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

Sonal Mahajan*§, Abdulmajeed Alameer*§, Phil McMinn†, William G. J. Halfond*
*University of Southern California, USA {alameer, spmahaja, halfond}@usc.edu
†University of Sheffield, UK {p.mcminn}@sheffield.ac.uk

***Abstract*—Internationalization enables companies to reach a global audience by adapting their websites to locale specific language and content. However, such translations can often introduce Internationalization Presentation Failures (IPFs) — distortions in the intended appearance of a website. It is challenging for developers to design websites that can inherently adapt to varying lengths of text from different languages. Debugging and repairing IPFs is complicated by the large number of HTML elements and CSS properties that define a web page's appearance. Tool support is also limited as existing techniques can only detect IPFs, with the repair remaining a labor intensive manual task. To address this problem, we propose a search-based technique for automatically repairing IPFs in web applications. Our empirical evaluation showed that our approach was able to successfully resolve 98% of the reported IPFs for 23 real-world web pages. In a user study, participants rated the visual quality of our fixes significantly higher than the unfixed versions.**

I. Introduction

Web applications enable companies to easily establish a global presence. To more effectively communicate with this global audience, companies often employ internationalization (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 [4]. Avoiding presentation problems, such as these, is important. Studies show that the design 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 [11], [13], [14].

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.

Existing techniques targeting internationalization problems, such as GWALI [5], 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 [19], which repairs cross-browser issues; and PhpRepair [36] and PhpSync [30], which repair malformed HTML.

In this paper, we present an approach for automatically repairing IPFs in web pages. Our approach is designed to 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 the large solution space defined by the HTML elements and CSS properties. 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. In an evaluation of the implementation of our approach, we found that it was effective at repairing IPFs, resolving over 98% of the detected IPFs; and also fast, requiring about four minutes on average to generate the repair. In a user study of the repaired web pages, we found that the repairs met with high user approval — over 70% of user responses rated the repaired pages as better than the faulty versions. Overall, these results are positive and indicate that

§ Equal contribution by Sonal Mahajan and Abdulmajeed Alameer.

our approach can help developers automatically resolve IPFs in web pages.

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 empirical study on a large set of real-world web pages whose results show that our approach is effective and fast in repairing IPFs,
3) 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 II we present background information about internationalization and IPFs. Then in Section III we describe the approach in detail and its evaluation in Section IV. We discuss related work in Section V and conclude in Section VI.

II. Background

Developers internationalize web applications by isolating language-specific content, such as text, icons, and media, into resource files. Different sets of resource files can then be utilized 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 2b. Here, the text of the page in Figure 2a has been translated, but the increased number of characters required by the translated text pushes the final link of the navigation bar under an icon, making it difficult to read and click. 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.

The complete process of debugging an IPF requires developers to (1) detect when an IPF occur 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. An existing technique, GWALI [5], 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.) 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", "contains" etc.) 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. Our approach leverages 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. This challenge motivates our approach, which we present in the next section.

III. Approach

The goal of our approach is to automatically repair IPFs that have been detected in a translated version of a web page. As described in Section II, 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.

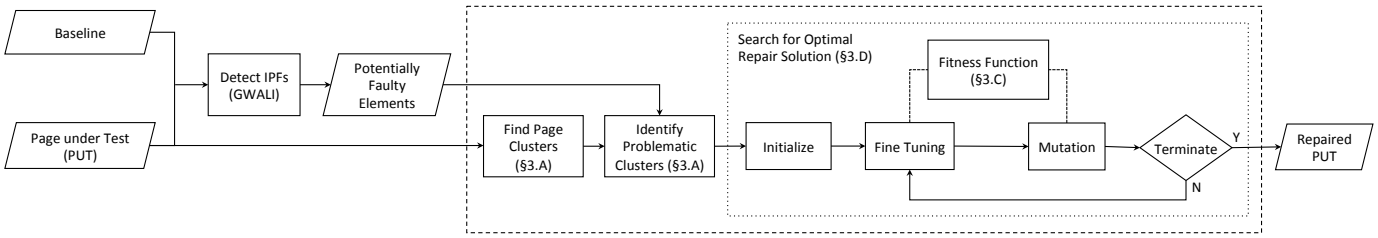
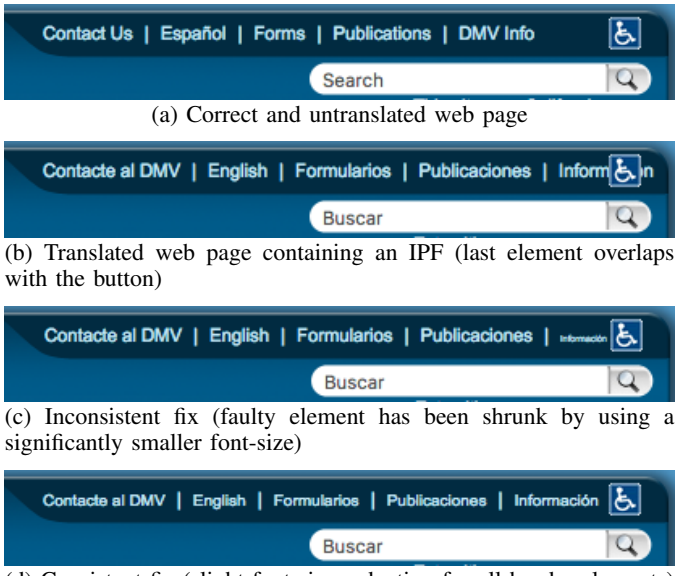


