Automated Repair of Layout Cross Browser Issues Using Search-Based Techniques

Sonal Mahajan
University of Southern California, USA

Abdulmajeed Alameer
University of Southern California, USA

Phil McMinn
University of Sheffield, UK

William G. J. Halfond
University of Southern California, USA

ABSTRACT

A consistent cross-browser user experience is crucial for the success of a website. Layout Cross Browser Issues (XBIs) can severely undermine a website's success by causing web pages to render incorrectly in certain browsers, thereby negatively impacting users' impression of the quality and services that the web page delivers. Existing Cross Browser Testing (XBT) techniques can only detect XBIs in websites. Repairing them is, hitherto, a manual task that is labor intensive and requires significant expertise. Addressing this concern, our paper proposes a technique for automatically repairing layout XBIs in websites using guided search-based techniques. Our empirical evaluation showed that our approach was able to successfully fix 86% of layout XBIs reported for 15 different web pages studied, thereby improving their cross-browser consistency.

CCS CONCEPTS

•Software and its engineering →Software testing and debugging; Search-based software engineering;

KEYWORDS

Cross-browser issues; automated search-based repair; web apps.

ACM Reference format:
Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated Repair of Layout Cross Browser Issues Using Search-Based Techniques. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
DOI: 10.1145/3092703.3092726

1 INTRODUCTION

The appearance of a web application's User Interface (UI) plays an important part in its success. Studies have shown that users form judgments about the trustworthiness and reliability of a company based on the visual appearance of its web pages [21, 22, 51, 52], and that issues degrading the visual consistency and aesthetics of a web page have a negative impact on an end user's perception of the website and the quality of the services that it delivers.

ISSTA'17, Santa Barbara, CA, USA

DOI: 10.1145/3092703.3092726

The constantly increasing number of web browsers with which users can access a website has introduced new challenges in preventing appearance related issues. Differences in how various browsers interpret HTML and CSS standards can result in *Cross Browser Issues (XBIs)* — inconsistencies in the appearance or behavior of a website across different browsers. Although XBIs can impact the appearance or functionality of a website, the vast majority — over 90% — result in appearance related problems [42]. This makes XBIs a significant challenge in ensuring the correct and consistent appearance of a website's UI.

Despite the importance of XBIs, their detection and repair poses numerous challenges for developers. First, the sheer number of browsers available to end users is large — an informal listing reports that there are over 115 actively maintained and currently available [59]. Developers must verify that their websites render and function consistently across as many of these different browsers and platforms as possible. Second, the complex layouts and styles of modern web applications make it difficult to identify the UI elements responsible for the observed XBI. Third, developers lack a standardized way to address XBIs and generally have to resolve XBIs on a case by case basis. Fourth, for a repair, developers must modify the problematic UI elements without introducing new XBIs. Predictably, these challenges have made XBIs an ongoing topic of concern for developers. A simple search on StackOverflow — a popular technical forum — with the search term "cross browser" results in over 23,000 posts discussing ways to resolve XBIs, of which approximately 7,000 are currently active questions [49].

Tool support to help developers debug XBIs is limited in terms of capabilities. Although tools such as Firebug [15] can provide useful information, developers still require expertise to manually analyze the XBIs (which involves determining which HTML elements to inspect, and understanding the effects of the various CSS properties defined for them), and then repair them by performing the necessary modifications so that the page renders correctly. XBI-oriented techniques from the research community (e.g., X-PERT [8, 42, 44] and Browserbite [47]) are only able to *detect and localize* XBIs (i.e., they address the first two of the four previously listed challenges), but are incapable of *repairing* XBIs so that a web page can be "fixed" to provide a consistent appearance across different browsers.

To address these limitations, we propose a novel search-based approach that enables the automated repair of a significant class of appearance related XBIs. The XBIs targeted by our approach are known as *layout XBIs* (also referred to as "structure XBIs" by Choudhary et al. [42]), which collectively refer to any XBI that relates to an inconsistent layout of HTML elements in a web page when viewed in different browsers. Layout XBIs appear in over

(a) **Correct rendering of the page with Internet Explorer 11.0.33**

(b) **The same page displaying an XBI when rendered with Mozilla Firefox 46.0.1**

Figure 1: Screenshots of the navigation bar of the IncredibleIndia homepage (http://incredibleindia.org), which has an XBI. When viewed with Firefox the text of the navigation menu bar is unreadable.

56% of the websites manifesting XBIs [42]. Our key insight is that the impact of layout XBIs can be quantified by a fitness function capable of guiding a search to a repair that minimizes the number of XBIs present in a page. To the best of our knowledge, our approach is the first automated technique for generating XBI repairs, and the first to apply search-based repair techniques to web pages. We implemented our approach as a tool, $\mathcal{X}$Fix, and evaluated it on 15 real world web pages containing layout XBIs. $\mathcal{X}$Fix was able to resolve 86% of the XBIs reported by X-PERT [42], a well-known XBI detection tool, and 99% of the XBIs observed by humans. Our results therefore demonstrate that our approach is potentially of high use to developers by providing automated fixes for problematic web pages involving layout XBIs.

The main contributions of this paper are as follows:

(1) A novel approach for automatically finding potential fixes for layout XBIs using guided search-based techniques.
(2) An extensive evaluation on a set of 15 real-world web pages, in which our approach resolved 86% of automatically detected XBIs and 99% observed by human subjects.
(3) A human study to assess the web pages' cross-browser consistency after repair by our approach.
(4) A study to compare the size similarity of our repair patches to XBI-addressing code in real-world web pages.

The rest of this paper is organized as follows: In Section 2, we give background information about web page rendering and introduce an illustrative example. We then present our approach in Section 3 and discuss its evaluation in Section 4. We discuss related work in Section 5 and summarize in Section 6.

2 BACKGROUND AND EXAMPLE

In this section we provide background information that details why layout XBIs occur, what the common practices are to repair them, and introduces an illustrative example.

Basic Terminology. Modern web applications typically follow the "Model-View-Controller (MVC)" design pattern in which the application code (the "Model" and "Controller") runs on a server accessible via the Internet and delivers HTML and CSS-based web pages (the "View") to a client running a web browser. The *layout engine* in a web browser is responsible for rendering and displaying the web pages. When a web browser receives a web page, the layout engine parses its HTML into a data structure called a Document Object Model (DOM) tree. Each HTML element may be referenced in the DOM tree using a unique expression, called an "XPath". To render a DOM tree, the layout engine calculates each DOM element's bounding box and applicable style properties based on the Cascading Style Sheets (CSS) style rules pertaining to the web page. A bounding box gives the physical display location and size of an HTML element on the browser screen.

Layout XBIs. Inconsistencies in the way browsers interpret the semantics of the DOM and CSS can cause layout XBIs — differences in the rendering of an HTML page between two or more browsers. These inconsistencies tend to arise from different interpretations of the HTML and CSS specifications, and are not per se, faults in the browsers themselves [1]. Additionally, some browsers may implement new CSS properties or existing properties differently in an attempt to gain an advantage over competing browsers [30].

Fixing Layout XBIs. When a layout XBI has been detected, developers may employ several strategies to adjust its appearance. For example, changing the HTML structure, replacing unsupported HTML tags, or adjusting the page's CSS. Our approach targets XBIs that can be resolved by finding alternate values for a page's CSS properties. There are two significant challenges to carrying out this type of repair. First, the appearance (e.g., size, color, font style) of any given set of HTML elements in a browser is controlled by a series of complex interactions between the page's HTML elements and CSS properties, which means that identifying the HTML elements responsible for the XBI is challenging. Second, assuming that the right set of elements can be identified, each element may have dozens of CSS properties that control its appearance, position, and layout. Each of these properties may range over a large domain. This makes the process of identifying the correct CSS properties to modify and the correct alternate values for those properties a labor intensive task.

Once the right alternate values are identified, developers can use browser-specific CSS qualifiers to ensure that they are used at runtime. These qualifiers direct the layout engine to use the provided alternate values for a CSS property when it is rendered on a specific browser [5, 58]. This approach is widely employed by developers. In our analysis of the top 480 websites (see Section 4), we found that 79% employed browser-specific CSS to ensure a consistent cross browser appearance. In fact, web developers typically maintain an extensive list of browser specific styling conditions [5] to address the most common XBIs.

