Efficiently Repairing Internationalization Presentation Failures By Solving Layout Constraints

Abdulmajeed Alameer, Paul T. Chiou and William G. J. Halfond
University of Southern California, USA
{alameer, paulchio, halfond}@usc.edu

Abstract—Web developers employ internationalization frameworks to automate web page translations and enable their web apps to more easily communicate with a global audience. However, the change of text size in different languages can lead to distortions in the translated web page's layout. These distortions are known as Internationalization Presentation Failures (IPFs). Debugging these IPFs can be a tedious and error-prone process. Previous research efforts to develop an automatic IPF repair technique could compromise the attractiveness and readability of the repaired web page. In this paper, we present a novel approach that can rapidly repair IPFs and maintain the readability and the attractiveness of the web page. Our approach models the correct layout of a web page as a system of constraints. The solution to the system represents the new and correct layout of the web page that resolves its IPFs. In the evaluation, we found that our approach could more quickly produce repairs that were rated as more attractive and more readable than those produced by a prior state-of-the-art technique.

I. INTRODUCTION

The web allows companies to easily offer their services and products to global customers. Web developers typically leverage internationalization (i18n) frameworks for their websites to effectively communicate with international audiences. The use of i18n frameworks allows web developers to easily maintain different versions of a website with localized content and translated text. However, the use of these frameworks can also come with unanticipated problems. The change of text size in different languages can cause elements to expand, contract, or move in order to handle the translated text, which ultimately can cause distortions to the layout of the translated web page. We refer to such distortions as Internationalization Presentation Failures (IPFs). These failures are very common in translated web pages, a recent study showed that they occurred in 77% of translated web pages [1]. Resolving IPFs in a web page is important, as website visitors form opinions about the site’s trustworthiness and quality based on the appearance of its web pages [2], [3], [4].

Repairing IPFs in a web page is challenging for web developers. Modern web pages may contain thousands of HTML elements, and each element may contain several CSS properties that control its appearance. In order to repair an IPF, developers have to find which HTML elements and CSS properties need to be adjusted, then try multiple values for the CSS properties until the IPF is resolved. This process is further complicated by the complex interaction among the elements of a web page. Modifying one element to resolve an IPF can also distort the layout of other parts of the web page, such as causing other elements to become misaligned, wrapped, or overlapped with other elements. This means that any attempted repair must be carefully evaluated to ensure that it does not introduce new problems into the repaired web page. All of these considerations make resolving IPFs a time consuming and error prone process, especially for large websites that support many different languages.

Existing techniques for repairing web pages have limitations that either reduce their usefulness or make them unsuitable for resolving IPFs. Related work that targets IPFs, TFix [5], uses a search-based technique to repair IPFs. Although the TFix technique could resolve a large number of IPFs, it could also negatively impact the readability of the page by reducing the size of translated text or reduce the page’s attractiveness by allowing the repaired page to deviate a lot from the intended layout of the original untranslated page. TFix could also be slow, requiring up to 19 minutes in some cases to find a repair for a single web page. Another technique, TranStrL [6], [7], aids developers in internationalizing an existing application by isolating hard-coded strings into external resource files. However, TranStrL cannot guarantee that the layout of the web page will adapt to the international text. Other web application repair techniques target conceptually different types of presentation problems, such as those caused by cross-browser issues [8], [9], mobile-friendly problems [10], and malformed HTML [11], [12]. Finally, techniques, such as GWALI [13], can only identify if an IPF is present in a web page, but cannot generate repairs for the identified IPFs.

To address the limitations of the existing technique, we designed a new approach for repairing IPFs in translated web pages. Our approach is designed to produce repairs faster than prior techniques and repairs that result in more attractive and readable repaired web pages. To do this, we developed techniques to model the layout of a web page as a system of constraints and leveraged the ability of state-of-the-art constraint solvers to quickly identify solutions that could be used as web page repairs. Using a constraint-based technique also allowed our approach to define constraints that could better preserve the aesthetics of the original web pages. In a side-by-side comparison of pages repaired with our approach and TFix, users rated our approach’s repaired pages as more attractive and readable. Our approach was also significantly faster, reducing the time that is required for the repair by 96%. Overall, these results were positive and indicate that our
approach can help web developers to produce more attractive and readable fixes for many web pages.

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

An internationalized web application can be built using one of two approaches. The first approach is to isolate the translated text and media into separate language-specific resource files. When the web page is requested by the web browser, the browser provides the user’s preferred language and a server-side internationalization framework loads the correct language-specific resource files and inserts the translated text and media into placeholders in the web page. This isolation of the language-specific content allows for easier management of the internationalized website. The second approach is to use online automated translation web services (e.g., Google Website Translator [14] and Bing Translator Widget [15]). In this approach, the website developers install an automated translation plugin in their website. The visitors to the site can select their desired language from a drop-down box and the plugin will extract all the text in the web page. Then, the plugin will send that text to an online automated translation service, which will reply with the translation of the text to the desired language. The plugin then replaces the original text in the web page with the translated text. This approach allows the developers to easily make their web applications available in a large number of languages without the need to manually extract and translate the content.