Fig. 1: Overview of our approach

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. This is illustrated 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, which subtly decreases the font size of all of the stylistically related elements in the header. This challenge requires that our 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 find 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. 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 [5]) and other metrics for measuring the amount of difference between two UI layouts. Therefore, the goal of our search-based approach is to find a solution (i.e., new CSS values) that minimizes this fitness function.

Figure 1 shows an overview of our approach. The inputs to the approach are: a version of the web page (*baseline*) that shows its correct layout, a translated version (*PUT*) that exhibits IPFs, and 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 a conservative list of possibly faulty HTML elements, but the use of an automated detection technique allows the debugging process to be fully automated. Our approach begins by analyzing the PUT and identifying the stylistically similar clusters that include the potentially faulty elements. Then, the approach performs a guided search to find the best CSS values for each of the identified clusters. When the search terminates, the best CSS values obtained from all of the clusters are converted to a web page CSS repair patch and provided as the output of the approach. We now explain the parts of the approach in more detail in the following subsections.



(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)

Fig. 2: Example of an IPF on the DMV homepage (https://www.dmv.ca.gov) when translated from English to Spanish and different ways of fixing the IPF

A. *Identifying Stylistically Similar Clusters*

The goal of this step is to group HTML elements in the page that are visually similar into Sets of Stylistically Similar Elements (*SimSets*). 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. 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) [8], Block-o-Matic [37], and R-Trees [23]. 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 [12]. 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 an element is placed into multiple SimSets, the search could define multiple change values for it, which may prevent the search from converging if the changes are conflicting.

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.

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.

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.

B. 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. 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, $0\rangle$, $\langle S_2$, font-size , $-1\rangle$, $\langle S_2$, width, $10\rangle$, $\langle S_2$, height, $0\rangle$). 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 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.

C. 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.

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. The approach also modifies the width and the height of any ancestor element that has a fixed width or height that prevents the children elements from expanding freely. An example of such an ancestor element is shown in Figure 3. 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.

To modify the elements that need to be changed in the PUT, our approach uses the following algorithm. Our approach

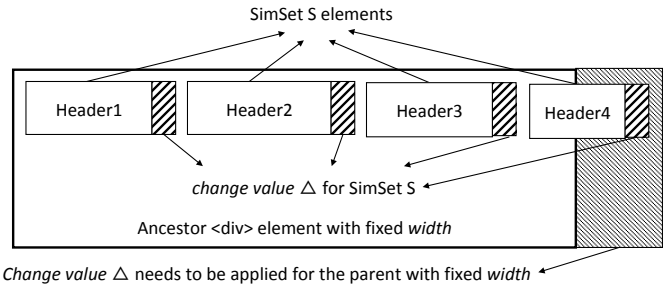


Fig. 3: Example of ancestor elements with fixed width that need to be adjusted together with SimSet elements

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: $e.p = e.p + \Delta$. Then 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 $a.width = a.width + (max(e_{x2}) - a_{x2})$. 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}.

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 minimal amount of changes to the page.

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 with 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 to scale the amount of change to a fraction between 0 and 1 and adding it to the

amount of layout inconsistency value. Using this, the overall fitness function is equal to *amount of layout inconsistency* + $sigmoid$(*amount of change*). We now describe the components of the fitness function in more detail.

