Automated Repair of Responsive Web Page Layouts

Ibrahim Althomali
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Phil McMinn
University of Sheffield

Abstract—**Responsive Web Design (RWD) is a strategy that allows developers to create webpages that adjust their layout according to available screen size. Since modern web applications must format correctly on the small displays of mobile devices up to the large displays on desktop computers, and given this dramatic difference in screen space, Responsive Layout Failures (RLFs) — visual discrepancies that are only apparent at certain screen sizes — can easily creep into live production webpages. These can include, for example, HTML elements protruding off the edge of the page or into one another as layout space becomes scarce. This leaves webpages looking unprofessional at best and non-functional at worst. This paper presents a technique for repairing RLFs, implemented into a tool called LAYOUT DR. After detecting an RLF, LAYOUT DR harvests layouts from the page's responsive design that are closest to the point of failure, but where the RLF does not occur. It then transforms these layouts so that they can be transplanted over the failure, effectively "hiding" the original RLF from the end user. We evaluated LAYOUT DR on 19 subjects, containing 55 RLFs in total. LAYOUT DR could find a suitable fix for each of them. When we conducted a human study of the repairs, 92% of the participants preferred the repaired version of the page compared to the original containing the RLF.**

I. INTRODUCTION

Both the smartphone era and the surging popularity of tablets and other web-enabled devices with touchscreens has resulted in a serious challenge for developers and designers: ensuring that their webpages format correctly on different, arbitrary screen sizes. Since developing multiple static webpage layouts for a small number of specific devices is no longer feasible, web developers use *Responsive Web Design* (RWD) strategies [24] to formulate a webpage layout that dynamically adjusts to the amount of space available to render a page, known as the *viewport*. However, since responsive web designs have to accommodate a very broad range of viewport sizes — from small mobile devices up to large desktop displays — ensuring that a webpage adapts to all screen sizes and always formats correctly is a difficult problem. *Responsive Layout Failures* (RLFs), or visual discrepancies in the layout of a responsive webpage, can creep into designs and may not be noticed until webpages go live, often because they occur at a few viewport sizes out of a very wide range. RLFs occur because there is not enough space for aspects of the webpage's design to display correctly at a particular viewport size, or range of sizes. RLFs include, for example, HTML elements being rendered off the edge of the screen or crashing into one another and overwriting each other's content.

Repairing RLFs in responsive pages is a non-trivial problem, and may involve one or two and up to tens or even hundreds of HTML elements and CSS properties. Some RLFs may be fixed by the judicious adjustment of only a few CSS properties while others may require a significant re-working of the page's design before they are completely resolved [27]. This repair process is complex and potentially time-consuming for developers, for whom automated assistance is needed. While there are automated repair approaches for non-responsive types of presentational failures in webpages, namely cross-browser issues [20], international presentation failures [10], [21], and mobile-friendly issues [19], there has been no work (to the best of our knowledge) on automatically repairing RLFs.

There has, however, been work on detecting problems in responsive pages. One such tool, called REDECHECK [28], provides several algorithms for identifying different types of common RLF present in a responsively designed webpage. Based on this initial detection process, we propose a technique for automatically repairing the RLFs discovered. We implemented this technique into a tool called LAYOUT DR (responsive *layout* failure *d*etection and *r*epair, pronounced "Layout Doctor"). Given an RLF and the range of viewports at which it occurs, LAYOUT DR sources two layouts either side of the range not affected by the RLF. It then scales and transforms them in the failure range to produce two potential "hotfixes" that the developer can choose from to provide a repair for the page. This hotfix addresses the symptoms of the RLF, giving developers time to accurately diagnose the underlying problem with the HTML and CSS and to then develop a durable solution to the problem. LAYOUT DR differs from the aforementioned approaches to other, non-responsive types of webpage failure repair, in that it uses layouts from other viewports of the page's own design, as opposed to a reference "correct" version of the webpage [10], [20], a formal layout specification [17], or guidance from usability metrics [21].

We empirically evaluated LAYOUT DR and showed that it could automatically and reliably created repairs for a wide range of RLFs found in real responsively designed webpages. A study with human participants revealed that, in 92% of the cases, participants preferred the repairs generated by LAYOUT DR as opposed to the original, buggy version of the page, suggesting that the hotfixes rarely degrade page layout. Furthermore, the presented tool never takes more than 40 seconds to produce a repair for any of the studied webpages. The contributions of this paper are therefore as follows:
1) A technique, implemented into a tool called LAYOUT DR, that, given a responsive webpage and an RLF, offers a developer the choice between repair(s) created with the page's own design at different viewports (Section III). **2)** An empirical study evaluating LAYOUT DR with 19 real responsively
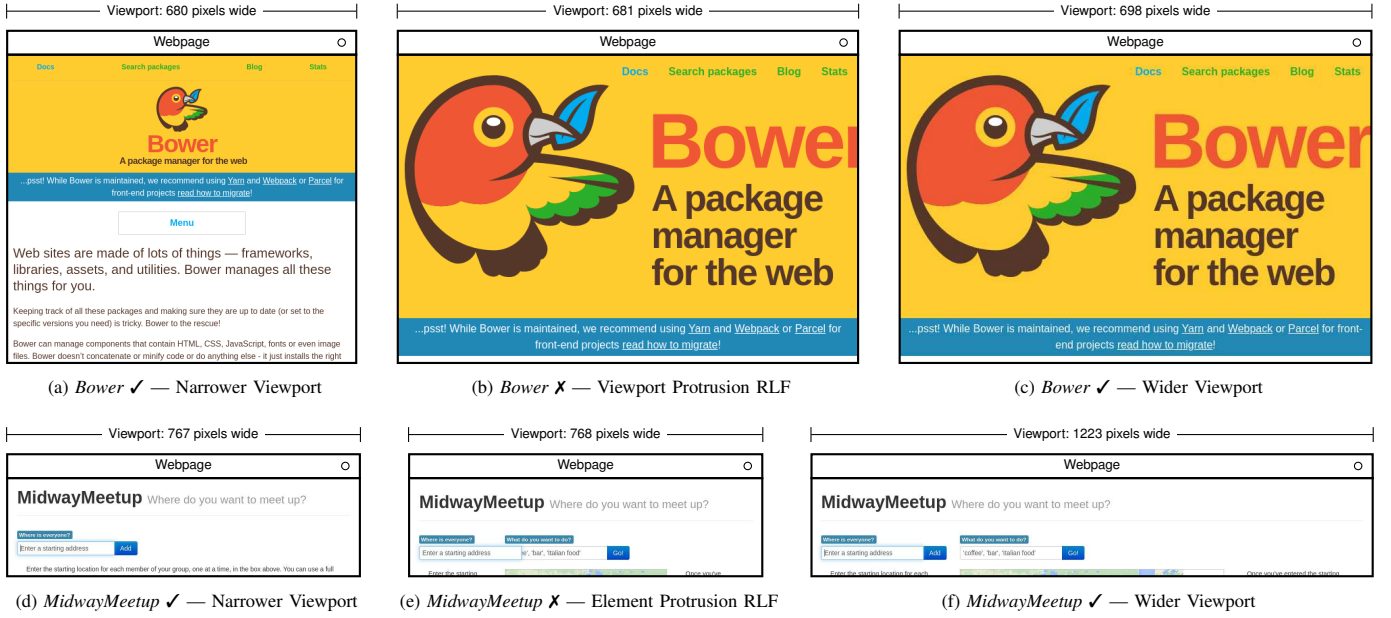
Fig. 1. Webpages with *Responsive Layout Failures* (RLFs), visual discrepancies that are only apparent when the page is rendered at certain viewport widths. For the *Bower* example, the "r" in the logo protrudes off the right edge of the viewport (part (b)). For *MidwayMeetup*, the first text input box protrudes into the second, and its associated button is hidden and no longer clickable (part (e)). Both pages render without issues at narrower and wider viewport widths.

designed webpages involving 55 RLFs, showing that **(a)** it reliably fixes those RLFs; **(b)** compared to the original, buggy page, humans prefer the repaired ones; and **(c)** LAYOUT DR is practical for developers to run, always taking less than 40 seconds to repair the studied RLFs (Section IV).