The two internalization approaches cannot guarantee that the layout of the web page can accommodate the size of the translated text. The change in the size of the text can cause it to overflow the HTML element in which it is contained, make the text become cut-off, or make it overlap with surrounding elements. Also, the element in which the text in the text is contained may expand to fit the text, which can, in turn, make it push other surrounding elements from their position, leading to disruption in the layout of the web page. An example of such IPFs is shown in Figure 1. When the text was translated from English to German, the expansion of the text caused elements to overlap with each other, affecting the readability and the usability of the web page.

Debugging IPFs can be a time consuming and error prone process. Developers need to check every page of a website in each language that it supports. This process is further complicated for large websites that support many different languages. In order to ensure a website is free of IPFs, a developer must (1) detect when they occur in a page, (2) localize the faulty HTML elements that are causing the IPFs to appear, and (3) repair the web page to ensure that the failure no longer occurs.

An existing technique, GWALI [13], can address the first and second part of this debugging process (i.e., detection and localization). The inputs to GWALI are two pages, a Page Under Test (PUT), and a baseline (untranslated) page that shows the correct layout. To detect IPFs, GWALI builds a model called a Layout Graph (LG). A difference in the LGs of the baseline and the PUT reveals that the relative positions between elements in the baseline web page have changed after the translation in the PUT, indicating a potential IPF.

Repairing IPFs in a web page is not a trivial process. Fixing one IPF in a web page can often introduce a new IPF. Consider the example in Figure 2. After translation, “first name” placeholder expands into “nome di battesimo”, which results in it being cut-off. To resolve this IPF, the width of “nome di battesimo” input field can be increased. However, this pushes the “il casato” input field to wrap to a new line resulting in a new IPF (Figure 2c). To repair this new IPF, the width of the outer DIV container’s width can be increased, but this makes “indirizzo email” no longer right-aligned with “il casato” input field (Figure 2d). This process demonstrates how repairing IPFs can be a tedious process, as repairing an IPF can introduce another one.

There exists an approach that targets the repairing process, ZFix [5] uses a search-based technique to find repairs for IPFs that are present in a web page. However, ZFix typically requires a long time to produce repairs. The technique uses a guided search algorithm to find the CSS values for the HTML elements that need to be adjusted. The algorithm tries different CSS values for each element and renders the web page in a browser. It then runs GWALI to determine if the number of IPFs has reduced. ZFix repeats this process hundreds of times until a repair is found. This process results in a lengthy time that is required to find a single repair. Another issue with ZFix is that it focuses only on the elements that are reported by the detection tool in order to reduce the search space, which often results in repairs that decrease the font size of these elements, negatively affecting the web page’s readability. When repairing an IPF, adjusting the size of elements other than the ones that are reported by detection tool often times could lead to a more attractive repair. This insight motivates our approach to account for all the possible elements that can be adjusted.
The goal of our approach is to produce a repair that will resolve IPFs in a translated web page in a way that is faster and results in more attractive and readable web pages than I2Fix. As with I2Fix, our approach’s basic premise is that IPFs in a page can be resolved by changing the values of elements’ layout CSS properties so that the elements’ size after their texts’ translation can be accommodated. However, to make our approach faster, we define the repair problem as a constraint system whose solution represents new and correct values for the relevant CSS properties. To make the pages more attractive and readable, we introduce constraints so that the solution will more closely conform to the design and layout of the original, correct web page.

Figure 3 shows an overview of our approach. The inputs to our approach are: (1) a baseline web page, which is typically the original version of the page that is already known to be correct; (2) a PUT, which is an internationalized version of the baseline that exhibits one or more IPFs; and (3) a set of IPFs that have been detected using an IPF detection tool (e.g., GWALI [13]). An IPF is represented as a tuple of the form \((e_1, e_2, r)\) where \(r\) is the type of the IPF that affects the two elements \(e_1\) and \(e_2\) (e.g., \(e_1\) and \(e_2\) are not aligned). In the first step, our approach analyzes the IPFs and the baseline to approximate a superset of the layout aspects that can be modified in order to resolve the IPFs. Then, in the second step, our approach converts these aspects of the layout into a system of constraints that can be solved and applied in the third step. Our approach repeats these three steps until either all IPFs have been resolved or no further improvements are made to the PUT as a result of the previous iteration.

A. Step 1: Extracting visual relationships between elements

The goal of the first step is to identify the aspects of the PUT that must be modeled by constraints in order to resolve the detected IPFs. The aspects to be modeled include both the HTML elements whose CSS properties may need to be adjusted and the visual relationships (e.g., Right-Aligned) among those elements that must be either changed or not allowed to change to restore the correct layout.

A naive solution to this problem would be to identify all visual relationships between \(e_1\) and \(e_2\) as the aspects to be modeled. However, for all but the simplest of IPFs, this set is inadequate to fully address our goals of attractiveness and readability. To illustrate this, consider the case where the size of one element in a horizontal menu bar has grown and caused the menu bar to wrap to a second line. A simple solution that focuses only on the expanded menu item would be limited to changing the size of its text or its margins and padding. This could cause the item to look different than the rest of the menu items, hence, reducing the design’s attractiveness and consistency. Instead, a better solution might be to increase the size of the containing parent element to allow more space for the expanded menu item and adjust surrounding elements to maintain any visual relationships with the menu bar, such as alignment relationships. The analysis in the first step of the approach identifies all such additional elements and their visual relationships.