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 page. To compute this amount, our approach calculates the percentage of change that is applied to each CSS property for every modified element in the page. The total amount of change is the summation of the squared percentages of changes. The intuition behind squaring the percentages of change is to penalize solutions more heavily if they represent a large change.

D. Search

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.

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:

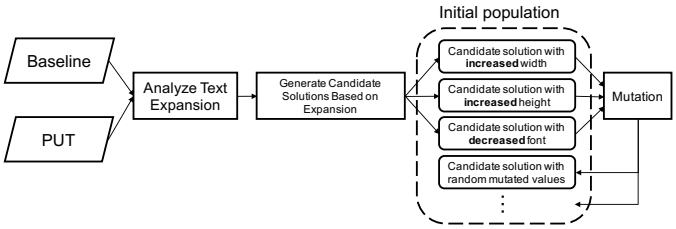


Fig. 4: Initializing the population

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 4 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 three candidate solutions 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 percentage. The rest of the candidate solutions in the initial population (i.e., fourth 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.

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. To do this, our approach uses the Alternating Variable Method (AVM) local search algorithm [15], [16]. 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 newly generated candidate solution is added 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.

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. The best candidate solutions are identified based on the fitness function described in Section III-C2. The selected candidate solutions are used as the population for the next iteration of the search.

Termination: The algorithm terminates when either of two conditions are satisfied. 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 cases, the search most likely converged to the best candidate solution it could find, and further iterations will not introduce more improvement.

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.

IV. Evaluation

To assess the effectiveness and performance of our approach, we conducted an empirical evaluation on 23 real-world subject web pages and answered three research questions:

RQ1: How effective is our approach 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?

A. Implementation

We implemented our approach in Java as a prototype tool named ℐFix [2]. 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 [5], which we obtained from its authors. For the search technique described in Section III, we used the following parameter values: population size = 100, mutation rate = 1.0, max number of iterations = 20, and saturation point = 2. For the Gaussian distribution, used by the mutation operator, we used a 50% decrease and increase as the min and max values, and $\sigma = (max - min)/8.0$ 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.

B. Subjects

For the evaluation we used 23 real-world subject web pages as shown in Table I. 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 gathered these subjects from the web pages used in the evaluation of GWALI [5]. The main criteria behind selecting this source was the presence of known IPFs in the study of GWALI and the diversity in size, layouts, and translation languages that the GWALI subjects offered. 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.

C. Experiment One

To answer RQ1 and RQ2, we ran ℐFix on each subject and recorded the set of IPFs before and after each run, as reported by GWALI, and measured the total time taken. To minimize the variance in the results that can be introduced from the non-deterministic aspects of the search, we ran ℐFix on each subject 30 times and used the mean values across the runs in the results. To further assess and understand the effectiveness of the two main features of our work, guided search and style similarity clustering, we conducted more experiment runs with three variations to ℐFix. The first variation replaced the guided search in the approach with a random search to evaluate the benefit of guided search with a fitness function. For every subject, we time bounded the random search by terminating it once the average time required by ℐFix for that subject had been utilized. The second variation removed the clustering component from ℐFix to evaluate the benefit of clustering stylistically similar elements in a page. The third variation combined the first and second variation. Similar to ℐFix, we ran the three variations 30 times on each subject. All of the experiments were run on a 64-bit Ubuntu 14.04 machine

