This is a repository copy of *Effective automated repair of internationalization presentation failures in web applications using style similarity clustering and search-based techniques*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/163401/

Version: Accepted Version

Article:

Mahajan, S., Alameer, A., McMinn, P. orcid.org/0000-0001-9137-7433 et al. (1 more author) (2021) Effective automated repair of internationalization presentation failures in web applications using style similarity clustering and search-based techniques. Software Testing, Verification and Reliability, 31 (10-2). e1746. ISSN 0960-0833

https://doi.org/10.1002/stvr.1746

This is the peer reviewed version of the following article: Mahajan, S, Alameer, A, McMinn, P, Halfond, W. Effective automated repair of internationalization presentation failures in web applications using style similarity clustering and search-based techniques. Softw. Test. Verif. Reliab. 2020;e1746, which has been published in final form at https://doi.org/10.1002/stvr.1746. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.

Reuse

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

Effective Automated Repair of Internationalization Presentation Failures in Web Applications Using Style Similarity Clustering and Search-Based Techniques

Sonal Mahajan[1] | Abdulmajeed Alameer[2] | Phil McMinn[3] | William G. J. Halfond*[4]

[1]Advanced Software & Algorithms Group, Fujitsu Laboratories of America, Inc., Sunnyvale, USA
[2]Department of Computer Science, King Saud University, Riyadh, Saudi Arabia
[3]Department of Computer Science, University of Sheffield, Sheffield, UK
[4]Department of Computer Science, University of Southern California, Los Angeles, USA

Correspondence
*Correspondence to: William G. J. Halfond, University of Southern California, USA. E-mail: Email: halfond@usc.edu

Summary

Companies often employ internationalization (i18n) frameworks to provide translated text and localized media content on their websites in order to effectively communicate with a global audience. However, the varying lengths of text from different languages can cause undesired distortions in the layout of a web page. Such distortions, called Internationalization Presentation Failures (IPFs), can negatively affect the aesthetics or usability of the website. Most of the existing automated techniques developed for assisting repair of IPFs either produce fixes that are likely to significantly reduce the legibility and attractiveness of the pages, or are limited to only detecting IPFs, with the actual repair itself remaining a labor intensive manual task. To address this problem, we propose a search-based technique for automatically repairing IPFs in web applications, while ensuring a legible and attractive page. The empirical evaluation of our approach reported that our approach was able to successfully resolve 94% of the detected IPFs for 46 real-world web pages. In a user study, participants rated the visual quality of our fixes significantly higher than the unfixed versions, and also considered the repairs generated by our approach to be notably more legible and visually appealing than the repairs generated by existing techniques.

KEYWORDS:
internationalization, presentation failures, automated repair, search-based software engineering, layout issues, web applications

1 Introduction

Web applications enable companies to easily establish a global presence. To more effectively communicate with this global audience, companies often employ i18n frameworks for their websites, which allow the websites to provide translated text or localized media content. However, because the length of translated text differs in size from text written in the original language of the page, the page's appearance can become distorted. HTML elements that are fixed in size may clip text or look too large, while those that are not fixed can expand, contract, and move around the page in ways that are inconsistent with the rest of the page's layout. Such distortions, called *Internationalization Presentation Failures (IPFs)*, reduce the aesthetics or usability of a website and occur frequently — a recent study reports their occurrence in over 75% of internationalized web pages [1]. Avoiding presentation problems, such as these, is important. Studies show that the design, legibility, and visual attractiveness of a website affects users' impressions of its credibility and trustworthiness, ultimately impacting their decision to spend money on the products or services that it offers [16, 18, 19].

[0]Equal contribution by Sonal Mahajan and Abdulmajeed Alameer.

Repairing IPFs poses several challenges for web developers. First, modern web pages may contain hundreds, if not thousands, of HTML elements, each with several CSS properties controlling their appearance. This makes it challenging for developers to accurately determine which elements and properties need to be adjusted in order to resolve an IPF. Assuming that the relevant elements and properties can be identified, the developers must still carefully construct the repair. Due to complex and cascading interactions between styling rules, a change in one part of a web page user interface (UI) can easily introduce further issues in another part of the page. This means that any potential repair must be evaluated in the context of not only how well it resolves the targeted IPF, but also its impact on the rest of the page's layout as a whole. This task is complicated because it is possible that more than one element will have to be adjusted together to repair an IPF. For example, if the faulty element is part of a series of menu items, then all of the menu items may have to be adjusted to ensure their new styling matches that of the repaired element.

Researchers have therefore proposed several automated techniques to help developers repair IPFs. One such technique is $\mathcal{I}$Fix [33], our prior work, that targets the repair of IPFs. $\mathcal{I}$Fix searches through the set of potentially faulty HTML elements to identify the best CSS values for them that can repair the observed IPF. Out of the several different CSS properties that control the appearance of an HTML element, $\mathcal{I}$Fix restricts the search for best repair to three CSS properties: `font-size`, `width`, and `height`. With this strategy, $\mathcal{I}$Fix was able to effectively repair a high number of IPFs. However, its user study revealed that the participants were not always impressed with the visual appeal of the repaired pages — 30% favored faulty pages over their repaired versions. Upon investigation of such pages, we found one dominant reason — $\mathcal{I}$Fix substantially reduced the `font-size` to resolve IPFs. Such changes are likely to significantly affect the readability of the page, reducing its visual attractiveness, thereby negatively affecting users' impressions of the website. Other existing techniques targeting internationalization problems, such as GWALI [2], are only able to detect IPFs, and cannot generate repairs. Meanwhile other web page repair approaches target fundamentally different UI problems and are not capable of repairing IPFs. These include XFix [31], which repairs cross-browser issues; MFix [29], which repairs mobile friendly problems; and PhpRepair [63] and PhpSync [51], which repair malformed HTML.

In this paper, we present an approach for automatically repairing IPFs in web pages. Our approach is designed to address the limitations of $\mathcal{I}$Fix discussed above and handle the practical and conceptual challenges particular to the IPF domain: To identify elements whose styling must be adjusted together, we designed a novel style-based clustering approach that groups elements based on their visual appearance and DOM characteristics. To find repairs, we designed a guided search-based technique that efficiently explores a diverse set of eleven different CSS properties, such as `padding` and `margin`, which control the size and placement of an HTML element, to find a repair that is potentially legible and attractive. This technique is capable of finding a repair solution that best fixes an IPF while avoiding the introduction of new layout problems. To guide the search, we designed a fitness function that leverages existing IPF detection techniques and UI change metrics. To overcome the limitation of $\mathcal{I}$Fix and generate a repair with `font-size` reduction only as a last resort, we defined a preference weighting in the fitness function to rank the candidate repairs based on the CSS properties used in them, with `font-size` assigned the least preference weight, while `padding` and `margin` properties assigned the highest weight. We implemented our approach in a tool called $\mathcal{I}$Fix^{++}. In the evaluation, we found that it was effective at repairing IPFs, resolving over 94% of the detected IPFs; and also reasonably fast, requiring approximately three minutes as the median average time needed to generate the repair. In a user study of the repaired web pages, we found that the repairs met with high user approval — over 60% of user responses rated the repaired pages as better than the faulty versions. We also compared the visual impact of the repairs generated by $\mathcal{I}$Fix and $\mathcal{I}$Fix^{++} via a user study. We observed a high user approval in this study as well — the pages repaired by $\mathcal{I}$Fix^{++} were rated better in legibility three times more than the pages repaired by $\mathcal{I}$Fix. Overall, these results are positive and indicate that our approach can help developers automatically resolve IPFs in web pages by producing more legible and attractive repairs.

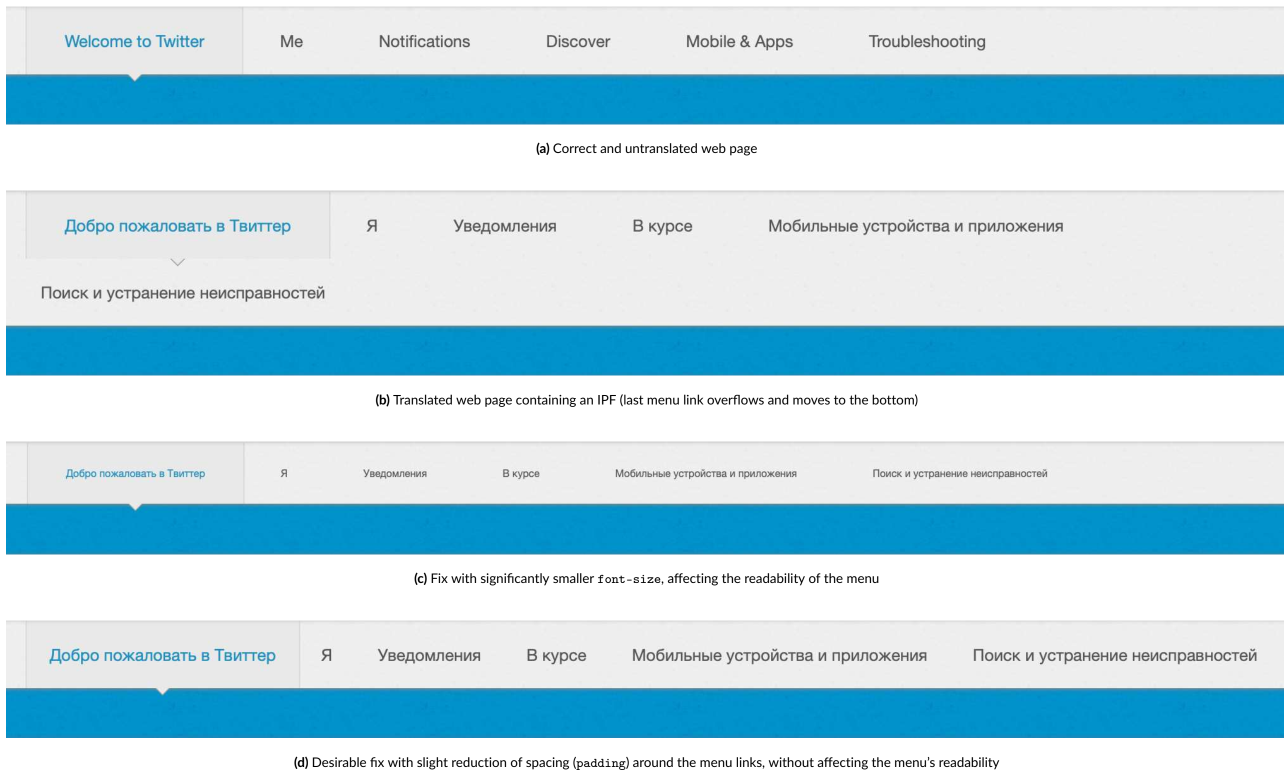
The contributions of this paper are therefore as follows:

1. An approach for automatically repairing IPFs in web pages that uses style similarity clustering and search-based techniques.
2. An approach that improves the repair strategy of our prior work, $\mathcal{I}$Fix [33], by including a diverse set of CSS properties to produce a repair patch that is legible and visually attractive.
3. An empirical study on a large set of real-world web pages whose results show that our approach is effective and fast in repairing IPFs,
4. A user study showing that the web pages repaired by our approach were rated more highly than the unrepaired versions.

The rest of the paper is organized as follows. In Section 2 we present background information about internationalization and IPFs. Then in Section 3 we describe the approach in detail and its evaluation in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Background and Motivation

In this section we discuss the process of internationalization in web pages and the presentation failures that could be caused in that process, the procedure of debugging such failures, and the limitations of existing techniques in repairing IPFs.



(a) Correct and untranslated web page

(b) Translated web page containing an IPF (last menu link overflows and moves to the bottom)

(c) Fix with significantly smaller `font-size`, affecting the readability of the menu

(d) Desirable fix with slight reduction of spacing (`padding`) around the menu links, without affecting the menu's readability

FIGURE 1 Example of an IPF on the Twitter Help page when translated from English to Russian and two ways of fixing the IPF

2.1 Internationalization Presentation Failures (IPFs)

Developers internationalize web applications by isolating language-specific content, such as text, icons, and media, into resource files. The web page can utilize different sets of resource files, depending on the user's language — a piece of information supplied by their browser — and inserted into placeholders in the requested page. This isolation of language-specific content allows a developer to design a universal layout for a web page, easing its management and maintenance, while also modularizing language specific processing.

However, the internationalization of web pages can distort their intended layout because the length of different text segments in a page can vary depending on their language. An increase in the length of a text segment can cause it to overflow the HTML element in which it is contained, be clipped, or spill over into surrounding areas of the page. Alternatively, the containing element may expand to fit the text, which can, in turn, cause a cascading effect that disrupts the layout of other parts of the page. IPFs can affect both the usability and the aesthetics of a web page. An example is shown in Figure 1b. Here, the text of the page in Figure 1a has been translated from English to Russian, but the increased number of characters required by the translated text pushes the final link of the navigation bar to the next line. Such an IPF not only distorts the appearance of the navigation bar, but also affects its usability as the sub-menu pop-up from the first link (in blue) is obstructed by the final link that is pushed down. Internationalization can also cause non-layout failures in web pages, such as corrupted text, inconsistent keyboard shortcuts, and incorrect/missing translations. Our approach does not target these non-layout related failures as we see the solutions as primarily requiring developer intervention to provide correct translations.

2.2 Debugging IPFs

The complete process of debugging an IPF requires developers to (1) detect when an IPF occurs in a page, (2) localize the faulty HTML elements that are causing the IPF to appear, and (3) repair the web page by modifying CSS properties of the faulty elements to ensure that the failure no longer occurs and introduces no further problems in the layout of the page. An existing technique, GWALI [2], has been shown to be an accurate *detection and localization* technique for IPFs. (i.e., it addresses the first and second part of the debugging process described above.) For completeness, we provide a summary of GWALI's algorithm and its evaluation results. The inputs to GWALI are a baseline (untranslated) page, which represents a correct rendering of the page, and a translated version (Page Under Test (PUT)), which is analyzed for IPFs. To detect IPFs, GWALI builds a model called a Layout Graph (LG), which captures the position of each HTML element in a web page relative to the other elements. Each node of the

graph represents a visible HTML element, while an edge between two nodes is annotated with a type of visual layout relationship (e.g., "East of", "intersects", "aligns with", and "contains") that exists between the two elements. After building the LGs for the two versions of a page, GWALI compares them and identifies edges whose annotations are different in the PUT. A difference in annotations indicates that the relative positions of the two elements are different, signaling a potential IPF. If an IPF is detected, GWALI outputs a list of HTML elements that are most likely to have caused it. GWALI was evaluated over 54 real-world web pages. It was found to be highly accurate — detecting IPFs with 91% precision and 100% recall, and localizing faulty element with a median rank of three. GWALI was also found to be time-efficient, requiring an average of 9.75 seconds per page. Due its accurate and efficient performance, we leverage the output of GWALI to initialize the repair process.

Assuming that an IPF has been detected and localized, there are several strategies developers can use to repair the faulty HTML elements. One of these is to change the translation of the original text, so that the length of the translated text closely matches the original. However, this solution is not normally applicable for two reasons. Firstly, the translation of the text is not always under the control of developers, having typically been outsourced to professional translators or to an automatic translation service. Secondly, a translation that matches the original text length may not be available. Therefore a more typical repair strategy is to adapt the layout of the internationalized page to accommodate the translation. To do this, developers need to identify the right sets of HTML elements and CSS properties among the potentially faulty elements, and then search for new, appropriate values for their CSS properties. Together, these new values represent a language-specific CSS patch for the web page. To ensure that the patch is employed at runtime, developers use the CSS `:lang()` selector. This selector allows developers to specify alternative values for CSS properties based on the language in which the page is viewed. Although this repair strategy is relatively straightforward to understand, complex interactions among HTML elements, CSS properties, and styling rules make it challenging to find a patch that resolves all IPFs without introducing new layout problems or significantly distorting the appearance of a web UI. Furthermore, web pages exhibiting IPFs need to be repaired on a case-by-case basis, since the unique design and layout of different pages makes it challenging to extract a common repair template.

2.3 Limitations of existing techniques

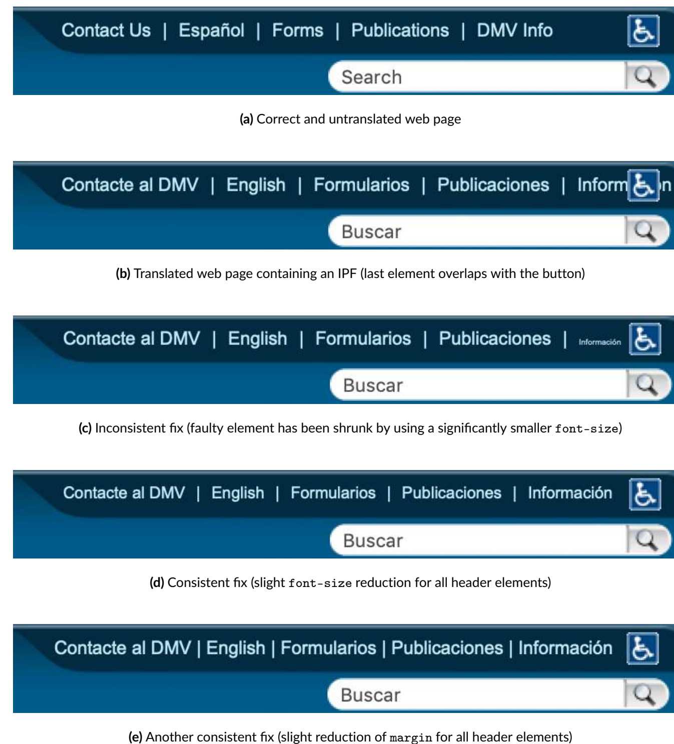
Our prior work, $\mathcal{I}$Fix [33], automates the repair of IPFs in web pages. It uses guided search-based techniques to find new values for the faulty CSS properties of the HTML elements that need to be adjusted in order to repair the observed IPF. Using this approach, $\mathcal{I}$Fix was able to resolve over 98% of the detected IPFs. Although effective at resolving IPFs, we observed that the repairs produced by $\mathcal{I}$Fix often decreased the `font-size` of the faulty HTML elements, thereby negatively affecting the aesthetics and readability of the page. This observation was also supported by the user study results, where almost 30% of the user responses rated the faulty pages as having more appearance similarity to the baseline pages than the repaired versions. Although appearance similarity included other aspects, such as spacing and alignment, our investigation revealed that text size was likely the most prominent aspect considered by the participants for judging appearance similarity.

The challenge in generating a successful repair, therefore, involves adjusting the size of HTML elements using CSS properties such as `padding`, `margin`, `width`, and `height`, which can resolve IPFs without affecting the page's readability, and decreasing the `font-size` only when a repair with other CSS properties is not feasible. Figure 1 shows an illustrative example of successful repairs for the Twitter Help page. To correct the overflow of the menu links in the example, the `font-size` can be decreased, which is the repair generated by $\mathcal{I}$Fix (Figure 1c). However, this change significantly affects the readability of the menu. This point was also underscored in the user study, where over 36% of the participants favored the unrepaired version of the Twitter page over the repaired version [33]. Therefore a more desirable fix, as illustrated in Figure 1d, would be to decrease the space surrounding the menu links, which is controlled by the `padding` and `margin` CSS properties, without having to modify the `font-size`. This challenge and insight motivates our approach, which is an enhanced version of $\mathcal{I}$Fix, to repair IPFs while also ensuring an attractive and readable page. We present this approach in the next section.

