Detecting and Localizing Internationalization Presentation Failures in Web Applications

Abdulmajeed Alameer, Sonal Mahajan and William G. J. Halfond
University of Southern California
Los Angeles, California, USA
{alameer, spmahaja, halfond}@usc.edu

Abstract—Web applications can be easily made available to an international audience by leveraging frameworks and tools for automatic translation and localization. However, these automated changes can distort the appearance of web applications since it is challenging for developers to design their websites to accommodate the expansion and contraction of text after it is translated to another language. Existing web testing techniques do not support developers in checking for these types of problems and manually checking every page in every language can be a labor intensive and error prone task. To address this problem, we introduce an automated technique for detecting when a web page’s appearance has been distorted due to internationalization efforts and identifying the HTML elements or text responsible for the observed problem. In evaluation, our approach was able to detect internationalization problems in a set of 54 web applications with high precision and recall and was able to accurately identify the underlying elements in the web pages that led to the observed problem.

I. INTRODUCTION

Web applications enable companies to easily offer their services and products on a worldwide basis. Although this capability provides companies with many benefits, it also introduces the challenges of internationalization. Developers must design their websites to handle different languages while maintaining the appearance and aesthetics of their web pages’ user interfaces (UIs). Maintaining the attractiveness of a web page is important, as previous studies have shown that users’ impression of a website can be formed within fifty milliseconds [18] of viewing a page and that users often base their impressions of trustworthiness and quality, and ultimately, decisions to purchase, on the design and visual appearance of a web page [15], [12], [14].

Extensive infrastructure exists to support internationalization by web developers [13]. This enables developers to isolate “need-to-translate” strings in language-specific resource files, and load the correct resource files for the languages accepted by the requesting browser at runtime. This mechanism enables developers to support localization for any one of the languages specified in the ISO 639-3 standard, which currently has more than 7,700 entries [6]. Alternatively, developers can integrate translation APIs into their websites, such as the Google Website Translator [4], which allows the visitors to select their preferred language from a drop down menu and have the API automatically translate text elements. Although these mechanisms enable the use of translated strings, they do not address the problem of maintaining the appearance of the translated version of the page. As we discuss in Section II, translated versions of text can be different lengths and heights, depending on the character set of the target language. This can lead to an Internationalization Presentation Failure (IPF), which is an undesired distortion of the page’s intended appearance as HTML elements expand, contract, or move in order to handle the translated text.

To prevent IPFs, there are best practices that developers can follow. However, most are too generic to do anything more than serve as a reminder. For example, one large software engineering company’s guidelines for internationalization states, “Provide for effective presentation of the UI after the expansion that results from translation.” [7] Although such guidelines are helpful, they do not provide concrete or specific guidance to developers on how to accommodate for such expansions. Given the lack of effective guidance for avoiding the problem, developers must resort to proactive detection of IPFs. Although it is possible to test for IPFs, this effort requires developers to manually examine each page with each language option and compare its appearance to the intended appearance. Unsurprisingly, this can be a labor intensive and error prone process.

Automated support in the research and practitioner community is limited with respect to supporting internationalization issues. One notable exception is work that locates strings in web applications that developers need to translate before internationalization can be considered complete [29]. Although this technique helps developers to carry out the internationalization process more thoroughly, it does not help developers detect when their translations have led to an IPF. Apple’s pseudo-localization testing [1] attempts to help developers identify IPFs, but requires them to check each page of an app manually. There are other techniques that focus on other types of presentation failures. Although not intended to detect IPFs, they can detect a subset of these failures. For example, Fighting-Layout-Bugs [27] uses image processing techniques to detect overlapping text, a common symptom of IPFs. Cross-Browser Testing (XBT) techniques (e.g., [10], [11], [26]) and general presentation failure finding techniques (e.g., WebSee [21]) can also detect many IPFs, but as we show in the evaluation, they also generate many false positives since they often detect any change in the page caused by the translations of the text as a presentation failure.

In this paper we present a novel approach that specifically
targets the detection of IPFs. Our technique assists developers in automatically detecting when internationalization efforts have caused a page’s appearance to become distorted and helps them in identifying the elements responsible for this problem. The basic intuition of our approach is to build rendering based models of web pages, focusing primarily on the visual relationships between translated text and layout related HTML elements. These models are then compared to identify elements whose appearance or relative position has changed between internationalized versions. Our approach then analyzes the identified elements to produce a ranked list of potentially faulty elements for developers. Our evaluation of the technique shows that it is very accurate, it can detect IPFs with 91% precision and 100% recall, and identify the faulty element with a median rank of three. Our technique is also fast and can perform this detection and localization for a given webpage in 9.75 seconds. Overall these results are very positive and indicate our approach could help developers in preventing IPFs.

The organization of our paper is as follows. In Section II we present some basic background information and definitions related to IPFs. We describe the approach in Section III and its evaluation in Section IV. A discussion of related work appears in Section V, and we conclude in Section VI.

II. BACKGROUND

An internationalized web application can be built using many approaches. However, two general techniques have become widespread and popular. The first approach is to isolate the “need-to-translate” text and images into separate language-specific resource files. A user’s web browser provides the user’s preferred languages and a server side framework loads the correct language specific resource files and inserts them into placeholders in the web page. This modularization of the language-specific content allows for easier management of the internationalized website. The second approach is to use online automated translation services (e.g., Google Website Translator [4]). With this approach, the developers install a plugin in their website. The visitors to the site can select their desired language from a drop-down box and the plugin will scan the web page to find all of the textual content. Then, the plugin will send that text to the online automated translation service, which will reply with the translation of the text. The plugin then replaces the original text 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. However, neither of these approaches provides any guarantee that the translated text will not cause an IPF, and it is up to the web developer to verify that the pages’ user interfaces have not been distorted.