TABLE I: SUBJECT DETAILS AND RESULTS FOR EFFECTIVENESS IN REDUCING IPFS

ID	Name	URL	#HTML	Baseline	Translated	#Before	#After (Average Reduction in %)			
							IFIX	Rand (Variation 1)	NoClust (Variation 2)	Rand-NoClust (Variation 3)
1	akamai	https://www.akamai.com	304	English	Spanish	6	0 (100)	2 (74)	0 (100)	0.20 (97)
2	caLottery	http://www.calottery.com	777	English	Spanish	4	0 (100)	0 (100)	1 (70)	0.73 (81)
3	designSponge	http://www.designsponge.com	1,184	English	Spanish	9	0.07 (99)	3 (63)	0.07 (99)	3 (71)
4	dmv	https://www.dmv.ca.gov	638	English	Spanish	18	0 (100)	4 (78)	2 (85)	9 (41)
5	doctor	https://sfplasticsurgeon.com	689	English	Spanish	21	0 (100)	0 (100)	6 (72)	21 (0)
6	els	https://www.els.edu	483	English	Portuguese	6	0 (100)	0 (100)	0 (100)	0 (100)
7	facebookLogin	https://www.facebook.com	478	English	Bulgarian	16	0 (100)	6 (65)	12 (25)	16 (0)
8	flynas	http://www.flynas.com	1,069	English	Turkish	9	0 (100)	0.07 (99)	0 (100)	0 (100)
9	googleEarth	https://www.google.com/earth	323	Italian	Russian	15	0 (100)	0 (100)	4 (72)	7 (55)
10	googleLogin	https://accounts.google.com	175	English	Greek	6	0 (100)	0 (100)	0 (100)	0 (100)
11	hightail	https://tinyurl.com/y9tpmro7	1,135	English	German	2	0 (100)	0 (100)	0 (100)	0 (100)
12	hotwire	https://www.hotwire.com	583	English	Spanish	30	0 (100)	0.47 (98)	4 (87)	4 (87)
13	ixigo	https://www.ixigo.com/flights	1,384	English	Italian	38	12 (68)	12 (68)	0 (100)	12 (68)
14	linkedin	https://www.linkedin.com	586	English	Spanish	22	0 (100)	0 (100)	12 (46)	19 (13)
15	mplay	http://www.myplay.com	3,223	English	Spanish	76	0.40 (99)	3 (96)	3 (95)	51 (33)
16	museum	https://www.amnh.org	585	English	French	32	0.40 (99)	0 (100)	12 (63)	19 (40)
17	qualitrol	http://www.qualitrolcorp.com	401	English	Russian	19	0 (100)	0 (100)	21 (-9)	22 (-16)
18	rentalCars	http://www.rentalcars.com	1,011	English	German	6	0 (100)	2 (74)	0 (100)	1 (99)
19	skype	https://tinyurl.com/ycuxxhso	495	English	French	3	0 (100)	0 (100)	0 (100)	0 (100)
20	skyScanner	https://www.skyscanner.com	388	French	Malay	4	0 (100)	0 (100)	0 (100)	0 (100)
21	twitterHelp	https://support.twitter.com	327	English	French	5	0 (100)	0 (100)	0 (100)	0.17 (97)
22	westin	https://tinyurl.com/ycq4o8ar	815	English	Spanish	11	1 (91)	1 (91)	1 (91)	1 (91)
23	worldsBest	http://www.theworlds50best.com	581	English	German	24	0 (100)	7 (69)	0 (100)	17 (29)
					Average	16	0.6 (98)	2 (90)	3 (82)	8 (65)

with 32GB memory, Intel Core i7-4790 processor, and screen resolution of 1920 $\times$ 1080. For rendering the subject web pages, we used Mozilla Firefox v46.0.01 with the browser window maximized to the screen size.

For RQ1, we used GWALI to determine the initial number of IPFs in a subject and the number of IPFs remaining after each of the 30 runs. We calculated the reduction in IPFs as a percentage of the before and after values for each subject.

For RQ2, we computed the average total running time of IFIX and variation 2 across 30 runs for each subject. We did not compare the performance of IFIX with its first and third variations since we time bounded their random searches, as described above. We also measured the time required for the two main stages in our approach; clustering stylistically similar elements (Section III-A) and searching for a repair patch (Section III-D).

Presentation of Results: Table I shows the results for RQ1. The initial number of IPFs are shown under the column "#Before". The columns headed "#After" show the average number of IPFs remaining after each of the 30 runs of IFIX for its three variations: "Rand", "NoClust", and "Rand-NoClust". (Since it is an average, the results under "#After" columns may show decimal values.) The average percentage reduction is shown in parenthesis.

Discussion of Results: The results show that IFIX was the most effective in reducing the number of IPFs, with an average 98% reduction, compared to its variations. This shows the effectiveness of our approach in resolving IPFs.

The results also strongly validate our two key insights of using guided search and clustering in the approach. The first key insight was validated as IFIX 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., dmv and facebookLogin) containing multiple IPFs concentrated in the same area that require a careful resolution of the IPFs by balancing the layout constraints without introducing new IPFs. Second, pages (e.g., akamai) that have 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 28% more changes, on average, compared to IFIX. The second key insight of using a style-based clustering technique was validated as IFIX 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.