3 Approach

The goal of our approach is to automatically repair IPFs that have been detected in a translated version of a web page. As we describe in Section 2, a translation can cause the text in a web page to expand or contract, which leads to text overflow, element movement, incorrect text wrapping, and misalignment. The placement and the size of elements in a web page is controlled by their CSS properties. Therefore, these failures can be fixed by changing the value of the CSS properties of elements in a page to allow them to accommodate the new size of the text after translation.

Finding these new values for the CSS properties is complicated by several challenges. The first challenge is that any kind of style change to one element must also be mirrored in stylistically related elements. We illustrated this in Figure 2. To correct the overlap shown in Figure 2b, the text size of the word "Informacion" can be decreased, resulting in the layout shown in Figure 2c. However, this change is unlikely to be visually appealing to an end user since the consistency of the header appearance has been changed. Ideally, we would prefer the change in Figure 2d (or Figure 2e), which subtly decreases the font size (or margin) of all of the stylistically related elements in the header. This challenge requires that our



(a) Correct and untranslated web page

(b) Translated web page containing an IPF (last element overlaps with the button)

(c) Inconsistent fix (faulty element has been shrunk by using a significantly smaller `font-size`)

(d) Consistent fix (slight `font-size` reduction for all header elements)

(e) Another consistent fix (slight reduction of `margin` for all header elements)

FIGURE 2 Example of an IPF on the DMV homepage when translated from English to Spanish and different ways of fixing the IPF

solution identify groupings of elements that are stylistically similar and adjust them together in order to maintain the aesthetics of a web page. The second challenge is that a change for any particular IPF may introduce new layout problems into other parts of the page. This can happen when the elements surrounding the area of the IPF move to accommodate the changed size of the repaired element. This challenge is compounded when there are multiple IPFs in a page or there are many elements that must be adjusted together, since multiple changes to the page increase the likelihood that the final layout will be distorted. This challenge requires that our solution finds new values for the CSS properties that fix IPFs while avoiding the introduction of new layout problems.

Two insights into these challenges guide the design of our approach. The first insight is that it is possible to automatically identify elements that are stylistically similar through an approach that uses traditional density based clustering techniques. We designed a clustering technique that is based on a combination of visual aspects (e.g., elements' alignment) and DOM-based metrics (e.g., XPath similarity). This allows our approach to accurately group stylistically similar elements that need to be changed together to maintain the aesthetic consistency of a web page's style. As a secondary effect, the clustering technique can also facilitate the repair of multiple IPFs in tandem. The second insight is that it is possible to quantify the amount of distortion introduced into a page by IPFs and use this value as a fitness function to guide a search for a set of new CSS values. We designed our approach's fitness function using existing detectors for IPFs (i.e., GWALI [2]) and other metrics for measuring the amount of difference between two UI layouts. Therefore, the goal of our approach is to find a solution (i.e., new CSS values) that minimizes this fitness function.

Our approach begins by analyzing the PUT and identifying the stylistically similar clusters using visual and DOM similarity metrics. Then, the approach performs a guided search to find the best CSS values for each of the identified clusters. The fitness function used to guide the search leverages, (1) a measurement of the layout dissimilarity in the PUT introduced by the IPFs, by using existing detection techniques, namely GWALI, and (2) the amount of CSS change needed for repairing the IPFs. Upon termination, the best CSS values from all of the clusters are returned as the output of the search. We now further introduce the steps of our approach in more detail, beginning with an overview of the complete algorithm.

3.1 Overall Algorithm

The overall algorithm of our approach is shown in Algorithm 1. The three inputs to the approach are: (1) B, a version of the web page (*baseline*) that shows its correct layout, (2) *PUT*, a translated version that exhibits IPFs, and (3) E, a list of HTML elements of the PUT that are likely to be faulty. The last input can be provided either by a detection technique, such as GWALI, or manually by developers. Developers could simply provide

Algorithm 1 Overall Algorithm

```
Input:  B: Baseline page showing the correct layout
        PUT: Web page under test (translated page showing IPFs)
        E: Potentially faulty HTML elements in PUT
Output: PUT′: Repaired PUT
 1: /* Phase 1 — Identify Stylistically Similar Clusters */
 2: allSimSets ← findPageClusters(PUT)                          ▷ sub-function defined in Algorithm 2
 3: SimSets ← {}
 4: for each S ∈ allSimSets do
 5:     for each e ∈ E do
 6:         if e ∈ S then
 7:             SimSets ← SimSets ∪ S
 8:         end if
 9:     end for
10: end for
11: /* Phase 2 — Search for Optimal Repair Solution */
12: iterationCnt ← 0
13: saturationCnt ← 0
14: /* Step 1: Initializing the population */
15: candidateSolutions ← initialize(SimSets, B, POPULATION_SIZE)  ▷ sub-function defined in Algorithm 3
16: while true do
17:     /* Step 2: Fine tuning using local search*/
18:     optimalSolutionSoFar ← getBestSolution(candidateSolutions)  ▷ sub-function defined in Algorithm 4
19:     fineTunedSolution ← fineTune(optimalSolutionSoFar)          ▷ sub-function defined in Algorithm 5
20:     candidateSolutions ← candidateSolutions ∪ fineTunedSolution
21:     /* Step 3: Mutation */
22:     for each solution ∈ candidateSolutions do
23:         mutatedSolution ← mutate(solution)                      ▷ sub-function defined in Algorithm 6
24:         candidateSolutions ← candidateSolutions ∪ mutatedSolution
25:     end for
26:     /* Step 4: Selection */
27:     candidateSolutions ← select(candidateSolutions, POPULATION_SIZE)  ▷ sub-function defined in Algorithm 7
28:     /* Step 5: Check Termination Criteria */
29:     optimalSolution ← getBestSolution(candidateSolutions)
30:     if optimalSolution = optimalSolutionSoFar then
31:         saturationCnt ← saturationCnt + 1
32:     else
33:         saturationCnt ← 0
34:     end if
35:     iterationCnt ← iterationCnt + 1
36:     if iterationCnt = MAX_ITERATIONS or saturationCnt = SATURATION_POINT then
37:         PUT′ ← applyRepair(PUT, optimalSolution)                ▷ sub-function defined in Algorithm 8
38:         return PUT′
39:     end if
40: end while
```

a conservative list of possibly faulty HTML elements, but the use of an automated detection technique allows the debugging process to be fully automated. The output of our approach is a page, PUT′, a repaired version of the PUT.

The overall algorithm, shown by Algorithm 1, comprises of two phases, as shown by the overview diagram in Figure 3. The different sub-functions used in the overall algorithm are shown in Algorithms 2 to 8.

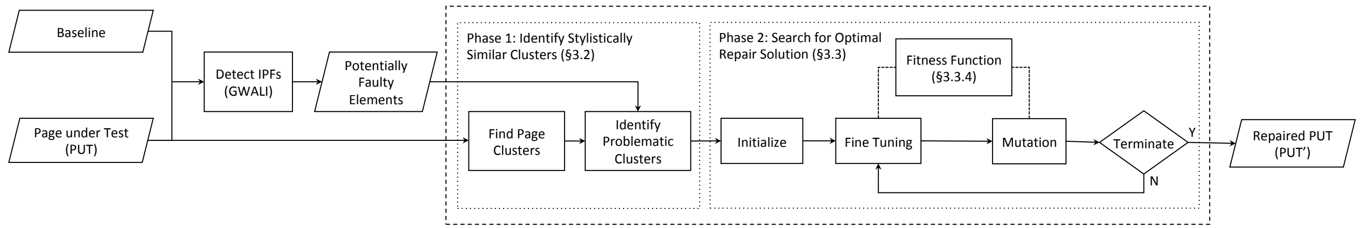


FIGURE 3 Overview of our approach

Phase 1 — Identify Stylistically Similar Clusters. The first phase of the approach identifies clusters of HTML elements in the PUT that are visually similar and whose properties should be adjusted together to maintain the visual consistency of the repaired page. First, the approach groups all of the HTML elements in the PUT into clusters (line 2) and then stores in *SimSets* only the clusters that include the potentially faulty elements (lines 3–10). We describe the process of identifying stylistically similar clusters employed by our approach in detail in Section 3.2.

Phase 2 — Search for Optimal Repair Solution. The second phase of the approach performs a guided search to find the best repair and is comprised of five steps (lines 12—40). The first step creates an *initial population* of n candidate solutions by analyzing the change in text length between the PUT and B (line 15). The second step finds the best solution from the available candidate solutions and uses a local search to *fine tune* the CSS change values to obtain an improved solution, which is then added to the pool of candidates (lines 18—20). To diversify the population, the third step *mutates* every candidate solution by changing their CSS values with a random amount and adds the mutated solutions to *candidateSolutions* (lines 22—25). The fourth step then *selects* the top n candidate solutions for processing in the next iteration (line 27). Finally, the fifth step determines whether the algorithm should terminate or proceed to another iteration of the search (lines 29—34). The approach terminates if a predefined number of MAX_ITERATIONS is reached or no improvement is observed in the *optimalSolution* for consecutive SATURATION_POINT iterations. When the search terminates, the *optimalSolution* is converted to a web page CSS repair patch and applied to the PUT to produce the repaired page, PUT′, which is then returned as the output of the approach (lines 37 and 38). We describe the search process in detail in Section 3.3

3.2 Phase 1: Identify Stylistically Similar Clusters

The primary goal of this step is to group HTML elements in the page that are visually similar into Sets of Stylistically Similar Elements (*SimSets*). Typically, there exists a one-to-one mapping between the IPFs in a page and the SimSets. The clustering step allows our approach to apply the same change value to all elements in a SimSet uniformly, to maintain their visual consistency while repairing the IPF. In case of multiple IPFs, changes are applied to different SimSets in synchronization with each other to find the overall repair. Clustering, therefore, helps our approach to find a repair that is aesthetically consistent with the style of the page, and also repair multiple IPFs efficiently.

To group a page's elements into SimSets, our approach computes visual similarity and DOM information similarity between each pair of elements in the page. We designed a distance function that quantifies the similarity between each pair of elements e_1 and e_2 in the page. Then our approach uses a density-based clustering technique to determine which elements are in the same SimSet. After computing these SimSets, our approach identifies the SimSet associated with each faulty element reported by GWALI. This subset of the SimSets serves as an input to the search.

Different techniques can be used to group HTML elements in a web page. A naive mechanism is to put elements having the same style *class* attribute into the same SimSet. In practice we found that the class attribute is not always used by developers to set the style of similar elements, or in some cases, it is not matching for elements in the same SimSet for page areas such as header and footer, where the current menu item in the navigation bar is typically highlighted differently than the other menu items using different class attributes. Another option could be to use computer vision techniques [36, 35, 38] to find visually similar elements. Since such techniques are typically pixel-based, they are highly sensitive to even slight differences in size or color. This can result in highly inaccurate results. There are several more sophisticated techniques that may be applied to group related elements in a web page, such as Vision-based Page Segmentation (VIPS) [10], Block-o-Matic [64], and R-Trees [36]. These techniques rely on elements' location in the web page and use different metrics to divide the web page into multiple segments. However, these techniques do not produce sets of visually similar elements as needed by our approach. Instead, they produce sets of web page segments that group elements that are located closely to each other and are not necessarily similar in appearance. The clustering in our approach uses multiple visual aspects to group the elements, while the aforementioned techniques rely solely on the location the elements, which makes them unsuitable for our approach.

To identify stylistically similar elements in the page, our approach uses a density-based clustering technique, DBSCAN [17]. A density-based clustering technique finds sets of elements that are close to each other, according to a predefined distance function, and groups them into clusters. Density-based clustering is well suited for our approach for several reasons. First, the distance function can be customized for the problem domain,

which allows our approach to use style metrics instead of location. Second, this type of clustering does not require prior knowledge of the number of clusters, which is ideal for our approach since each stylistically similar group may have a different number of elements, making the total number of clusters unknown beforehand. Third, the clustering technique puts each element into only one cluster (i.e., hard clustering). This is important because if one element is placed into multiple SimSets, the search could define multiple change values for the same element, which may prevent the search from converging if the changes are conflicting. This hard clustering, in fact, allows our approach to search for fixes within a cluster independently, and across clusters collectively. For example, increasing the width of the elements in two clusters in tandem to repair an IPF.

Our approach's distance function uses several metrics to compute the similarity between pairs of elements in a page. At a high-level, these metrics can be divided into two types of similarity: (1) similarity in the visual appearance of the elements, including width, height, alignment, and CSS property values and (2) similarity in the DOM information, including XPath, HTML *class* attribute, and HTML tag name. We include DOM related metrics in the distance function because only using visual similarity metrics may produce inaccurate clusters in cases where the elements belonging to a cluster are intentionally made to appear different. For example, to highlight the link of the currently rendered page from a list of navigational menu links. Since the different metrics have vastly different value ranges, our approach normalizes the value of each metric to a range [0, 1], with zero representing a match for the metric and 1 being the maximum difference. The overall distance computed by the function is the weighted sum of each of the normalized metric values. The metrics' weights were determined based on experimentation on a set of web pages and are the same for all subjects. Next, we provide a detailed description of each of the metrics our approach uses in the distance function, also shown in Algorithm 2.

3.2.1 Visual Similarity Metrics

These metrics are based on the similarity of the visual appearance of the elements. Our approach uses three types of visual metrics to compute the distance between two elements e_1 and e_2. These are:

Elements' *width* and *height* match: Elements that are stylistically similar are more likely to have matching width and/or height. Our approach defines width and height matching as a binary metric. If the widths of the two elements e_1 and e_2 match, then the width metric value is set to 0, otherwise it is set to 1. The height metric value is computed similarly.

Elements' alignment match: Elements that are similar are more likely to be aligned with each other. This is because browsers render a web page using a grid layout, which aligns elements belonging to the same group either horizontally or vertically. Alignment includes left edge alignment, right edge alignment, top edge alignment, and bottom edge alignment. These four alignment metrics are binary metrics, so they are computed in a way similar to the width and height metrics.

Elements' CSS properties similarity: Aspects of the appearance of the elements in a web page, such as their color, font, and layout, are defined in the CSS properties of these elements. For this reason, elements that are stylistically similar typically have the same values for their CSS properties. Our approach computes the similarity of the CSS properties as the ratio of the matching CSS values over all CSS properties defined for both elements. For this metric, our approach only considers explicitly defined CSS properties, so it does not take into account default CSS values and CSS values that are inherited from the *body* element in the web page. These values are matching for all elements and are not helpful in distinguishing elements of different SimSets.

3.2.2 DOM Information Similarity Metrics

These metrics are based on the similarity of features defined in the DOM of the web page. Our approach uses three types of DOM related metrics to compute the distance between two elements e_1 and e_2. These are:

Elements' tag name match: Elements in the same SimSet have the same type, so the HTML tag names for them need to match. HTML tag names are used as a binary metric, i.e., if e_1 and e_2 are the same tag name, then the metric value is set to 0, otherwise it is set to 1.

Elements' XPath similarity: Elements that are in the same SimSet are more likely to have similar XPaths. The XPath similarity between two elements quantifies the commonality in the ancestry of the two elements. In HTML, elements in the page inherit CSS properties from their parent elements and pass them on to their children. More ancestors in common between two elements means more inherited styling information is shared between them. To compute XPath distance, our approach uses the Levenshtein distance between elements' XPath. More formally, XPath distance is the minimum number of HTML tags edits (insertions, deletions or substitutions) required to change one XPath into the other.

Elements' class attribute similarity: As mentioned earlier, an HTML element's *class* attribute is often insufficient to group similarly styled elements. Nonetheless, it can be a useful signal; therefore we use class attribute similarity as one of the our metrics for style similarity. An HTML element can have multiple class names for the class attribute. Our approach computes the similarity in class attribute as the ratio of class names that are matching over all class names that are set.

Algorithm 2 Find stylistically similar clusters

```
1:  function findPageClusters(PUT)
2:      allElements ← getXpathsOfAllElements (PUT)
3:      allSimSets ← DBSCAN (allElements, distance)
4:      return C
5:  end function

6:  function distance(e1, e2)
7:      d ← 0.0
8:      /* Visual Similarity Metrics */
9:      if e1.width == e2.width then
10:         d ← d + (W_width × 1)
11:     end if
12:     if e1.height == e2.height then
13:         d ← d + (W_height × 1)
14:     end if
15:     if e1.left == e2.left then
16:         d ← d + (W_left × 1)
17:     end if
18:     if e1.right == e2.right then
19:         d ← d + (W_right × 1)
20:     end if
21:     if e1.top == e2.top then
22:         d ← d + (W_top × 1)
23:     end if
24:     if e1.bottom == e2.bottom then
25:         d ← d + (W_bottom × 1)
26:     end if
27:     if e1.bottom == e2.bottom then
28:         d ← d + (W_bottom × 1)
29:     end if
30:     d ← d + (W_css × (|e1.css ∩ e2.css| / |e1.css ∪ e2.css|))
31:     /* DOM Information Similarity Metrics */
32:     if e1.tag == e2.tag then
33:         d ← d + (W_tag × 1)
34:     end if
35:     d ← d + (W_xpath × levenshtein (e1.xpath, e2.xpath))
36:     d ← d + (W_class × (|e1.classNames ∩ e2.classNames| / |e1.classNames ∪ e2.classNames|))
37:     return d
38: end function
```

3.3 Phase 2: Search for Optimal Repair Solution

The goal of the search is to find values for the CSS properties of each SimSet that make the baseline page and the PUT have LGs that are matching with minimal changes to the page. Our approach generates candidate solutions using the search operations we define in this section. Then our approach evaluates each candidate solution it generates using the fitness function to determine if the candidate solution produces a better version of the PUT. We begin with a discussion of the CSS properties used by our approach to generate a repair, which we follow with a description of the representation of the candidate solution, an overview of the search algorithm used, and finally explanation of the fitness function employed by our approach.

3.3.1 Relevant CSS Properties

Identifying relevant CSS properties is instrumental in finding an optimal repair. In the context of IPFs, the expansion or contraction of text length due to language translation causes the enclosing HTML elements to display layout failures, such as overlap with neighboring elements, text overflow, and clipping. Therefore the general intuition of this step is to find CSS properties that can impact the size of an HTML element in view of accommodating the translated text. For finding such properties we have two key insights. First, the number of relevant CSS properties identified should be small. This is important to allow the search to converge on a solution in a reasonable amount of time. Second, the identified CSS properties should be diverse to allow the search to find different viable candidate solutions, such as one with `font-size` reduction and the other with increase in `width`, in order to converge on the optimal one. Through analysis of the different CSS properties defined by the W3C, we identified eleven CSS properties that can affect the size of an HTML element and be used to successfully repair IPFs. This set of CSS properties holds true for all web applications without requiring developer intervention.