II. BACKGROUND

Responsive Web Design (RWD) is a design paradigm for webpages in which the layout of HTML elements adjusts to accommodate the space available [24]. RWD allows for a web page to be rendered on a range of different devices with a comparable user experience — including from the relatively small screens of mobile phones to larger displays such as those provided by modern desktop monitors. With RWD, developers design webpages in consideration of a broad range of *viewport widths*, or the amount of space available horizontally to render the page in a web browser. This is because a properly designed RWD page adjusts its content to fit in the horizontal constraints of the browser so that the user does not have to pan the page back and forth sideways to read and access content. Importantly, this means that the user should only have to scroll the page vertically. The typical assortment of viewport widths a developer must consider ranges from the very small ones of just 320 pixels wide for a mobile device, up to very wide widths of 1400 pixels and beyond for larger desktop displays.

RWD utilizes the concepts of fluid grids, flexible media, and CSS media queries for accommodating different viewport sizes [24]. Fluid grids allow HTML elements to be arranged into layouts that adjust in relation to the current viewport size, while flexible media refer to images or videos that expand and contract in size according to space available. CSS media queries allow developers to switch groups of CSS rules on and off depending on the configuration and size of the browser.

Despite these innovations, and frameworks such as Bootstrap [3] and more recent alternatives (e.g., Bulma [4], Tailwind-CSS [8], and WindiCSS [9]) that help developers to create an RWD page, designing layouts so that they dynamically adjust to fit each viewport width in a broad range of viewport sizes is still challenging. *Responsive Layout Failures* (RLFs) — visual discrepancies in the layout of an RWD page — are frequent and find their way onto live sites, as reported by Walsh et al. [28], who identified five common RLF types:

Viewport Protrusion occurs when elements cannot be accommodated within the space of the current viewport and spill off the edge of the page. An example of this can be seen with the *Bower* webpage depicted in Figure 1. At the wider viewport shown in part (c), the "Bower" title in the masthead is properly located within the confines of the space of the page. However, as the viewport narrows in part (b), there is no longer enough horizontal space, and the title no longer fits. The "r" protrudes off the edge of the page. Viewport protrusion potentially forces the end user to pan sideways to access content, thereby breaking one of the key principles of RWD. At a narrower viewport width, shown by part (a), the responsive layout reformats to fit the more confined space.

Element Protrusion occurs when an HTML element is contained within the space of another, but as the viewport width decreases the child element does not have sufficient space to accommodate its contents within the constraints of its parent. As such, the child element protrudes out of its container. An example of element protrusion is shown by Figure 1(e) with the *MidwayMeetup* webpage. At the wider viewport, shown by part (f), the text box and "Add" button in the left column fit into the space afforded to it. In part (e), however, there is no longer enough horizontal space in the column, so the textbox protrudes out, and the "Add" button is hidden behind another

element. The webpage reformats correctly for tighter screen sizes with even less horizontal space, as shown by part (d).

Element Collision is when the viewport width reduces such that elements do not have the required space to accommodate their contents without being rendered on top of other elements.

Wrapping Failures occur when HTML elements are supposed to be rendered together on one line (for example, the menu items in the navigation bar of a page), but as the viewport width reduces, they can no longer fit side by side, causing one or more elements to wrap to a new line on the page.

Finally, **Small-Range Failures** occur when a page's layout chaotically changes for a small number of contiguous viewport widths, often due to off-by-one errors made by developers when encoding media queries in the CSS of a webpage. For example, CSS rules pertaining to the media queries `@media (min-width: 768px)` and `@media (max-width: 768px)` would both be active at a viewport width of 768 pixels, since the ranges defined by both queries are inclusive. When rules are switched on that the developer did not intend, the design of a page can look very erratic for particular viewport widths.

To automatically detect these issues, Walsh et al. [28] proposed a data structure called the Responsive Layout Graph (RLG). The RLG tracks the relative alignment of a webpage's HTML elements with respect to one another as the viewport width of the page changes. Walsh et al.'s method builds the RLG by sampling the layout of a webpage at intervals throughout a range of viewport widths, leveraging the Document Object Model (DOM) at each width sampled to retrieve the coordinates of each HTML element. It employs a binary search to locate precise viewport widths between intervals relating to exact points of layout change. Walsh et al. also proposed algorithms that use the RLG to identify each of the five common responsive layout failures previously introduced, reporting the viewport range of the RLF and details about the elements involved. They implemented all of these techniques into a tool called REDECHECK [29]. Even though REDECHECK reliably detects the failures in a responsive page, a developer must manually fix the CSS problems. That is, REDECHECK does not automatically repair a reported RLF.

In general, fixing RLFs is a non-trivial problem. Tens or hundreds of HTML elements may be involved, each with an individual set of CSS properties that may vary depending on the viewport width. For some RLFs, only a single CSS property may need to be adjusted. Other RLFs may have major implications for the overall design of the page, requiring significant re-working of the HTML and CSS code [27].

III. AUTOMATED RLF REPAIR WITH LAYOUT DR

This section introduces an automatic responsive layout failure repair technique, which we implemented into a tool named "LAYOUT DR" (responsive _layout_ failure _d_etection and _r_epair, pronounced "Layout Doctor"). The presented technique first determines whether there are RLFs in a responsive webpage. If any RLFs are found, it reports their type (according to the categories defined in Section II) and the *RLF range*, denoted $\{fail_{min}..fail_{max}\}$, where $fail_{min}$ is the smallest viewport width where the RLF occurs and $fail_{max}$ is the largest one.

The key idea behind our technique is to harness the webpage layouts on either side of the failure range as a basis for the repair. We refer to these layouts as the *bordering layouts*. The *narrower bordering layout* exists at the viewport width $fail_{min}-1$, while the *wider bordering layout* exists at the viewport width $fail_{max}+1$; that is, they are at either side of the RLF range. From the entire range of viewport widths that the webpage can be viewed at, the bordering layouts of the failure represent the ones likely to most closely represent the layout where the RLF occurred, but without, themselves, exhibiting the same failure. LAYOUT DR then selects a bordering layout and extracts the HTML elements. Each viewport width within the failure range uses a version of this layout that is dynamically scaled to that width using an automatically created CSS patch that overrides the prior layout behavior of the page. LAYOUT DR thereby creates a temporary "hotfix" for a live webpage that can then afford a developer the time needed to realize a properly engineered patch for the page's CSS. The webpages in Figure 1 show the potential merit in our premise. The narrower and wider viewport widths either side of the RLF are layouts that could be scaled up or down to fit the viewport width with the RLF, thus covering over the failure.

In the following, we describe the operation of our technique in more detail, as summarized by Figure 2, which provides an overview schematic of its implementation in the LAYOUT DR tool. As shown by the figure, LAYOUT DR takes a responsive webpage, and the *responsive range* of viewport widths it is designed to be displayed for, $\{range_{min}..range_{max}\}$.

RLF Detection. Figure 2 shows that our technique begins by taking a responsively designed page and running it through a process called *RLF Detection*, which seeks to find any RLFs that may be present. For this step, we re-implemented RE-DECHECK [29] and the RLF detection routines introduced in Section II. We re-implemented REDECHECK to streamline the incorporation of its algorithms into our tool, while also giving us the chance to fix some bugs noted by Althomali et al. [13] and update its outdated libraries. For example, REDECHECK uses versions of Selenium to drive Firefox that were released in 2016; in contrast LAYOUT DR uses a recent release of the more modern Puppeteer to drive the Chromium browser. RLF Detection outputs a set of one or more reports for a webpage and its intended responsive range. Each report refers to a specific detected RLF, comprising its type (one of *Viewport Protrusion*, *Element Protrusion*, *Element Collision*, *Wrapping*, or *Small Range*); details about the HTML elements involved in the RLF; and the *RLF range*, $\{fail_{min}..fail_{max}\}$, or the set of viewports in which the layout failure occurs.