Example XBI and Repair. Figure 1 shows screenshots of the menu bar of one of our evaluation subjects, IncredibleIndia, as rendered in Internet Explorer (IE) (Figure 1a) and Firefox (Figure 1b). As can be seen, an XBI is present in the menu bar, where the text of the navigational links is unreadable in the Firefox browser (Figure 1b).

An excerpt of the HTML and CSS code that defines the navigation bar is shown in Listing 1. To resolve the XBI, an appropriate value for the `margin-top` or `padding-top` CSS property needs to

be found for the HTML element corresponding to the navigation bar to push it down and into view. In this instance, the fix is to add "`margin-top: 1.7%`" to the CSS for the Firefox version. The inserted browser-specific code is shown in the red box in Listing 1. The "`-moz`" prefixed selector declaration directs the layout engine to only use the included value if the browser type is Firefox (i.e., Mozilla), and other browsers' layout engines will ignore this code.

```
1  <style>
2      .menubar {
3          position: relative;
4      }
5
6    @-moz-document url-prefix("") {
7       .menubar {
8         margin-top: 1.7%;
9       }
10   }
11
12 </style>
13 <body>
14     <div class="menubar">
15         ...
16     </div>
17 </body>
```

Listing 1: HTML and CSS excerpt of the IncredibleIndia example shown in Figure 1. The highlighted section (lines 6–10) represents the fix added to the CSS to address the XBI.

This particular example was chosen because the fix is straightforward and easy to explain. However, most XBIs are much more difficult to resolve. Typically multiple elements may need to be adjusted, and for each one multiple CSS properties may also need to be modified. A fix itself may introduce new XBIs, meaning that several alternate fixes may need to be considered.

3 APPROACH

The goal of our approach is to find potential fixes that can repair the layout XBIs detected in a web page. Layout XBIs[1] result in the inconsistent placement of UI elements in a web page across different browsers. The placement of a web page's UI elements is controlled by the page's HTML elements and CSS properties. Therefore to resolve the XBIs, our approach attempts to find new values for CSS properties that can make the faulty appearance match the correct appearance as closely as possible.

Formally, XBIs are due to one or more HTML-based *root causes*. A root cause is a tuple $\langle e, p, v \rangle$, where e is an HTML element in the page, p is a CSS property of e, and v is the value of p. Given a set of XBIs X for a page *PUT* and a set of potential root causes, our approach seeks to find a set of fixes that resolve the XBIs in X. We define a *fix* as a tuple $\langle r, v' \rangle$, where r is a root cause and v' is the suggested new value for p in the root cause r. We refer to a set of XBI-resolving fixes as a *repair*.

Our approach generates repairs using guided search-based techniques [9, 17]. Two aspects of the XBI repair problem motivate this choice of technique. The first is that the number of possible repairs is very large, since there can be multiple XBIs present in a page, each of which may have several root causes, and for which the relevant CSS properties range over a large set of possible values. Second, fixes made for one particular XBI may interfere with those for another, or, a fix for any individual XBI may itself cause additional XBIs, requiring a tradeoff to be made among possible fixes. Search-based techniques are ideal for this type of problem because they can explore large solution spaces intelligently and efficiently, while also identifying solutions that effectively balance a number of competing constraints. Furthermore, the visual manifestation of XBIs also lends itself to quantification via a fitness function, which is a necessary element for a search-based technique. A *fitness function* computes a numeric assessment of the "closeness" of candidate solutions found during the search to the solution ultimately required. Our insight is that a good fitness function can be built that leverages a measurement of the number of XBIs detected in a *PUT*, by using well-known XBI detection techniques, and the similarity of the layout of the *PUT* when rendered in the reference and test browsers, by comparing the size and positions of the bounding boxes of the HTML elements involved in each XBI identified.

Our approach works by first detecting XBIs in a page and identifying a set of possible root causes for those XBIs. Then our approach utilizes two phases of guided search to find the best repair. The first search takes the CSS property of each root cause and tries to find a new value for it that is most optimal with respect to the fitness function. This optimized property value is referred to as a *candidate fix*. The second search then seeks to find an optimal combination of candidate fixes identified in the first phase. This additional search is necessary since not all candidate fixes may be required, as the CSS properties involved may have duplicate or competing effects. For instance, the CSS properties `margin-top` and `padding-top` may both be identified as root causes for an XBI, but can be used to achieve similar outcomes — meaning that only one may actually need to be included in the repair. Conversely, other candidate fixes may be required to be used in combination with one another to fully resolve an XBI. For example, an HTML element may need to be adjusted for both its `width` *and* `height`. Furthermore, candidate fixes produced for one XBI may have knock-on effects on the results of candidate fixes for other XBIs, or even introduce additional and unwanted XBIs. By searching through different combinations of candidate fixes, the second search aims to produce a suitable subset — a repair — that resolves as many XBIs as possible for a page when applied together.

We now introduce the steps of our approach in more detail, beginning with an overview of the complete algorithm.

3.1 Overall Algorithm

The top level algorithm of our approach is shown by Algorithm 1. Three inputs are required: the page under test, *PUT*, which exhibits XBIs. The *PUT* is obtained via a URL that points to a location on the file system or network that provides access to all of the necessary HTML, CSS, Javascript, and media files for rendering *PUT*. The second input is the reference browser, R, that shows the correct rendering of *PUT*. The third input is the test browser, T, in which the rendering of *PUT* shows XBIs with respect to R. The output of our approach is a page, *PUT*′, a repaired version of *PUT*.

The overall algorithm, shown by Algorithm 1, comprises five stages, as shown by the overview diagram in Figure 2.

Stage 1 — Initial XBI Detection. The initial part of the algorithm (lines 1–4) involves obtaining the set of XBIs X when *PUT* is rendered in R and T. To identify XBIs, we use the X-PERT tool [42],

[1]Hereafter, we refer to layout XBIs as simply XBIs.

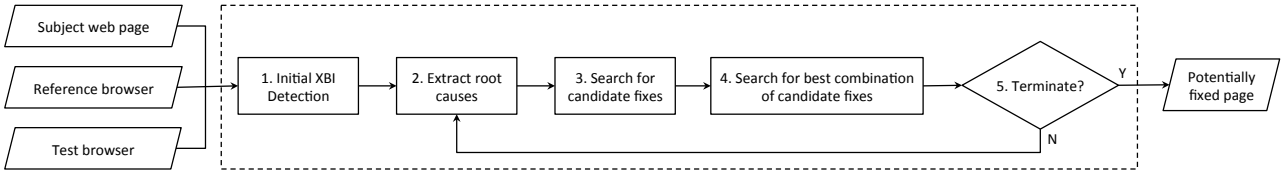


Figure 2: An overview of our search-based XBI repair approach, as detailed by Algorithm 1

which is represented by the "getXBIs" function called on line 2. X-PERT returns a set of identified XBIs, X, in which each XBI is represented by a tuple of the form $\langle label, \langle e_1, e_2 \rangle \rangle$, where e_1 and e_2 are the XPaths of the two HTML elements of the *PUT* that are rendered differently in T versus R, and *label* is a descriptor that denotes the original (correct) layout position of e_1 that was violated in T. For example, $\langle$*top-align*$, e_1, e_2 \rangle$ indicates that e_1 is pinned to the top edge of e_2 in R, but not in T. After identifying the XBIs, the algorithm then enters its main loop, which comprises Stages 2–5.

Stage 2 — Extract Root Causes. The second stage of the algorithm (lines 6–16) extracts the root causes relevant to each XBI. The key step in this stage identifies CSS properties relevant to the XBI's label (shown as "getCSSProperties" at line 9). For example, for the *top-align* label, the CSS properties `margin-top` and `top` can alter the top alignment of an element with respect to another and would therefore be identified in this stage. We identified this mapping through analysis of the CSS properties and it holds true for all web applications without requiring developer intervention. Each relevant CSS property forms the basis of two root causes, one for e_1, and one for e_2. These are added to the running set *rootCauses*, with the values of the CSS properties extracted for each element (v_1 and v_2 respectively) extracted from the DOM of the *PUT* when it is rendered in T (lines 11 and 13).