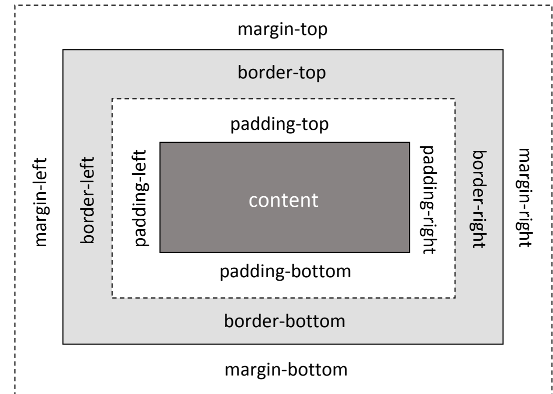


FIGURE 4 CSS Box Model of an HTML Element

To help us explain the eleven CSS properties and the rationale behind their selection, we first discuss the CSS box model of an HTML element. The CSS box model shown in Figure 4 is a standard defined by the W3C for calculating the size of an HTML element rendered in a browser. The box model consists of four different parts: content, padding, border, and margin. The "content" area is where the text, image, or other media content of the element appears. The "padding" area is the transparent space between the box's content and border. "Border" is the line between the box's padding and margin. "Margin" is the transparent area between the box and the surrounding boxes. The size of each of the four areas as well as the individual sides of the areas, i.e., top, left, bottom, and right, can be set using different CSS properties. We now explain the eleven CSS properties we have employed in our approach.

`font-size`: This CSS property is used to set the size of font of text in HTML elements. Unless defined, the content size of a text element primarily depends on the length of the text that it contains and the `font-size` assigned to it. Therefore, increasing or decreasing the `font-size` value can cause the content area of an element to expand or contract, which can be used to repair IPFs. This is illustrated in Figure 2d, where a slight reduction in the `font-size` of all the menu links causes their size to shrink, thereby correcting the overlap.

`width, height`: The `width` and `height` CSS properties are used to set the size of the content area of an HTML element. For text content, if the length of the text is longer than the specified `width` and `height` value, then the content could overflow, get wrapped, get clipped, or spill into surrounding areas, resulting in an IPF. Therefore adjusting the values of `width`, `height`, or both can be used to effectively repair IPFs.

`padding-top, padding-bottom, padding-left, padding-right`: The padding properties are used to set the size of the padding area in the box model. Therefore decreasing (increasing) the `padding` value can decrease (increase) the overall size of the element. This characteristic can be leveraged to repair IPFs that typically cause elements to overflow or wrap as shown in the motivating example in Figure 1. On analyzing the use of padding properties in different real-world web pages, we found that {`padding-left`, `padding-right`} and {`padding-top`, `padding-bottom`} are generally set together to center align the content in the element. Based on this insight, we define compound notations, "`padding-left-right`" and "`padding-top-bottom`", to allow our approach to adjust the left-right and top-bottom padding areas symmetrically to produce an aesthetically appealing repair. Our idea is to use the change value found by the search for `padding-left-right` notation and apply the same value for both, `padding-left` and `padding-right`. The approach follows the same process for `padding-top-bottom`. Therefore along with a better visual repair, the compound notations also allow our approach to reduce the search space for the padding properties by a factor of two, helping reduce the runtime.

`margin-top, margin-bottom, margin-left, margin-right`: The margin properties are used to set the spacing between an element and its neighbors. Decreasing or increasing the margin values can cause the surrounding elements to move closer or away from the element under consideration. Similar to padding properties, margin properties are effective in repairing IPFs caused by overflowing or wrapping of elements. Similar to the padding compound notations, we define "`margin-left-right`" and "`margin-top-bottom`" notations for the margin properties. An example of a repair using the margin properties is shown in Figure 2e.

Note that the first three properties, `font-size`, `width`, and `height`, were also used in $\mathcal{I}$Fix [33], however, the remaining eight properties, four of `padding` and four of `margin`, are introduced in our approach.

There exist other CSS properties, such as `border-top`, `border-bottom`, `border-left`, `border-right`, `letter-spacing`, `word-spacing`, and `font-weight`, that can also potentially affect the overall size of an HTML element. However, we did not include them in the set of relevant properties used in our approach as through experiments and analysis we found that their impact on the absolute size of HTML elements was very small, making them marginally useful in the repair of IPFs.

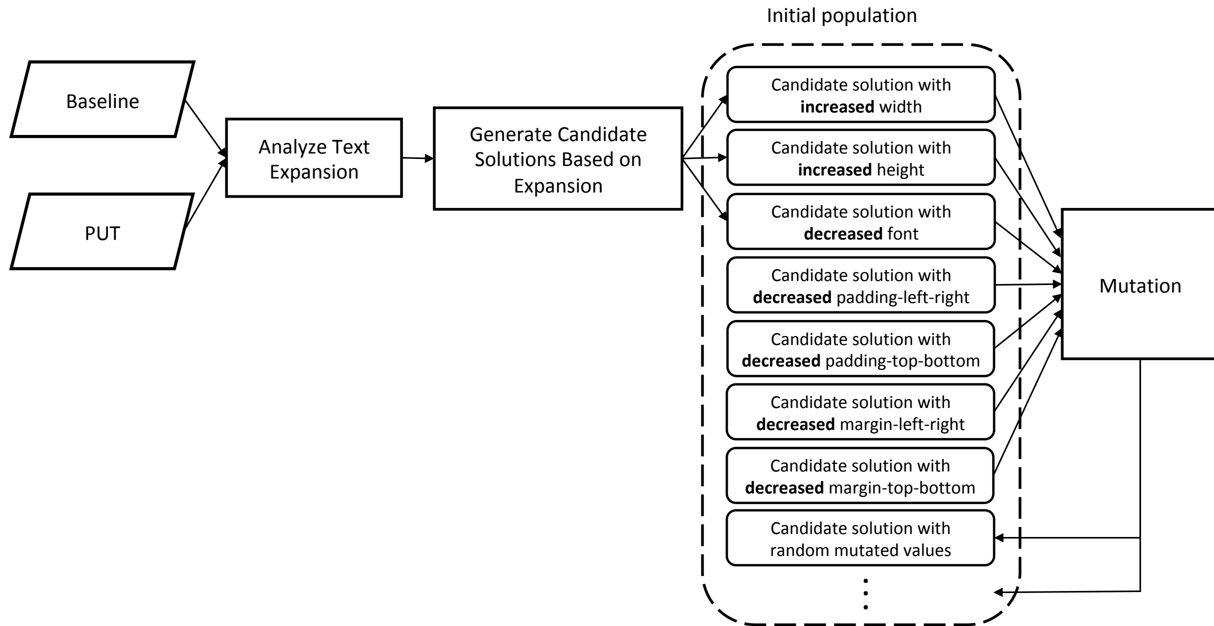


FIGURE 5 Initializing the population

3.3.2 Candidate Solution Representation

A repair for the PUT is represented as a collection of changes for each of the SimSets identified by the clustering technique. More formally, we define a potential repair as a *candidate solution*, which is a set of change tuples. Each change tuple is of the form $\langle S, p, \Delta \rangle$ where Δ is the *change value* that our approach applies to a specific CSS property p for a particular SimSet S. Here, $p \in \{$`font-size, width, height, padding-left-right, padding-top-bottom, margin-left-right, margin-top-bottom`$\}$ as discussed in Section 3.3.1. The change value can be positive or negative to represent an increase or decrease in the value of p. Note that a candidate solution can have multiple change tuples for the same SimSet as long as they target different CSS properties.

An example candidate solution is ($\langle S_1,$ `font-size`, $-1\rangle$, $\langle S_1,$ `width`, $0\rangle$, $\langle S_1,$ `height`, $0\rangle$, $\langle S_1,$ `padding-left-right`, $-5\rangle$, $\langle S_1,$ `padding-top-bottom`, $0\rangle$, $\langle S_1,$ `margin-left-right`, $0\rangle$, $\langle S_1,$ `margin-top-bottom`, $0\rangle$, $\langle S_2,$ `font-size` , $-1\rangle$, $\langle S_2,$ `width`, $10\rangle$, $\langle S_2,$ `height`, $0\rangle$)), $\langle S_2,$ `padding-left-right`, $0\rangle$, $\langle S_2,$ `padding-top-bottom`, $0\rangle$, $\langle S_2,$ `margin-left-right`, $0\rangle$, $\langle S_2,$ `margin-top-bottom`, $0\rangle$. This candidate solution represents a repair to the PUT that decreases the `font-size` of the elements in S_1 by one pixel, decreases the left and right padding of the elements in S_1 by five pixels, decreases the `font-size` of the elements in S_2 by one pixel, and increases the `width` of the elements in S_2 by ten pixels. Note that the value "0" means that there is no change to the elements in the SimSet for the specified property.

3.3.3 Search Algorithm

The approach operates by going through multiple iterations of the search. In each iteration, the approach generates a population of candidate solutions. Then, the approach refines the population by keeping only the best candidate solutions and performing the search operations on them for another iteration. The search terminates when a termination condition is satisfied. After the search terminates, the approach returns the best candidate solution in the population. More formally, the iteration includes five main steps (1) initializing the population, (2) fine-tuning the best solution using local search, (3) performing mutation, (4) selecting the best set of candidate solutions, (5) and terminating the search if a termination condition is satisfied. The following is a description of each step in more detail:

Initializing the population: This step creates an initial population of candidate solutions that our approach performs the search on. The goal of this step is to create a diverse initial population that allows the search to explore different areas of the solution space. Figure 5 shows an overview of the process of initializing the population. In the figure, the first set of candidate solutions represents modifications to the elements that are computed based on text expansion that occurred to the PUT. To generate this set of candidate solutions, our approach computes the average percentage of text expansion in the elements of each SimSet that includes a faulty element. Then our approach generates seven candidate solutions, one for each of the identified relevant CSS properties (Section 3.3.1), based on the expansion percentage. The first candidate solution increases the `width` of the elements in the SimSets by a percentage equal to the percentage of the text expansion. The second candidate solution increases the height by the same percentage. The third candidate solution decreases the `font-size` of the elements in the SimSets by the same

Algorithm 3 Initialize population

```
1: function initialize(SimSets, B, N)
2:     CSS ← {width, height, font-size, padding-left-right, padding-top-bottom, margin-left-right, margin-top-bottom}
3:     candidateSolutions ← {}
4:     for each S ∈ SimSets do
5:         totalTextExpansion ← 0
6:         for each e_PUT ∈ S do
7:             e_B ← findMatchingElement (e_PUT.xpath, B)
8:             totalTextExpansion ← totalTextExpansion + |e_PUT.text| / |e_B.text|
9:         end for
10:        avgTextExpansion ← totalTextExpansion / |S|
11:        for each p ∈ CSS do
12:            if p == width OR p == height then
13:                soln ← ⟨S, p, avgTextExpansion⟩
14:            else
15:                soln ← ⟨S, p, −avgTextExpansion⟩
16:            end if
17:            for each other_p ∈ (CSS − p) do
18:                soln ← soln ∪ ⟨S, other_p, 0⟩
19:            end for
20:            candidateSolutions ← candidateSolutions ∪ soln
21:        end for
22:    end for
23:    remainingPopulationSize ← N − |candidateSolutions|
24:    for i ← 1 to remainingPopulationSize do
25:        randomSolution ← selectRandomSolution (candidateSolutions)
26:        candidateSolutions ← candidateSolutions ∪ mutate (randomSolution)
27:    end for
28:    return candidateSolutions
29: end function
```

percentage. The last four candidates decrease the `padding` and `margin` by the same percentage. The rest of the candidate solutions in the initial population (i.e., eight candidate solution in the figure) are generated by creating copies of the current candidate solutions and mutating the copies using the mutation operation described in the mutation step below. Algorithm 3 presents a step-by-step view of the initialization step.

Fine tuning using local search: This step works by selecting the best candidate solution in the population and fine tuning the *change values* Δ in it in order to get the best possible fix, as shown in Algorithms 4 and 5. To do this, our approach uses the Alternating Variable Method (AVM) local search algorithm [24, 22, 40]. Our approach performs local search by iterating over all the change tuples in the candidate solution and for each change tuple it tries a new value in a specific direction (i.e., it either increases or decreases the *change value* Δ for the CSS property), then evaluates the fitness of the new candidate solution to determine if it is an improvement. If there is an improvement, the search keeps trying larger values in the same direction. Otherwise, it tries the other direction. This process is repeated until the search finds the best possible *change values* Δ based on the fitness function. The search adds the newly generated candidate solution to the population.

Mutation: The goal of the mutation step is to diversify the population and explore *change values* that may not be reached during the AVM search. Our approach performs standard Gaussian mutation operations to the change values in the candidate solutions. It iterates over all the candidate solutions in the population and generates a new mutant for each one. Our approach creates a mutant by iterating over each tuple in the candidate solution and changing its value with a probability of 1 / (number of change tuples). The new change value is picked randomly from a Gaussian distribution around the old value. The newly generated candidate solutions are added to the population to be evaluated in the selection step. Algorithm 6 shows details of the mutation step.

Selection: Our approach evaluates all of the candidate solutions in the current population and selects the best `n` candidate solutions, where `n` is the predefined size of the population (Algorithm 7). The best candidate solutions are identified based on the fitness function described in Section 3.3.4. The selected candidate solutions are used as the population for the next iteration of the search.

Algorithm 4 Find best solution

```
1:  function getBestSolution(candidateSolutions)
2:      minFitnessScore ← ∞
3:      bestSolution ← {}
4:      for each solution ∈ candidateSolutions do
5:          if solution.fitnessScore < minFitnessScore then
6:              minFitnessScore ← solution.fitnessScore
7:              bestSolution ← solution
8:          end if
9:      end for
10:     return bestSolution
11: end function
```

Algorithm 5 Fine tune solution

```
1:  function fineTune(solution)
2:      fineTunedSolution ← {}
3:      for each ⟨S, p, Δ⟩ ∈ solution do
4:          ⟨S, p, Δ′⟩ ← AVM (⟨S, p, Δ⟩)
5:          fineTunedSolution ← fineTunedSolution ∪ ⟨S, p, Δ′⟩
6:      end for
7:      return fineTunedSolution
8:  end function
```

Algorithm 6 Mutate solution

```
1:  function mutate(solution)
2:      mutatedSolution ← {}
3:      probability ← 1 / |solution|
4:      for each ⟨S, p, Δ⟩ ∈ solution do
5:          if random(0, 1) < probability then
6:              Δ′ ← gaussian (Δ)
7:              mutatedSolution ← mutatedSolution ∪ ⟨S, p, Δ′⟩
8:          else
9:              mutatedSolution ← mutatedSolution ∪ ⟨S, p, Δ⟩
10:         end if
11:     end for
12:     return mutatedSolution
13: end function
```

Termination: The algorithm terminates when it satisfies one of two conditions. The first condition is when a predefined maximum number of iterations is reached. This condition is used to bound the execution time of the search and prevents it from running for a long time without converging to a solution. The second condition is when the search reaches a saturation point (i.e., no improvement in the candidate solutions for multiple consecutive iterations). In this case, the search most likely converged to the best candidate solution it could find, and further iterations will not introduce more improvement. Upon termination, the best candidate solution is then applied as a repair patch to the PUT, as shown by Algorithm 8.

Our approach could fail to find an acceptable fix under two scenarios. The first scenario is when GWALI does not include the actual faulty HTML element in its reported list. Our approach assumes that the initial set of elements provided as the input contains the faulty elements. If this assumption is violated, our approach will not be able to find a repair. The second scenario is when the search does not converge to an acceptable fix. This could occur due to the non-determinism of the search.

Algorithm 7 Select solutions

```
1: function select(solutions, n)
2:     topSolutions ← {}
3:     solutions ← sortInAscendingOrderOfFitnessScore (solutions)
4:     for i ← 1 to n do
5:         topSolutions ← topSolutions ∪ solutions[i]
6:     end for
7:     return topSolutions
8: end function
```

Algorithm 8 Apply Repair

```
1: function applyRepair(PUT, solution)
2:     for each ⟨S, p, Δ⟩ ∈ solution do
3:         for each e ∈ S do
4:             v ← getValue(PUT, e.p)
5:             v′ ← v + Δ
6:             setValue(PUT, e.p, v′)
7:         end for
8:     end for
9: end function
```

3.3.4 Fitness Function

To evaluate each candidate solution, our approach first generates a PUT′ by adjusting the elements of the PUT based on the values in the candidate solution. The approach then calculates the fitness score of the PUT′ when it is rendered in a browser. We now describe both these steps in detail.

3.3.4.1 Generating the PUT′

To generate the PUT′, our approach modifies the PUT according to the values in the candidate solution that will subsequently be evaluated. To modify the elements that need to be changed in the PUT, our approach uses the following general algorithm. Our approach iterates over each change tuple $\langle S, p, \Delta \rangle$ in the candidate solution and modifies the elements $e \in S$ by changing their CSS property values: $e.p = e.p + \Delta$. If p is one of the four compound notations (`padding-left-right`, `padding-top-bottom`, `margin-left-right`, or `margin-top-bottom`), then the approach applies the same Δ value to the two involved CSS properties. For example, if p is `padding-left-right` with Δ = -5, then the values of both `padding-left` and `padding-right` of e are decreased by five pixels. The changes to `width` and `height` properties for S to take effect require further processing, as explained below.

In addition to modifying the `width` and `height` of every element e in the SimSet S, the approach also modifies the `width` and the `height` of any ancestor element that has a fixed `width` or `height` that prevents the child elements from expanding freely. An example of such an ancestor element is shown in Figure 6. In the example, increasing the `width` of the elements in SimSet S requires modification to the fixed `width` value of the ancestor `div` element in order to make space for the children elements' expansion. For doing this our approach uses the following algorithm. After applying the change value to `width` for every element $e \in S$ using `e.width`$=$ `e.width`$+\Delta$, our approach computes the cumulative increase in `width` and `height` for all the elements in S and determines the new coordinates $\langle x1, y1 \rangle$,$\langle x2, y2 \rangle$ of the Minimum Bounding Rectangles (MBRs) of each element e. Then our approach finds the new position of the right edge of the rightmost element $\max(e_{x2})$, and the new position of the bottom edge of the bottommost element $\max(e_{y2})$. After that, our approach iterates over all the ancestors of the elements in S. For each ancestor a, if a has a fixed value for the `width` CSS property and $\max(e_{x2})$ is larger than a_{x2}, then our approach increases the `width` of the ancestor

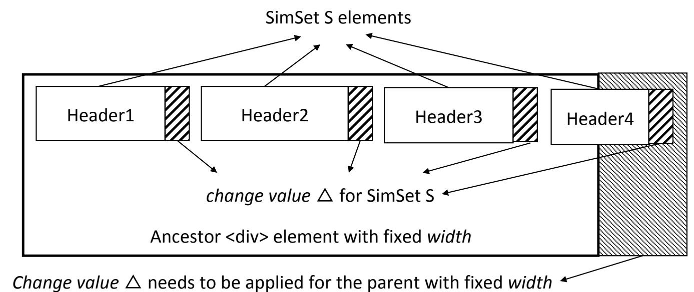


FIGURE 6 Example of ancestor elements with fixed `width` that need to be adjusted together with SimSet elements

$\texttt{a.width} = \texttt{a.width} + (\max(e_{x2}) - a_{x2})$. For any of these elements, if the calculated `width` exceeds the predefined `max-width` CSS property, then the approach increases the element's `max-width` to the calculated `width` value, i.e., $\texttt{a.max-width} = \texttt{a.width}$. A similar increase is applied to the `height`, if the ancestor has a fixed value for the `height` CSS property and $\max(e_{y2})$ is larger than a_{y2}.