RLF Filtering. The next step of the process, *RLF Filtering*, is a manual one that happens outside of the LAYOUT DR tool. Since RLF detection is prone to false positives, a human must examine the RLF reports and check whether each RLF is visibly evident and requires repair. This is because DOM-based RLF detection, such as that employed by REDECHECK,
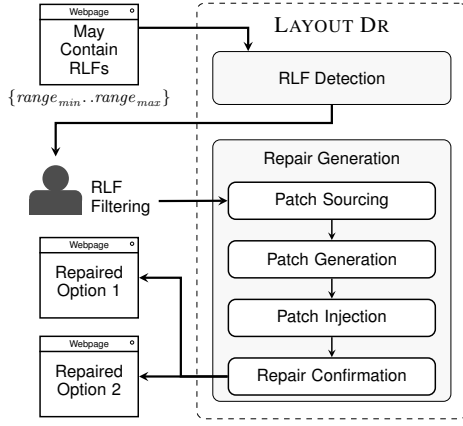
Fig. 2. Overview schematic showing the operation of LAYOUT DR.

is particularly prone to a class of false positives known as *non-observable issues* — issues that are apparent in the DOM but which are not visibly evident in the page itself [13]. For instance, two HTML elements may overlap, as discovered from checking the coordinates of each element in the DOM, but if they are both transparent and the content of one does not overwrite that of the other, the collision of elements will not be visible to a human looking at the page. Since the focus of this paper is on RLF repair, we judge that this is not a significant issue for the presented technique. However, in the future, we plan to implement techniques that extend REDECHECK to alleviate these issues in LAYOUT DR, such as VERVE [13], which applies further visual checks to discern visible RLFs in these kinds of situations. It is worth noting, however, that even the VERVE tool is not 100% accurate, so it is inevitable that some level of manual checking may still be required.

Repair Generation. LAYOUT DR then sets about repairing the RLF(s), identified by the *RLF Filtering step*. The four stages of repair, as shown by Figure 2, are *Patch Sourcing*, *Patch Generation*, *Patch Injection*, and *Repair Confirmation*.

Patch Sourcing. The presented technique works to produce a repair by sourcing a layout from the narrower and wider bordering layouts at $fail_{min}-1$ and $fail_{max}+1$, respectively.

Each bordering layout must pass LAYOUT DR's *applicability test* to ensure it is acceptable for use in a repair. First, its viewport width must be within the responsive range, $\{range_{min}..range_{max}\}$, originally sampled by the failure detection component. That is, $fail_{min}-1 \geq range_{min} \wedge fail_{max}+1 \leq range_{max}$. Suppose the responsive range of a page is 320–1400 pixels, and LAYOUT DR detects an RLF between 320 and 600 pixels wide. As the layout at the viewport of 319 pixels wide is not part of the original responsive range (i.e., $fail_{min}-1 < range_{min}$), this bordering layout is not used, and patch generation from the narrower bordering layout ($fail_{min}-1$) does not proceed. Equally, if the failure occurred at the viewports between 1000 and 1400 pixels wide, the layout at the viewport width at 1401 pixels would similarly not be considered since $fail_{max}+1 > range_{max}$. While, practically, the layout at 319 pixels could form a viable repair, the responsive range of the page inputted to our

technique is considered to be the range of valid viewports at which it may be viewed. Furthermore, the cut-off must be established somewhere, since even if $range_{min}=1$, the layout at 0 pixels wide would not be usable by LAYOUT DR.

The second test is that a bordering layout must be free of a transformed version of the RLF under repair. For example, an *element protrusion* failure may transform into a *viewport protrusion* failure if, at the narrower bordering layout, an element protrudes not only out of its container, but also out of the viewport itself. Our tool, and the REDECHECK tool before it, will report these as two distinct RLFs with two different ranges. Therefore, the patch sourcing element of LAYOUT DR performs a check on the DOM of the bordering layout — depending on the type of failure — to ensure that it is suitable for use as a patch. For example, when handling element protrusion, LAYOUT DR checks that in the bordering layout, the child element does indeed reside inside its containing element, and thus has not spilled out of the viewport as well.

Assuming one or both of the bordering layouts is a viable layout according to the two aforementioned criteria, the technique proceeds to generate one or two patches for the failure. If neither is suitable, then LAYOUT DR fails to produce a repair.

Patch Generation. Patch generation begins by collecting the CSS from each bordering layout (if applicable), rescaling it, and applying it to all viewports in the failure range. Figure 3 illustrates the stages of the patch generation process with wireframe webpage layouts, starting with a page containing an RLF (part (a) of the figure), where the element marked "D" collides with the element marked "E". Using the two bordering layouts (the narrower shown in part (b) of the figure, the wider in part (f)), our approach generates two possible repairs for the RLF that the developer can choose from to fix the page.

To produce a repair from a bordering layout, the tool drives a browser to open the webpage with the viewport set at that layout's particular viewport width, and then extracts the CSS properties of all of its elements. These CSS properties are a result of the browser resolving all CSS rules and property settings defined in the webpage's CSS files and through inline `style` HTML elements. To do this, LAYOUT DR embeds JavaScript code into the page to invoke the `getComputedStyle()` method on all elements of the page. This particular method returns all possible CSS settings (including computed positional and dimensional values in pixels) associated with each HTML element in the page after the browser applies all CSS styles from the webpage's CSS style files, HTML `style` elements, and any relevant browser default style settings. Starting at the root element of the DOM (i.e., the `html` element), LAYOUT DR traverses the DOM to capture all the CSS properties of each element in the tree. The CSS snippet in Figure 3(k) shows CSS properties of some of the HTML elements of the wider bordering layout in 3(f).