Out of the 23 subjects, IFIX was able to completely resolve all of the reported IPFs in 18 subjects in each of the 30 runs and in 21 subjects in more than 90% of the runs. We investigated the two subjects, *ixigo* and *westin*, where IFIX was not able to completely resolve all of the reported IPFs. We found that the dominant reason for the *ixigo* subject was false positive IPFs that were reported by GWALI. This occurred because the footer area of the page had significant differences in terms of layout and structure between the baseline and translated page. Therefore CSS changes made by IFIX were not sufficient to resolve the IPFs in the footer area. For the *westin* subject, 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 IFIX from finding a suitable fix.

The total running time of IFIX ranged from 73 seconds to 17 minutes, with an average of just over 4 minutes and a median of 2 minutes. IFIX was also three times faster, on average, than its second variation (no clustering). This

Fig. 5: UI snippets in the same equivalence class (Hotwire)

was primarily because clustering enabled a narrowing of the search space by grouping together potentially faulty elements reported by GWALI that were also stylistically similar. Thereby a single change to the cluster was capable of resolving multiple IPFs. Moreover, the clustering overhead in ℐFIX was negligible, requiring less than a second, on average. Due to space limitations, the detailed timing results are omitted from the paper, but can be found at the project website [2].

D. Experiment Two

For addressing RQ3, we conducted a user study to understand the visual quality of ℐFIX's suggested fixes from a human perspective. The general format of our survey was to present, in random order, an IPF containing a UI snippet from a subject web page before and after repair. The participants were then asked to compare the two UI snippets on a 5-point Likert scale with respect to their appearance similarity to the corresponding UI snippet from the baseline version. 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 5. To select the "after" version of a subject, we used the run with the best fitness score across the 30 runs of ℐFIX in Experiment One.

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 5 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.

To make the survey length manageable for the participants, we divided the 23 subjects over five different surveys, with each containing four or five subjects. The participants of our user study were 37 undergraduate level students. Each participant was assigned to one of the five surveys. The participants were instructed to use a desktop or laptop for answering the survey to be able to view the IPF UI snippets in full resolution.

Presentation of Results: The results for the appearance similarity ratings given by the participants for each of the IPFs in the 23 subjects are shown in Figure 6. On the x-axis, the ID and number of IPFs for a subject are shown. For example, 4a, 4b, and 4c represent the *dmv* subject with three IPFs. The blue colored bars above the x-axis indicate the number of ratings in favor of the *after* (repaired) version. The dark blue color

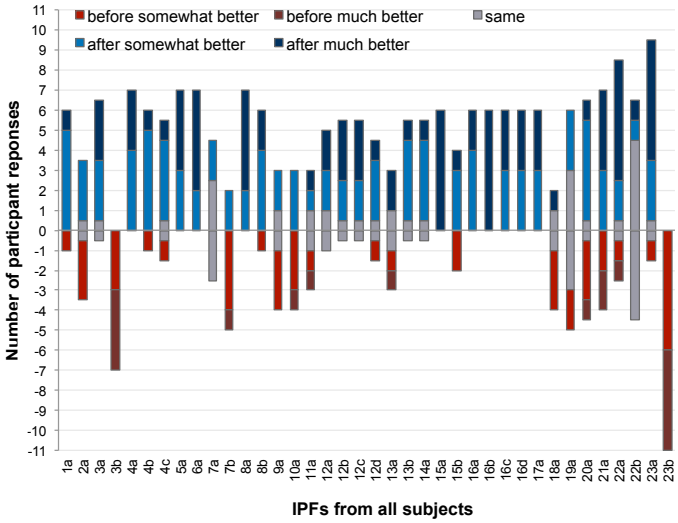


Fig. 6: Similarity ratings given by user study participants

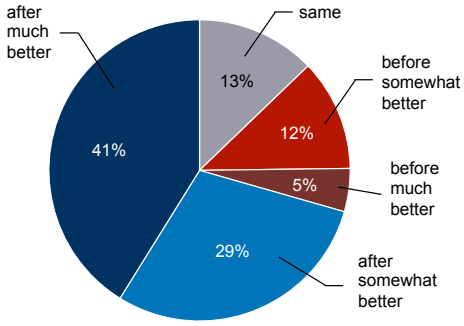


Fig. 7: Weighted distribution of the ratings

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 appearance similarity to the baseline. For example, IPF 23a had a total of 11 responses, six for the after version being much better, three for the after version being somewhat better, one reporting both the versions as the same, and one reporting the before version as being somewhat better. As can be seen from Figure 6, 64% of the participant responses favored the after repair versions, 21% favored the before repair versions, and 15% reported both versions as the same.