To identify the set of HTML elements, \(E\), whose CSS properties need to be adjusted, our approach analyzes the baseline web page and the reported IPFs. For each reported IPF, the approach employs one of two strategies based on the IPF’s type, \(r\). The first strategy is used if (1) \(r\) is a Directional issue — when one element is placed in a specific direction to another element in the baseline (e.g., West) but it is placed in a different direction in the PUT (e.g., North) or (2) if \(r\) is a Containment issue — when one element contains (i.e., bounds) another element in the baseline, but not in the PUT. To identify elements that need to be modeled to address Directional and Containment issues, our approach finds the element \(l\) that is the Lowest Common Ancestor (LCA) of \(e_1\) and \(e_2\) in the Document Object Model (DOM) tree of the baseline. Then our approach adds \(l\) and all of its successors to \(E\). The reason for this selection of elements is that HTML elements are typically arranged using relative positioning, and modifying sibling or parent elements can influence the position of the other elements. Adding \(l\) and all of its successors to \(E\) is a conservative heuristic to ensure that all parent and the sibling elements are modeled so that they could be modified to resolve the IPFs. The second strategy is employed if \(r\) is an Alignment issue — when the two elements were aligned with each other.
Fig. 3: An overview of our approach

along some dimension in the baseline, but not in the PUT. This is the simple situation discussed earlier and our approach only needs to add the two elements $e_1$ and $e_2$ that became unaligned to $E$. The mapping from the set of IPF types, $R$, to each of these three issue types (Directional, Containment, and Alignment) will vary based on the information provided by the IPF detector. Creating this mapping is a one-time effort that must be expended for every unique detector used to provide the input list of IPFs.

Once the elements in $E$ have been identified, the approach identifies $V : E \times E \rightarrow \mathcal{P}(S)$, where $S$ is the set of visual relationships that our approach captures between elements. For each pair of elements, $V$ represents the visual relationships among those elements that must be either changed or not allowed to change to repair the PUT.

The visual relationships between elements in the baseline web page represent the correct layout that is needed in the PUT. To identify these relationships, our approach renders the baseline in a web browser and uses the browser’s API to traverse the DOM tree of the rendered baseline web page. For each HTML element $h$ in the DOM, our approach collects $h$’s position and size. If $h$ contains a text element $t$ inside it (e.g., text between $<p>$ tags or the placeholder text of an $<input>$ element), then our approach also collects $t$’s position and size. After collecting the positions and sizes of all the elements in the baseline, our approach computes the visual relationships between the elements in $E$. For each pair of elements $e_1, e_2 \in E$, our approach computes the visual relationship between them by comparing their coordinates. For example, if $e_1-bottom \leq e_2-top$ then the relationship set would include Top-Contain. Similarly, if $e_1-left = e_2-left$ then the set would include Left-Aligned. The other relationships are computed in an analogous manner.

Table I shows the set of visual relationships $S$ that our approach captures between the elements in $E$. The first column shows a complete list of the visual relationships our approach computes between the elements. A pair of elements $(e_1, e_2)$ could have one or more relationships between them. These relationships have three types. (1) Directional: left-right, which means $e_1$ is on the left side of $e_2$, or top-bottom, which means $e_1$ is on top of $e_2$. (2) Alignment: right-aligned, left-aligned, top-aligned, and bottom-aligned. They determine if $e_1$ and $e_2$ are aligned with each other along one (or more) sides. (3) Containment: Horizontal-Contains, if the left and right sides of $e_1$ bound the left and right sides of $e_2$. Contains, if the left,

right, top, and bottom sides of $e_1$ bound those of $e_2$. Same, if both $e_1$ and $e_2$ have the same position and size.

B. Step 2: Converting visual relationships to constraints

The goal of this step is to translate the visual relationships among the elements in $E$ into a system of constraints. The variables in this system will represent the sizes and positions of the elements in $E$, and the constraints among these variables will represent their correct intended layout, which can be obtained by analyzing the baseline web page. This system of constraints will then be solved in the third step of our approach (Section III-C) to generate a repair for the IPFs.

Our approach defines the system of constraints for the PUT layout as a Linear Program (LP). This representation is a good fit for our approach since the variables can represent the numeric values of layout related CSS properties and all of the visual relationships shown in Table I can be represented using linear constraints. Additionally, the use of LP allows our approach to specify an objective function so it can pick a preferred solution if multiple solutions satisfy the constraints.

The following is a detailed description of how the variables and constraints of the LP are defined in our approach.

1) Variables: Our approach defines the variables in the LP to be the CSS properties that can be set for each element in $E$ to change the way it is sized or positioned in a page. Properties that can be set for this purpose and their relationship
TABLE I: Constraints that need to be enforced between the variables of two elements $e_1$ and $e_2$ in the PUT based on the visual relationships between these elements in the baseline web page.