Stage 3 — Search for Candidate Fixes. Comprising the first phase search, this stage produces individual candidate fixes for each root cause (lines 17–22). The fix is a new value for the CSS property that is optimized according to a fitness function, with the aim of producing a value that resolves, or is as close as possible to resolving the layout deviation. This optimization process occurs in the "searchForCandidateFix" procedure, which we describe in detail in Section 3.2.

Stage 4 — Search for the Best Combination of Candidate Fixes. Comprising the second phase search, the algorithm makes a call to the "searchForBestRepair" procedure (line 24) that takes the set of candidate fixes in order to find a subset, *repair*, representing the best overall repair. We describe this procedure in Section 3.3.

Stage 5 — Check Termination Criteria. The final stage of the algorithm (lines 25–36) determines whether the algorithm should terminate or proceed to another iteration of the loop and two-phase search. Initially, the fixes in the set *repair* are applied to a copy of *PUT* by adding test browser (T) specific CSS code to produce a modified version of the page PUT' (line 26). The approach identifies the set of XBIs, X' for PUT', with another call to the "getXBIs" function (line 27).

Ideally, all of the XBIs in *PUT* will have been resolved by this point, and X' will be empty. If this is the case, the algorithm returns the repaired page PUT'. If the set X' is identical to the original set of XBIs X (originally determined on line 2), the algorithm has made no improvement in this iteration of the algorithm, and so the PUT' is returned, having potentially only been partially fixed as a result of the algorithm rectifying a subset of XBIs in a previous iteration of the loop.

If the number of XBIs has increased, the current repair introduces further layout deviations. In this situation, *PUT* is returned (which may reflect partial fixes from a previous iteration of the loop, if there were any). However, if the number of XBIs has been reduced, the current repair represents an improvement that may be improved further in another iteration of the algorithm.

Broadly, there are two scenarios under which our approach could fail: (1) X-PERT does not initially include the faulty HTML element in X; or (2) the search does does not identify an acceptable fix, which could happen due to the non-determinism of the search.

Algorithm 1 Overall Algorithm

```
Input:  PUT: Web page under test
        R: Reference browser
        T: Test browser
Output: PUT′: Modified PUT with repair applied
1:  /* Stage 1 — Initial XBI Detection */
2:  X ← getXBIs(PUT, R, T)
3:  DOM_R ← buildDOMTree(PUT, R)
4:  DOM_T ← buildDOMTree(PUT, T)
5:  while true do
6:      /* Stage 2 — Extract root causes */
7:      rootCauses ← {}
8:      for each ⟨label, ⟨e1, e2⟩⟩ ∈ X do
9:          props ← getCSSProperties(label)
10:         for each p ∈ props do
11:             v1 ← getValue(e1, p, DOM_T)
12:             rootCauses ← rootCauses ∪ ⟨e1, p, v1⟩
13:             v2 ← getValue(e2, p, DOM_T)
14:             rootCauses ← rootCauses ∪ ⟨e2, p, v2⟩
15:         end for
16:     end for
17:     /* Stage 3 — Search for Candidate Fixes */
18:     candidateFixes ← {}
19:     for each ⟨e, p, v⟩ ∈ rootCauses do
20:         candidateFix ← searchForCandidateFix(⟨e, p, v⟩, PUT, DOM_R, T)
21:         candidateFixes ← candidateFixes ∪ candidateFix
22:     end for
23:     /* Stage 4 — Search for Best Combination of Candidate Fixes */
24:     repair ← searchForBestRepair(candidateFixes, PUT, R, T)
25:     /* Stage 5 — Check Termination Criteria */
26:     PUT′ ← applyRepair(PUT, repair)
27:     X′ ← getXBIs(PUT′, R, T)
28:     if X′ = ∅ or X′ = X then
29:         return PUT′
30:     else if |X′| > |X| then
31:         return PUT
32:     else
33:         X ← X′
34:         PUT ← PUT′
35:         DOM_T ← buildDOMTree(PUT′, T)
36:     end if
37: end while
```

3.2 Search for Candidate Fixes

The first search phase (represented as the procedure "searchForCandidateFix") focuses on each potential root cause $\langle e, p, v \rangle$ in isolation of the other root causes, and attempts to find a new value v' for the

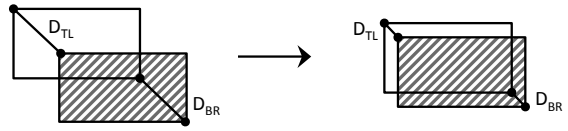
root cause that improves the similarity of the page when rendered in the reference browser R and the test browser T. Guidance to this new value is provided by a fitness function that quantitatively compares the relative layout discrepancies between e and the elements that surround it when PUT is rendered in R and T. We begin by giving an overview of the search algorithm used, and then explain the fitness function employed.

Algorithm 2 Fitness Function for Candidate Fixes

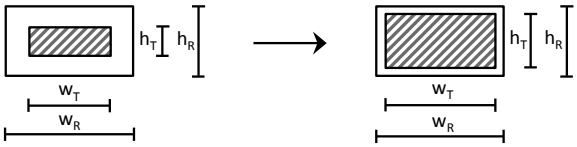
Input: e: XPath of HTML element under analysis
p: CSS property of HTML element, e
$\hat{v}$: Value of CSS property, p
PUT: Web page under test
DOM_R: DOM tree of PUT rendered in R
T: Test browser
Output: *fitness*: Fitness value of the hypothesized fix $\langle e, p, \hat{v} \rangle$

1: $\widehat{PUT} \leftarrow$ applyValue(e, p, $\hat{v}$, PUT)
2: $DOM_T \leftarrow$ buildDOMTree($\widehat{PUT}$, T)
3: /* Component 1 — Difference in location of e with respect to R and T */
4: $\langle x_1^t, y_1^t, x_2^t, y_2^t \rangle \leftarrow$ getBoundingBox(DOM_T, e)
5: $\langle x_1^r, y_1^r, x_2^r, y_2^r \rangle \leftarrow$ getBoundingBox(DOM_R, e)
6: $D_{TL} \leftarrow \sqrt{(x_1^t - x_1^r)^2 + (y_1^t - y_1^r)^2}$
7: $D_{BR} \leftarrow \sqrt{(x_2^t - x_2^r)^2 + (y_2^t - y_2^r)^2}$
8: $\Delta_{pos} \leftarrow D_{TL} + D_{BR}$
9: /* Component 2 — Difference in size of e with respect to R and T */
10: $width_R \leftarrow x_2^r - x_1^r$
11: $width_T \leftarrow x_2^t - x_1^t$
12: $height_R \leftarrow y_2^r - y_1^r$
13: $height_T \leftarrow y_2^t - y_1^t$
14: $\Delta_{size} \leftarrow |width_R - width_T| + |height_R - height_T|$
15: /* Component 3 — Differences in locations of neighboring elements of e */
16: $neighbors_T \leftarrow$ getNeighbors(e, DOM_T, N_r)
17: $\Delta_{npos} \leftarrow 0$
18: **for each** $n \in neighbors_T$ **do**
19: $n' \leftarrow$ getMatchingElement(n, DOM_R)
20: $\langle x_1^t, y_1^t, x_2^t, y_2^t \rangle \leftarrow$ getBoundingBox(DOM_T, n)
21: $\langle x_1^r, y_1^r, x_2^r, y_2^r \rangle \leftarrow$ getBoundingBox(DOM_R, n')
22: $D_{TL} \leftarrow \sqrt{(x_1^t - x_1^r)^2 + (y_1^t - y_1^r)^2}$
23: $D_{BR} \leftarrow \sqrt{(x_2^t - x_2^r)^2 + (y_2^t - y_2^r)^2}$
24: $\Delta_{pos} \leftarrow D_{TL} + D_{BR}$
25: $\Delta_{npos} \leftarrow \Delta_{npos} + \Delta_{pos}$
26: **end for**
27: /* Compute final fitness value */
28: *fitness* $\leftarrow (w_1 * \Delta_{pos}) + (w_2 * \Delta_{size}) + (w_3 * \Delta_{npos})$
29: **return** *fitness*