Since LAYOUT DR now has all the CSS properties needed to reproduce the layout at another viewport width, it proceeds to generate a CSS patch that can be applied to the page. As a first step, it creates a CSS selector for each HTML element in the bordering layout that will become part of the repair, so
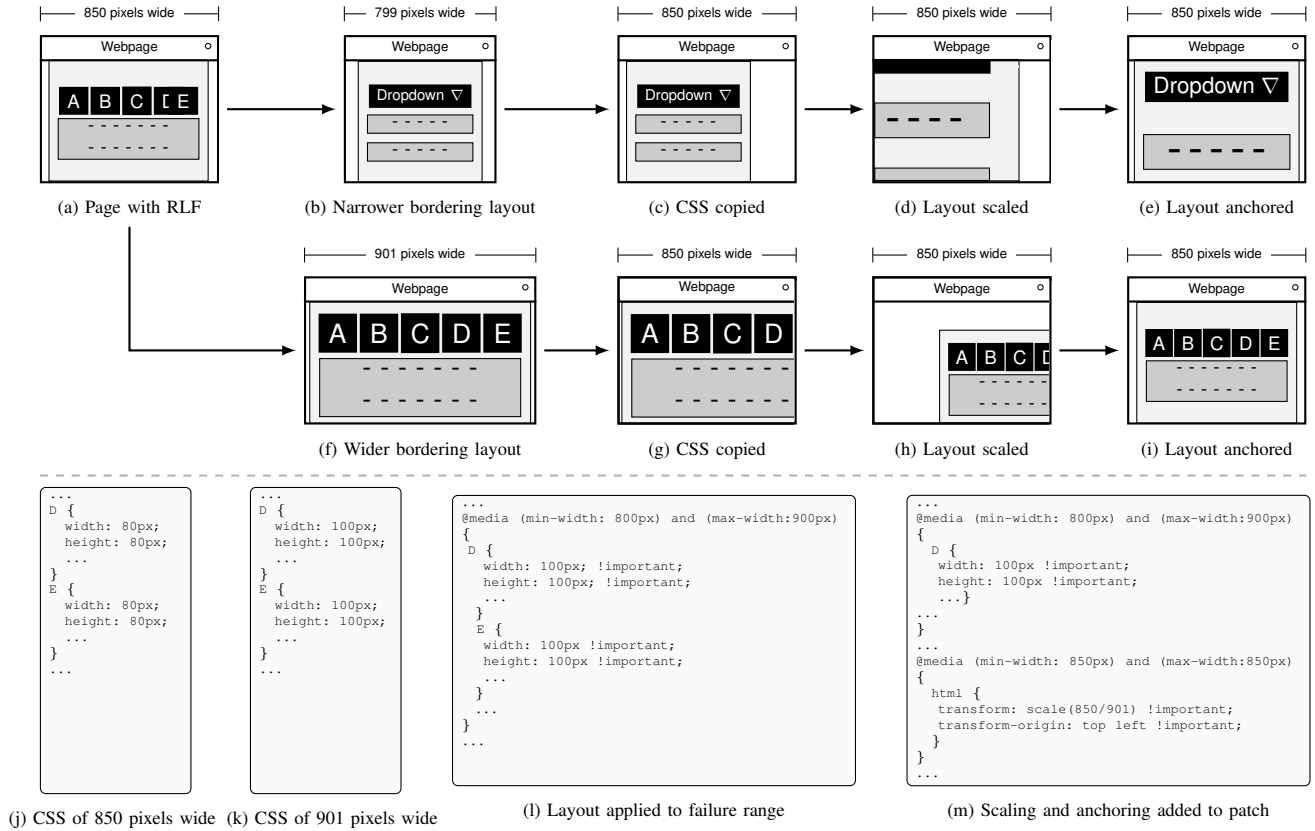
Fig. 3. A wireframe example of the steps involved in producing repair, with CSS snippets for the patch generated using the page's wider bordering layout.

(a) Page with RLF — 850 pixels wide
(b) Narrower bordering layout — 799 pixels wide
(c) CSS copied — 850 pixels wide
(d) Layout scaled — 850 pixels wide
(e) Layout anchored — 850 pixels wide
(f) Wider bordering layout — 901 pixels wide
(g) CSS copied — 850 pixels wide
(h) Layout scaled — 850 pixels wide
(i) Layout anchored — 850 pixels wide

(j) CSS of 850 pixels wide
```
...
D {
   width: 80px;
   height: 80px;
   ...
}
E {
   width: 80px;
   height: 80px;
   ...
}
...
```

(k) CSS of 901 pixels wide
```
...
D {
   width: 100px;
   height: 100px;
   ...
}
E {
   width: 100px;
   height: 100px;
   ...
}
...
```

(l) Layout applied to failure range
```
...
@media (min-width: 800px) and (max-width:900px)
{
   D {
     width: 100px; !important;
     height: 100px; !important;
     ...
   }
   E {
     width: 100px !important;
     height: 100px !important;
     ...
   }
   ...
}
...
```

(m) Scaling and anchoring added to patch
```
...
@media (min-width: 800px) and (max-width:900px)
{
   D {
     width: 100px !important;
     height: 100px !important;
     ...}
   ...
}
...
@media (min-width: 850px) and (max-width:850px)
{
   html {
     transform: scale(850/901) !important;
     transform-origin: top left !important;
   }
}
...
```

that the properties can be reapplied to the same element they were extracted from. For this, our technique uses the XPath of each element to generate a unique CSS selector which will encompass all the properties of an individual element in the patch. Since the CSS properties in the patch will be competing with others from the code base of the webpage, the !important flag is added to all properties in the patch to override any competing declarations. With the selectors set to target each element, the tool must now target the viewports for which the patch should take effect. Otherwise, the patch will be applied to all viewports. To restrict the patch to the failure range, the selectors and their properties are encapsulated within a media rule spanning the failure range. These modifications to the CSS are demonstrated by part (l) of Figure 3, with the failure range appearing in the media query segment (the line beginning "@media...") as 800–900 pixels.

Without further improvements to the patch, the webpage will cease to be responsive where the patch is applied. This is because the patch contains absolute positional values (i.e., any CSS property measured in pixels) specific to a single viewport — the viewport width of a particular bordering layout — and need adaptation to "fit" into a smaller viewport size if the wider bordering layout is under adaptation, or seamlessly use all the space if it is the narrower layout. To scale the layout to occupy the full viewport width and nothing more or less, our technique uses the scale() CSS method in conjunction with the transform CSS property. The transform property

modifies the coordinates of the associated element to rotate, scale, skew, or translate from its original coordinates. The technique uses the scale() method to scale an element's coordinates to be smaller or larger than the originals, resulting in a zoom effect. Since the application of this property and method on an element exceeds the element itself and affects all descendant elements in the DOM tree, the tool applies the scale() property on the root element of the DOM, the html element, to scale all elements of the layout appropriately. This scale value is calculated based on the ratio of the browser's "current" viewport to the bordering layout viewport, to make the patch itself, when applied to the webpage, responsive. These adjustments are shown by Figure 3(m). For the example viewport width of 850 pixels, elements are scaled with the amount 850/901 (i.e., this specific repair viewport width, divided by the width of the bordering layout used to generate the patch, 901 pixels). The full patch contains more similar declarations for the other viewport widths in the RLF range.

Although our technique scales the bordering layout using the transform property, the scaled layout transformation for the wider bordering viewport is anchored to the center of the original coordinates. The result is a webpage with empty space to the top, right, bottom, and left of the page. Parts (d) and (h) illustrate the result of applying the scale() method on the example layout from parts (c) and (g) respectively. The result is a scaled down version of the layout that shrinks to the center. Worse than the empty space is the portion of the page that
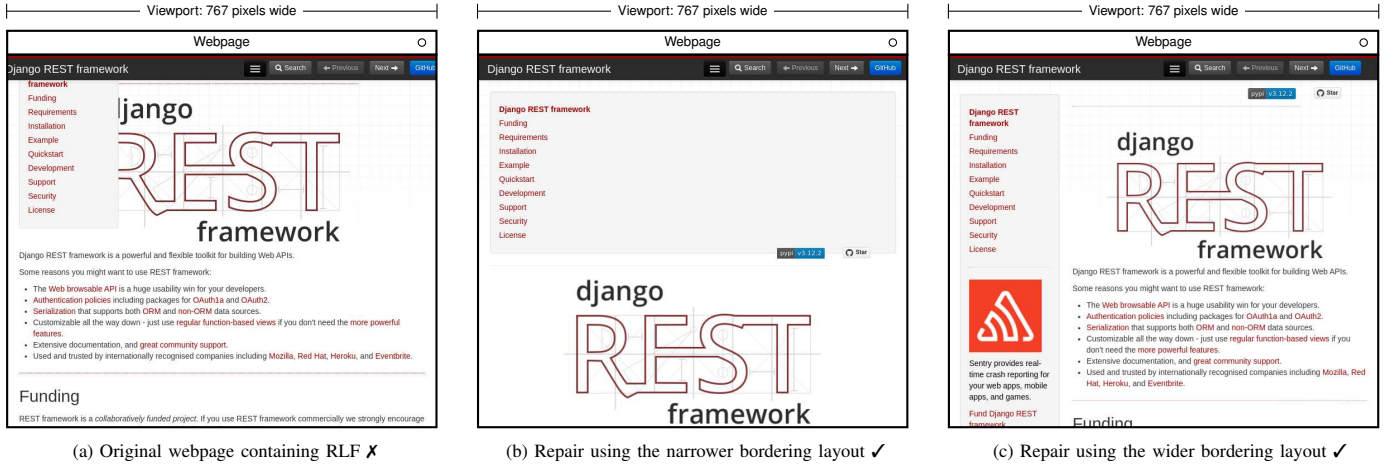
Fig. 4. Two repairs produced by LAYOUT DR for the *DjangoREST* subject as part of this paper's empirical evaluation.