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Constraint(s) to enforce it in the PUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-Right</td>
<td>$e_1$-right + $e_1$-margin-right $\leq$ $e_2$-left - $e_2$-margin-left</td>
</tr>
<tr>
<td>Top-Bottom</td>
<td>$e_1$-bottom + $e_1$-margin-bottom $\leq$ $e_2$-top - $e_2$-margin-top</td>
</tr>
<tr>
<td>Left-Aligned</td>
<td>$e_1$-left $= e_2$-left</td>
</tr>
<tr>
<td>Top-Aligned</td>
<td>$e_1$-top $= e_2$-top</td>
</tr>
<tr>
<td>Right-Aligned</td>
<td>$e_1$-right $= e_2$-right</td>
</tr>
<tr>
<td>Bottom-Aligned</td>
<td>$e_1$-bottom $= e_2$-bottom</td>
</tr>
<tr>
<td>Contains</td>
<td>$e_1$-left + $e_1$-border-left + $e_1$-padding-left $\leq$ $e_2$-left - $e_2$-left-margin</td>
</tr>
<tr>
<td></td>
<td>$e_1$-top + $e_1$-border-top + $e_1$-padding-top $\leq$ $e_2$-top - $e_2$-top-margin</td>
</tr>
<tr>
<td></td>
<td>$e_1$-right - $e_1$-border-right - $e_1$-padding-right $\geq$ $e_2$-right + $e_2$-right-margin</td>
</tr>
<tr>
<td></td>
<td>$e_1$-bottom - $e_1$-border-bottom - $e_1$-padding-bottom $\geq$ $e_2$-bottom + $e_2$-padding-bottom</td>
</tr>
<tr>
<td>Horizontal-Contains</td>
<td>$e_1$-left + $e_1$-border-left + $e_1$-padding-left $\leq$ $e_2$-left - $e_2$-left-margin</td>
</tr>
<tr>
<td></td>
<td>$e_1$-right - $e_1$-border-right - $e_1$-padding-right $\geq$ $e_2$-right + $e_2$-right-margin</td>
</tr>
<tr>
<td>Same</td>
<td>$e_1$-left + $e_1$-border-left + $e_1$-padding-left $= e_2$-left + $e_2$-border-left + $e_2$-padding-left</td>
</tr>
<tr>
<td></td>
<td>$e_1$-top + $e_1$-border-top + $e_1$-padding-top $= e_2$-top + $e_2$-border-top + $e_2$-padding-top</td>
</tr>
<tr>
<td></td>
<td>$e_1$-right - $e_1$-border-right - $e_1$-padding-right $= e_2$-right - $e_2$-border-right - $e_2$-padding-right</td>
</tr>
<tr>
<td></td>
<td>$e_1$-bottom - $e_1$-border-bottom - $e_1$-padding-bottom $= e_2$-bottom - $e_2$-border-bottom - $e_2$-padding-bottom</td>
</tr>
</tbody>
</table>

to each other are formally defined by the W3C’s CSS Box Model [16]. Figure 4 shows the different properties and relationships defined by the CSS Box Model. The box model consists of four areas: (1) The content area, which represents the area where the actual content of the element appears. (2) The padding area, which is an empty transparent area around the content. (3) The border area, which surrounds the padding and the content areas. (4) The margin area, which is an empty transparent area outside the border. Each one of these four areas can be either defined for all four sides of the box or individual sides can be defined using the modifiers “top,” “bottom,” “left,” or “right.” In total, this represents 16 variables that can be defined for each HTML tag element — four variables representing the positions of the four sides of the HTML element, and 12 variables representing the size of each side of the padding, border, and margin areas. By HTML definition, text elements are represented only by the positions of the four sides representing the content area of the text and they do not have attributes to represent their margin, padding, or border.

2) Constraints: Our approach uses the constraints in the LP to describe the correct visual relationships among the elements. The constraints are expressed as linear constraints comprised of the variables defined for each element. Our approach generates two types of constraints:

The first type of constraint is derived directly from the visual relationships, $V$, identified in the previous step. These constraints restrict the variables in the LP to have values that place the elements in the PUT in a layout that resembles the baseline web page’s layout. For each extracted relationship $v \in V$ between two elements $e_1, e_2 \in E$, our approach adds constraints for the variables in the LP to enforce the same visual relationships on the PUT. The approach generates different types of constraints for the different types of visual relationships. Table I shows the mapping of visual relationship type (first column) to the specific generated constraints (second column).

To illustrate, consider the example in Figure 2. For the left-right visual relationship between firstname input ($e_1$) and lastname input ($e_2$) the following constraint is added to the LP:

$e_1$-right + $e_1$-right-margin $\leq$ $e_2$-left - $e_2$-left-margin

For the right-aligned relationship between lastname input ($e_2$) and email input ($e_3$) the following constraint is added to the LP:

$e_2$-right $= e_3$-right

The second type of constraint preserves the size of text in the repaired PUT. We add this type of constraint to the system to prevent the approach from reducing the text size in the new layout, which could negatively impact readability of the repaired PUT. Our approach does this indirectly by constraining the variables that represent the positions of the four sides of the content area of each text element to values that maintain the size of the text. For each text element $t$ that has a width $w$ and height $h$ in the PUT, the approach adds a constraint to the LP $t$-right $- t$-left $= w$ and another constraint $t$-bottom $- t$-top $= h$.

For our running example, the following constraints will be added to the LP for the variables representing “nome di battesimo” placeholder, $e_p$ : $e_p$-right $- e_p$-left $= \text{width of } e_p$ (constant), and $e_p$-bottom $- e_p$-top $= \text{height of } e_p$ (constant). Where $e_p$-left, $e_p$-right, $e_p$-top, and $e_p$-bottom are the variables representing the four sides of the placeholder text box.