3.3.4.2 Fitness Function Components

As mentioned earlier, a challenge in fixing IPFs is that any change to fix a particular IPF may introduce layout problems into other parts of the page. In addition, larger changes that are applied to the page make it more likely that the final layout will be distorted. This motivates the goal of the fitness function, which is to minimize the differences between the layout of the PUT and the layout of the baseline while making as few changes to the page as possible.

To address this goal, our approach's fitness function involves two components. The first is the *Amount of Layout Inconsistency* component. This component measures the impact of IPFs by quantifying the dissimilarity between the PUT′ layout and the baseline layout. The second part of the fitness function is the *Amount of Change* component. This component quantifies the amount of change the candidate solution applies to the page in order to repair it. To combine the two components of the fitness function, our approach uses a prioritized fitness function model in which minimizing the amount of layout inconsistency has a higher priority than minimizing the amount of change. The amount of layout inconsistency is given higher priority because it is strongly tied to resolving the IPFs, which is the goal of our approach, while amount of change component is used after resolving the IPFs to make the changes as minimal as possible. The prioritization is done by using a sigmoid function (Equation (2)) to scale the amount of change to a fraction and adding it to the amount of layout inconsistency value, which is a whole number. Since the amount of change is an aggregate of the percentage change applied to the concerned CSS properties, i.e., is always a positive value, the output of the sigmoid function ranges from 0.5 to 1.0. The overall fitness score is calculated as shown in Equation (1). We now describe the components of the fitness function in more detail.

$$\textit{Fitness score} = \textit{amount of layout inconsistency} + sigmoid(\textit{amount of change}) \tag{1}$$

$$sigmoid(x) = \frac{2\arctan(x) + \pi}{2\pi} \tag{2}$$

Amount of Layout Inconsistency: This component represents a quantification of the dissimilarity between the baseline and the PUT′ Layout Graphs (LGs). To compute the value for this component, our approach computes the coordinates of the MBRs of each element and the inconsistencies in the PUT as reported by GWALI. Then our approach computes the distance (in pixels) required to make the relationships in the two LGs match. The number of pixels is computed for every inconsistent relationship reported by GWALI. For alignment inconsistencies, if two elements `e1` and `e2` are top-aligned in the baseline and not top-aligned in the PUT′, our approach computes the difference in the vertical position of the top side of the two elements $|e1_{y1} - e2_{y1}|$. A similar computation is performed for bottom-alignment, right-alignment, and left-alignment. For direction inconsistencies, if `e1` is situated to the "West" of `e2` in the baseline, and is no longer "West" in the PUT′, our approach computes the number of pixels by which `e2` needs to move to be to the West of `e1`, which is $e1_{x2} - e2_{x1}$. A similar computation is performed for East, North, and South relationships. For containment inconsistencies, if `e1` bounds (i.e., contains) `e2` in the baseline, and no longer bounds it in the PUT′, our approach computes the vertical and horizontal expansion needed for each side of `e1`'s MBR to make it bound `e2`. The number of pixels computed for each of these inconsistent relationships (alignment, directional, and bounding) is added to get the total amount of layout inconsistency.

Amount of Change: This component represents the amount of change a candidate solution causes to the PUT, i.e., the total difference in CSS values with respect to the original (unmodifed) PUT. To compute this amount, our approach calculates the percentage of change that is applied to each CSS property for every modified element in the PUT. Every percentage of change is then multiplied with a weight, $\mathcal{W}_p$, where $\mathcal{W}$ is the weight value for the CSS property p. The total amount of change is calculated as the sum of the product of percentage of change and its preference weight $\mathcal{W}_p$. The intuition behind using $\mathcal{W}_p$ is to establish a priority in the solutions based on the CSS properties used in the repair. Through experiments, we set $\mathcal{W}_p$ for the `padding` and `margin` properties as 0.1, for `width` and `height` properties as 1.0, and for `font-size` as 4.0. This means that our approach's first preference is for solutions with `padding`/`margin` changes, followed by `width`/`height`, and lastly `font-size` changes. The intuition behind this prioritization is to overcome the limitation of $\mathcal{I}$Fix and guide the search towards solutions that can likely have minimum impact on the readability of the PUT while repairing IPFs. Referring back to the example, our approach rates the fix using `margin` shown in Figure 2e as better than the `font-size` fix shown in Figure 2d based on this preference weighting.

3.3.4.3 Fitness Function Calculations for Illustrative Example

To illustrate the impact of the fitness function in guiding the search to select the best repair, consider the fitness calculations shown in Figure 7. We show here the search progression for correcting the text overlap shown in Figure 2b using the `font-size` property. The Δ change value applied for `font-size` is shown as an absolute value, while the percentage change is given in parentheses. For simplifying the calculation for the amount

(a) `font-size` change = 0 (0%), amount of layout inconsistency (overlap) = 52px, fitness score = 52 + sigmoid(0) = 52.5

(b) `font-size` change = +2 (20%), amount of layout inconsistency (overlap) = 103px, fitness score = 103 + sigmoid(20) = 103.98

(c) `font-size` change = +1 (10%), amount of layout inconsistency (overlap) = 20px, fitness score = 20 + sigmoid(10) = 20.97

(d) `font-size` change = -5 (50%), amount of layout inconsistency (overlap) = 0px, fitness score = 0 + sigmoid(50) = 0.99

(e) `font-size` change = -1 (10%), amount of layout inconsistency (overlap) = 0px, fitness score = 0 + sigmoid(10) = 0.96

FIGURE 7 Illustration of fitness calculations for correcting the text overlap on the DMV homepage

of change component in the illustrative example, we ignore the preference weight as it would be the same for all of the considered `font-size` candidates. Therefore, here, the amount of change is the same as the percentage change. The amount of layout inconsistency is shown as the overlap between the last header element (highlighted with orange box) and the accessibility icon (highlighted with yellow box). At the beginning with `font-size` change = 0, the amount of overlap is reported by GWALI to be 52px, resulting in a fitness score of 52.5 (Figure 7a). When the search tries a new +2 change value, meaning the `font-size` of all header elements is increased by 2px, the overlap increases to 103px, resulting in a higher fitness score than before (Figure 7b). This informs the search that it is going in the wrong direction. Correcting the direction, the search then tries a lower change value resulting in a reduced 20px overlap, indicating that the search is now progressing in the right direction (Figure 7c). Continuing further in this direction, the search produces a candidate fix with a -5 change value, which corrects the overlap (i.e., repairs the IPF), however, results in a significantly smaller `font-size` (Figure 7d). The search then adjusts the change value to -1, which still corrects the overlap but produces a visually better fix by reducing the `font-size` only slightly for all header elements (Figure 7e). No further fitness improvements are observed for the `font-size` changes, indicating that the search has found the best solution with respect to the `font-size` property.

4 Evaluation

To assess the effectiveness and performance of our approach, we conducted an empirical evaluation on 46 real-world subject web pages and answered three research questions:

RQ1: How effective is our approach in reducing IPFs?

RQ1.a: How do the results compare with the previous version of the tool ($\mathcal{I}$Fix)?

RQ1.b: What is the contribution of guided search in reducing IPFs?

RQ1.c: What is the contribution of style similarity clustering in reducing IPFs?

RQ2: How long does it take for our approach to generate repairs?

RQ3: What is the quality of the fixes generated by our approach?

RQ4: How accurate is our style similarity clustering algorithm?

4.1 Implementation

We implemented our approach in Java as a prototype tool named $\mathcal{I}$Fix^{++} [30]. We used the *Apache Commons Math3* library implementation of the DBSCAN algorithm to group similarly styled HTML elements. We used *JavaScript* and *Selenium WebDriver* for dynamically applying candidate fix values to the pages and for extracting the rendered Document Object Model (DOM) information, such as element MBRs and XPath. We used the *jStyleParser* library for extracting explicitly defined CSS properties for HTML elements in a page. For obtaining the set of IPFs, we used the latest version of GWALI [2]. For the search technique described in Section 3, we selected the following parameter values empirically: population size = 100, mutation rate = 1.0, max. number of iterations = 20, and saturation point = 2. The values for population size and max. number of iterations were determined empirically with the view of allowing $\mathcal{I}$Fix^{++} to converge in a reasonable amount of time without trading off accuracy. We chose to keep a high mutation rate in the implementation of $\mathcal{I}$Fix^{++} in order to perturb the population frequently to prevent the search from converging on a local optima. The value, 2, of saturation point indicates that the approach terminates after the current iteration if no improvement in the fitness score was observed over the previous iteration. For the Gaussian distribution, used by the mutation operator, we used a 50% decrease and increase as the min and max values, and $\sigma = (\mathtt{max} - \mathtt{min})/8.0$ as the standard deviation. For clustering, we used the following weights for the different metrics: 0.1 for width/height and alignment, 0.3 for CSS properties similarity, 0.4 for tag name, 0.3 for XPath similarity, and 0.2 for class attribute similarity. These weights were determined empirically. The implementation of $\mathcal{I}$Fix^{++} and GWALI can be found in our replication package [30].

4.2 Subjects

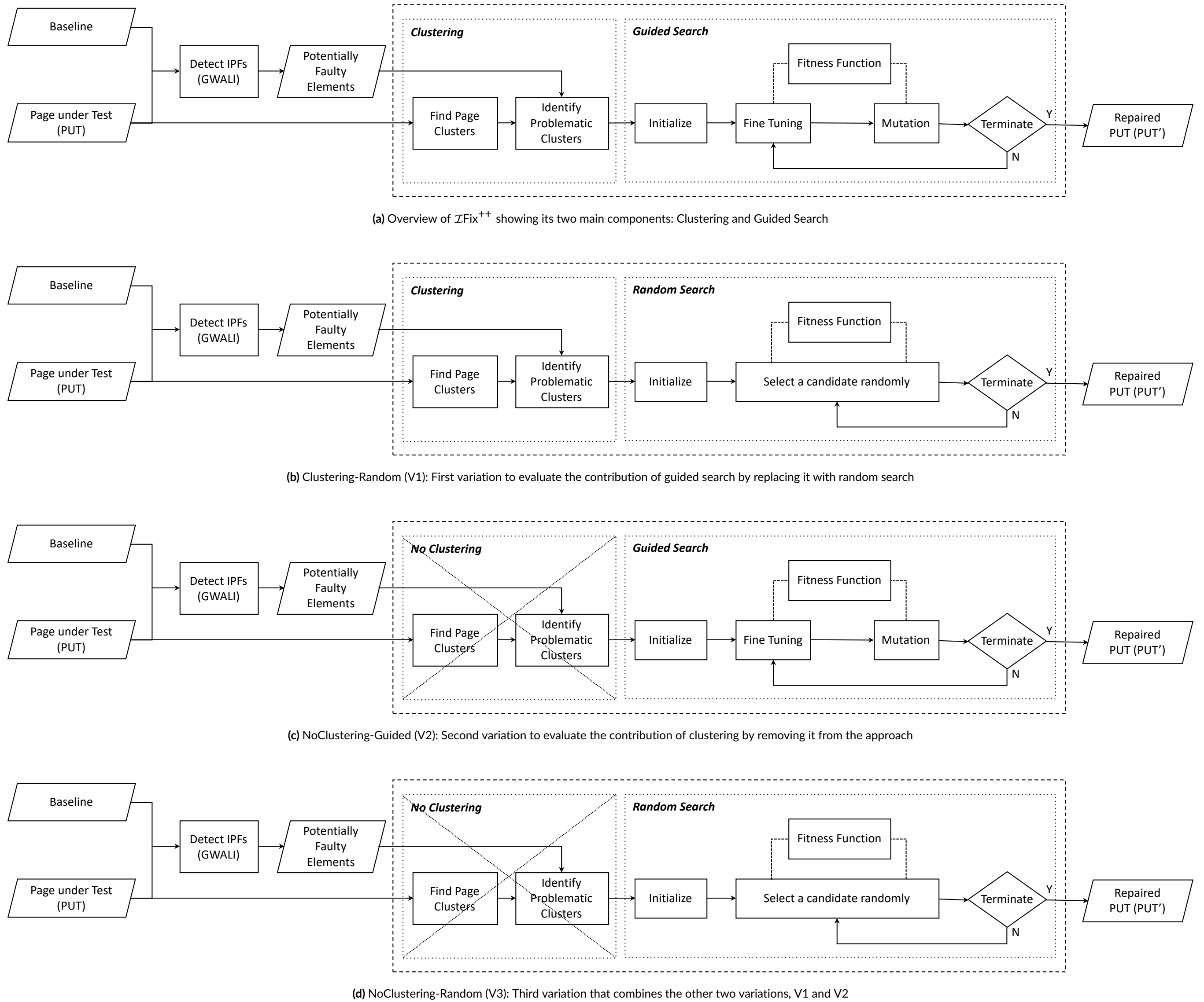
For the evaluation we used 46 real-world subject web pages as shown in Table 1. The column "#HTML" shows the total number of HTML elements in the subject page, giving a rough estimate of its size and complexity. The column "Baseline" shows the language of the subject used in the baseline version that shows the correct appearance of the page, and "Translated" shows the language that exhibits IPFs in the subject with respect to the baseline. We collected the subjects from three sources: (1) web pages used in the evaluation of GWALI [2] and $\mathcal{I}$Fix [33], (2) web pages used in a large-scale empirical study of IPFs [1], and (3) the random URL generator, UROULETTE [69]. Subjects 1–23 came from the first source, subjects 24–38 were chosen from the second source, and the remaining eight subjects, 39–46, were gathered from the third source. The main criteria behind selecting the first and second source was the presence of known IPFs and the diversity in size, layouts, and translation languages that the different subjects offered. From the first source, out of the total 54 subject pages used in the evaluation of GWALI, we filtered and selected only those web pages for which at least one IPF was reported. From the second source, out of the 1,020 web pages used in the empirical study, we randomly selected 16 subject web pages for the evaluation of our approach. For the third source, we used GWALI to select those subjects that contained at least one IPF.

4.3 Experiment One

To answer RQ1, we evaluated the performance and repairs generated by $\mathcal{I}$Fix^{++}. For doing this, we ran $\mathcal{I}$Fix^{++} on each subject and recorded the set of IPFs before and after each run, as reported by GWALI. To minimize the variance in the results that can be introduced from the non-deterministic aspects of the search, we ran $\mathcal{I}$Fix^{++} on each subject 10 times and used the mean values across the runs in the results. We calculated the *reduction in IPFs* as a percentage of the before and after values for each subject.

TABLE 1 Subjects

ID	Name	URL	#HTML	Baseline	Translated
1	akamai	https://www.akamai.com	304	English	Spanish
2	caLottery	http://www.calottery.com	777	English	Spanish
3	designSponge	http://www.designsponge.com	1,184	English	Spanish
4	dmv	https://www.dmv.ca.gov	638	English	Spanish
5	doctor	https://sfplasticsurgeon.com	689	English	Spanish
6	els	https://www.els.edu	483	English	Portuguese
7	facebookLogin	https://www.facebook.com	478	English	Bulgarian
8	flynas	http://www.flynas.com	1,069	English	Turkish
9	googleEarth	https://www.google.com/earth	323	Italian	Russian
10	googleLogin	https://accounts.google.com	175	English	Greek
11	hightail	https://tinyurl.com/y9tpmro7	1,135	English	German
12	hotwire	https://www.hotwire.com	583	English	Spanish
13	ixigo	https://www.ixigo.com/flights	1,384	English	Italian
14	linkedin	https://www.linkedin.com	586	English	Spanish
15	mplay	http://www.myplay.com	3,223	English	Spanish
16	museum	https://www.amnh.org	585	English	French
17	qualitrol	http://www.qualitrolcorp.com	401	English	Russian
18	rentalCars	http://www.rentalcars.com	1,011	English	German
19	skype	https://tinyurl.com/ycuxxhso	495	English	French
20	skyScanner	https://www.skyscanner.com	388	French	Malay
21	twitterHelp	https://support.twitter.com	327	English	Russian
22	westin	https://tinyurl.com/ycq4o8ar	815	English	Spanish
23	worldsBest	http://www.theworlds50best.com	581	English	German
24	deptOfEducation	http://www.arkansased.gov	654	English	Japanese
25	essex	https://www.essex.gov.uk	425	English	Spanish
26	marionCounty	http://www.co.marion.or.us	647	English	Russian
27	namibia	http://www.namibiatourism.com.na	777	English	Russian
28	nashville	https://www.nashville.gov	839	English	Polish
29	nevadaGoverner	http://gov.nv.gov/	472	English	French
30	princeGeorgeCity	http://www.princegeorge.ca	1,023	English	Russian
31	thunderbird	https://www.thunderbird.net	284	English	Russian
32	familySearch	https://www.familysearch.org	249	English	Spanish
33	limburg	https://www.limburg.de	836	German	Turkish
34	skyteam	https://www.skyteam.com	157	English	Russian
35	worldCustoms	http://www.wcoomd.org	594	English	French
36	bookCrossing	https://www.bookcrossing.com/	265	English	French
37	hattrick	https://www.hattrick.org/	427	English	German
38	googleGroups	https://groups.google.com/	433	English	German
39	portland	http://www.ci.portland.me.us/	2,590	English	French
40	denham	https://denham.house.gov	440	English	Spanish
41	sbc	http://www.sbc.net	1,102	English	Dutch
42	hachijojima	http://www.hachijo.gr.jp/	325	Japanese	English
43	geneva	https://www.geneve.com/	822	French	Russian
44	franceComte	http://www.franche-comte.org/	578	French	Dutch
45	tenable	https://www.tenable.com/	2,340	English	German
46	ruhr	https://www.metropoleruhr.de/	285	Turkish	English



(a) Overview of $\mathcal{I}$Fix^{++} showing its two main components: Clustering and Guided Search

(b) Clustering-Random (V1): First variation to evaluate the contribution of guided search by replacing it with random search

(c) NoClustering-Guided (V2): Second variation to evaluate the contribution of clustering by removing it from the approach

(d) NoClustering-Random (V3): Third variation that combines the other two variations, V1 and V2

FIGURE 8 $\mathcal{I}$Fix^{++} and its variations to evaluate the contribution of guided search and style similarity clustering

To further assess and understand the effectiveness of the main features of our work, we conducted further experiment runs with different variations of $\mathcal{I}$Fix^{++} (RQ1.a-c). For RQ1.a, we evaluated the two new features (additional CSS properties and modified fitness function with preference weighting) in $\mathcal{I}$Fix^{++} by comparing the results with that of the previous version of the tool, $\mathcal{I}$Fix. To answer RQ1.b and RQ1.c, we evaluated the two main contributions of our work, guided search and style similarity clustering, by implementing three variations of $\mathcal{I}$Fix^{++} shown in Figure 8.