Discussion of Results: The results of our user study show that the participants largely rated the after (repaired) pages as better than the before (faulty) versions. This indicates that our approach generates repairs that are high in visual quality. The IPFs presented in the user study, however, do not comprehensively represent all of the IPFs reported for the subjects as the surveys only contained one representative from each equivalence class. Therefore we weighted the survey

responses by multiplying each response from an equivalence class with the size of the class. The results are shown in Figure 7. With the weighting, 70% responses show support for the after version. Also, interestingly, the results show the strength of support for the after version — 41% of responses rate the after version as *much* better, while only 5% responses rate the before version as *much* better.

Two of the IPFs, 3b and 23b, had no participant responses in favor of the after version. We inspected these subjects in more detail and found that the primary reason for this was that $\mathcal{I}$FIX substantially reduced the font-size (e.g., from 13px to 5px for 3b) to resolve the IPFs. 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, 7a, 19a, and 22b, had a majority of the participant responses reporting both versions as the same. $\mathcal{I}$FIX was unable to resolve 22b, implying that the before and after versions *were* practically the same. The issue with 7a and 19a was slightly different. Both 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.

E. Threats to Validity

The first 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 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 5. To further support this, we have made the surveys and subject pages publicly available [2] for verification. A 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.

V. Related Work

Different techniques exist that target detection of internationalization failures in web applications. GWALI [5] and i18n checker [3] are automated techniques, while Apple's pseudo-localization [1] requires manual checking to identify IPFs. There are also techniques [7], [6], [34] that perform automated checks for identifying internationalization problems, such as corrupted text, inconsistent keyboard shortcuts, and incorrect/missing translations. Other techniques, such as X-PERT [10], [9], [35], REDECHECK [40], [39], WebSee [23], [24], [22], [26], [25], [21], and Fighting Layout Bugs [38], detect certain types of presentation failures in web pages. However, none of them are designed to repair IPFs.

Another technique related to internationalization in web pages is TranStrL [41]. 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 [19], [20] and MFix [18] 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 [36] and PhpSync [30] 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 [42]. Cassius [33] 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.

There has been extensive work on the automated repair of software programs [17], [43], [31], [45], [32]. However, these techniques are not capable of repairing presentation failures (e.g., IPFs) in web pages because they are structured to work for general-purpose programming languages (e.g., Java and C).

Lastly, there are several techniques in the field of GUI testing by Memon et. al. [27], [44], [28], [29] 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.

VI. Conclusion

Internationalization Presentation Failures (IPFs) are distortions in the intended appearance of a web page that are caused by the relative expansion or contraction of translated text. In this paper, we introduce an automated approach for repairing IPFs in web pages. Our approach first uses clustering to group stylistically similar elements in a page. It then performs a guided search to find suitable CSS fixes for the identified clusters. In the evaluation, our approach was able to resolve 98% of the reported IPFs. In the user study, 70% of the participant responses rated the fixed versions as better than the unfixed versions. Overall, these are positive results and signify 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] Apple Internationalization and Localization Guide. https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPInternational/TestingYourInternationalApp/TestingYourInternationalApp.html.

[2] IFix Evaluation Data. https://github.com/USC-SQL/ifix.

[3] W3C Internationalization Checker. https://validator.w3.org/i18n-checker/.

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

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

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

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

[8] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. VIPS: a Vision-based Page Segmentation Algorithm. Technical report, November 2003.

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

[10] S. R. Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications. In *Proceedings of the 35th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE)*, May 2013.

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

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

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

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

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

[16] B. Korel. Automated Software Test Data Generation. In *IEEE Transactions on Software Engineering*, volume 16, pages 870–879, 1990.

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

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

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

[20] S. Mahajan, A. Alameer, P. McMinn, and W. G. Halfond. XFix: Automated Tool for Repair of Layout Cross Browser Issues. In *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA) – Tool Track*, July 2017.

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

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

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

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

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

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

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

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

[29] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: An innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105, Mar. 2014.

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

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

[32] R. Nokhbeh Zaeem and S. Khurshid. Contract-based data structure repair using alloy. In T. D'Hondt, editor, *ECOOP – Object-Oriented Programming*, 2010.

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

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

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

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

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

[38] M. Tamm. Fighting Layout Bugs. https://code.google.com/p/fighting-layout-bugs/.

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

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

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

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

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

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

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