C. Step 3: Solving constraints and producing a repair

The third and last step of our approach solves the constraint system and generates the repaired PUT. The solution to the
Our approach penalizes undesirable than changing the variables that control the content paddings, and border variables to make changing them less the variable. Our approach sets these variables obtained from the unfixed PUT. More formally, our objective function is:

\[
\text{minimize } \sum_{i=1}^{m} c_i |\text{var}_i - \text{org}_i| 
\]

Where \(\text{var}_1, \text{var}_2, \ldots, \text{var}_m\) are all the variables defined in the LP and \(\text{org}_1, \text{org}_2, \ldots, \text{org}_m\) are the original values for these variables obtained from the unfixed PUT. \(c_1, c_2, \ldots, c_m\) are constant coefficients that are set depending on the type of the variable. Our approach sets \(c_i\) to high values for margins, paddings, and border variables to make changing them less desirable than changing the variables that control the content size or the positions of the elements. Our approach penalizes margins, paddings, and border changes more because they typically have small values, and a minor change in their values has a disproportionately large impact on a page’s layout.

The objective function defined above is non-linear, which requires us to linearize it to use it in the LP. A well-known linearization technique for such an objective function is described in linear programming literature [17]. The basic idea of this linearization technique is that if we have an LP in this form:

\[
\text{minimize } c_1|\text{var}_1 - \text{org}_1| + c_2|\text{var}_2 - \text{org}_2| \\
\text{such that: } \text{var}_1 > \text{var}_2
\]

It is equivalent to:

\[
\text{minimize } c_1t_1 + c_2t_2 \\
\text{such that: } \text{var}_1 - \text{org}_1 \leq t_1 \\
\text{var}_1 - \text{org}_1 \geq -t_1 \\
\text{var}_2 - \text{org}_2 \leq t_2 \\
\text{var}_2 - \text{org}_2 \geq -t_2 \\
\text{var}_1 > \text{var}_2
\]

This linearization technique reformulates our objective function by introducing new variables, \(t_i\), to replace each absolute value term \(|\text{var}_i - \text{org}_i|\), and adding into the LP two new constraints \(t_i \geq \text{var}_i - \text{org}\), and \(-t_i \leq \text{var}_i - \text{org}\) for each newly introduced variable \(t_i\).

After solving the LP, our approach modifies the PUT by changing the content size, padding, margin, and border of the elements in the PUT using the values produced by the solver. The new width and new height of each element \(e\) in the PUT is computed based on the value of the element’s CSS property box-sizing. This CSS property determines whether the padding and border are included when computing the width and height of the element. Figure 4 shows the difference between the two box-sizing options. If the value of box-sizing is set to “border-box” the new width is \(e\)-right \(- e\)-left, and the new height is \(e\)-bottom \(- e\)-top. If the value of box-sizing is set to “content-box” the new width is \((e\)-right \(- e\)-border-right \(- e\)-padding-right \(- (e\)-left \(+ e\)-border-left \(+ e\)-padding-left\)), and the new height is \((e\)-bottom \(- e\)-border-bottom \(- e\)-padding-bottom\) \(- (e\)-top \(+ e\)-border-top \(+ e\)-padding-top\). In both cases, if the newly computed width value exceeds the value of the max-width CSS property, the value of max-width will be adjusted to the new width. A similar process will be applied for the min-width, max-height, and min-height CSS properties. The padding, border, and margin of each element \(e\) in the PUT is updated with the new values produced by the solver for the variables related to padding, border, and margin that are defined for the element \(e\) in the LP.

Our approach applies the new values for the CSS properties using a language-specific CSS patch for the web page. The patch is embedded in a CSS :lang() selector. This selector allows our approach to specify new alternative values for CSS properties based on the language in which the page is viewed.
IV. Evaluation

We conducted an empirical evaluation that focused on the following research questions:

**RQ1:** How effective is our approach in repairing IPFs?

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

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

### A. Implementation

Our approach is implemented as a Java prototype tool, CBRepair (Constraints-Based Repair), which leveraged several third-party libraries. Selenium Webdriver was used to automatically load the web pages and extract the rendered DOM information. We used the Java interface of Google Optimization Tool’s (OR-Tools) linear programming solver to solve the constraints as discussed in Section III-C [18]. Javascript code was executed using Selenium Webdriver to retrieve the Box Model and XPath locator for each element as well as apply the repair patch to the PUT. To maintain consistency, all web pages were rendered with Firefox version 46.0.1 maximized to a fixed screen resolution of 1920 x 1080 pixels. All experiments were conducted on a single Intel Core i7-4790 64-bit machine with 32GB memory, running Ubuntu Linux 16.04 LTS.

### B. Subjects

We conducted our evaluation on a set of 23 real-world subject web pages, which are shown in Table II. The subjects included web pages containing one IPF and web pages with multiple IPFs occurring together. These subjects are the same subjects that were used in the evaluation of TFix [5] and GWALI [13]. The authors of GWALI chose these particular subjects to cover a wide range of translation technologies and frameworks with diversity in size, layouts, and styles. The original sources of these subjects are: (1) builtinwith.com, a website that indexes web applications built using various technologies, (2) Alexa top 100, and (3) popular, high-profile websites that targeted international audiences with multiple languages. We used the same set of subjects to facilitate a more direct comparison between the two approaches. The complete set of subjects’ files can be found on the project website [19].