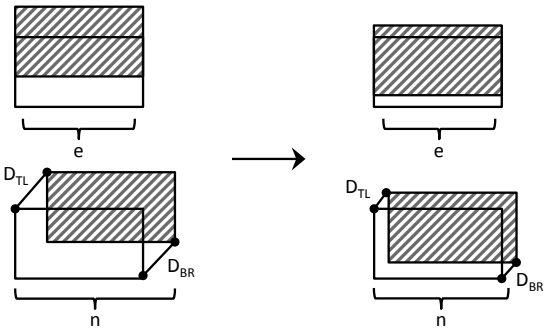
3.2.1 Search Algorithm. The inputs to the search for a candidate fix are the page under test, PUT, the test browser, T, the DOM tree from the reference browser, DOM_R, and the root cause tuple, $\langle e, p, v \rangle$. The search attempts to find a new value, v', for p in the root cause. The search process used to do this is inspired by the variable search component of the *Alternating Variable Method (AVM)* [18, 19], and specifically the use of "exploratory" and "pattern" moves to optimize variable values. The aim of exploratory moves is to probe values neighboring the current value of v to find one that improves fitness when evaluated with the fitness function. Exploratory moves involve adding small delta values (i.e., [-1 ,1]) to v and observing the impact on the fitness score. If the fitness is observed to be improved, pattern moves are made in the same "direction" as the exploratory move to accelerate further fitness improvements through step sizes that increase exponentially. If a pattern move fails to improve fitness, the method establishes a new direction from the current point in the search space through further exploratory moves. If exploratory moves fail to yield a new direction (i.e., a local optima had been found), this value is returned as the best candidate fix value. The fix tuple, $\langle e, p, v, v' \rangle$, is then returned to the main algorithm (line 20 of Algorithm 1).



(a) **Component 1: Δ_{pos} = D_{TL} + D_{BR}, where D_{TL} and D_{BR} is the Euclidean distance between the top left (TL) and bottom right (BR) corners, respectively, of e rendered in R and T. Δ_{pos} decreases as the boxes move closer.**



(b) **Component 2: Δ_{size} = $|w_R - w_T| + |h_R - h_T|$, where w_R and h_R are the respective width and height of e rendered in R, and w_T and h_T are the respective width and height of e rendered in T. Δ_{size} decreases as the boxes become similar in size.**



(c) **Component 3: Δ_{npos} = D_{TL} + D_{BR}, where D_{TL} and D_{BR} is the Euclidean distance between the top left (TL) and bottom right (BR) corners, respectively, of e's neighbor n rendered in R and T. Δ_{npos} decreases as e's boxes move closer, which causes n's boxes to also move closer.**

Figure 3: Diagrammatic representation of the fitness function components. Rectangles with a solid background correspond to the bounding boxes of elements rendered in R and the rectangles with diagonal lines correspond to the bounding boxes of elements rendered in T.

3.2.2 Fitness Function. The fitness function for producing a candidate fix is shown by Algorithm 2. The goal of the fitness function is to quantify the relative layout deviation for PUT when rendered in R and T following the change to the value of a CSS property for an HTML element. Given the element e in PUT, the fitness function considers three components of layout deviation between the two browsers: (1) the difference in the location of e; (2) the difference in the size of e; and (3) any differences in the location of e's neighbors. Figure 3 shows a diagrammatic representation of these components. Intuitively, all three components should be minimized as the evaluated fixes make progress towards resolving an XBI without introducing any new differences or introducing further XBIs for e's neighbors. The fitness function for an evaluated fix is the weighted sum of these three components.

The first component, location difference of e, is computed by lines 3–8 of Algorithm 2, and assigned to the variable Δ_{pos}. This value is calculated as the sum of the Euclidean distance between the top-left (TL) and bottom-right (BR) corners of the bounding box of e when it is rendered in R and T. The bounding box is obtained from the DOM tree of the page for each browser.

The second component, difference in size of e, is calculated by lines 10–14 of the algorithm, and is assigned to the variable Δ_{size}. The value is calculated as the sum of the differences of e's width and height when rendered in R and T. The size information is obtained from the bounding box of e obtained from the DOM tree of the page in each browser.

The third and final component of the fitness function, finding the location difference of e's neighbors occurs on lines 16–26 of the algorithm, and is assigned to the variable Δ_{npos}. The neighbors of e are the set of HTML elements that are within N_r hops from e in PUT's DOM tree as rendered in T. For example, if $N_r = 1$, then the neighbors of e are its parents and children. If $N_r = 2$, then the neighbors are its parent, children, siblings, grandparent, and grandchildren. For each neighbor, the approach finds its corresponding element in the DOM tree of PUT rendered in R and calculates Δ_{pos} for each pair of elements.

The final fitness value is then formed from the weighted sum of the three components Δ_{pos}, Δ_{size}, and Δ_{npos} (line 28).

3.3 Search for the Best Combination of Candidate Fixes

The goal of the second search phase (represented by a call to "searchForBestRepair" at line 24 of Algorithm 1) is to identify a subset of *candidateFixes* that together minimize the number of XBIs reported for the PUT. This step is included in the approach for two reasons. Firstly, a fix involving one particular CSS property may only be capable of partially resolving an XBI and may need to be combined with another fix to fully address the XBI. Furthermore, the interaction of certain fixes may have emergent effects that result in further unwanted layout problems. For example, suppose a submit button element appears below, rather than to the right of a text box. Candidate fixes will address the layout problem for each HTML element individually, attempting to move the textbox down and to the left, and the button up and to the right. Taking these fixes together will result in the button appearing to the top right corner of the text box, rather than next to it. Identifying a selection of fixes, a candidate repair, that avoids these issues is the goal of this phase. To guide this search, we use the number of XBIs that appear in the PUT after the candidate repair has been applied.

The search begins by evaluating a candidate repair with a single fix — the candidate fix that in the first search phase produced the largest fitness improvement. Assuming this does not eradicate all XBIs, the search continues by generating new candidate repairs in a biased random fashion. Candidate repairs are produced by iterating through the set of fixes. A fix is included in the repair with a probability imp_{fix}/imp_{max}, where imp_{fix} is the improvement observed in the fitness score when the fix was evaluated in the first search phase divided by the maximum improvement observed over all of the fixes in *candidateFixes*. Each candidate repair is evaluated for fitness in terms of the number of resulting XBIs, with the best repair retained. A history of evaluated repairs is maintained, so that any repeat solutions produced by the biased random generation algorithm are not re-evaluated.

The random search terminates when (a) a candidate repair is found that fixes all XBIs, (b) a maximum threshold of candidate repairs to be tried has been reached, or (c) the algorithm has produced a sequence of candidate repairs with no improvement in fitness.

4 EVALUATION

We conducted empirical experiments to assess the effectiveness and efficiency of our approach, with the aim of answering the following four research questions:

RQ1: How effective is our approach at reducing layout XBIs?

RQ2: What is the impact on the cross-browser consistency of the page when the suggested repairs are applied?

RQ3: How long does our approach take to find repairs?

RQ4: How similar in size are our approach generated repair patches to the browser-specific code present in real-world websites?

4.1 Implementation

We implemented our approach in a prototype tool in Java, which we named "$\mathcal{X}$Fix" [13]. We leveraged the `Selenium WebDriver` library for making dynamic changes to web pages, such as applying candidate fix values. For identifying the set of layout XBIs, we used the latest publicly available version [7] of the well-known XBI detection tool, X-PERT [42, 43]. We made minor changes to the publicly available version to fix bugs and add accessor methods for data structures. We used this modified version throughout the rest of the evaluation. The fitness function parameters for the search of candidate fixes discussed in Section 3.2.2 are set as: $N_r = 2$, and $w_1 = 1$, $w_2 = 2$, and $w_3 = 0.5$ for the weights for Δ_{pos}, Δ_{size}, and Δ_{npos} respectively. (The weights assigned prioritize Δ_{size}, Δ_{pos} and Δ_{npos} in that order. We deemed size of an element as most important, because of its likely impact on all three components, followed by location, which is likely to impact its neighbors.) For the termination conditions (b) and (c) of the search for the best combination of candidate fixes (Section 3.3), the maximum threshold value is set to 50 and the sequence value is set to 10. More implementation details about $\mathcal{X}$Fix are available in our tool demo paper [24].