- **Clustering-Random (V1)**: The first variation replaced the guided search in the approach with random search to evaluate the benefit of guided search with fitness function feedback. Figure 8b shows an overview of this variation. The random search operated by randomly selecting an unvisited candidate solution from the population and evaluating its fitness score. For every subject, we time bounded the random search by terminating it once the average time required by $\mathcal{I}$Fix^{++} for that subject had been utilized. Upon termination, the solution with the minimum fitness score was reported as the optimal solution. To allow a fair comparison, the random search used the same initial population and fitness function as that of guided search (i.e., $\mathcal{I}$Fix^{++}).

- **NoClustering-Guided (V2)**: The second variation removed the clustering component from $\mathcal{I}$Fix^{++} to evaluate the benefit of clustering stylistically similar elements in a page. An overview of this variation is shown in Figure 8c. The potentially faulty elements reported by GWALI were directly fed to the initialization step of the search.

- **NoClustering-Random (V3)**: The third variation combined the first and second variation (Figure 8d).

For RQ2, we computed the average total running time of $\mathcal{I}$Fix, $\mathcal{I}$Fix^{++}, and V2 across 10 runs for each subject. We did not compare the performance of $\mathcal{I}$Fix^{++} with V1 and V3 since we time bounded their random searches, as described above. We also measured the time required for the two main phases in our approach; clustering stylistically similar elements (Section 3.2) and searching for a repair patch (Section 3.3).

All of the experiments were run on a 64-bit Ubuntu 16.04 machine with 32GB memory, Intel Core i7-4790 processor, and screen resolution of 1920×1080. For rendering the subject web pages, we used Mozilla Firefox v46.0.01 with the browser window maximized to the screen size. Note that the number of IPFs used in our experiments and shown in Table 2 are with respect to this hardware configuration. Different hardware setup (screen size, resolution, etc.) may result in a different number of IPFs.

4.3.1 Presentation of Results

Table 2 shows the results for RQ1 and RQ2. The initial number of IPFs are shown under the column "#Before". The column headed "#After" shows the average number of IPFs remaining after each of the 10 runs of $\mathcal{I}$Fix^{++}, its three variations ("V1", "V2", and "V3"), and $\mathcal{I}$Fix. (Since it is an average, the results under "#After" columns may show decimal values.) The average percentage reduction is shown in parenthesis. The columns under "Time" show the average runtime in seconds across 10 runs of $\mathcal{I}$Fix, $\mathcal{I}$Fix^{++}, and V2. A breakdown of the total time required for phases 1 and 2 of the approach is presented under the columns "P1" and "P2", respectively.

4.3.2 Discussion of Results

The results show that $\mathcal{I}$Fix^{++} reported an average 94% reduction in IPFs, with a median of 100%. This shows that $\mathcal{I}$Fix^{++} was effective in finding fixes for the observed IPFs. A similar effectiveness value was also demonstrated by $\mathcal{I}$Fix. This is expected since the core search algorithm employed by both the techniques is the same, with the primary difference lying in the set of relevant CSS properties used to generate the repairs. This difference has a potential impact on the attractiveness and readability of the page, which we investigate in Experiment 2 via a user study (Section 4.4).

Out of the 46 subjects, $\mathcal{I}$Fix^{++} was able to completely resolve all of the reported IPFs in 37 subjects in each of the 10 runs and in 38 subjects in more than 80% of the runs. We investigated the subjects where $\mathcal{I}$Fix^{++} was not able to completely resolve all of the reported IPFs. We found that there are three reasons where $\mathcal{I}$Fix^{++} failed to completely repair the webpage. The first reason (for subjects *westin*, *essex*, and *denham*) was that elements surrounding the unrepaired IPF were required to be modified in order to completely resolve it. However, these elements were not reported by GWALI, thereby precluding $\mathcal{I}$Fix^{++} from finding a suitable fix. The second reason (for subjects *deptOfEducation* and *sbc*) was inaccuracy in the clustering part of our approach, where many unrelated elements were included in the clusters of faulty elements. This made any attempt from our approach to repair the faulty elements result in introducing new IPFs. The third reason (in subjects *ixigo*, *designSponge*, and *portland*) was due to false positive IPFs that were reported by GWALI. These unresolved IPFs were either for invisible elements in the web page, or they were falsely reported because the baseline and the translated web page have inherently significant differences in their structure and layout. Note that although GWALI shows some negative impact on results of $\mathcal{I}$Fix^{++}, we used GWALI since it was the only tool available to make $\mathcal{I}$Fix^{++} fully automated. The use of GWALI in our implementation can be easily replaced with any other IPF detection tool or can be provided manually.

$\mathcal{I}$Fix^{++} was also found to be the most effective in reducing the number of IPFs compared to its variations. This result strongly validates our two key insights of using guided search and clustering in the approach. The first key insight was validated as $\mathcal{I}$Fix^{++} was able to outperform a random search that had been given the same amount of time. Our approach was substantially more successful in primarily two scenarios. First, pages (e.g., hotwire) containing multiple IPFs concentrated in the same area that required a careful resolution of the IPFs by balancing the layout constraints without introducing new IPFs. Second, pages (e.g., akamai) that had strict layout constraints, permitting only a very small range of CSS values to resolve the IPFs. We also found that, overall, the repairs generated by random search were not visually pleasing as they often involved a substantial reduction in the `font-size` of text, indicating that guidance was helpful for our approach. This observation was also reflected in the total amount of change made to a page, captured by the fitness function, which reported that random search introduced 52% more changes, on average, compared to $\mathcal{I}$Fix^{++}. The second key insight of using a style-based clustering technique was validated as $\mathcal{I}$Fix^{++} not only rendered the pages more visually consistent compared to its non-clustered variations, but also increased the effectiveness by resolving a relatively higher number of IPFs. Particularly interesting is the comparison between $\mathcal{I}$Fix^{++} and its second variant (V2), which ignores clustering but performs guided search. One may presume that, barring the obvious visual difference, V2 may produce the same result as $\mathcal{I}$Fix^{++} for pages with a single cluster (e.g., qualitrol, denham, doctor, and googleEarth). However, it can be observed that V2 often reports a significantly lower repair rate than $\mathcal{I}$Fix^{++}. The primary reason for this is that V2 adjusts different elements reported by GWALI sequentially, i.e., one at a time. This often becomes unsuccessful as the fitness function does not show any improvement if the surrounding elements are still faulty. As opposed to this, $\mathcal{I}$Fix^{++} adjusts all the elements in a cluster together, helping resolve the IPF effectively. For example, consider the excerpt of the qualitrol page shown in Figure 17c. The IPF is caused because of the two faulty menu links positioned incorrectly on a new line. In this case, unless the two menu links are adjusted simultaneously, the IPF cannot be resolved.

The total running time of $\mathcal{I}$Fix^{++} ranged from 25 seconds to 34 minutes, with an average of just over 7 minutes and a median of 3 minutes. $\mathcal{I}$Fix^{++} was also 4.6 times faster, on average, than its second variation (no clustering). This was primarily because clustering enabled a narrowing of

TABLE 2 Results for effectiveness in reducing IPFs and average run time in seconds

Name	#Before	#After (Average Reduction in %)					Average Time in Sec				
		$\mathcal{I}$Fix	$\mathcal{I}$Fix^{++}	*V1*	*V2*	*V3*	$\mathcal{I}$Fix^{++} P1	$\mathcal{I}$Fix^{++} P2	$\mathcal{I}$Fix^{++} Total	$\mathcal{I}$Fix Total	*V2* Total
akamai	6	0 (100)	0 (100)	1.4 (77)	0.2 (97)	0.6 (90)	0.2	126.8	127.0	107.1	237.6
caLottery	4	0 (100)	0 (100)	0.2 (95)	0 (100)	2.7 (33)	0.3	88.6	88.9	97.5	153.1
designSponge	6	1 (83)	1 (83)	1 (83)	0 (100)	0 (100)	4.6	557.9	562.5	622.0	1161.1
dmv	13	0 (100)	0 (100)	4 (69)	0 (100)	6.7 (48)	0.4	597.9	598.3	325.9	769.0
doctor	21	0 (100)	0 (100)	0 (100)	0.5 (98)	10.5 (50)	0.3	114.7	115.0	76.8	345.3
els	6	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.4	72.1	72.5	98.1	124.0
facebookLogin	16	0 (100)	0 (100)	2.5 (84)	0.4 (98)	17 (-6)	0.2	339.6	339.8	282.4	988.0
flynas	9	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	1.8	542.9	544.7	246.6	625.7
googleEarth	15	0 (100)	0 (100)	0 (100)	0.8 (95)	8.6 (43)	0.1	164.5	164.6	99.3	447.9
googleLogin	6	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.0	64.1	64.2	57.2	74.5
hightail	2	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.5	89.9	90.4	81.2	92.7
hotwire	30	0 (100)	0 (100)	2.8 (91)	4 (87)	4 (87)	1.0	375.2	376.1	248.9	1734.5
ixigo	40	12 (70)	12 (70)	13.8 (66)	0 (100)	14 (65)	1.3	1224.9	1226.2	751.9	3539.6
linkedin	19	0 (100)	0 (100)	0 (100)	4.9 (74)	16.9 (11)	0.4	100.0	100.4	85.1	1542.6
mplay	76	0.4 (99)	0 (100)	3.2 (96)	29.8 (61)	75 (1)	2.4	1121.4	1123.8	834.1	26115.4
museum	32	0.4 (99)	0 (100)	0 (100)	14 (56)	17 (47)	0.3	535.4	535.7	322.0	3943.9
qualitrol	19	0 (100)	0 (100)	0 (100)	17.4 (8)	21 (-11)	0.4	107.8	108.3	81.1	436.0
rentalCars	6	0 (100)	0 (100)	1.2 (80)	0 (100)	0.2 (97)	0.9	526.0	526.9	380.4	863.9
skype	3	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.8	121.3	122.1	124.5	100.4
skyScanner	4	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.2	100.2	100.4	62.9	74.9
twitterHelp	5	0 (100)	0 (100)	0 (100)	0 (100)	0.3 (94)	0.2	72.7	72.9	64.4	224.2
westin	11	1 (91)	1.2 (89)	2.2 (80)	1 (91)	1 (91)	0.3	192.4	192.7	123.0	275.6
worldsBest	20	0 (100)	0 (100)	6.2 (69)	0 (100)	14.8 (26)	0.4	259.7	260.1	208.1	1678.1
deptOfEducation	13	7.2 (45)	6.2 (52)	5 (62)	11.2 (14)	11.2 (14)	2.6	905.1	907.7	1280.0	409.9
essex	91	8 (91)	8 (91)	8 (91)	17.6 (81)	44.4 (51)	1.5	1742.7	1744.3	1045.8	12226.7
marionCounty	8	0 (100)	0.4 (95)	2.2 (73)	0.4 (95)	0.3 (96)	0.7	379.1	379.8	275.2	486.6
namibia	19	0 (100)	0 (100)	0 (100)	2 (89)	6 (68)	2.0	456.6	458.5	427.6	1390.0
nashville	14	0 (100)	0 (100)	0 (100)	1.8 (87)	2 (86)	5.2	749.2	754.4	381.3	1943.2
nevadaGoverner	36	0 (100)	0 (100)	0 (100)	17.1 (53)	12.5 (65)	1.0	906.7	907.7	565.8	2940.4
princeGeorgeCity	4	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	3.7	1289.5	1293.1	1267.4	1033.3
thunderbird	9	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.1	146.8	146.9	137.1	195.8
familySearch	6	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.1	68.7	68.8	52.0	80.1
limburg	2	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.9	104.7	105.6	163.2	127.3
skyteam	8	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.0	45.4	45.4	35.0	48.0
worldCustoms	2	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.2	71.0	71.3	60.0	78.3
bookCrossing	2	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.1	94.7	94.8	76.4	87.3
hattrick	8	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.1	76.9	77.0	77.6	122.0
googleGroups	2	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.2	24.8	25.0	33.7	34.1
portland	12	7 (42)	7 (42)	8.2 (32)	0 (100)	0 (100)	11.3	1850.7	1862.0	1601.8	3148.5
denham	22	4 (82)	4 (82)	4 (82)	3.6 (84)	9.2 (58)	1.2	220.6	221.8	217.0	1593.5
sbc	2	2 (0)	2 (0)	2 (0)	0 (100)	2 (0)	3.4	2036.7	2040.1	2102.2	542.8
hachijojima	6	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	1.5	256.4	257.9	259.3	315.2
geneva	2	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.3	68.4	68.8	87.0	144.4
franceComte	10	0 (100)	0 (100)	0.2 (98)	0 (100)	2.4 (76)	0.3	173.6	173.9	118.0	633.1
tenable	20	0 (100)	0 (100)	0 (100)	0 (100)	0 (100)	0.7	537.7	538.4	342.5	1396.1
ruhr	4	0 (100)	0 (100)	1.8 (55)	0 (100)	0 (100)	0.5	150.2	150.7	107.8	199.1
Average	**14.6**	**0.9 (94)**	**0.9 (94)**	**1.5 (90)**	**2.7 (81)**	**6.5 (73)**	**1.2**	**431.6**	**432.8**	**349.9**	**1624.4**

the search space by grouping together potentially faulty elements reported by GWALI that were also stylistically similar. Therefore a single change to the cluster was capable of resolving multiple IPFs. Breaking down the total runtime shows that the clustering overhead in $\mathcal{I}$Fix^{++} was negligible, requiring less than a second, on average. A majority, 99.7% of the total runtime, on average, was required by the search phase. Overall, $\mathcal{I}$Fix^{++} required 23% more time than $\mathcal{I}$Fix in order to find repairs. This slowdown is expected since the set of relative CSS properties used in the search, which has a direct impact on the search space, is almost four times larger than that used in $\mathcal{I}$Fix. ($\mathcal{I}$Fix^{++} uses eleven CSS properties in the search, while $\mathcal{I}$Fix uses only three.) The runtime of $\mathcal{I}$Fix^{++} can be further improved via parallelization, as has been achieved in related work [25, 38]. We plan to do this in future work.

4.4 Experiment Two

For addressing RQ3, we conducted two variants of a user-based survey to understand the visual quality of $\mathcal{I}$Fix^{++}'s suggested fixes from a human perspective. The survey format of our first variant was to present, in random order, an IPF containing a UI snippet from a subject web page before and after repair by $\mathcal{I}$Fix^{++}, and a snippet of its baseline version. The participants were then asked to compare the two UI snippets to their baseline, provide ratings on a 5-point Likert scale for each of the three features of *legibility*, *attractiveness*, and *appearance similarity*. Each UI snippet showing an IPF was captured in context of its surrounding region to allow participants to view the IPF from a broader perspective. Examples of UI snippets are shown in Figure 2b and Figure 10. An optional text area was also provided to the participants to explain their choice of the answers. To select the "after" version of a subject, we used the run with the best fitness score across the 10 runs of $\mathcal{I}$Fix^{++} in Experiment One. We followed the same setup for the second variant, but this time comparing the repairs generated by $\mathcal{I}$Fix^{++} and $\mathcal{I}$Fix, to understand the impact of employed enhancements; addition of `padding` and `margin` CSS properties and preference weighting to reduce the likelihood of a repair generated using `font-size` reduction. We also instructed the participants to use a desktop or laptop for answering the survey to be able to view the IPF UI snippets in full resolution.

Figure 9 shows a sample question from our survey for subject #21 (*twitterHelp*). We finalized the format and phrasing of the question by learning improvements from the pilot studies that we performed internally and with Amazon Mechanical Turk (AMT) participants. PDF copies of all of the surveys used in our user study and the study results can be found in our project repository [30].

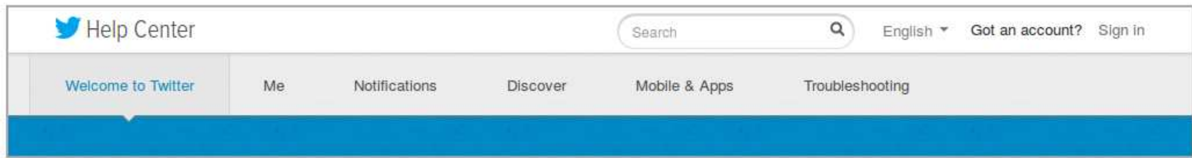
To figure out the number of IPFs to be shown for each subject, we manually analyzed the IPFs reported by GWALI and identified groups of IPFs that shared a common visual pattern. We called these groups "equivalence classes". Figure 10 shows an example of an equivalence class from the hotwire subject, where the two IPFs caused by the price text overflowing the container are highly similar. One IPF from each equivalence class was presented in the survey. We identified and represented equivalence classes in three of our subjects: *hotwire* (two equivalence classes with two and five members, respectively), *mplay* (one equivalence class with nine members), and *worldsBest* (one equivalence class with two members). Thus for the first variant, we presented a total of 64 IPFs across all subjects but one as shown in Table 1. Subject #41 (*sbc*) was excluded from the study since $\mathcal{I}$Fix^{++} was unable to generate a repair for it.

For the second variant, we only considered those IPFs for which a different repair patch was generated by $\mathcal{I}$Fix and $\mathcal{I}$Fix^{++}. This resulted in the evaluation of a total of 13 IPFs from subjects #1, 5, 6, 8, 17, 18, 21, 22, 24, 26, 36, and 45. We selected the 13 IPFs by using the following process. For each subject, we first identified the repair with the best fitness score across the 10 generated by $\mathcal{I}$Fix and $\mathcal{I}$Fix^{++}. Then we used the `diff` utility to compare the repair patches, and only included those IPFs that were reported to have a difference. A qualitative analysis of the repair patches generated by $\mathcal{I}$Fix^{++} for these 13 IPFs revealed that the patches included a fix consisting of `padding` or `margin` properties exclusively or in combination with other CSS properties, such as `width` or `height`. This means that over 20% (13 out of 64) of the IPFs were repaired by $\mathcal{I}$Fix^{++} using the `padding` and `margin` CSS properties.

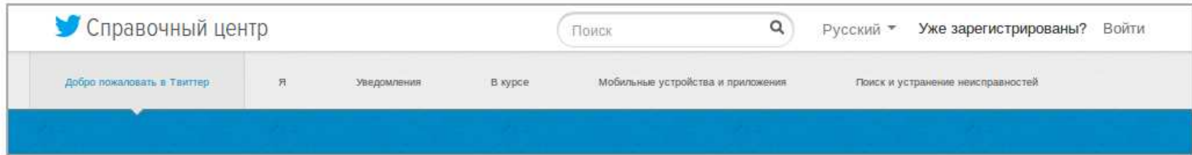
FIGURE 10 UI snippets in the same equivalence class (Hotwire)

To make the survey length manageable for the participants, we divided the 64 IPFs of the first variant over 11 different surveys, with each containing five or six IPFs. Similarly, 13 IPFs of the second variant were presented in three different surveys. Thus we conducted a total of 14 surveys using the AMT service. The participants of our surveys were anonymous users (workers) of the Amazon Mechanical Turk (AMT). For selecting qualified workers to complete our surveys, we only allowed workers that had at least a 95% approval rating for their previously completed tasks and had completed more than 1,000 approved tasks. We had 20 to 40 anonymous participants complete each survey, resulting in a total of 277 completed surveys for the first variant and 80 for the second variant. Each participant was paid $0.20 for completing a survey. To reduce bias, a participant could attempt a selected survey only once, and resubmissions were not allowed.