(a) Original webpage containing RLF ✗   (b) Repair using the narrower bordering layout ✓   (c) Repair using the wider bordering layout ✓

will not be visible without scrolling horizontally and breaking one of the responsive design principles. To resolve this issue, LAYOUT DR also adds the `transform-origin` CSS property to the patch to position the scaled layout appropriately at the top-left of the browser window. This property identifies the position around which a transformation is applied using the `transform` property. This addition is shown as the last declaration in the snippet of the patch shown by part (m) of the figure, anchoring the scaled layout to the proper location, as seen in parts (i). Along with the repair generated from the narrower bordering layout, these two final repairs form two options that the developer can use to fix the webpage.

Patch Injection. LAYOUT DR proceeds to inject each patch into the webpage. LAYOUT DR achieves this by creating a `style` element that contains the CSS code of the patch and attaches it to the end of the DOM tree. On refreshing the page, the browser then uses the additional CSS comprising the patch to render the scaled bordering layout at viewports within the original failure range. The patch is then assessed by LAYOUT DR in the next stage, *Repair Confirmation*. LAYOUT DR then removes the `style` element from the page.

Repair Confirmation. Re-using the RLF detection algorithms, LAYOUT DR checks the RLF range of the repaired page to confirm that the repair has successfully removed the original responsive layout failure. If both bordering layouts led to successful patches, the developer may pick the one to use. Otherwise, if there was only one successful patch, then LAYOUT DR applies it by default to the final page. Figure 4 gives the *DjangoREST* subject used in the evaluation. Part (a) shows obvious visual discrepancies, while parts (b) and (c) show two repairs produced by LAYOUT DR, with patches respectively sourced from the narrower and wider bordering layouts.

IV. EVALUATION

To evaluate Section III's technique, we implemented it into the LAYOUT DR tool and applied it to 19 webpages that contained a total of 55 RLFs. Since the first author's thorough visual investigation always confirmed that LAYOUT DR was

successful at automatically fixing each RLF, we designed the empirical study to answer these three research questions:

RQ1: Which bordering layout is likely to result in a repair? This RQ aims to evaluate how effective LAYOUT DR is at repairing failures in responsive webpages, investigating which bordering layout (i.e., the narrower bordering layout or the wider bordering layout) tends to result in successful repairs.

RQ2: Do humans prefer the repaired version of the webpage produced by LAYOUT DR *compared to the version exhibiting the failure?* Along with determining if LAYOUT DR's repairs are acceptable to humans viewing the webpage, this RQ also investigates whether humans prefer LAYOUT DR's repairs at the narrower bordering layout or the wider bordering layout.

RQ3: How long does LAYOUT DR *take to generate patches?* This RQ intends to find out if the time taken by LAYOUT DR is reasonable for developers who wish to apply it in practice.

Tool and Experimental Runtime Environment. We implemented LAYOUT DR in JavaScript and ran it using Node version 14.15.4 with NPM version 6.14.10 installed on a workstation running the 64-bit version of the Ubuntu 20.04.2 operating system. The workstation had 16GB of RAM and an Intel Core i7-4720HQ processor. LAYOUT DR also used Puppeteer version 4.0.1, a Node library with an API to control a Chromium browser that we configured to run in headless mode, with a fixed viewport height of 1000 pixels. To assess LAYOUT DR's effectiveness at producing and confirming patches for the viewports of mobile devices up to desktop monitors, we set a page's responsive range to 320–1400 pixels.

Subjects. To answer the RQs we needed a subject set of webpages containing RLFs for our technique to repair. To compile our subject set, we began with the 26 webpages studied by Walsh et al. [28] and available in their online repository [6], which contains examples of responsive webpages in which their REDECHECK tool was able to find real examples of RLFs. To this we added further webpages harvested from the web. In particular, we sought to find examples of RLFs in the websites of open source tools, using the search terms "open source software" and "top open source software", while also

manually searching for webpages linked from the repositories of software with high numbers of stars hosted on GitHub. We used GNU Wget version 1.20.3 to download all candidate webpages. and ran LAYOUT DR with these examples and studied the detected RLFs. As part of the necessarily manual *RLF Filtering* phase (see Section III), we discarded RLF reports for which we could not see a visual discrepancy in the layout of the page. LAYOUT DR's detection routines are based on REDECHECK, which is DOM-based, and therefore prone to reporting *non-observable issues*, a prevalent form of false positive [13]. For instance, two HTML elements may overlap, as discovered from the DOM, but if they are both transparent and the content of one does not overwrite that of the other, the collision of elements will not be visible to a human looking at the page. Since the LAYOUT DR tool is primarily focussed on repair, this is not a major issue. However, in the future, we intend to implement the techniques that extend REDECHECK to alleviate these issues, such as VERVE [13], which applies extra visual checks to discern these kinds of situations. We further discarded reports when we found a visual "disturbance" that did not significantly influence the page in a way that a human developer would want to repair. An example of this is when there is not enough horizontal space to fit every social media icon side-by-side in the footer of the page, with one of the icons wrapping to the new line, thus triggering a wrapping RLF. However, since the icons are still centered, this wrapping behavior is likely not worthy of a developer's further attention.

Only considering the definite and visually evident RLFs resulted in a total of 55 from 19 webpages overall, comprising 9 pages from Walsh et al.'s original subject set and 10 unique to this paper. Table I lists these webpages and the RLF counts, revealing that these pages vary in their size and complexity and thus form a diverse subject set suitable for this study.

Methodology. To answer *RQ1*, we ran the *Repair Generation* stage of LAYOUT DR on the 19 webpages. We recorded the bordering viewport layouts (i.e., narrower and/or wider) that it used to generate repairs (i.e., passed both the tool's applicability tests and its automated repair confirmation check following the repair's creation). The first author then visually inspected each repair to check that (a) it had correctly removed the original RLF; and (b) had not introduced more RLFs.

To determine whether the repairs were acceptable to human users in general, and to answer *RQ2*, we conducted a human study on Amazon Mechanical Turk [1]. We designed a web-based questionnaire where participants were asked to judge the repairs. The basis of each question is an RLF that was fixed by LAYOUT DR, with a repair generated from both the narrower and wider bordering layout. The questionnaire asked participants to compare an image of the original webpage containing the RLF and images of each of the two repairs, and to select which version of the page they prefer in each case. To maintain as much authenticity as possible within the scope of the questionnaire, we did not scale the snapshot images of the different webpage versions, presenting images that were the same width as the originals. Since it was not

TABLE I
SUBJECT WEBPAGES USED IN THE EXPERIMENTS

In this table, #RLFs is the number of RLFs found in the subject, #HTML is the number of HTML elements on the page, and #CSS is the number of CSS properties for each of those elements. Note that a subject may no longer be available in its studied form at the URL listed — please refer to our replication package for all of a subject's details [7].

Subject	Original URL	#RLFs	#HTML	#CSS
3MinuteJournal	3minutejournal.com	4	80	5499
Ardour	ardour.org	2	222	3774
Bottender	bottender.js.org	5	243	2202
Bower	bower.io	1	370	844
BugMeNot	bugmenot.com	1	42	658
ConsumerReports	consumerreports.org	7	1042	8005
Django	djangoproject.com	1	242	4732
DjangoREST	django-rest-framework.org	1	610	3787
Duolingo	duolingo.com	1	856	4260
ElasticSearch	elastic.co/elasticsearch	2	1243	21467
Honey	joinhoney.com/install	1	461	7903
HotelWiFiTest	hotelwifitest.com	1	359	6746
MantisBT	mantisbt.org	3	247	7731
MarkText	marktext.app	15	560	1890
MidwayMeetup	midwaymeetup.com	1	86	4147
OrchardCore	orchardcore.net	5	234	6352
PepFeed	pepfeed.com	1	343	7276
Selenium	selenium.dev	1	286	4980
WillMyPhoneWork	willmyphonework.net	2	782	6576
Total		55	8308	108829

possible to present whole webpages in their entirety without vertical scrolling, we presented the images within a frame, so that participants could scroll around the image as they would a normal webpage in a browser window. The questionnaire presented each image in a tab, allowing participants to flip between tabs, comparing each image. When the RLF was not at the top of the page, the questionnaire would present the image automatically scrolled vertically to the position of the failure, so that each version of the page could be compared directly by switching tabs. Finally, if a participant's screen was not big enough to accommodate the width of the snapshot and the questionnaire, the questionnaire displayed an error message. Users needed to have a minimum screen resolution of $1400{\times}780$ to accommodate both webpage screenshots and the surrounding GUI elements of the questionnaire itself.