4.2 Subjects

For our evaluation we used 15 real-world subjects as listed by Table 1. The columns labeled "#HTML" and "#CSS" report the total number of HTML elements present in the DOM tree of a subject, and the total number of CSS properties defined for the HTML elements in the page respectively. These metrics of size give an estimate of a page's complexity in debugging and finding potential fixes for the observed XBIs. The "Ref" column indicates the reference browser in which the subject displays the correct layout; while the column "Test" refers to the browser in which the subject shows a layout XBI. In these columns, "CH", "FF", and "IE" refer to the Chrome, Firefox, and Internet Explorer browsers respectively.

We collected the subjects from three sources: (1) websites used in the evaluation of X-PERT [42], (2) the authors' prior interaction with websites exhibiting XBIs, and (3) the random URL generator,

Table 1: Subjects

Name	URL	#HTML	#CSS	Ref	Test
BenjaminLees	http://www.benjaminlees.com	317	1,525	CH	FF
Bitcoin	https://bitcoin.org/en/	207	1,957	FF	IE
Eboss	http://www.e-boss.gr	439	789	IE	FF
EquilibriumFans	http://www.equilibriumfans.com	340	868	CH	FF
GrantaBooks	http://grantabooks.com	325	6,545	FF	IE
HenryCountyOhio	http://www.henrycountyohio.com	300	983	IE	FF
HotwireHotel	https://goo.gl/pH9d6d	1,457	10,618	FF	IE
IncredibleIndia	http://incredibleindia.org	251	2,172	IE	FF
Leris	http://clear.uconn.edu/leris/	195	1,262	FF	CH
Minix3	http://www.minix3.org	118	821	IE	CH
Newark	http://www.ci.newark.ca.us	598	17,426	FF	IE
Ofa	http://www.ofa.org	578	5,381	IE	CH
PMA	http://www.pilatesmethodalliance.org	456	10,159	FF	IE
StephenHunt	http://stephenhunt.net	497	13,743	FF	IE
WIT	http://www.wit.edu	300	3,249	FF	IE

UROULETTE [53]. The "GrantaBooks" subject came from the first source. The other subjects from X-PERT's evaluation could not be used because their GUI had been reskinned or the latest version of the IE browser now rendered the pages correctly. The "HotwireHotel" subject was chosen from the second source, and the remaining thirteen subjects were gathered from the third source.

The goal of the selection process was to select subjects that exhibited human perceptible layout XBIs. We did not use X-PERT for an initial selection of subjects because we found that it reported many subjects with XBIs that were difficult to observe. For selecting the subjects, we used the following process: (1) render the page, *PUT*, in the three browser types; (2) visually inspect the rendered *PUT* in the three browsers to find layout XBIs; (3) if layout XBIs were found in the *PUT*, select the browser showing a layout problem, such as overlapping, wrapping, or distortion of content, as the test browser, and one of the other two browsers showing the correct rendering as the reference browser; (4) try to manually fix the *PUT* by using the developer tools in browsers, such as Firebug for Firefox, and record the HTML elements to which the fix was applied; (5) run X-PERT on the *PUT* with the selected reference and test browsers; and (6) use the *PUT* as a subject, if the manually recorded fixed HTML elements were present in the set of elements reported by X-PERT. We included steps 4–6 in the selection process to ensure that if X-PERT reported false negatives, they would not bias our evaluation results.

4.3 Methodology

For the experiments, the latest stable versions of the browsers, Mozilla Firefox 46.0.1, Internet Explorer 11.0.33, and Google Chrome 51.0, were used. These browsers were selected for the evaluation as they represent the top three most widely used desktop browsers [36, 50]. The experiments were run on a 64-bit Windows 10 machine with 32GB memory and a 3rd Generation Intel Core i7-3770 processor. Since the set of XBIs reported by X-PERT can vary based on screen resolution, we also report our test monitor setup, which had a resolution of 1920×1080 and size of 23 inches. The subjects were rendered in the browsers with the browser viewport size set to the screen size.

Each subject was downloaded using the Scrapbook-X Firefox plugin and the `wget` utility, which download an HTML page along with all of the files (e.g., CSS, JavaScript, images, etc.) it needs to display. We then commented out portions of the JavaScript files and HTML code that made active connections with the server, such as Google Analytics, so that the subjects could be run locally in an offline mode. The downloaded subjects were then hosted on a local Apache web server.

We ran X-PERT on each of the subjects to collect the set of initial XBIs present in the page. We then ran $\mathcal{X}$Fix 30 times on each of the subjects to mitigate non-determinism in the search, and measured the run time in seconds. After each run of $\mathcal{X}$Fix on a subject, we ran X-PERT on the repaired subject and recorded the remaining number of XBIs reported, if any.

We also conducted a human study with the aim of judging $\mathcal{X}$Fix with respect to the human-perceptible XBIs, and to gauge the change in the cross-browser consistency of the repaired page. Our study involved 11 participants consisting of PhD and post-doctoral researchers whose field of study was Software Engineering. For the study, we first captured three screenshots of each subject page: (1) rendered in the *reference* browser, (2) rendered in the test browser *before* applying $\mathcal{X}$Fix's suggested repair, and (3) rendered in the test browser *after* applying the suggested fixes. We embedded these screenshots in HTML pages provided to the participants. We varied the order in which the before (pre-$\mathcal{X}$Fix) and after (post-$\mathcal{X}$Fix) versions were presented to participants, to minimize the influence of learning on the results and referred to them in the study as $version_1$ and $version_2$ based on the order of their presentation.

Each participant received a link to an online questionnaire and a set of printouts of the renderings of the page. We instructed the participants to individually (i.e., without consultation) answer four questions per subject: The first question asked the users to compare the *reference* and $version_1$ by opening them in different tabs of the same browser and circle the areas of observed visual differences on the corresponding printout. The second question asked the participants to rate the similarity of $version_1$ and *reference* on a scale of 0–10, where 0 represents no similarity and 10 means identical. Note that the similarity rating includes the participants reaction to intrinsic browser differences as well since we did not ask them to exclude these. The third and fourth questions in the questionnaire were the same, but for $version_2$.

For RQ1, we used X-PERT to determine the initial number of XBIs in a subject and the average number of XBIs remaining after each of the 30 runs of $\mathcal{X}$Fix. From these numbers we calculated the *reduction of XBIs* as a percentage.

For RQ2, we classified the similarity rating results from the human study into three categories for each subject: (1) **improved**: the *after* similarity rating was higher than that of the *before* version, (2) **same**: the *after* and *before* similarity ratings were exactly the same, and (3) **decreased**: the *after* similarity rating was lower than that of the *before* version. The human study data can be found at the project website [13].

For RQ3, we collected the average total running times of $\mathcal{X}$Fix and for Stages 3 and 4, the search phases, of our algorithm.