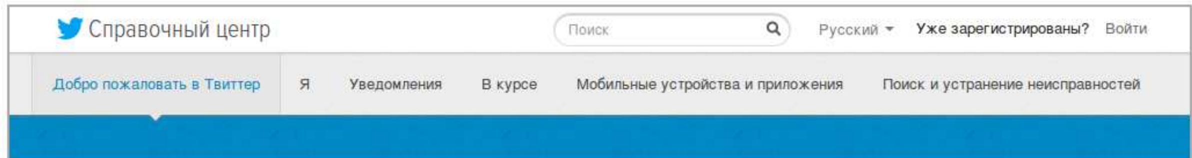
Q. For the below images, how would you rate the ***legibility*** and ***attractiveness*** of **Version 1** and **Version 2**, and their ***appearance similarity*** to the **Reference**?



Reference



Version 1



Version 2

	Which version is more legible? (Legible = are the letters clear? easy to read?)	**Which version is more attractive?** *(Attractive = looks visually appealing?)*	**Which version is more similar in appearance to the Reference?** (Appearance similarity = similar text size, spacing, alignment, etc.?)
Version 1 is much better than Version 2	○	○	○
Version 1 is somewhat better than Version 2	○	○	○
Both versions are the same	○	○	○
Version 2 is much better than Version 1	○	○	○
Version 2 is much better than Version 1	○	○	○
I am not able to decide	○	○	○

Q. Could you explain your answers? We would like to know the reasons for your choice of version for legibility, attractiveness, and appearance similarity. (Optional)

FIGURE 9 A sample question from our user study survey for the subject "twitterHelp"

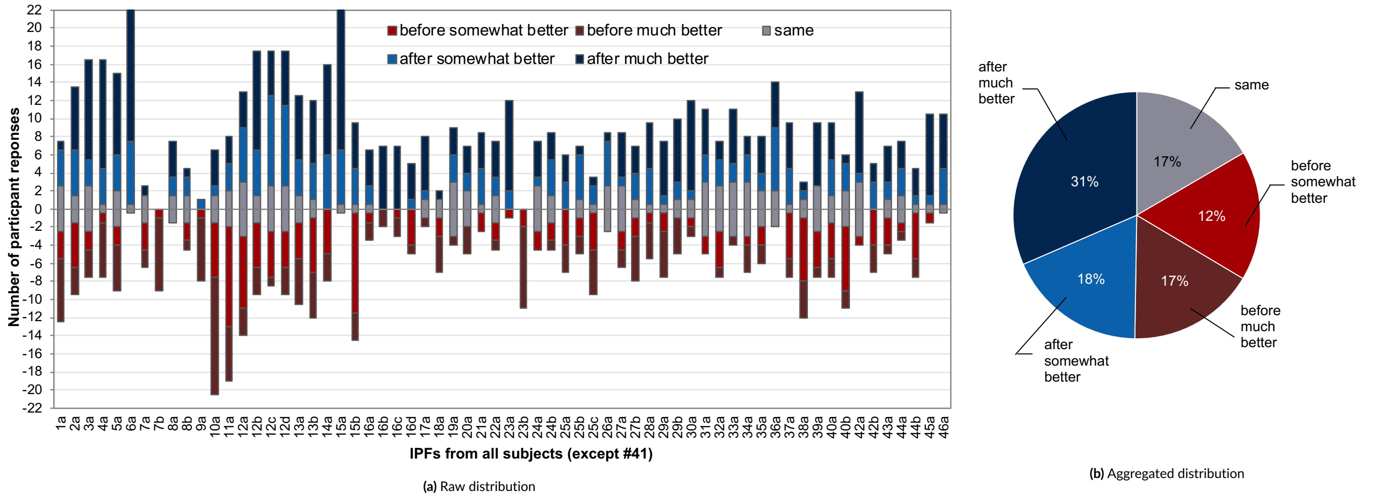
We employed several different ways to ensure analysis of genuine responses in our surveys. To prevent participants from using automated techniques, such as bots, to randomly select answers in the survey, we included a captcha and attention-check questions in our surveys. For each question about attractiveness, readability, and appearance similarity, we also provided the participants with a way to opt out from answering that question if they were unable to decide their preference between the two UI snippets shown. We employed this mechanism to avoid forcing participants to unwillingly select a snippet. Thus, for our analysis, we only considered those participant responses that passed the attention question and captcha checks, and provided some decision about their UI snippet preference. Following this, we had 1,019 participant responses for variant one and 175 participant responses for variant two for analysis, after removing 27% and 33% failed responses from variant one and two, respectively.

4.4.1 Presentation and Discussion of Results for Variant One

Figures 11, 12 and 13 show the results for legibility, attractiveness, and appearance similarity for the first variant of the user study, which compares the PUT before and after repair by $\mathcal{I}$Fix^{++}. The ratings given by the participants for each of the IPFs in the 46 subjects (except the *sbc* subject) are shown in Figures 11a, 12a and 13a, respectively, for the three aspects: legibility, attractiveness, and appearance similarity. Figures 11b, 12b and 13b convey the same information as the bar charts, but in an aggregate format for all of the IPFs under consideration. On the x-axis, the ID and number of IPFs for a subject are shown. For example, Figures 12a, 12b, 12c, and 12d correspond to the *hotwire* subject with four IPFs. The blue colored bars above the x-axis indicate the number of ratings in favor of the *after* version ,i.e., the version of the PUT repaired by $\mathcal{I}$Fix^{++}. The dark blue color shows participants' response for the after version being *much* better than the before version, while the light blue color shows the response for the after version being *somewhat* better than the before version. Similarly, the red bars below the x-axis indicate the number of ratings in favor of the *before* repair version, with dark and light red showing the response for the before version being *much* and *somewhat* better than the after version, respectively. The gray bars show the number of ratings where the participants responded that the before and after versions had the same legibility, attractiveness, or appearance similarity to the baseline. For example, IPF 5a for appearance similarity had a total of 24 responses, thirteen for the after version being much better, five for the after version being somewhat better, two reporting both the versions as the same, three reporting the before version as being somewhat better, and one reporting the before version as being much better. As can be deduced from Figure 13a for appearance similarity, 60% of the participant responses favored the after repair versions, 26% favored the before repair versions, and 14% reported both versions as the same. Similarly, 58% of the participant responses indicated the after repair versions more attractive than the before versions, 24% rated the before versions as more attractive, and 18% found both versions as equally attractive (Figure 12a). For legibility, almost 50% of the participant responses denoted after versions are more legible, 33% favored the before versions, and 17% rated both versions as the same (Figure 11a).

The results of the first variant of our user study show that the participants largely rated the after (repaired) pages as better than the before (faulty) versions on all aspects of legibility, attractiveness, and appearance similarity. This indicates that our approach generates repairs that are high in visual quality. Also, interestingly, the results show the strength of support for the after version — for example, 42% of responses rate the after version as *much* better in appearance similarity, while only 13% responses rate the before version as *much* better. A high number of participants also left insightful comments explaining their choice for the after repair version. One participant rated the after repair version *much better* than the before version for the IPF shown in Figure 2 with the following sentence: "*in {before}, the handicapped sign is blocking the words. how ironic*". Other examples of insightful comments in favor of the after repair versions given by the participants are: "*{Before} has a word moved down into another frame, the text body, and the font is changed. Ugly.*" for IPF 6a, "*{After} has better spacing on the text and {before} the text runs into each other.*" for IPF 15a, and "*{Before} has overlapped words and is much harder to read. {After} is very similar to the original.*" for IPF 23a. Note that in the above quoted sentences the references to UI snippet versions are replaced with their appropriate before or after counterparts.

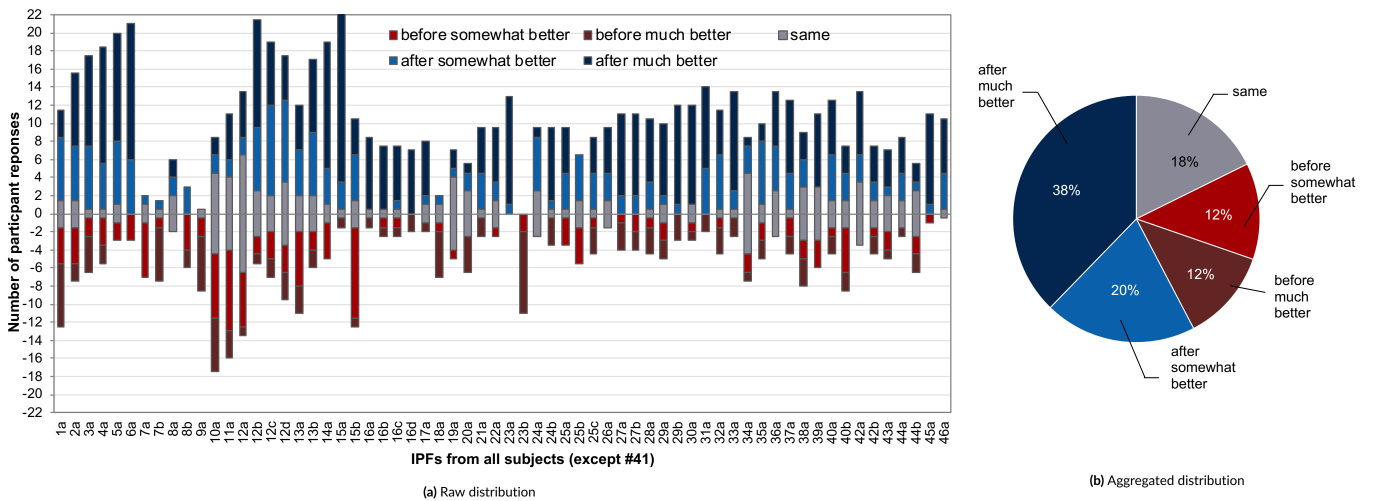
Three of the IPFs, 7b, 9a, and 23b, had more than 80% of the participant responses in the favor of the before version across all of the three aspects of legibility, attractiveness, and appearance similarity. We inspected these subjects in more detail and hypothesized that the primary reason for this was that $\mathcal{I}$Fix^{++} substantially reduced the `font-size` (e.g., from 12px to 8px for 7b) to resolve the IPFs. Our hypothesis was confirmed by the explanation provided the participants for the choice of before versions for these IPFs: e.g., "*{after} is way too tiny and unreadable, {before} looks legible and easy to read*" and "*{After} is more accurate in terms of size but {Before} is better in every other possible way.*". Although these changes were visually unappealing, we were able to confirm that these extreme changes were the only way to resolve the IPFs. We also found that IPFs, 12a, 19a, and 34b, had a majority of the participant responses reporting both versions as the same, also supported by the provided explanation text, for example, "*They're almost the exact same. I can't find a difference.*" and "*They both look the same to me.*". These IPFs were caused by guidance text in an input box being clipped because the translated text exceeded the size of the input box. Unless the survey takers could understand the target language translation, there was no way to know that the guidance text was missing words.



(a) Raw distribution

(b) Aggregated distribution

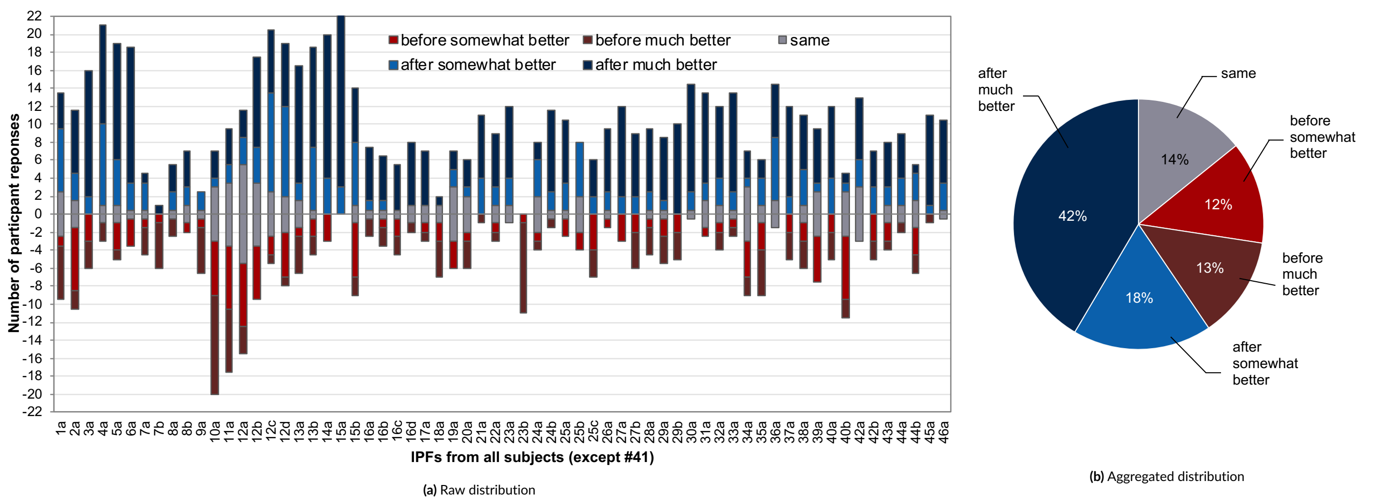
FIGURE 11 Legibility ratings given by user study participants for variant one (before vs after repair by $\mathcal{I}$Fix^{++})



(a) Raw distribution

(b) Aggregated distribution

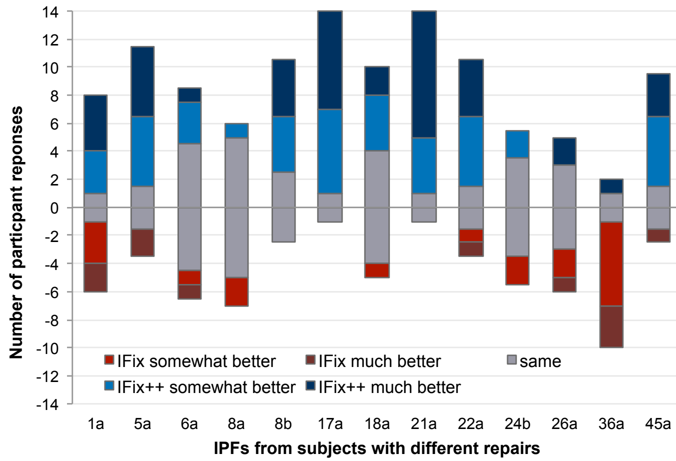
FIGURE 12 Attractiveness ratings given by user study participants for variant one (before vs after repair by $\mathcal{I}$Fix^{++})



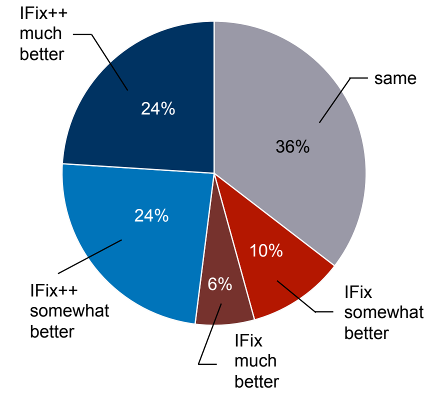
(a) Raw distribution

(b) Aggregated distribution

FIGURE 13 Appearance similarity ratings given by user study participants for variant one (before vs after repair by $\mathcal{I}$Fix^{++})

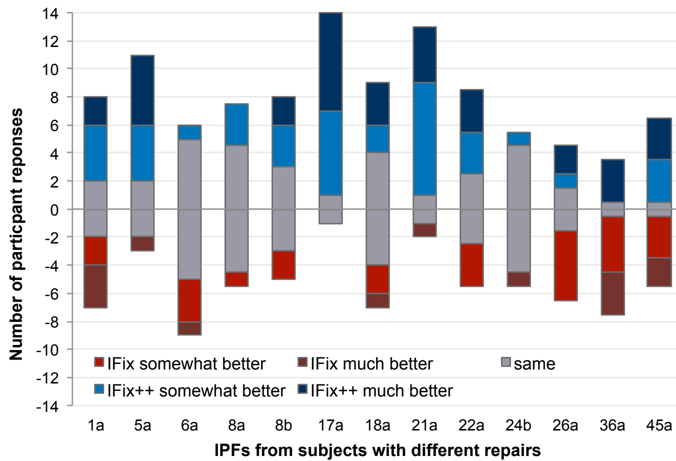


(a) Distribution by IPFs

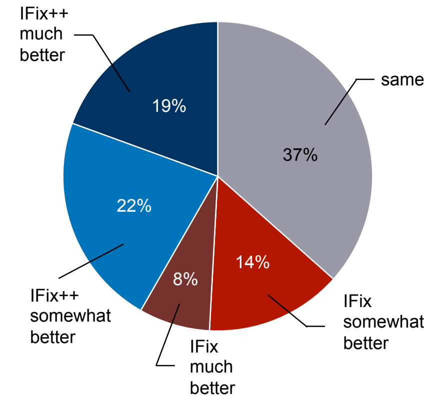


(b) Aggregated distribution

FIGURE 14 Legibility ratings given by user study participants for variant two (repairs generated by $\mathcal{I}$Fix vs $\mathcal{I}$Fix^{++})

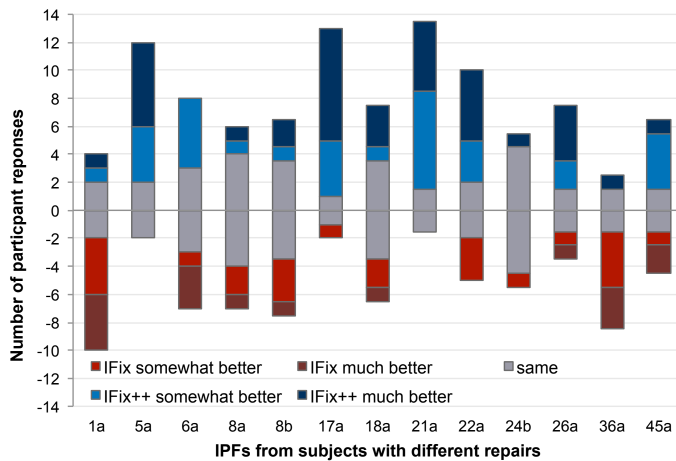


(a) Distribution by IPFs

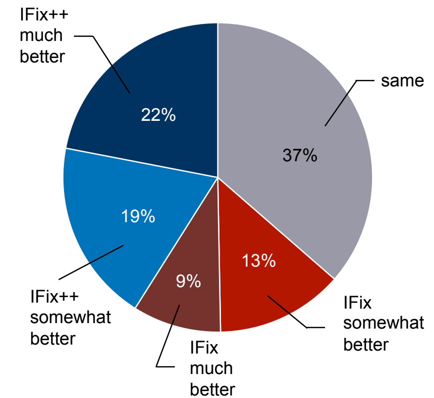


(b) Aggregated distribution

FIGURE 15 Attractiveness ratings given by user study participants for variant two (repairs generated by $\mathcal{I}$Fix vs $\mathcal{I}$Fix^{++})



(a) Distribution by IPFs



(b) Aggregated distribution

FIGURE 16 Appearance Similarity ratings given by user study participants for variant two (repairs generated by $\mathcal{I}$Fix vs $\mathcal{I}$Fix^{++})

4.4.2 Presentation and Discussion of Results for Variant Two

Figures 14, 15 and 16 show the results for the second variant of the user study, which compares the UI snippets of the IPFs repaired by $\mathcal{I}$Fix and $\mathcal{I}$Fix^{++}. The main intent of the study was to evaluate the effectiveness of the new features employed in $\mathcal{I}$Fix^{++}, namely, padding and margin relevant CSS properties (Section 3.3.1) and preference weighting in the fitness function (Section 3.3.4), in repairing IPFs and producing more legible and visually appealing pages compared to $\mathcal{I}$Fix. The ratings given by the participants for each of the 13 IPFs across 12 subjects for which a different repair patch was generated by $\mathcal{I}$Fix and $\mathcal{I}$Fix^{++} are shown in Figures 14a, 15a and 16a, respectively, for the three evaluation aspects: legibility, attractiveness, and appearance similarity. The layout of the graphs is the same as the graphs discussed in variant one, with the only difference that here the red colored bars correspond to the repairs generated by $\mathcal{I}$Fix. Figures 14b, 15b and 16b convey the same information as the bar charts, but in an aggregated format for all the IPFs under consideration.