We used 20 RLFs from the 14 webpage subjects for the human study, which were specifically the RLFs for which LAYOUT DR successfully generated repairs for both the narrower bordering layout and wider bordering layout of the RLF. So as to mitigate the potential effects of fatigue affecting participants, we limited each questionnaire to only ten questions. Each questionnaire for each participant featured ten questions randomly selected from the overall pool of 20. We terminated the availability of the study on Mechanical Turk after we had reached over 100 responses. Since Mechanical Turk necessitates that we remunerate participants, we paid them $1 for a median of just under 5 minutes of their time to complete the questionnaire. The amount we paid was similar to other studies in software engineering of a similar style and length (e.g., [19], [32]). As part of controlling the quality of the data from the survey, we added code to the web-based questionnaire to monitor the number of clicks on each of the

This table records the number of RLFs for a subject that has a particular type of bordering layout (**N**, **W**, **N|W**, and **N&W**) used by LAYOUT DR for a repair. The second figure (in parentheses) is the number of those layouts manually verified as RLF-free (including the non-presence of further RLFs).

N and **W** count RLFs with applicable narrower and wider bordering layouts, respectively; **N&W** counts RLFs with an applicable narrower *and* wider layout, **N|W** counts RLFs with an applicable narrower *or* wider layout.

Subject	#RLFs	N		W		N&W		N|W	
3MinuteJournal	4	2	(2)	4	(4)	2	(2)	4	(4)
Ardour	2	0	(0)	2	(2)	0	(0)	2	(2)
Bottender	5	0	(0)	5	(1)	0	(0)	5	(1)
Bower	1	1	(1)	1	(1)	1	(1)	1	(1)
BugMeNot	1	0	(0)	1	(1)	0	(0)	1	(1)
ConsumerReports	7	2	(2)	7	(7)	2	(2)	7	(7)
Django	1	0	(0)	1	(1)	0	(0)	1	(1)
DjangoREST	1	1	(1)	1	(1)	1	(1)	1	(1)
Duolingo	1	1	(1)	1	(1)	1	(1)	1	(1)
ElasticSearch	2	1	(1)	2	(2)	1	(1)	2	(2)
Honey	1	1	(1)	1	(1)	1	(1)	1	(1)
HotelWiFiTest	1	1	(1)	1	(1)	1	(1)	1	(1)
MantisBT	3	2	(1)	3	(3)	2	(1)	3	(3)
MarkText	15	3	(2)	15	(2)	3	(0)	15	(4)
MidwayMeetup	1	1	(1)	1	(1)	1	(1)	1	(1)
OrchardCore	5	0	(0)	5	(5)	0	(0)	5	(5)
PepFeed	1	1	(1)	1	(1)	1	(1)	1	(1)
Selenium	1	1	(1)	1	(1)	1	(1)	1	(1)
WillMyPhoneWork	2	2	(2)	2	(2)	2	(2)	2	(2)
Total	**55**	**20**	**(18)**	**55**	**(38)**	**20**	**(16)**	**55**	**(40)**

three tabs involving either the RLF or one of the repairs. To account for the possibility of participants voting for an option without viewing all of them first, we filtered the results to show only the responses of the participants who clicked on each of the tabs at least once, firing the JavaScript load event for each of the webpage images in the questionnaire application.

To answer *RQ3*, we recorded the time taken, in milliseconds, by LAYOUT DR in the *Repair Generation* phase on each of the 19 webpage subjects in Table I. Finally, to obtain a reliable average, we repeated these timing experiments 10 times.

V. ANSWERING THE RESEARCH QUESTIONS

Answer to RQ1. For each RLF listed in Table II, we recorded whether each bordering layout formed the basis of LAYOUT DR's repair. Table II also presents how many repairs pass the first author's visual checks for being free of (a) the original RLF and (b) any further RLFs that may have been inadvertently copied into or created as part of the repair.

For all RLFs, LAYOUT DR could always use wider bordering layout to generate a repair. However, for the narrower bordering layout only 20 (36%) of the original 55 narrower layouts passed the applicability test and were subsequently used by LAYOUT DR to generate a repair. For the 35 RLFs with bordering layouts that failed the applicability test, 17 had a viewport range starting at 320 pixels wide, for which the narrower bordering layout was excluded by LAYOUT DR. For the other 18 RLFs, the failure transformed to a different RLF type at the narrower viewport and so were also inappropriate, thus causing LAYOUT DR to discard them as patch sources.

Further manual inspection revealed that not all repairs were free of interference from additional RLFs. In each case, this was because of additional RLFs in the page that happened to be present in the bordering layout used as part of the repair. LAYOUT DR repairs each RLF independently; it does not currently account for other RLFs detected/filtered prior to the repair process. (We leave this for future work, while RQ2 addresses whether these repairs were acceptable to human participants in general.) However, as Table II shows in the "N|W" column, 40 of the 55 RLFs had at least one viable repair that was also free of some other RLF. The majority of these repairs were sourced from the wider bordering layout. RLFs for which neither repair was free of additional RLFs were found only in 3 of the 19 subjects — *Bottender*, *MantisBT*, and *MarkText* — the subjects that had some of the higher numbers of RLFs detected in them, and therefore are prone to this issue.

> **Conclusion for RQ1.** The wider bordering layout is more likely to result in a repair than the narrower bordering layout.

Answer to RQ2. Table III gives the 20 RLFs and the 14 webpage subjects from which we drew them. They are, specifically, the RLFs from Table II where LAYOUT DR generated two repairs sourced from each of the narrower and wider bordering layouts. This table shows that the set of RLFs includes failures of each of the different types identified in Section II, demonstrating their suitability for the human study.

In total, 101 participants took our questionnaire. As explained in Section IV, we disregarded individual answers when the participant did not view each image (i.e., the original page containing the RLF and the two repairs generated from each bordering layout) before responding. We hereafter refer to non-disregarded answers as "votes" for a particular version of a webpage — either the original or a particular repair — only accepting a vote when the participant did view each questionnaire tab before answering. Overall, the participant responses to our questionnaire resulted in a total of 738 votes.

Table III shows an overwhelming preference for one of the repairs compared to the original version of the webpage containing the RLF. For all RLFs, the number of votes for the original page was small, and never greater than either repair. This result suggests that LAYOUT DR effectively produces a hotfix that does not degrade a webpage, giving developers extra time to diagnose and resolve a responsive layout failure.

This was true even though, as noted in our answer to RQ1, a few of bordering layouts themselves involved other RLFs. Four of these RLFs involved repairs that had further visible RLFs. The two Viewport Protrusion RLFs for *MarkText* had further RLFs appearing in the repair generated from the wider bordering layout; while the second wrapping RLF for *MantisBT* had a further RLF in the repair generated for the narrower bordering layout. For these, participants voted for the repairs generated from the alternative layout, except for the second Viewport Protrusion RLF for *MarkText*. Here, the RLF in the wider bordering layout appears to have not been noticed or regarded as insignificant by the participants. Both repairs for the Element Protrusion RLF of *MarkText* had further RLFs.

