecting this issue and highlighting the offending elements with coloured boxes, ReDeCheck clearly alerts the developer to both the presence and nature of the problem. In this instance, the developer would investigate the carousel's style rules, modifying them so that this element scales down its width in accordance with the viewport, thereby ensuring the arrow is not obscured at the highlighted width. Without ReDeCheck's assistance a developer would not only have to inspect the web page at one of the faulty viewport widths, but also manually check the layout with sufficient care so as to notice the failure — neither of which is guaranteed.

4 RELATED WORK

To the best of our knowledge, ReDeCheck is the first tool that automatically checks a responsive web page; previous research prototypes for web testing have targeted orthogonal problems. X-Pert [16] and gwali [3] used the same concept of relative layout to detect cross-browser inconsistencies (XBIs) and internationalization presentation failures (IPFs), respectively. WebSee [10] and Wraith [12] adopt a different approach, using image comparison to report presentation failures to the user. Cornipickle [6] supports user-defined layout constraints describing how the web page should look, alerting the user if any of the constraints are not met. Yet, none of these tools specifically address the challenges associated with automated responsive web page checking. Unlike ReDeCheck's failure checking mode, they also require an oracle.

Developers also have access to many tools for testing responsively designed web pages. Most modern browsers come with a "responsive mode" that renders the current page at the developer's chosen resolution. Multi-screenshot tools (e.g., [8]) give developers a simple overview of a page's responsive behaviour by rendering it at a series of common viewport widths, while others (e.g., [21]) provide the option to tune the display to a specific resolution, enabling fine-grained checking. While these tools are useful, they still require the developer to study the page at each resolution. Finally, Galen [17] is a responsive layout testing framework with which a developer describes the intended layout of a page, verifying it at chosen viewport resolution(s). Unlike ReDeCheck, this tool requires the developer to write tests with one or more oracles.

5 CONCLUSIONS AND FUTURE WORK

The paper presents ReDeCheck, a tool that uses automatic layout checking to improve the quality assurance process for responsively designed web pages. Supporting two checking modes, ReDeCheck can detect both potential unintended regressions in layout and a set of common layout failures often observed in responsive pages. In addition to summarizing recent experimental studies of ReDeCheck, the paper presents case studies that illustrate how the automatically constructed failure reports help developers understand and repair layout problems. Platform independent and compatible with a range of web browsers (e.g., Chrome, Firefox, Safari, and PhantomJS), ReDeCheck is well documented and currently available on GitHub under an open-source license [20], thus supporting use by practitioners and further study by researchers.

In future work, we plan to improve the report generation module so that it outputs an interactive and easy-to-interpret report in which the most "important" RLFs are presented first. We also intend to enable ReDeCheck to handle dynamic pages that use JavaScript and to check the layout of entire sites, rather than just individual pages. We will also extend the regression checking mode so that it allows developers to check a page's layout in two different browsers, thereby enabling the detection of responsive XBIs. Finally, to ensure that developers adopt and integrate our tool into their development suites, we plan to create browser plugins for ReDeCheck.

REFERENCES

[1] Pew research center: Mobile fact sheet. http://www.pewinternet.org/fact-sheet/mobile/.
[2] Selenium: Web browser automation http://www.seleniumhq.org/.
[3] A. Alameer, S. Mahajan, and W. G. Halfond. Detecting and localizing internationalization presentation failures in web applications. In *Proc. of 9th ICST*.
[4] D. Cyr, M. Head, and A. Ivanov. Design aesthetics leading to m-loyalty in mobile commerce. *Inf. Manag.*, 43(8), 2006.
[5] C. Dougherty. Google adds 'mobile friendliness' to its search criteria. *The New York Times*, 2015.
[6] S. Hallé, N. Bergeron, F. Guerin, G. L. Breton, and O. Beroual. Declarative layout constraints for testing web applications. *J. Log. Algebr. Meth. Program.*, 85, 2016.
[7] R. Hof. Google research: No mobile site = lost customers. *Forbes*, 2012.
[8] M. Kersley. Responsive design testing http://mattkersley.com/responsive/.
[9] W. Li, M. J. Harrold, and C. Görg. Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors. In *Proc. of 25th ASE*, 2010.
[10] S. Mahajan and W. G. Halfond. WebSee: A tool for debugging HTML presentation failures. In *Proc. of the 8th ICST*, 2015.
[11] E. Marcotte. *Responsive Web Design.* A Book Apart, 2014.
[12] BBC News. Wraith https://github.com/bbc-news/wraith/.
[13] ZURB Corporation. Foundation http://foundation.zurb.com/.
[14] M. Otto and J. Thornton. Bootstrap: Mobile-first and responsive front-end framework http://getbootstrap.com/.
[15] J. Poushter. Smartphone ownership and Internet usage continues to climb in emerging economies. In *Pew Research Cener – Global Attitudes and Trends*, 2016.
[16] 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.
[17] I. Shubin. Galen framework http://galenframework.com/.
[18] T. A. Walsh, G. M. Kapfhammer, and P. McMinn. Automated layout detection for responsive web pages without an explicit oracle. In *Proc. of ISSTA*, 2017.
[19] T. A. Walsh, P. McMinn, and G. M. Kapfhammer. Automatic detection of potential layout faults following changes to responsive web pages. In *Proc. of 30th ASE*.
[20] T. A. Walsh, P. McMinn, and G. M. Kapfhammer. ReDeCheck. https://github.com/redecheck/redecheck/.
[21] M. Wassermann. Viewport resizer http://lab.maltewassermann.com/viewport-resizer/.