### C. Experiment One

To answer RQ1 and RQ2, we evaluated and compared the repairs performed by CBRepair and TFix. For RQ1, we measured the number of IPFs detected by GWALI before and after the repair. The experiment was carried out in three steps: (1) GWALI was run on each subject web page’s translated version (i.e. the PUT) to determine the number of IPFs present before the repairs. (2) CBRepair and TFix were independently run on each subject web page to produce a repaired version (PUT’). (3) GWALI was run on each subject web page’s PUT’ to determine the number of unresolved IPFs remaining in each page. Since TFix uses a non-deterministic search-based technique, we ran TFix on each subject 30 times, averaging the number of resulting IPFs.

For RQ2, we evaluated our approach by measuring and comparing the running times for CBRepair and TFix to generate the repairs. To mitigate the non-determinism of TFix, each subject’s TFix running time was also computed as an average across the 30 runs.

1) **Presentation of Results:** Table II shows the results of Experiment 1. The initial number of IPFs for each subject are shown under the column “#Before”. Under RQ1, the columns “#CBR” and “#IFix” correspond to the number of IPFs remaining after applying CBRepair and TFix respectively (Note that values for TFix are averages of 30 runs and may contain decimal values). Under RQ2, column “CBR” shows the running time of each subject (in seconds) for CBRepair. Column “IFix” shows the average running time (in seconds) of each subject across the 30 runs of TFix. The column “Δ” shows the percentage reduction when comparing CBRepair’s running time and TFix’s average running time. The running time of CBRepair averaged 13 seconds with a median of 4.7 seconds, and the running time of TFix averaged 315 seconds with a median of 163 seconds.

2) **Discussion of Results:** Overall, the results of experiment one show that CBRepair was able to significantly reduce the number of IPFs in the subject applications. For 19 of the 23 subjects, CBRepair was able to decrease the number of IPFs detected in the after version of the page, with an overall average reduction across all subjects of 54%. Of those 19 improved subjects, eight were completely repaired, with GWALI reporting zero remaining IPFs. After visually inspecting the other 11 subjects, we determined that five of them were completely repaired by CBRepair, but GWALI is erroneously reporting them as still containing IPFs (i.e., GWALI reported five false positives). We marked them with “FP” in Table II. This brought the average IPF reduction of CBRepair up to 65%.

<table>
<thead>
<tr>
<th>Subject</th>
<th>RQ1</th>
<th>RQ2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Before</td>
<td>#IFix</td>
</tr>
<tr>
<td>akamai</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>cal.Lottery</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>designSponge</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>dmz</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>doctor</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>els</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>facebookLogin</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>flynas</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>googleEarth</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>googleLogin</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>hightral</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>hotwire</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>itxgo</td>
<td>38</td>
<td>12</td>
</tr>
<tr>
<td>linkedin</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>mplay</td>
<td>76</td>
<td>0.53</td>
</tr>
<tr>
<td>museum</td>
<td>32</td>
<td>0.13</td>
</tr>
<tr>
<td>qualitol</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>rentalCars</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>skype</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>skyScanner</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>twitterHelp</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>westin</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>worldsBest</td>
<td>24</td>
<td>0</td>
</tr>
</tbody>
</table>
We investigated the IPFs that our approach could not repair and found two scenarios where this occurred. The first, which occurred in one subject, was that some constraint systems were infeasible and a satisfying solution could not be found by our solver. We confirmed this by manually inspecting the subject and found it impossible to repair the IPFs without reducing the font size. The second reason, for the remaining non-repaired subjects, was that these web pages used CSS properties that could not be accurately modeled by a linear constraint system. An example of this is the “float” CSS property. We believe that these types of properties could be modeled with more expressive constraint systems and intend to explore this in future work.

Despite the successful reduction of IPFs by our approach, \(\mathcal{IFix}\) was able to achieve a higher reduction. Overall, \(\mathcal{IFix}\) had an average IPF reduction of 98% and could completely resolve IPFs in 18 of the 23 subjects. We analyzed the results to better understand the reasons why \(\mathcal{IFix}\) was able to outperform our approach in RQ1. Our investigation found that our approach’s restriction on modifying the font size was a likely cause for this disparity — \(\mathcal{IFix}\) reduced the font size in 21 of 23 subjects’ repairs. To evaluate this possibility, we modified \(\mathcal{IFix}\) to prevent it from allowing repairs that reduced font size (we refer to this version as \(\mathcal{IFix}'\)). We found that \(\mathcal{IFix}'\) was able to resolve only four of the 23 subjects, with a 29% average reduction of IPFs. This indicates that \(\mathcal{IFix}\) has a limited ability in repairing IPFs without modifying the font size. As we show in RQ3, although these repairs allowed \(\mathcal{IFix}\) to perform well in RQ1, the user-perceived quality of the \(\mathcal{IFix}\) repairs was lower than those generated by our approach — a result that could reflect the impact of the font size reductions.

For RQ2, results indicate that \(\mathcal{CBRepair}\)’s analysis was significantly faster than \(\mathcal{IFix}\) in generating repairs. Breaking down the average time of our approach to the granularity of individual steps, 4.76 seconds (37%) was required to extract relationships (Section III-A), 0.11 seconds (1%) to extract constraints (Section III-B), and 7.86 seconds (62%) to solve and repair (Section III-C). \(\mathcal{CBRepair}\) had an average repair time of less than 13 seconds versus an average repair time of over 5 minutes for \(\mathcal{IFix}\), an average time reduction of 96%.