In this table, "**O**" is the number of votes for original page involving the RLF, "**N**" and "**W**" are votes for the repairs generated from the narrower and wider bordering layouts, respectively, "**N+W**" is the sum of votes for both repairs.

In terms of RLF types, "*EC*" is Element Collision, "*EP*" is Element Protrusion, "*SR*" is Small Range, "*VP*" is Viewport Protrusion, "*W*" is Wrapping.

Subject (RLF Type)	O ☐	N ▨	W ☐	(N+W)	
3MinuteJournal (*EP*)	3	7	25	(32)	
3MinuteJournal (*VP*)	2	22	12	(34)	
Bower (*VP*)	3	27	2	(29)	
ConsumerReports (*VP*)	5	33	2	(35)	
ConsumerReports (*EP*)	2	9	20	(29)	
DjangoREST (*VP*)	3	6	29	(35)	
Duolingo (*VP*)	4	8	25	(33)	
ElasticSearch (*EC*)	6	9	21	(30)	
Honey (*EC*)	3	11	27	(38)	
HotelWiFiTest (*VP*)	1	22	10	(32)	
MantisBT (*W*)	4	5	40	(45)	
MantisBT (*W*)	4	10	26	(36)	
MarkText (*VP*)	2	20	12	(32)	
MarkText (*EP*)	6	20	8	(28)	
MarkText (*VP*)	0	12	21	(33)	
MidwayMeetup (*EP*)	1	15	24	(39)	
PepFeed (*VP*)	3	20	10	(30)	
WillMyPhoneWork (*EC*)	5	14	22	(36)	
Selenium (*W*)	2	7	27	(34)	
WillMyPhoneWork (*SR*)	1	15	23	(38)	
Total	60	292	386	(678)	
	(8%)	(40%)	(52%)	(92%)	

Here also, the page with the original RLF was not the preferred choice, since the repair presented one less visual issue.

Overall, the wider bordering layout was the preferred source of the repair, scoring the most votes for 13 of the 20 RLFs. We surmise this is because the wider layout is often most similar to the one with the failure, while the narrower layout is often a scaled-up "mobile" view of the webpage. We manually analyzed the repairs to ascertain why participants may have opted for the narrower repair in the instances they did, and found that in four cases (i.e., the Viewport/Element Protrusions for *3MinuteJournal*, *Bower*, *ConsumerReports*, and *MarkText*) the wider layout solved the protrusion but pushed elements up to the edge of the viewport or their container, making the narrower repair more appealing. Figure 1 evidences this issue for the *Bower* RLF. Although the "r" in the logo is no longer clipped in part (c) — and the RLF does not occur — it is still on the right edge of the viewport boundary. For the remaining cases, the wider layout was scaled down to the point that the text was hard to read, which again makes the narrower repair more appealing. Future work needs to create methods for better taking into account these preferences of participants.

> **Conclusion for RQ2.** Participants preferred a repaired version of the webpage generated by LAYOUT DR over the original page containing the RLF. Generally, participants preferred the repairs created from the wider bordering layout over those originating from the narrow bordering one.

Answer to RQ3. Figure 5 plots the time taken to repair the RLFs listed in Table II. The plot shows that the time taken never exceed 40 seconds (i.e., for RLFs of the largest subject in terms of HTML elements, *ElasticSearch*) making the repair approach practical to use. It reveals a positive relationship between time and the number of HTML elements in the page (Spearman's correlation coefficient, ρ, is 0.81). Our timing analysis did not reveal a practically significant difference between the bordering layout used (i.e., narrower or wider) nor between RLF type, nor a strong correlation with the number of CSS properties. This is an intuitive result, since our method involves copying layouts pertaining to all the page's HTML elements, for which not all CSS properties defined by the page may be in active use for the particular viewport width used.

> **Conclusion for RQ3.** Repairs by LAYOUT DR took no longer than 40 seconds for the subject webpages studied, which is a very practical amount of time for developers to apply the tool in practice. The time taken is related to a page's complexity in terms of its number of HTML elements.

Threats to Validity. One threat to the validity of this paper's results is the extent to which they generalize to other webpages. We mitigated this by selecting webpages of a range of sizes. Table I shows that our subjects ranged in complexity from 42–1,243 HTML elements, and having between 658–21,467 CSS rules. The functionality and design of these webpages also varies, including online language learning (i.e., *Duolingo*) to browser automation tools (i.e., *Selenium*). Another potential threat to validity is the RLFs used as part of our study. These were found by re-implementing the algorithms of the REDECHECK tool, and were manually checked to remove false positives. This intrinsically manual, yet necessary, process was a straightforward one in which we discarded any reports of RLFs that were not visible or did not significantly impact the page — and were therefore unsuitable for our study. Finally, we attempted to avoid any potential subjectivity associated with confirming LAYOUT DR's repair(s) for each RLF by having the first author verify them with a thorough visual inspection and asking humans to judge the repairs.

There are also threats to the validity of the results from the human study. We only asked the participants to pick between the orginal, defective webpage and the repairs generated by LAYOUT DR, aiming to confirm that the hotfixes do not degrade a page's layout. We did not compare LAYOUT DR's repairs to alternatives created by tools like XFIX or IFIX because they do not automatically repair responsive layouts.

Another human study threat is the selection of participants. We used Amazon Mechanical Turk [1] to recruit anonymous participants from a relatively large pool. To ensure authentic results, we discarded any questionnaire responses if the participant had not clicked on each webpage image in order to assess the layout of each of the particular options provided. Another threat involves the devices on which participants viewed the study's webpages. To ensure that participants used a device with a display that correctly rendered each responsive page at the required viewport width in a frame of the overall questionnaire page, the system showed an error message if the participants's resolution was below 1400×768 and prevented
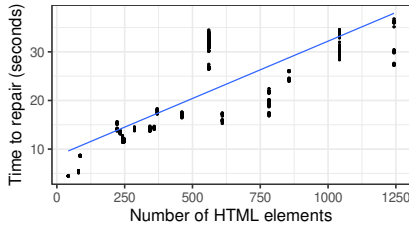
Fig. 5. Number of HTML elements in a webpage compared to its repair time. In this plot the blue goodness-of-fit line has an R^2 value of 0.62.

them from proceeding onto the questions. To mitigate threats regarding differences in participants' web browsers, we displayed the original webpages and their repairs as static, yet scrollable, images. Even though this meant that people could not interact with the webpages, this was not necessary as the study's goal was for participants to evaluate page layout.

Finally, there are validity threats in the implementation of the LAYOUT DR tool itself, which may contain defects. We mitigated this by ensuring that we wrote unit tests during its development, and manually checking all results. To support the replication of our results and the availability of our tool set, we have made our scripts, tool, and other artefacts associated with this paper's study available in a replication package [7].

VI. RELATED WORK

Detecting different types of presentation failures in web applications is a topic that has been extensively explored in the literature, including *cross browser issues (XBIs)* [15], [14], [25], a type of presentation failure that occurs when a web page is rendered with one particular browser, but is free of failures with at least one other browser; and *internationalization presentation failures* (IPFs) [11] — layout issues that arise from the translation of a web page due to differences in the space occupied by translated text compared to text in the page's original language. There have also been methods for detecting generalized presentation failures by comparing images of intended layout (e.g., mockups provided by a graphic designer) with actual layout in a web application [18], [23] and through the use of verification methods that require, for example, a developer to provide a layout specification [16].

For responsively designed web pages, Walsh et al. presented two versions of the REDECHECK tool, one that works to identify regressions between two consecutive versions of a webpage [30], [31] and one that checks for layout failures according to the implicit oracles reviewed in Section II. Furthermore, Ryou et al. [26] proposed VFDETECTOR, which also finds responsive layout failures, including those triggered by human interaction due to the incorporation of dynamic elements on the page coded with CSS and/or JavaScript.