The results for the second variant of the survey underscored the importance of the new features (inclusion of `padding` and `margin` CSS properties, and preference weighting in selecting a repair) employed by $\mathcal{I}$Fix^{++} compared to $\mathcal{I}$Fix, and their impact on the end users' perception of the page's legibility and overall visual appearance. The results show that the $\mathcal{I}$Fix^{++} repaired versions received *three times* more favorable participant responses for legibility than the $\mathcal{I}$Fix repaired versions. The primary reason for this success is because $\mathcal{I}$Fix^{++} repairs consisted of only spacing adjustments using the `padding` and `margin` properties, without reducing the `font-size`. On the contrary, the $\mathcal{I}$Fix repaired the IPFs by reducing the `font-size`, sometimes significantly as shown in Figure 1c for IPFs 21a, largely affecting the readability of the page. The `font-size` versus `non-font-size` repairs also had an influence on the participant responses for the aspects of attractiveness and appearance similarity. $\mathcal{I}$Fix^{++} repaired versions were rated as more attractive by almost the double the number of participant responses that rated the $\mathcal{I}$Fix versions as better. Interestingly, although both, $\mathcal{I}$Fix^{++} and $\mathcal{I}$Fix, repaired the IPFs, the $\mathcal{I}$Fix^{++} versions were rated as more similar in appearance to the baseline (untranslated) than the $\mathcal{I}$Fix versions by more than twice the number of participant responses rating $\mathcal{I}$Fix versions as better. Comments left by the participants largely highlighted the impact of `font-size` reduction on the overall impression of the page. Examples of such comments are: "*In regards to the font size, {$\mathcal{I}$Fix^{++} repaired version} is much better overall than {$\mathcal{I}$Fix repaired version}. It is more pleasing to the eye.*" for IPF 21a, "*{$\mathcal{I}$Fix^{++} repaired version} fonts are larger and more legible, that makes it much better than {$\mathcal{I}$Fix repaired version}.*" for IPF 17a, and "*I believe {$\mathcal{I}$Fix^{++} repaired version} is better looking than {$\mathcal{I}$Fix repaired version} due to font size.*" for IPF 5a. These results strongly indicate that the new features employed in $\mathcal{I}$Fix^{++} play an important role in generating patches that make the pages more legible and visually appealing to end users.

4.5 Experiment Three

To answer RQ4, we evaluated the accuracy of our style similarity clustering algorithm. For doing this, we compared the clusters produced by $\mathcal{I}$Fix^{++} with the ground truth clusters. Since there exists no other available technique to visually cluster HTML elements in a page, we constructed the ground truth manually. To generate the ground truth for a subject page *p*, we used the following process. (1) take a full-page screenshot of *p* rendered in a browser (e.g., Firefox); (2) obtain the set of potentially faulty elements (*E*) reported by GWALI for *p*; and (3) for each $e \in E$, visually inspect e in the screenshot and find other elements that "look" similar to e, and group them in a cluster. Since creating such ground truth is a time-consuming task, we selected a subset of 15 subjects as shown in Table 3. All 15 subjects, except *deptOfEducation* and *sbc*, were selected randomly. We intentionally added these two subjects to the pool to confirm our hypothesis presented in Experiment One (Section 4.3.2) that $\mathcal{I}$Fix^{++} was unable to generate accurate repairs for these subjects because of the inaccuracies in clustering. To mitigate any bias, we employed an external participant for creating the ground truth for 11 subjects, while we handled the four remaining ones.

We calculated the accuracy of $\mathcal{I}$Fix^{++} in identifying stylistically similar clusters using precision and recall. Specifically, for each subject, we calculated the intra-pair accuracy for each cluster. For a cluster $\mathcal{C}$, we defined *true positive* as an element that is contained in both $\mathcal{C}_g$ and $\mathcal{C}_i$, where $\mathcal{C}_g$ and $\mathcal{C}_i$ represent the ground truth and the result of $\mathcal{I}$Fix^{++}, respectively. A *false positive* is defined as an element that appears in $\mathcal{C}_i$, but not in $\mathcal{C}_g$. Similarly, a *false negative* is an element that is present in $\mathcal{C}_g$, but not in $\mathcal{C}_i$.

4.5.1 Presentation of Results

Table 3 shows the results for RQ4. The number of clusters used for the evaluation are shown under the column, "Cluster". Note that the number of IPFs (i.e., the potentially faulty elements reported by GWALI) shown under the column *#Before* in Table 2 is likely more than the number of clusters, since multiple elements may correspond to the same cluster. The number of true positives, false positives, and false negatives for each cluster are shown under the columns "TP", "FP", and "FN". The columns headed "Precision" and "Recall" show the degree of similarity between the ground truth and the clusters reported by $\mathcal{I}$Fix^{++}.

Figure 17 shows examples for the accuracy calculations. Figures 17a, 17d and 17g depict the ground truth, while Figures 17b, 17e and 17h report the clusters identified by $\mathcal{I}$Fix^{++}. The cluster elements are shown with red boxes. For example, the cluster in the ground truth for the *qualitrol*

TABLE 3 Results for accuracy of clustering

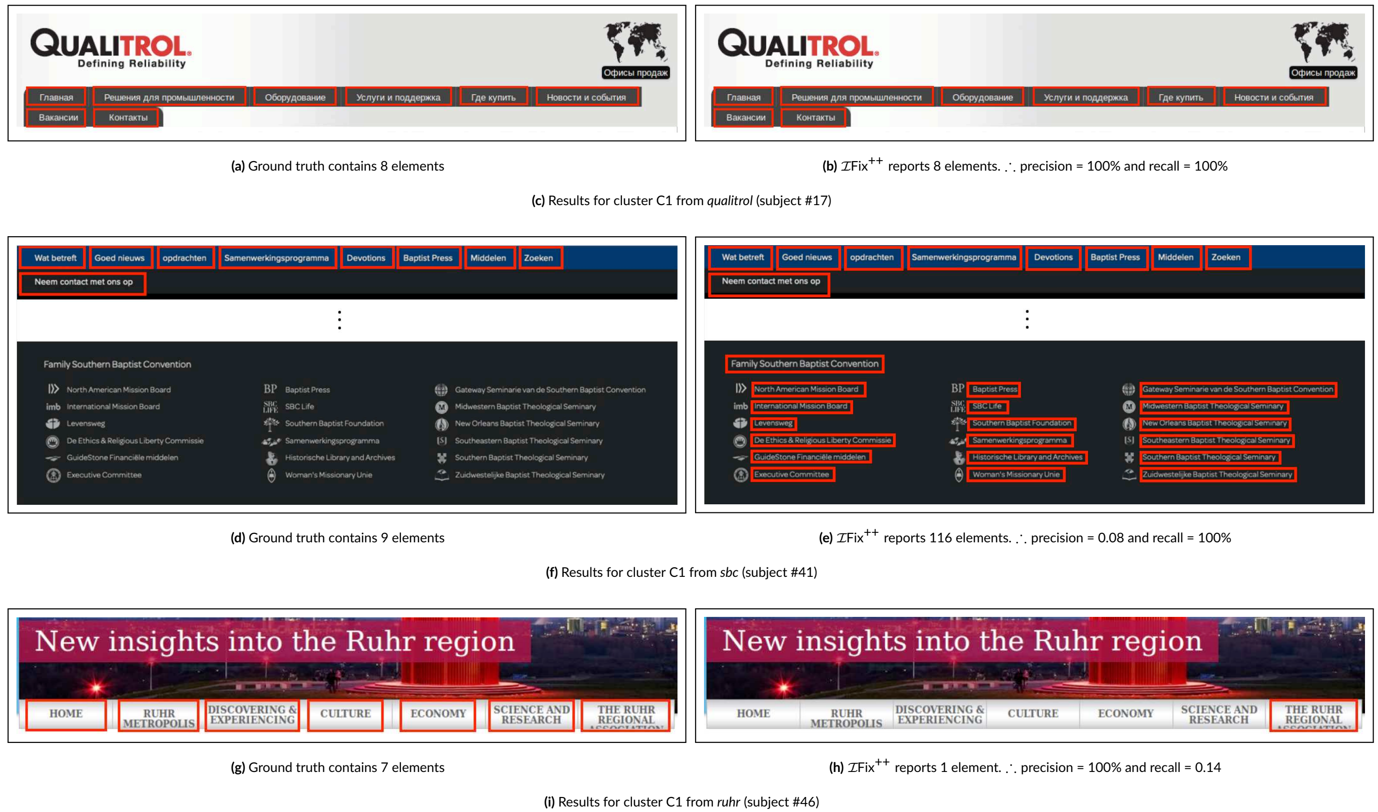
ID	Name	Cluster	TP	FP	FN	Precision	Recall
5	doctor	C1	9	0	0	100%	100%
6	els	C1	3	0	0	100%	100%
9	googleEarth	C1	10	0	0	100%	100%
11	hightail	C1	3	0	0	100%	100%
12	hotwire	C1	1	0	0	100%	100%
		C2	10	20	0	33%	100%
		C3	10	20	0	33%	100%
17	qualitrol	C1	8	0	0	100%	100%
19	skype	C1	1	0	0	100%	100%
24	deptOfEducation	C1	1	0	0	100%	100%
		C2	1	0	0	100%	100%
		C3	26	2	0	93%	100%
		C4	2	273	0	1%	100%
27	namibia	C1	2	0	0	100%	100%
		C2	11	149	0	7%	100%
35	worldCustoms	C1	2	0	0	100%	100%
		C2	2	1	0	67%	100%
38	googleGroups	C1	2	0	0	100%	100%
40	denham	C1	7	79	0	8%	100%
41	sbc	C1	9	107	0	8%	100%
43	geneva	C1	2	0	0	100%	100%
46	ruhr	C1	1	0	6	100%	14%
Average						**75%**	**96%**
Median						**100%**	**100%**

subject contains eight elements (Figure 17a). Similarly, $\mathcal{I}$Fix^{++} reports 116 elements in cluster C1 for *sbc*. Note that in Figure 17e we only show an excerpt of the cluster for this subject.

4.5.2 Discussion of Results

The results show that the clustering algorithm employed in $\mathcal{I}$Fix^{++} achieves a high accuracy: on average, 75% precision and 96% recall, with a median of 100% for both. This demonstrates that the clustering produced by $\mathcal{I}$Fix^{++} follows the human intuition of stylistically similar elements.

$\mathcal{I}$Fix^{++} reported a perfect accuracy (precision = 100% and recall = 100%) in almost 60% of the evaluated clusters (13 out of 22). An example of such a case is shown in Figure 17c. We investigated the cases where $\mathcal{I}$Fix^{++} was not able to achieve a perfect accuracy, i.e., reported false positives or false negatives. Across the evaluated subjects, $\mathcal{I}$Fix^{++} reported a very low false negative rate, less than 5%, thereby resulting in a high recall. $\mathcal{I}$Fix^{++} missed grouping all relevant elements together in only one case, *ruhr*, as shown in Figure 17i. All of the seven menu elements are visually similar and should have been grouped together, however, $\mathcal{I}$Fix^{++} forms a single-element cluster consisting of only the faulty element. On further inspection we found the primary reason for this was that although all of the seven elements look similar visually, the faulty element had different CSS values than the other six elements, setting it apart. For example, the `width` of the faulty element is set to 124px, while the other elements have their `width` value set to 119px. Such differences resulted in a low similarity score causing the faulty element to be clustered by itself. Regardless, this clustering inaccuracy did not have a negative effect on the ability of $\mathcal{I}$Fix^{++} in successfully repairing the IPF, as can be seen from Table 2. False positives were reported in eight clusters across six subjects. Upon examination we found that the main reason for this was when unrelated elements accidentally happened to have same tag names, CSS properties, class attributes, width, height, etc., then by virtue of our similarity metrics everything (related and unrelated) was grouped into one cluster. We found that this problem is more pronounced for web pages that use the Google Translate Framework [20] for internationalization. This is because the framework wraps all text elements in the page with ⟨`font`⟩ tags that make them very similar to each other structurally. In addition to this, when the algorithm finds similarity with respect to the visual metrics (e.g., CSS properties and width/height) as well, then it is not able accurately distinguish the elements into proper clusters. Such



(a) Ground truth contains 8 elements

(b) $\mathcal{I}$Fix^{++} reports 8 elements. ∴ precision = 100% and recall = 100%

(c) Results for cluster C1 from *qualitrol* (subject #17)

(d) Ground truth contains 9 elements

(e) $\mathcal{I}$Fix^{++} reports 116 elements. ∴ precision = 0.08 and recall = 100%

(f) Results for cluster C1 from *sbc* (subject #41)

(g) Ground truth contains 7 elements

(h) $\mathcal{I}$Fix^{++} reports 1 element. ∴ precision = 100% and recall = 0.14

(i) Results for cluster C1 from *ruhr* (subject #46)

FIGURE 17 Examples showing the accuracy of the clustering algorithm used in $\mathcal{I}$Fix^{++}

inaccuracy can have varying effects on the ability of $\mathcal{I}$Fix^{++} in finding a successful repair. For subjects *deptOfEducation* and *sbc*, which are indeed designed using the Google Translate Framework, the high false positive rate restricted $\mathcal{I}$Fix^{++} in finding a successful repair. Figure 17f shows the clustering problem in the *sbc* subject. Along with the true positive menu elements, many unrelated elements such as footer menu items and page body headings, were also included in the cluster. Conversely, the false positives did not affect three (*hotwire*, *namibia*, *and worldCustoms*) of the six subjects; $\mathcal{I}$Fix^{++} was able to perfectly resolve all of the IPFs. $\mathcal{I}$Fix^{++} was not able to completely resolve the IPF in *denham*, however, as discussed in Section 4.3.2 this was due to the inaccuracy in GWALI, not related to the clustering false positives.

Overall, these results are promising and indicate that the clustering algorithm can indeed group stylistically similar elements together with a high accuracy. As can also be seen, the imperfections in the clustering do not significantly impede the ability of $\mathcal{I}$Fix^{++} in finding a successful repair. Neither do the inaccuracies have a strong negative impact on the visual appeal of the pages, as was observed from the user study responses (Section 4.4).

4.6 Threats to Validity

External Validity: A potential threat is that a majority of the subjects (#1–38) used in the evaluation are from our previous work [2, 33, 1]. We chose to use these subjects only because they were known to exhibit IPFs. Moreover, these subjects in their respective studies were selected using an unbiased criteria from varied sources, such as Alexa top 100 most visited websites, random URL generator (URLOUTTE), websites listed on *builtwith.com*, and high-profile websites targeting international audiences (e.g., travel-related and telecom company websites). To further mitigate any bias and check generalizability of the results, we used eight new subjects (#39–46) for evaluating $\mathcal{I}$Fix^{++} which were drawn from a random URL generator. A potential threat is bias in the selection of participants for our user study in Experiment Two. To address this threat, we used AMT that allowed us to engage anonymous participants to undertake the surveys. We also followed AMT best practices to select the participants to ensure authentic results as explained in Section 4.4. Another possible threat is that the target language (i.e., the language in which the PUT is translated) may affect $\mathcal{I}$Fix^{++}'s results. However, we observed that the target language does not have any correlation with the effectiveness of $\mathcal{I}$Fix^{++}. For example, subjects *skype* and *portland* both are translated from English to French, however the IPFs of only the former were fully resolved. More generally, the design and layout of the page impacts the IPFs, rather than the target language. Therefore, it cannot be affirmatively concluded

which target languages can be problematic that developers should be wary of. Another threat is that $\mathcal{I}$Fix^{++} does not support RTL translations (e.g., Arabic). We inherit this limitation from GWALI, which currently only supports LTR (e.g., German) and TBRL (e.g., Japanese) writing systems. In the future, we plan to extend the support of both, GWALI and $\mathcal{I}$Fix^{++}, for RTL languages.

Internal Validity: One potential threat is the use of only GWALI for detecting IPFs. However, there exist no other available automated tools that can detect IPFs and report potentially faulty HTML elements. Another potential threat to internal validity is that $\mathcal{I}$Fix^{++} may not be able to repair IPFs in the absence of CSS (i.e., the page is purely HTML with no explicitly defined CSS). In such cases, $\mathcal{I}$Fix^{++} uses the default CSS values that are defined by the browser to find a repair. For example, the default value of `font-size` for ⟨`h1`⟩ element is set to 2em in Firefox [48]. Another potential threat is that $\mathcal{I}$Fix^{++} may generate a repair with very small values for CSS properties (e.g., `margin` and `padding`). A simple way to overcome this problem can be to prevent the search from decreasing the value below a predefined threshold. However, this may prevent $\mathcal{I}$Fix^{++} from finding a successful repair. Therefore, in our approach we designed the amount of change component in the fitness function to penalize such extreme reductions if there exist other repairs.

Construct Validity: A potential threat is that we manually categorized IPFs into equivalence classes for the user study. However, this categorization was fairly straightforward, and in practice there was no ambiguity regarding membership in an equivalence class, for example, as shown in Figure 10. To further support this, we have made the surveys and subject pages publicly available [30] for verification. Another potential threat to construct validity is that we presented UI snippets of the subject pages to the participants, rather than full-page screenshots, which might have an impact on their appearance similarity ratings. We opted for this mechanism as the full page screenshots of the subjects were large in size, making it difficult to view all three screenshots, baseline, before (faulty), and after (repaired), in one frame for comparison. The benefit of this mechanism was that it allowed the participants to focus only on the areas of the pages that contained IPFs and were thus modified by $\mathcal{I}$Fix^{++}.