### D. Experiment Two

To answer RQ3, we conducted user surveys to measure the visual quality of the generated repairs from a human perspective. In the surveys, we asked users to compare a series of two side-by-side user interface (UI) snippets against snippets from the baseline page. A UI snippet is an image of the area where an IPF occurred and was obtained by cropping the subject web pages’ screenshot. There were two variants of the side-by-side snippets. The first compared snippets from the PUT against the version repaired by \(\mathcal{CBRepair}\), and the second compared snippets from the version repaired by \(\mathcal{CBRepair}\) against snippets from the version repaired by \(\mathcal{IFix}\). Within each variant, the order in which the snippets were displayed was randomized and only labeled Version 1 and Version 2. We only compared the snippets for the subjects where \(\mathcal{CBRepair}\) was able to reduce the number of IPFs, resulting in a total of 24 evaluated IPFs. Participants were asked to rate the two UI snippets based on three metrics: (1) attractiveness, (2) readability, and (3) similarity to the baseline page. The ratings were based on a numeric scale from 1 to 10, where 1 represents the least-attractive/least-readable/least-similar and 10 represents the most-attractive/most-readable/most-similar.

To conduct the surveys about the repairs, we used the Amazon Mechanical Turk (AMT) service. The participants of our study were anonymous to us; however, to ensure the quality of the responses, we only allowed participants who had a minimum 95% approval rating on at least 100 previously completed tasks. We followed AMT best practices by employing a captcha and a check-question to reduce the likelihood that the survey was completed by a bot. Due to failing these checks, 35% of the completed surveys were removed from the analysis. For each survey, we had 20 unique workers participate, each was paid $0.10 for completing the survey.

1) Presentation of Results: The results for the appearance, readability, and similarity ratings provided by the participants are shown in Figure 6. The box plot in Figure 6a shows the results when snippets of \(\mathcal{CBRepair}\) were compared with those of the PUT. The box plot in Figure 6b shows the results when snippets from \(\mathcal{CBRepair}\) were compared against those from \(\mathcal{IFix}\). The boxes in the figures represent the distribution of the numeric ratings given by the participants with the average rating marked as an ‘x’ and the median marked by a bar inside the box.

As can be seen from Figure 6a, when compared to the PUT, the average attractiveness score increased from 6.3 to 7.1 (12%), the average readability score increased from 6.6 to 7.0 (7%), and the average baseline-similarity score increased from 6.3 to 7.4 (18%). These results were statistically significant using the Wilcoxon signed-rank test with p-values of 3.948e-05, 0.01654, and 3.68e-08, respectively. The Wilcoxon signed-rank test was used for this analysis because we were comparing paired ratings from two different groups and the measured ratings were not normally distributed.

When compared with \(\mathcal{IFix}\), Figure 6b shows that the average attractiveness score increased from 6.3 to 6.8 (8%), the average readability score increased from 5.9 to 7.0 (19%), and the average baseline-similarity score increased from 6.8 to 7.0 (2%). Both attractiveness and readability were statistically significant with p-values of 0.006957 and 1.358e-07, but baseline-similarity was not.

2) Discussion of Results: Overall the results of experiment two show that users perceived the quality of the \(\mathcal{CBRepair}\) generated repairs as more attractive and readable than the PUT and repairs generated by \(\mathcal{IFix}\). Below we discuss the results for each of the three metrics in more detail.

For readability, our approach significantly outperformed \(\mathcal{IFix}\), with an average increase in score of 19%. The results versus the PUT were also improved, but by only 7%. We investigated the repairs in more detail to identify possible reasons for this improvement.
the impact of the IPFs’s layout distortion on the readability of where the font size was the same. Here we saw a smaller comparison of our approach’s results against those of the PUT, played a role in this score change is also supported by the from 12px to 8px (a 33% reduction). The idea that font size readability score dropped 50%, and the font size was reduced 7px (a 40% reduction). In the second most extreme case, the dropped 68% and the font-size was reduced from 12px to 8px (a 33% reduction). The idea that font size played a role in this score change is also supported by the comparison of our approach’s results against those of the PUT, where the font size was the same. Here we saw a smaller improvement in the readability score, likely only reflecting the impact of the IPFs’s layout distortion on the readability of the page.

For attractiveness, the repairs generated by our approach were rated more attractive than both the PUT and the pages generated by IFFix. While the improved attractiveness versus the PUT is not meaningful by itself, it represents a sanity check that the resulting repair is something that users would consider an improvement over the un-repaired web page. We investigated the repairs in more depth to understand the reasons behind our approach’s higher ratings versus IFFix. To do this, we again examined the results that showed the largest decreases in their attractiveness ratings and found that these also generally correlated with the largest decreases in font size in the repairs generated by IFFix.

For similarity to the baseline, the results were also positive. Users perceived our repairs as leading to pages significantly more similar to the baseline than the pages that contained the IPFs (i.e., the PUTs). The results also showed that there was no significant difference between the similarity scores awarded to IFFix and CBRepair. This is a reasonable result and shows that both approaches are able to produce pages that look similar to the original baseline page.

Since our results analysis indicated that font size was a likely significant factor affecting attractiveness and readability, we investigated this issue further. We repeated experiment two but compared the repairs generated by the font-size restricted IFFix (IFFix' with the repairs generated by CBRepair. Our results showed that the user perceived quality for IFFix’ increased significantly and there was no statistically significant difference between the attractiveness and readability scores of IFFix’ and CBRepair. These results suggest that our decision to prevent changes to font size helped our approach to generate repairs that were more attractive and readable than those generated by IFFix. However, it is important to note that with this restriction IFFix’ was only able to repair four of the 23 subjects. So incorporating this restriction into IFFix would then involve other design tradeoffs and adaptations to make it as effective as our approach.