While there has been much work on detecting presentation failures, there has been comparatively less work on automated approaches to repairing them. Mahajan et al. proposed a suite of tools for fixing different types of presentation failures. XFIX [20] implements a search-based approach for automatically repairing XBIs. IFIX [21], [22] also used a search-based technique for repairing IPFs. Meanwhile, Alameer et al. [10]

took a constraint solving approach to the same problem. Finally, MFIX [19] patches webpages so that they passed "Mobile Friendly" tests (e.g., those implemented in tools by Google [5] and Microsoft [2]) designed to rate web pages based on their suitability for rendering in a mobile browser (e.g., font sizing and "tap target" spacing). Jacquet et al. [17] also presented an approach that uses linear programming to repair layout failures such as element protrusions, overlaps, and misalignment. Their technique requires a web developer to furnish a constraint specification of the desired layout of the page. Moreover, it only appears to work with a webpage's layout at a fixed viewport width and, in contrast to LAYOUT DR, is therefore unsuitable for repairing responsive webpages.

All of the repair techniques discussed in this section tackle specific types of presentation failures. None of them, however, tackle RLFs as does this paper. Another key difference between the technique presented in this paper and others is where the "oracle" information comes from that guides or drives the repair. For XFIX, it is the "correct" webpage rendering in the reference browser. For IFIX, it is the original, untranslated, webpage rendering. For MFIX, it is the information supplied by mobile-friendliness test tools; while for the approach of Jacquet et al. [17], it is the constraint specification describing the correct layout of the page. For LAYOUT DR, presented in this paper, the repair originates from renderings of the same page at viewport widths either side of the range of an RLF.

VII. CONCLUSIONS AND FUTURE WORK

This paper presented a technique, implemented in a tool called LAYOUT DR, that repairs layout failures in responsive webpages. The tool "hotfixes" the page, meaning that, although the repair is not necessarily a suitable long-term repair, it addresses the symptoms of the original failure so that it cannot be "seen" by users of the page. The presented approach automatically generates a worthwhile fix, buying a developer time to diagnose and address issues appearing on live sites, which is a time-consuming and challenging process that may involve changes to many HTML elements and CSS properties or even necessitate a partial re-design of the responsive page.

Since the human study revealed that 92% of the participants preferred the repaired version of a webpage compared to the original one containing the RLF, we plan to improve LAYOUT DR as part of future work. For instance, we will provide further automatic support to the developer in identifying and fixing the particular components involved in the RLF. We also plan to further extend the presented approach by strengthening its RLF detection capability through the incorporation of ideas from tools such as VERVE [12], [13], thereby providing support for identifying RLFs that are not yet visually evident in a page. Future work will also provide support for repairing overlapping RLFs, in which both bordering layouts for a particular RLF involve some other, further RLF, detected for the page. Ultimately, the combination of these innovations with the efficient and effective baseline established by the current version of LAYOUT DR will result in a complete solution to automatically repairing responsive layout faults in webpages.

REFERENCES

[1] Amazon Mechanical Turk. Online: https://www.mturk.com/.
[2] Bing Mobile Friendliness Test Tool. Online: https://www.bing.com/webmaster/tools/mobile-friendliness.
[3] Bootstrap. Online: https://getbootstrap.com.
[4] Bulma CSS Framework. Online: https://bulma.io.
[5] Google Mobile Friendliness Test Tool. Online: search.google.com/test/mobile-friendly.
[6] ReDeCheck tool and ISSTA results archive. Online: http://redecheck.org/issta17.
[7] Replication package for this paper. Online: https://bitbucket.org/responsiverepair/replicationpackage.
[8] TailwindCSS. Online: https://tailwindcss.com.
[9] Windi CSS. Online: https://windicss.org/.
[10] Abdulmajeed Alameer, Paul T Chiou, and William GJ Halfond. Efficiently repairing internationalization presentation failures by solving layout constraints. In *International Conference on Software Testing, Verification and Validation (ICST 2019)*, pages 172–182, 2019.
[11] Abdulmajeed Alameer, Sonal Mahajan, and William GJ Halfond. Detecting and localizing internationalization presentation failures in web applications. In *International Conference on Software Testing, Verification and Validation (ICST 2016)*, pages 202–212, 2016.
[12] Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. Automatic visual verification of layout failures in responsively designed web pages. In *International Conference on Software Testing, Verification and Validation (ICST 2019)*, pages 183–193, 2019.
[13] Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. Automated visual classification of DOM-based presentation failure reports for responsive web pages. *Software Testing, Verification and Reliability*, 31(4), 2021.
[14] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. CrossCheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *International Conference on Software Testing, Verification and Validation (ICST 2012)*, pages 171–180, 2012.
[15] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. WebDiff: Automated identification of cross-browser issues in web applications. In *International Conference on Software Maintenance (ICSM 2010)*, pages 1–10, 2010.
[16] Sylvain Hallé, Nicolas Bergeron, Francis Guérin, Gabriel Le Breton, and Oussama Beroual. Declarative layout constraints for testing web applications. *Journal of Logical and Algebraic Methods in Programming*, 8, 2016.
[17] Stéphane Jacquet, Xavier Chamberland-Thibeault, and Sylvain Hallé. Automated repair of layout bugs in web pages with linear programming. In *International Conference on Web Engineering (ICWE 2021)*, pages 423–439, 2021.
[18] S. Mahajan, B. Li, P. Behnamghader, and W. G. J. Halfond. Using visual symptoms for debugging presentation failures in web applications. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 191–201, 2016.
[19] Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G.J. Halfond. Automated repair of mobile friendly problems in web pages. In *International Conference on Software Engineering (ICSE 2018)*, pages 140–150. ACM, 2018.
[20] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. Automated repair of layout cross browser issues using search-based techniques. In *International Conference on Software Testing and Analysis (ISSTA 2017)*, pages 249–260, 2017.
[21] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. Automated repair of internationalization failures using style similarity clustering and search-based techniques. In *International Conference on Software Testing, Validation and Verification (ICST 2018)*. IEEE, 2018.
[22] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. Effective automated repair of internationalization presentation failures in web applications using style similarity clustering and search-based techniques. *Software Testing, Verification and Reliability*, 31(1–2), 2021.
[23] Sonal Mahajan and William G. J. Halfond. WebSee: A tool for debugging HTML presentation failures. In *International Conference on Software Testing, Verification and Validation Tools Track (ICSE 2015)*, pages 1–8, 2015.
[24] Ethan Marcotte. *Responsive Web Design*. A Book Apart, 2014.
[25] Ali Mesbah and Mukul R Prasad. Automated cross-browser compatibility testing. In *International Conference on Software Engineering (ICSE 2011)*, pages 561–570, 2011.
[26] Yeonhee Ryou and Sukyoung Ryu. Automatic detection of visibility faults by layout changes in HTML5 web pages. In *International Conference on Software Testing, Validation and Verification (ICST 2018)*, pages 182–192, 2018.
[27] Thomas A. Walsh. *Automatic Identification of Presentation Failures in Responsive Web Pages*. PhD thesis, University of Sheffield, 2018.
[28] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. Automated layout failure detection for responsive web pages without an explicit oracle. In *International Conference on Software Testing and Analysis (ISSTA 2017)*, pages 192–202, 2017.
[29] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. ReDeCheck: An automatic layout failure checking tool for responsively designed web pages. In *International Conference on Software Testing and Analysis (ISSTA 2017)*, pages 360–363, 2017.
[30] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. Automatically identifying potential regressions in the layout of responsive web pages. *Software Testing, Verification and Reliability*, 30(6), 2020.
[31] Thomas A. Walsh, Phil McMinn, and Gregory M. Kapfhammer. Automatic detection of potential layout faults following changes to responsive web pages. In *International Conference on Automated Software Engineering (ASE 2015)*, pages 709–714, 2015.
[32] Westley Weimer. Advances in automated program repair and a call to arms. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE 2013)*, 2013.