For RQ4, we compared the size, measured by the number of CSS properties, of browser specific code found in real-world websites to that of our automatically generated repairs. We used size for comparing similarity because CSS has a simple structure and does not contain any branching or looping constructs. We used `wget` to download the homepages of 480 websites in the Alexa Top 500 Global Sites [3] and analyzed their CSS to find the number of websites containing browser specific code. Twenty sites could not be downloaded as they pointed to URLs without UIs — for instance

Table 2: Number of XBIs reported by X-PERT

Subject	#Before XBIs	Avg. #After XBIs	Reduction (%)
BenjaminLees	25	0	100
Bitcoin	37	0	100
Eboss	49	29	41
EquilibriumFans	117	6	95
GrantaBooks	16	0	100
HenryCountyOhio	11	0	100
HotwireHotel	40	4	90
IncredibleIndia	20	12	40
Leris	13	0	100
Minix3	11	0.73	93
Newark	42	2	95
Ofa	16	3	83
PMA	39	10	75
StephenHunt	159	33	79
WIT	40	3	92
Mean	42	7	86
Median	37	3	93

Table 3: Average run time in seconds

Subject	Search for Candidate Fixes	Search for Best Combination	Total
BenjaminLees	159	14	204
Bitcoin	144	42	358
Eboss	1,729	780	2,685
EquilibriumFans	822	225	1,208
GrantaBooks	41	7	86
HenryCountyOhio	219	41	291
HotwireHotel	3,281	2,036	5,582
IncredibleIndia	599	247	908
Leris	105	46	169
Minix3	18	6	43
Newark	477	232	841
Ofa	122	113	257
PMA	3,050	1,384	4,488
StephenHunt	5,535	1,114	6,639
WIT	3,725	1,409	4,980
Mean	369	90	1916
Median	194	48	841

the `googleadservices.com` and `twimg.com` web services. To find whether a website has browser specific CSS, we parsed its CSS files using the CSS Parser tool [48] and searched for browser specific CSS selectors, such as the one shown in Listing 1, based on well-known prefix declarations: `-moz` for Firefox, `-ms` for IE, and `-webkit` for Chrome. To calculate the size, we summed the numbers of CSS properties declared in each browser specific selector. To establish a comparable size metric for each subject web page used with $\mathcal{X}$Fix, we added the size of each subject's previously existing browser specific code for T, the test browser, to the average size of the repair generated for T.

4.4 Threats to Validity

External Validity: The first potential threat is that we used a manual selection of the subjects. To minimize this threat, we only performed a manual filtering of the subjects to ensure that the subjects showed human perceptible XBIs and that X-PERT did not miss the observed XBIs (i.e., have a false negative). We also selected subjects from three different sources, including a random URL generator, to make the selection process generalizable across a wide variety of subjects. All our subjects had multiple XBIs reported by X-PERT (Table 2), and a mix of single (e.g., Bitcoin and IncredibleIndia) and multiple (e.g., HotwireHotel and Grantabooks) human-observable XBIs. A second potential threat is the use of only three browsers. To mitigate this threat, we selected the three most widely used browsers, as reported by different commercial agencies studying browser statistics [36, 50]. Furthermore, our approach is not dependent on the choice of browsers, so our results should generalize to other browsers.

Internal Validity: One potential threat is the use of X-PERT. However, there are no other publicly available tools for detecting XBIs that report the level of detail required by $\mathcal{X}$Fix to produce repairs. A further threat is represented by the changes we made to X-PERT favored our approach. However, the changes made were to provide access to existing information (and so do not change XBI-identifying behavior) or to address specific bugs. An example of one of the defects we found was a mismatch in the data type of a `DomNode` object being checked to see if it is contained in an array of `String` specifying the HTML tags to be ignored. We corrected this defect by adding a call to the `getTagName()` method of the `DomNode` object that returns the `String` HTML tag name of the node. We have made our patched version of X-PERT publicly available [13], with the download containing a `README.txt` file detailing the defects that were corrected.

The fact that the authors' judgment was used to determine which browser rendering was the reference is not a threat to validity. This is because the metrics used were relative comparisons (e.g., consistency) and flipping the choice of reference rendering would have produced the same difference. Human participant understanding as to what constituted an XBI was not a threat to the correctness of our protocol either since we only asked them to spot differences between the renderings.

A potential threat is the number of real-world (Alexa) websites found to be using browser-specific styling. There exist numerous other ways to declare browser specific styling [5, 58] than the simple prefix selector declarations we used, and therefore the number of Alexa websites we found to be using browser-specific styling and the browser-specific code sizes calculated for each only represents a lower bound.

Construct Validity: A potential threat is that the similarity metric used in the human study is subjective. To mitigate this threat we used the relative similarity ratings given by the users, as opposed to the absolute value, to understand the participants' relative notion of consistency quality. A second potential threat to validity is that screenshots of the subjects were used in the human study instead of actual HTML pages. We opted for this mechanism as not all of the users had our required environment (OS and browsers). Also, to mitigate this threat we designed the HTML pages containing the screenshots to scale based on the width of the user's screen. Another potential threat is that the browser-specific code found in real-world (Alexa) websites might not necessarily be repair code for XBIs, so it might not be fair to compare that with our repair patches. However, to the best of our knowledge the primary purpose of browser-specific code is to target a particular browser and ensure cross-browser consistency.

4.5 Discussion of Results

4.5.1 RQ1: Reduction of XBIs. Table 2 shows the results of RQ1. The results show that $\mathcal{X}$Fix reported an average 86% reduction in XBIs, with a median of 93%. This shows that $\mathcal{X}$Fix was effective in finding XBI fixes. Of the 15 subjects, $\mathcal{X}$Fix was able to resolve all of the reported XBIs for 33% of the subjects and was able to resolve more than 90% of the XBIs for 67% of the subjects.

We investigated the results to understand why our approach was not able to find suitable fixes for all of the XBIs. We found that the dominant reason for this was that there were pixel-level differences between the HTML elements in the test and reference browsers that were reported as XBIs. In many cases, perfect matching at the pixel level was not feasible due to the complex interaction among the HTML elements and CSS properties of a web page. Also, the different implementations of the layout engines of the browser meant that a few pixel-level differences were unavoidable. After examining these cases, we hypothesized that these differences would not be human perceptible.

To investigate this hypothesis, we inspected the user-marked printouts of the *before* and *after* versions from the human study. We filtered out the areas of visual differences that represented inherent browser-level differences, such as font styling, font face, and native button appearance, leaving only the areas corresponding to XBIs.

We found that, for all but one subject, the majority of participants had correctly identified the areas containing layout XBIs in the *before* version of the page but had not marked the corresponding areas again in the *after* version. This indicated that the *after* version did not show the layout XBIs after they had been resolved by $\mathcal{X}$Fix. Overall, this analysis showed an average 99% reduction in the human observable XBIs (median 100%), confirming our hypothesis that almost all of the remaining XBIs reported by X-PERT were not actually human observable.

RQ1: $\mathcal{X}$Fix *reduced X-PERT-reported XBIs by a mean average of 86% (median 93%). Human-observable layout XBIs were reduced by a mean of 99% (median 100%).*

4.5.2 RQ2: Impact on Cross-browser Consistency. We calculated the impact of our approach on the cross-browser consistency of a subject based on the user ratings classifications, *improved*, *same*, or *decreased*. We found that 78% of the user ratings reported an *improved* similarity of the *after* version, implying that the consistency of the subject pages had improved with our suggested fixes. 14% of the user ratings reported the consistency quality as *same*, and only 8% of the user ratings reported a *decreased* consistency. Figure 4 shows the distribution of the participant ratings for each of the subjects. As can be seen, all of the subjects, except two (Eboss and Leris), show a majority agreement among the participants in giving the verdict of *improved* cross-browser consistency. The *improved* ratings without considering Eboss and Leris rise to 85%, with the ratings for *same* and *decrease* dropping to 10% and 4%, respectively.

We investigated the two outliers, Eboss and Leris, to understand the reason for high discordance among the participants. We found that the reason for this disagreement was the significant number of inherent browser-level differences related to font styling and font face in the pages. Both of the subject pages are text intensive and contain specific fonts that were rendered very differently by the respective reference and test browsers. In fact, we found that the browser-level differences were so dominant in these two subjects that some of the participants did not even mark the areas of layout XBIs in the *before* version. Since our approach does not suggest fixes for resolving inherent browser-level differences, the judgment of consistency was likely heavily influenced by these differences, thereby causing high disagreement among the users. To further

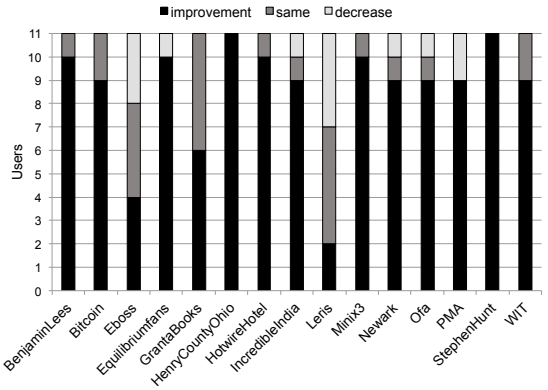


Figure 4: Similarity ratings given by participants in the human study

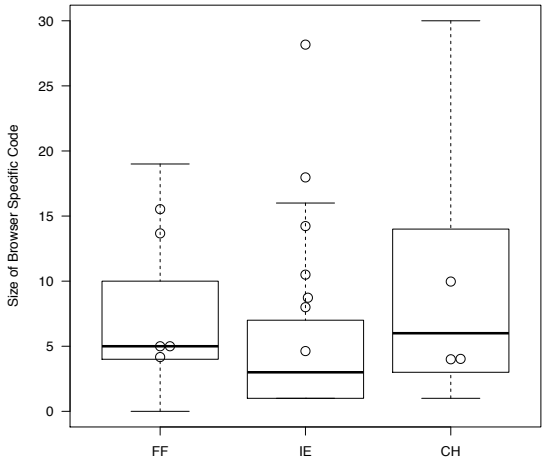


Figure 5: Size of browser specific code observed in real-world (Alexa) websites (shown by boxes) and $\mathcal{X}$Fix subjects (shown by circles)

quantify the impact of the intrinsic browser differences on participant ratings, we controlled for intrinsic differences, as discussed in Section 4.5.1. This controlled analysis showed a mean of 99% reduction in XBIs, a value consistent with the results in Table 2.