Overall, results from RQ3 strongly indicate that our approach, CBRepair, can repair IPFs with improvements in attractiveness and readability over the repairs generated by IFFix while maintaining the intended baseline look.

E. Threats to Validity

A potential threat to external validity is that our approach was only applied to web applications chosen by GWALI and IFFix. However, the subjects were originally chosen to represent a variety of different translation technologies and layout styles. This helps to ensure that our results can be generalized to a wider selection of web pages.

A threat to construct validity is that the number of IPFs (RQ1) is based on an automated tool. However, GWALI is currently the only available automated tool that can detect and quantify IPFs. Furthermore, we included a user study in RQ3 to support the conclusion that the repairs represented an improvement to the pages.

Another threat to construct validity is that human perception or judgment may be too subjective to correctly judge the repair quality. Some users may judge the quality based on the attractiveness, readability, or similarity to the baseline. We addressed this issue by using all three evaluation metrics to evaluate different aspects of web page’s quality from the users’ perspective.

Another potential threat is that the participants on AMT may not understand the translated language to provide a valid interpretation for the “readability” metric. However, all survey questions included two snippets, which allows the users’ to give relative ratings to indicate if they can easily recognize the different characters in the text without the need to actually understand the meaning of the text.
V. RELATED WORK

The most closely related technique to our work is ZFix [5], which uses a search-based technique to automatically repair IPFs in web pages. The technique works by exploring a large number of values for the CSS properties that could repair the faulty elements in a web page. ZFix suffers from two main issues. First, it requires extensive time (up to 19 minutes) to find a repair for a single web page. Second, for many web pages, ZFix generates repairs where the repaired version has a font-size reduced to a very small value, which affects the readability and the attractiveness of the web page.

Different techniques [20], [21], [22], [23], [24] have been developed to perform automated checks for several common internationalization problems, such as corrupted text, incorrect encoding, and incorrect/missing translations. These techniques are designed to run a list of pre-defined tests on the application to detect IPFs, but are not capable of repairing IPFs.

Responsive Web Design (RWD) approaches (e.g., [25], [26], [27]) are effective in designing layouts that adapt to different screen sizes and their use can help reduce the appearance of IPFs. However, RWD cannot guarantee that the resulting pages will be IPF free. Frameworks, such as Bootstrap, require developers to annotate elements with classes that have pre-defined responsive behaviors, while techniques, such as DECOR, 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. Underscoring this point, in our evaluation, two of the experiment’s subjects (doctor and twitterHelp) were designed using Bootstrap and contained IPFs that could be repaired by our approach.

Other techniques can be used to detect different types of presentation failures (e.g., mock-up driven development [28], [29], [30], [31] and RWD [32], [33], [34], [35]). These techniques are analogous to GWALI as they all detect presentation problems. However, previous work shows that such techniques can report a large number of false positives when used to detect IPFs in web pages [13], which makes them unsuitable as a starting point for our approach.

The repair of other types of presentation problems, such as mobile-friendliness and Cross-Browser Issue (XBI), cannot be handled by our approach at this time. These other types of problems require modeling more complex constraints that seem to require more expressiveness than the linear constraints that are used for IPFs. However, in future work, we plan to investigate this topic to identify new types of constraint systems that could solve these problems and compare them against existing techniques [9], [8], [10].

Cassius [36] and its extension VizAssert [37] provide an extensible framework for reasoning about web pages’ layout. The framework can be used to repair faulty CSS in web applications by assuming the availability of a set of faulty CSS values and page layout examples that the technique can use to synthesize a repair. The technique uses the CSS from the page layout examples as the oracle to identify the fix values for the faulty CSS. In the IPF domain, however, the page’s before and after translation share the same CSS file. Therefore these techniques are not applicable for repairing IPFs.

An extensive work in the area of automated GUI testing [38], [39], [40] focuses on testing the behavior of the system based on event sequences triggered from the user interfaces. These techniques differ from our approach in that they are not focused on testing the appearance of the GUI, but instead they use the GUI to test the behavior of applications.

Lastly, a group of techniques focuses on repairing different types of failures in web applications, but none of them can repair IPFs. PhpRepair [11] and PhpSync [12], 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 [41] assumes that an HTML/CSS fix has been found and focuses on propagating it to the server-side using hybrid analysis.

VI. CONCLUSION

The translation of a web page into different languages can cause expansion or contraction in the text, leading to distortions in the translated web page appearance. These distortions are known as Internationalization Presentation Failures (IPFs). Existing techniques to repair IPFs are slow, and often negatively impact the readability and the attractiveness of the web page. In this paper, we introduced a new approach to overcome the limitations of existing IPF repair technique. Our approach models the layout of a web page as a system of constraints. Then it uses constraint solvers to quickly find solutions that could be used to repair the web page. The use of constraint solvers allows our approach to maintain the attractiveness of the original web page. The evaluation of our approach shows that it is significantly faster. Despite that existing technique can repair a larger subset of IPFs, our evaluation indicates that the quality of our approach’s repairs are substantially better in both attractiveness and readability.

ACKNOWLEDGMENT

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

REFERENCES