5 Related Work

Different techniques exist that target detection of internationalization failures in web applications. GWALI [2] and i18n checker [70] are automated techniques, while Apple's pseudo-localization [4] requires manual checking to identify IPFs. There are also techniques [6, 5, 59] that perform automated checks for identifying internationalization problems, such as corrupted text, inconsistent keyboard shortcuts, and incorrect/missing translations. Conversely, another technique [74] provides automated domain specific translations using recurrent neural networks. Several techniques [13, 12, 60, 81, 55, 77, 79] are designed to detect and Cross-Platform Incompatibilities and Cross-Browser Issues (XBIs) in web pages by analyzing HTML, CSS, and Javascript. Yet other techniques, such as ReDeCheck [3, 73, 71, 72], WebSee [36, 37, 35, 39, 38, 34], VFDetector [61], CANVASURE [7], Ply [27], Fighting Layout Bugs [67], Sikuli [11], and techniques based on computer vision [68] and CSS/Javascript analysis [47], detect certain types of presentation failures in web pages. There also exists a group of parallel techniques [44, 43, 45] focusing on the detection and reporting of GUI violations in mobile apps. However, none of them are designed to repair IPFs.

Another technique related to internationalization in web pages is TranStrL [75]. It assists developers by identifying strings in a web application that need to be translated during the process of its internationalization, and as such is not designed for repairing IPFs.

A group of approaches from the research community focus on repairing different types of UI problems in web applications, but none of them can repair IPFs. XFix [31, 32] and MFix [29] use search-based techniques to repair Cross-Browser Issues (XBIs) and mobile friendly problems in web pages, respectively. However, the correctness criteria of these UI problems is different from the domain of IPFs, making XFix and MFix not applicable for the repair of IPFs. PhpRepair [63] and PhpSync [51] focus on repairing problems arising from malformed HTML. IPFs are, however, not caused by malformed HTML, meaning these techniques would not resolve IPFs. Another technique assumes that an HTML/CSS fix has been found and focuses on propagating it to the server-side using hybrid analysis [76]. Cassius [58, 57, 56] is a framework for repairing faulty CSS in web pages by using the CSS information extracted from the given page layout examples as the oracle. In the IPF domain, however, the pages before and after translation share the same CSS files. Therefore this technique cannot be used for repairing IPFs. Monperrus et al. provide yet other techniques [15, 54, 53, 66, 26] for debugging different problems in web applications in their survey on automated program repair [42]. However, none of these techniques can be used for the debugging of IPFs.

Responsive Web Design (RWD) techniques and frameworks are effective in designing websites that can automatically adapt to different screen resolutions and their use can help reduce the appearance of IPFs. However, using RWD cannot guarantee that the resulting web pages will be IPF free. Frameworks, such as Bootstrap [9], require developers to annotate HTML elements with classes that have pre-defined responsive behaviors, while techniques, such as DECOR [65] and ORC layout [21], require developers to provide "user-specified design constraints". These specifications are limited, which make these techniques unable to prevent all types of IPFs. Also, these specifications are provided manually by developers, which makes specifying them time consuming and error-prone; it is easy for the developers to specify wrong annotations or constraints. Further, if IPFs are observed in such responsively designed pages, our approach can be directly run on them to find repairs. This is because our approach operates

on HTML and CSS of the page after it has rendered in a browser, this makes our approach agnostic to the underlying framework (e.g., RWD) used to develop the web page.

Automated program repair (APR) has been an area of active research. GenProg [78], PAR [23], SPR [28], and NPEFix [14] are examples of template-based generate-and-validate APR approaches, which compile and test each candidate patch to collect validated patches. Other approaches (e.g., [52, 82, 50]) use data-driven or synthesis based techniques to generate repairs. However, none of them are capable of repairing presentation failures (e.g., IPFs) in web applications because they are structured to work for general-purpose programming languages (e.g., Java and C). Furthermore, repair templates for IPFs cannot be defined a priori, since their fixes are hard to generalize given the app-specific complex interaction between HTML and CSS.

Lastly, there are several techniques in the field of GUI testing by Memon et al. [41, 80, 46, 49] that are focused on testing the functionality of the software systems by triggering test sequences from the systems' Graphical User Interface (GUI). Although they could be helpful for aiding in the detection of IPFs they are not able to repair GUI problems.

6 Conclusion

Translation of a web page into different languages can cause changes in the text size. These changes make HTML elements expand, shrink, or move in order to handle the translated text, which can lead to distortions in the layout of the translated web page. These distortions are known as Internationalization Presentation Failures (IPFs). In this paper, we extend our previous work for repairing IPFs in web pages. Our approach uses clustering to group stylistically similar elements in a page. Then it performs a guided search to find suitable CSS fixes for the identified clusters. In the evaluation, our approach was able to resolve 94% of the reported IPFs on a set of 46 web pages. In the user study, participant responses rated the fixed versions generated by our approach as more legible, more attractive, and more similar to the baseline than the fixed versions generated by our previous work. Overall, these results are positive and indicate that our approach is helpful in automatically repairing IPFs in web pages.

Acknowledgment

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

References

[1] Alameer, A. and W. G. Halfond, 2016: An empirical study of internationalization failures in the web. *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*.

[2] Alameer, A., S. Mahajan, and W. G. Halfond, 2016: Detecting and localizing internationalization presentation failures in web applications. *Proceedings of the 9th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*.

[3] Althomali, I., G. M. Kapfhammer, and P. McMinn, 2019: Automatic visual verification of layout failures in responsively designed web pages. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 183–193.

[4] Apple, 2017: *Apple Internationalization and Localization Guide*. https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPInternational/TestingYourInternationalApp/TestingYourInternationalApp.html.

[5] Archana, J., S. R. Chermapandan, and S. Palanivel, 2013: Automation framework for localizability testing of internationalized software. *International Conference on Human Computer Interactions (ICHCI)*.

[6] Awwad, A. M. A. and W. Slany, 2016: Automated Bidirectional Languages Localization Testing for Android Apps with Rich GUI. *Mobile Information Systems*.

[7] Bajammal, M. and A. Mesbah, 2018: Web canvas testing through visual inference. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 193–203.

[8] Bavishi, R., H. Yoshida, and M. R. Prasad, 2019: Phoenix: Automated data-driven synthesis of repairs for static analysis violations. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*,

Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, 613–624.
URL https://doi.org/10.1145/3338906.3338952

[9] Bootstrap, 2019: *Responsive Web Design Framework*.
URL http://getbootstrap.com

[10] Cai, D., S. Yu, J.-R. Wen, and W.-Y. Ma, 2003: VIPS: a Vision-based Page Segmentation Algorithm. Microsoft Technical Report, MSR-TR-2003-79.

[11] Chang, T.-H., T. Yeh, and R. C. Miller, 2010: GUI testing using computer vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '10, 1535–1544.
URL http://doi.acm.org/10.1145/1753326.1753555

[12] Choudhary, S. R., M. R. Prasad, and A. Orso, 2012: CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST.

[13] — 2013: X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications. *Proceedings of the 35th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE)*.

[14] Cornu, B., T. Durieux, L. Seinturier, and M. Monperrus, 2015: NPEFix: Automatic runtime repair of null pointer exceptions in Java. 1512.07423. Arxiv.
URL https://arxiv.org/pdf/1512.07423.pdf

[15] Durieux, T., Y. Hamadi, and M. Monperrus, 2018: "Fully Automated HTML and Javascript Rewriting for Constructing a Self-healing Web Proxy. *29th IEEE International Symposium on Software Reliability Engineering*.

[16] Egger, F. N., 2000: "Trust Me, I'm an Online Vendor": Towards a Model of Trust for e-Commerce System Design. *CHI Extended Abstracts on Human Factors in Computing Systems*, ACM.

[17] Ester, M., H. peter Kriegel, J. S, and X. Xu, 1996: A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD.

[18] Everard, A. and D. F. Galletta, 2006: How Presentation Flaws Affect Perceived Site Quality, Trust, and Intention to Purchase from an Online Store. *Journal of Management Information Systems*, **22**, 56–95.

[19] Fogg, B. J., J. Marshall, O. Laraki, A. Osipovich, C. Varma, N. Fang, J. Paul, A. Rangnekar, J. Shon, P. Swani, and M. Treinen, 2001: What Makes Web Sites Credible?: A Report on a Large Quantitative Study. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI.

[20] Google, 2019: *Google Translate Tools*.
URL https://translate.google.com/intl/en/about/

[21] Jiang, Y., R. Du, C. Lutteroth, and W. Stuerzlinger, 2019: ORC layout: Adaptive GUI layout with OR-constraints. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '19, 413:1–413:12.
URL http://doi.acm.org/10.1145/3290605.3300643

[22] Kempka, J., P. McMinn, and D. Sudholt, 2015: Design and Analysis of Different Alternating Variable Searches for Search-Based Software Testing. *Theoretical Computer Science*, volume 605, 1–20.

[23] Kim, D., J. Nam, J. Song, and S. Kim, 2013: Automatic patch generation learned from human-written patches. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, ICSE '13, 802–811.

[24] Korel, B., 1990: Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, **16**, no. 8, 870–879.

[25] Le Goues, C., M. Dewey-Vogt, S. Forrest, and W. Weimer, 2012: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. *Proceedings of the 34th International Conference on Software Engineering*, ICSE, 3–13.

[26] Leotta, M., A. Stocco, F. Ricca, and P. Tonella, 2018: Pesto: Automated migration of DOM-based Web tests towards the visual approach. *Software Testing, Verification and Reliability*, **28**, no. 4, doi:10.1002/stvr.1665.
URL https://doi.org/10.1002/stvr.1665

[27] Lim, S., J. Hibschman, H. Zhang, and E. O'Rourke, 2018: Ply: A visual web inspector for learning from professional webpages. *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '18, 991–1002.
URL http://doi.acm.org/10.1145/3242587.3242660

[28] Long, F. and M. Rinard, 2015: Staged program repair with condition synthesis. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE.

[29] Mahajan, S., N. Abolhassani, P. McMinn, and W. G. J. Halfond, 2018: Automated Repair of Mobile Friendly Problems in Web Pages. *Proceedings of the 40th International Conference on Software Engineering (ICSE)*.

[30] Mahajan, S. and A. Alameer, 2018: *IFix Evaluation Data*. https://github.com/USC-SQL/ifix.

[31] Mahajan, S., A. Alameer, P. McMinn, and W. G. Halfond, 2017: Automated Repair of Layout Cross Browser Issues using Search-Based Techniques. *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA)*.

[32] — 2017: XFix: Automated Tool for Repair of Layout Cross Browser Issues. *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA) – Tool Track*.

[33] Mahajan, S., A. Alameer, P. McMinn, and W. G. J. Halfond, 2018: Automated Repair of Internationalization Presentation Failures in Web Pages Using Style Similarity Clustering and Search-Based Techniques. *Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[34] Mahajan, S., K. B. Gadde, A. Pasala, and W. G. J. Halfond, 2016: Detecting and Localizing Visual Inconsistencies in Web Applications. *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC) – Short paper*.

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

[36] — 2015: Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[37] — 2015: WebSee: A Tool for Debugging HTML Presentation Failures. *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST) – Tool track*.

[38] Mahajan, S., B. Li, P. Behnamghader, and W. G. J. Halfond, 2016: Using Visual Symptoms for Debugging Presentation Failures in Web Applications. *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[39] Mahajan, S., B. Li, and W. G. J. Halfond, 2014: Root Cause Analysis for HTML Presentation Failures Using Search-based Techniques. *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST)*.

[40] McMinn, P. and G. M. Kapfhammer, 2016: AVMf: An open-source framework and implementation of the alternating variable method. *International Symposium on Search-Based Software Engineering (SSBSE 2016)*, Springer, volume 9962 of *Lecture Notes in Computer Science*, 259–266.

[41] Memon, A. M., I. Banerjee, and A. Nagarajan, 2003: What Test Oracle Should I Use for Effective GUI Testing? *ASE*.

[42] Monperrus, M., 2018: The living review on automated program repair. hal-01956501. HAL/archives-ouvertes.fr.

[43] Moran, K., 2018: Automating software development for mobile computing platforms. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 749–754.

[44] Moran, K., B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, 2018: Automated reporting of GUI design violations for mobile apps. *Proceedings of the 40th International Conference on Software Engineering*, ACM, New York, NY, USA, ICSE '18, 165–175.
URL http://doi.acm.org/10.1145/3180155.3180246

[45] Moran, K., C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk, 2018: Detecting and summarizing GUI changes in evolving mobile apps. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, New York, NY, USA, ASE 2018, 543–553. URL http://doi.acm.org/10.1145/3238147.3238203

[46] Moreira, R. M. L. M., A. C. R. Paiva, and A. Memon, 2013: A Pattern-Based Approach for GUI Modeling and Testing. *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE.

[47] Moyeen, M. A., G. G. M. N. Ali, P. H. J. Chong, and N. Islam, 2016: An automatic layout faults detection technique in responsive web pages considering javascript defined dynamic layouts. *2016 3rd International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, 1–5.

[48] Mozilla, 2019: *Default Styles for Firefox*.
URL https://dxr.mozilla.org/mozilla-central/source/layout/style/res/html.css

[49] Nguyen, B. N., B. Robbins, I. Banerjee, and A. Memon, 2014: Guitar: An innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, **21**, no. 1, 65–105, doi:10.1007/s10515-013-0128-9.
URL http://dx.doi.org/10.1007/s10515-013-0128-9

[50] Nguyen, H. D. T., D. Qi, A. Roychoudhury, and S. Chandra, 2013: Semfix: Program repair via semantic analysis. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, ICSE '13, 772–781.

[51] Nguyen, H. V., H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, 2011: Auto-locating and Fix-propagating for HTML Validation Errors to PHP Server-side Code. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE.

[52] Nguyen, T. T., H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, 2010: Recurring Bug Fixes in Object-oriented Programs. *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE.

[53] Ocariza, F. S., Jr., K. Pattabiraman, and A. Mesbah, 2014: Vejovis: Suggesting fixes for JavaScript faults. *Proceedings of the 36th International Conference on Software Engineering*, ACM, New York, NY, USA, ICSE 2014, 837–847.
URL http://doi.acm.org/10.1145/2568225.2568257

[54] Ocariza Jr., F. S., K. Pattabiraman, and A. Mesbah, 2012: AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE Computer Society, Washington, DC, USA, ICST '12, 31–40.

[55] Paes, F. C. and W. M. Watanabe, 2018: Layout cross-browser incompatibility detection using machine learning and DOM segmentation. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ACM, New York, NY, USA, SAC '18, 2159–2166.
URL http://doi.acm.org/10.1145/3167132.3167364

[56] Panchekha, P., M. D. Ernst, Z. Tatlock, and S. Kamil, 2019: Modular verification of web page layout. *Proc. ACM Program. Lang.*, **3**, no. OOPSLA, 151:1–151:26, doi:10.1145/3360577.
URL http://doi.acm.org/10.1145/3360577

[57] Panchekha, P., A. T. Geller, M. D. Ernst, Z. Tatlock, and S. Kamil, 2018: Verifying that web pages have accessible layout. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, PLDI 2018, 1–14.
URL http://doi.acm.org/10.1145/3192366.3192407

[58] Panchekha, P. and E. Torlak, 2016: Automated Reasoning for Web Page Layout. *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA.

[59] Ramler, R. and R. Hoschek, 2017: How to test in sixteen languages? automation support for localization testing. *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[60] Roy Choudhary, S., H. Versee, and A. Orso, 2010: WEBDIFF: Automated identification of cross-browser issues in web applications. *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10.

[61] Ryou, Y. and S. Ryu, 2018: Automatic detection of visibility faults by layout changes in HTML5 web pages. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 182–192.

[62] Saha, R. K., Y. Lyu, H. Yoshida, and M. R. Prasad, 2017: Elixir: Effective object oriented program repair. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, ASE 2017, 648–659.

[63] Samimi, H., M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, 2012: Automated repair of HTML generation errors in PHP applications using string constraint solving. *Proceedings of the International Conference on Software Engineering*, ICSE.

[64] Sanoja, A. and S. Gançarski, 2014: Block-o-Matic: A web page segmentation framework. *Proceedings of the International Conference on Multimedia Computing and Systems*, ICMCS.

[65] Sinha, N. and R. Karim, 2015: Responsive designs in a snap. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, New York, NY, USA, ESEC/FSE 2015, 544–554.
URL http://doi.acm.org/10.1145/2786805.2786808

[66] Son, S., K. S. Mckinley, and V. Shmatikov, 2013: Fix me up: Repairing access-control bugs in web applications. *In Network and Distributed System Security Symposium*.

[67] Tamm, M., 2019: *Fighting Layout Bugs*. https://code.google.com/p/fighting-layout-bugs/.

[68] Tanno, H. and Y. Adachi, 2018: Support for finding presentation failures by using computer vision techniques. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 356–363.

[69] Uroulette, 2019: *Random URL Generator*.
URL http://www.uroulette.com/

[70] W3C, 2019: *W3C Internationalization Checker*. https://validator.w3.org/i18n-checker/.

[71] Walsh, T., G. Kapfhammer, and P. McMinn, 2017: Automated Layout Failure Detection for Responsive Web Pages without an Explicit Oracle. *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA)*.

[72] Walsh, T. A., G. M. Kapfhammer, and P. McMinn, 2017: ReDeCheck: An automatic layout failure checking tool for responsively designed web pages. *International Conference on Software Testing and Analysis (ISSTA 2017)*, 360–363.

[73] Walsh, T. A., P. McMinn, and G. M. Kapfhammer, 2015: Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages. *International Conference on Automated Software Engineering (ASE)*.

[74] Wang, X., C. Chen, and Z. Xing, 2019: Domain-specific machine translation with recurrent neural network for software localization. *Empirical Software Engineering*.

[75] Wang, X., L. Zhang, T. Xie, H. Mei, and J. Sun, 2010: Locating Need-to-Translate Constant Strings in Web Applications. *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE.

[76] Wang, X., L. Zhang, T. Xie, Y. Xiong, and H. Mei, 2012: Automating presentation changes in dynamic web applications via collaborative hybrid analysis. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE.

[77] Watanabe, W. M., G. L. Amêndola, and F. C. Paes, 2019: Layout cross-platform and cross-browser incompatibilities detection using classification of dom elements. *ACM Trans. Web*, **13**, no. 2, 12:1–12:27, doi:10.1145/3316808.
URL http://doi.acm.org/10.1145/3316808

[78] Weimer, W., T. Nguyen, C. Le Goues, and S. Forrest, 2009: Automatically finding patches using genetic programming. *Proceedings of the 31st International Conference on Software Engineering*, ICSE.

[79] Wu, G., M. He, H. Tang, and J. Wei, 2016: Detect cross-browser issues for JavaScript-based web applications based on record/replay. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 78–87.

[80] Xie, Q. and A. M. Memon, 2006: Studying the Characteristics of a "Good" GUI Test Suite. *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE.

[81] Xu, S., C. Zhou, Z. Gu, G. Wu, W. Chen, and J. Wei, 2018: X-diag: Automated debugging cross-browser issues in web applications. *2018 IEEE International Conference on Web Services (ICWS)*, 66–73.

[82] Zhang, S., H. Lü, and M. D. Ernst, 2013: Automatically Repairing Broken Workflows for Evolving GUI Applications. *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA.

How to cite this article: S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond (2019), Effective Repair of Internationalization Presentation Failures in Web Applications Using Style Similarity Clustering and Search-Based Techniques, *Softw Test Verif Reliab.*, *Journal volume*.