RQ2: *78% of participant responses reported an improvement in the cross-browser consistency of pages fixed by* $\mathcal{X}$Fix.

4.5.3 RQ3: Time Needed to Run $\mathcal{X}$Fix. Table 3 shows the average time results over the 30 runs for each subject. These results show that the total analysis time of our approach ranged from 43 seconds to 110 minutes, with a median of 14 minutes. The table also reports time spent in the two search routines. The "searchForCandidateFix" procedure was found to be the most time consuming, taking up 67% of the total runtime, with "searchForBestRepair" occupying 32%. (The remaining 1% was spent in other parts of the overall algorithm, for example the setup stage.) The time for the two search techniques was dependent on the size of the page and the number of XBIs reported by X-PERT. Although the runtime is lengthy for some subjects, it can be further improved via parallelization, as has been achieved in related work [20, 28].

RQ3: $\mathcal{X}$Fix *had a median runtime of 14 minutes to resolve XBIs.*

4.5.4 RQ4: Similarity of Repair Patches to Real-world Websites' Code. Our analysis of the 480 Alexa websites revealed that browser specific code was present in almost 80% of the websites and therefore highly prevalent. This indicates that the patch structure of XFix's repairs, which employs browser specific CSS code blocks, follows a widely adopted practice of writing browser specific code.

Figure 5 shows a box plot for browser specific code size observed in the Alexa websites and XFix subjects. The boxes represent the distribution of browser specific code size for the Alexa websites for each browser (i.e., Firefox (FF), Internet Explorer (IE), and Chrome (CH)), while the circles show the data points for XFix subjects. In each box, the horizontal line and the upper and lower edges show the median and the upper and lower quartiles for the distribution of browser specific code sizes, respectively. As the plot shows, the size of the browser specific code reported by Alexa websites and XFix subjects are in a comparable range, with both reporting an average size of 9 CSS properties across all three browsers (Alexa: FF = 9, IE = 7, CH = 10 and XFix: FF = 9, IE = 13, CH = 6).

RQ4: XFix *generates repair patches that are comparable in size to browser specific code found in real-world websites.*

5 RELATED WORK

Automatic repair of software programs has for long been an area of active research. Several techniques that use search-based algorithms have been proposed. Two examples include GenProg [20, 56], which uses genetic programming to find viable repairs for C programs, and SPR [23], which uses a staged repair strategy to search through a large space of candidate fixes. Alternative analytical approaches also exist, including FixWizard [39], which analyzes bug fixes in a piece of code and suggests comparable fixes to similar parts of the code base; and FlowFixer [62], which repairs sequences of GUI interactions in modified test scripts for Java programs. A group of techniques exists that can detect and repair HTML syntax problems in web applications [38, 45]. However, these techniques cannot find XBIs and repair them. Another technique [55] can automatically repair dynamic web applications for a given presentation change (fix). However, this technique cannot find the fix automatically. To our knowledge, no techniques have been proposed that repair presentation problems, such as XBIs, in web applications.

Simple CSS resetting techniques, such as Normalize CSS [14] and YUI 3 CSS Reset [41], establish a consistent CSS baseline for different browsers to minimize the browser differences that can lead to XBIs. However, such techniques cannot handle complex XBIs that are application dependent and are caused by complex interaction between HTML and CSS. When applied to our 15 evaluation subjects, Normalize CSS and YUI 3 CSS Reset could not fix any of the reported layout XBIs, but rather introduced new layout failures in some of them.

Cross Browser Testing (XBT) techniques, such as X-PERT [8, 42, 44], CrossT [32], Browserbite [47], Browsera [4], and Webmate [12], are effective in detecting XBIs. However, debugging the reported XBIs and finding fixes when using these techniques must still be performed manually. Crossfire [10] presents a protocol for XBI debugging by extending browser developer tools, such as Firefox's Firebug, to enable cross-browser support. However, the task of using the debugger to find potential fixes is developer-driven.

There exist several detection and localization techniques in the field of web app presentation testing. Techniques such as WebSee [25–27] and FieryEye [28, 29], focus on detecting presentation failures — a discrepancy in the actual and intended appearance of a web page — and localizing them to HTML elements and CSS properties in the page. GWALI [2] focuses on detecting presentation failures in internationalized web pages and finding faulty HTML elements. The ReDeCheck technique [54] uses a layout graph to find regression failures in responsive web pages that adjust their layout according to the size of the browser's viewport. However, debugging and finding potential fixes for presentation problems detected by these techniques is still a manual process.

Another technique, Cassius [40], helps debug and repair faulty CSS using automated reasoning. However, it does not specifically focus on repairing XBIs and can only handle repairs for a single browser with different browser settings.

A group of web testing techniques (e.g., Cucumber [11], Sikuli [6, 61], Crawljax [33], Selenium [46], Cornipickle [16]) require developers to manually write test cases or specify invariants to be checked against the application. However, unless developers exhaustively specify a correctness variant for each element and style combination, they cannot be reliably used to localize faults and fix them.

Browser plug-ins, such as "PerfectPixel" [57] for Chrome and "Pixel Perfect" [34] for Firefox, can help developers in detecting XBIs by overlaying a screenshot of the reference browser rendered web page on the test browser. Similarly, the tool, "Fighting Layout Bugs" can be used to automatically find application agnostic XBIs, such as overlapping text. However, the process of finding fixes for such XBIs is still a manual process.

Finally, work in the area of GUI testing by Memon et al. [31, 35, 37, 60] tests the behavior of a software system by triggering event sequences from the GUI. Their work is not focused on fixing presentation issues (e.g., XBIs), in the GUI, but rather on using the GUI as a driver to find behavioral problems in the system.

6 CONCLUSION

In this paper, we introduced a novel search-based approach for repairing layout XBIs in web applications. Our approach uses two phases of guided search. The first phase finds candidate fixes for each of the root causes identified for an XBI. The second phase then finds a subset of the candidate fixes that together minimizes the number of XBIs in the web page. In the evaluation, our approach was able to resolve 86% of the X-PERT reported XBIs and 99% of the human observed XBIs. In a human study assessing the improvement in consistency between the repaired and reference page, 78% of the participant ratings reported an improvement in the cross-browser consistency of the repaired web pages. Our repair patches were comparable in size to the browser-specific code present in real-world websites. Overall, these are strong results and indicate that our approach can be useful and effective in repairing layout XBIs in web pages.

ACKNOWLEDGMENTS

This work was supported by U.S. National Science Foundation grant CCF-1528163.

REFERENCES

[1] 2015. How Do Browsers Display Web Pages, and Why Don't They Ever Look the Same? Retrieved Jan 2017 from http://www.makeuseof.com/tag/how-do-browsers-display-web-pages-and-why-dont-they-ever-look-the-same/

[2] Abdulmajeed Alameer, Sonal Mahajan, and William G.J. Halfond. 2016. Detecting and Localizing Internationalization Presentation Failures in Web Applications. In *Proceeding of the 9th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*.

[3] Alexa. 2017. Alexa Top 500 Global Sites. Retrieved Jan 2017 from http://www.alexa.com/topsites

[4] Browsera. 2017. Automated Browser Compatibility Testing. Retrieved Jan 2017 from http://www.browsera.com/

[5] browserhacks.com. 2016. Browser Specific CSS Hacks. Retrieved Jan 2017 from http://browserhacks.com/

[6] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1535–1544.

[7] Shauvik Roy Choudhary. 2015. X-PERT Code. Retrieved Jan 2017 from https://github.com/gatech/xpert

[8] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Washington, DC, USA, 171–180.

[9] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, and others. 2003. Reformulating software engineering as a search problem. In *IEE Proceedings-Software*, Vol. 150. IET, 161–175.

[10] Michael G. Collins and John J. Barton. 2011. Crossfire: Multiprocess, Cross-browser, Open-web Debugging Protocol. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA)*. 115–124.

[11] Cucumber. 2017. Cucumber for BDD. Retrieved Jan 2017 from https://cucumber.io/

[12] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. 2012. WebMate: A Tool for Testing Web 2.0 Applications. In *Proceedings of the Workshop on JavaScript Tools (JSTools)*. ACM, 11–15.

[13] Sonal Mahajan et. al. 2016. XFix Project Page. Retrieved Jan 2017 from https://github.com/sonalmahajan/xfix

[14] Nicolas Gallagher and Jonathan Neal. 2016. Normalize CSS. Retrieved Jan 2017 from https://necolas.github.io/normalize.css/

[15] Firebug Working Group. 2017. Firebug. Retrieved Jan 2017 from https://addons.mozilla.org/en-US/firefox/addon/firebug/

[16] Sylvain Hallé, Nicolas Bergeron, Francis Guerin, and Gabriel Le Breton. 2015. Testing Web Applications Through Layout Constraints. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 1–8.

[17] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. In *Information and software Technology*, Vol. 43. Elsevier, 833–839.

[18] Joseph Kempka, Phil McMinn, and Dirk Sudholt. 2015. Design and Analysis of Different Alternating Variable Searches for Search-Based Software Testing. In *Theor. Comput. Sci.*, Vol. 605. 1–20.

[19] B. Korel. 1990. Automated Software Test Data Generation. In *IEEE Trans. Softw. Eng.*, Vol. 16. 870–879.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. 3–13.

[21] Gitte Lindgaard, Cathy Dudek, Devjani Sen, Livia Sumegi, and Patrick Noonan. 2011. An Exploration of Relations Between Visual Appeal, Trustworthiness and Perceived Usability of Homepages. In *ACM Trans. Comput.-Hum. Interact.*, Vol. 18. 1:1–1:30.

[22] Gitte Lindgaard, Gary Fernandes, Cathy Dudek, and Brown J. 2006. Attention web designers: You have 50 milliseconds to make a good first impression!. In *Behaviour & Information Technology*, Vol. 25. 115–126.

[23] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 166–178.

[24] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. 2017. XFix: Automated Tool for Repair of Layout Cross Browser Issues. In *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA) – Tool Track*.

[25] Sonal Mahajan and William G. J. Halfond. 2014. Finding HTML Presentation Failures Using Image Comparison Techniques. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE) – New Ideas track*.

[26] Sonal Mahajan and William G. J. Halfond. 2015. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[27] Sonal Mahajan and William G. J. Halfond. 2015. WebSee: A Tool for Debugging HTML Presentation Failures. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST) – Tool track*.

[28] Sonal Mahajan, Bailan Li, Pooyan Behnamghader, and William G. J. Halfond. 2016. Using Visual Symptoms for Debugging Presentation Failures in Web Applications. In *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[29] Sonal Mahajan, Bailan Li, and William G. J. Halfond. 2014. Root Cause Analysis for HTML Presentation Failures Using Search-based Techniques. In *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST)*.

[30] David Sawyer McFarland. 2006. *CSS: The Missing Manual*. O'Reilly.

[31] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. What Test Oracle Should I Use for Effective GUI Testing?. In *Proceedings of the International Conference on Automated So!ware Engineering (ASE)*. 164–173.

[32] Ali Mesbah and Mukul R. Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 561–570.

[33] Ali Mesbah and Arie van Deursen. 2009. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Washington, DC, USA, 210–220.

[34] Jan Odvarko Mike Buckley, Lorne Markham. 2017. Pixel Perfect Firefox. Retrieved Jan 2017 from https://addons.mozilla.org/en-us/firefox/addon/pixel-perfect/

[35] Rodrigo M. L. M. Moreira, Ana C. R. Paiva, and Atif Memon. 2013. A Pattern-Based Approach for GUI Modeling and Testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 288 – 297.

[36] NetMarketShare. 2017. Browser Net Market Share. Retrieved Jan 2017 from https://www.netmarketshare.com/

[37] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software. In *Automated Software Engg.*, Vol. 21. 65–105.

[38] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2011. Auto-locating and Fix-propagating for HTML Validation Errors to PHP Server-side Code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Washington, DC, USA, 13–22.

[39] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring Bug Fixes in Object-oriented Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE)*. 315–324.

[40] Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[41] CSS Reset. 2016. YUI 3 CSS Reset. Retrieved Jan 2017 from http://cssreset.com/scripts/yahoo-css-reset-yui-3/

[42] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2013. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. 702–711.

[43] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2014. X-PERT: A Web Application Testing Tool for Cross-browser Inconsistency Detection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 417–420.

[44] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. 1–10.

[45] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 277–287.

[46] Selenium. 2017. Selenium HQ. Retrieved Jan 2017 from http://docs.seleniumhq.org/

[47] Nataliia Semenenko, Marlon Dumas, and Tnis Saar. 2013. Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, Washington, DC, USA, 528–531.

[48] Sourceforge. 2017. CSS Parser. Retrieved Jan 2017 from http://cssparser.sourceforge.net/

[49] Stackoverflow. 2017. Stackoverflow Cross-browser Posts. Retrieved Jan 2017 from http://stackoverflow.com/questions/tagged/cross-browser

[50] StatCounter. 2016. Browser Statcounter. Retrieved Jan 2017 from http://gs.statcounter.com/#desktop-browser-ww-monthly-201506-201606-bar

[51] Noam Tractinsky, Avivit Cokhavi, Moti Kirschenbaum, and Tal Sharfi. 2006. Evaluating the Consistency of Immediate Aesthetic Perceptions of Web Pages. In *International booktitle of Human-Computer Studies*, Vol. 64. 1071 – 1083.

[52] Alexandre N. Tuch, Eva E. Presslaber, Markus StöCklin, Klaus Opwis, and Javier A. Bargas-Avila. 2012. The Role of Visual Complexity and Prototypicality Regarding First Impression of Websites: Working Towards Understanding Aesthetic Judgments. In *Int. J. Hum.-Comput. Stud.*, Vol. 70.

[53] Uroulette. 2017. Random URL Generator. Retrieved Jan 2017 from http://www.uroulette.com/

[54] Thomas A. Walsh, Phil McMinn, and Gregory M. Kapfhammer. 2015. Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages. In *International Conference on Automated Software Engineering (ASE)*. ACM, 709–714.

[55] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*. ACM, New York, NY, USA, 16:1–16:11.

[56] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. 364–374.

[57] WellDoneCode. 2017. Perfect Pixel Chrome. Retrieved Jan 2017 from https://chrome.google.com/webstore/detail/perfectpixel-by-welldonec/dkaagdgjmgdmbnecmcefdhjekcoceebi?hl=en

[58] Wikipedia. 2016. CSS Hacks. Retrieved Jan 2017 from https://en.wikipedia.org/wiki/CSS_hack

[59] Wikipedia. 2017. List of Browsers. Retrieved Jan 2017 from https://en.wikipedia.org/wiki/List_of_web_browsers

[60] Qing Xie and Atif M. Memon. 2006. Studying the Characteristics of a "Good" GUI Test Suite. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Washington, DC, USA, 159–168.

[61] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New York, NY, USA, 183–192.

[62] Sai Zhang, Hao Lü, and Michael D. Ernst. 2013. Automatically Repairing Broken Workflows for Evolving GUI Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 45–55.