Figure 1 shows a simplified version taken from the Hotwire website. In the Mexican version of the page, the price text overflows its container. This overflow is an example of an IPF. In general, the reason this type of a problem occurs is because the size of the text varies significantly based on its language. The resulting change in text size causes the text to overflow its intended containing element or, if the containing element grows with the text, the container growth can cause other elements to move, changing the UI layout. This change in the size of the text is mainly affected by three factors: number of characters in the translated text, the language’s characters’ width, and the language’s characters’ height. Some of these changes can be quite dramatic. For example, in English the word “views” translates to “visualizzazioni” in Italian, which is almost three times longer. More generally, IBM has found that an English text that is less than ten characters could increase in size from 100% to 200% when translated to another language [7]. The potential for this kind of expansion must be anticipated by developers in the design of their web pages. Additionally, the correct rendering of a page must be verified, as the fact that it renders properly for one language does not imply that it will render correctly for other languages.

It is possible for a developer to manually check for the presence of IPFs. To do this though, the developer would have to manually compare the layout of a baseline page, which is known to be correct, against the translated versions. Depending on how many languages are supported, this could quickly grow to an unmanageable amount of comparisons. For example, Google Translate can allow websites to be translated to up to 90 different languages. Furthermore, once an IPF has been detected, the underlying root cause can be difficult to determine since the appearance of modern web pages is controlled by a complex rendering interaction of HTML, CSS, and JavaScript. This means that the connection between an observed failure and the underlying fault is often not straightforward. These complications makes both detection and localization of IPFs a time consuming and error-prone process.

III. APPROACH

The goal of our approach is to automatically detect IPFs and identify the translated text that is responsible for the failure. Our key insight is that IPFs are caused by changes in the size of translated text. Therefore, our approach defines and builds a model, called the Layout Graph (LG), that captures the visual relationships and relative positioning of HTML tags and text elements in a web page (Section III-A). To use our approach, a tester provides two web pages as input: the first is the Page Under Test (PUT) and the second is a baseline version of the page that shows the correct layout. Typically, the baseline would be the original version of the page, which is already known to be correct and will be translated to another language, as represented in the PUT. Our approach first builds a LG for each of these pages (Section III-B). The approach then compares these two LGs and identifies differences between them that represent potentially faulty elements (Section III-C). Finally, the approach analyzes and filters these elements to produce a ranked list of elements for the developer (Section III-D).
A. Layout Graph Definition

The LG is a model of the visual relationships of the elements of a web page. As compared to models used in related work, such as the alignment graph [11] and R-tree [21], the LG focuses on capturing the relationships of not only the HTML tags, but also the text contained within the tags. The reason for this is that the primary change to a web page after internationalization is that the text contained within the HTML tags has been translated to another language. The translated text may expand or shrink, which can cause an IPF. Therefore, the LG includes the text elements so that these changes can be more accurately modeled and compared.

The LG is a complete graph defined by the tuple \( (V, F) \), where \( V \) is the set of nodes in the graph and \( F \) is a function \( F : V \times V \to \mathcal{P}(R) \) that maps each edge to a set of visual relationships defined by \( R \). Each node in \( V \) represents an element that has a visual impact on the page. A node is represented as a tuple \( (t, c_1, c_2, x) \), where \( t \) is the node type and is either “Element” (i.e., an HTML tag) or “Text” (i.e., text inside of an HTML tag), \( c_1 \) is the coordinate \( (x_1, y_1) \) representing the upper left corner of the node’s position on the page, \( c_2 \) is the coordinate \( (x_2, y_2) \) representing the lower right corner of the node, and \( x \) is the XPath representing the node. The two coordinates represent the Minimum Bounding Rectangle (MBR) that encloses the element or text. The set \( R \) of possible visual relationships can be broken into three categories, direction (i.e., North, South, East, West), alignment (i.e., top, bottom, left, right), and containment (i.e., contains and intersects).

![Fig. 1: Part of a web page and its localized version](image)

![Fig. 2: The MBRs of the two versions of the page](image)

B. Building the Layout Graph

In the first phase, the approach analyzes the PUT and baseline page to build an LG of each. The approach first analyzes the Document Object Model (DOM) of each page to define the LG’s nodes (i.e., \( V \)) and then identifies the visual relationship between the nodes (i.e., \( F \)).

The first step of building the layout graph is to analyze the baseline page and PUT and compute the nodes in the LG. For each of these pages, this process proceeds as follows. The page is rendered in a browser, whose viewport size has been set to a predefined value. This chosen viewport size has to be the same for both pages for a given comparison, but can be varied to detect IPFs at different resolutions. After the page is rendered in a browser, the approach uses the browser’s API to traverse the page’s DOM. For each HTML tag \( h \) in the DOM, the approach collects \( h \)’s XPath ID (i.e., \( x \)), finds \( h \)’s MBR based on the browser’s rendering of \( h \) (i.e., \( c_1 \) and \( c_2 \)), and assigns the type “Element” to the tag. If the node contains text (e.g., text between \( <p> \) tags or as the default value of an \( <input> \) text box) then the approach also creates a node for the text itself. For this type of node, the XPath is the XPath of the containing node plus the suffix “/text()”, the MBR is based on the size and shape of the text within the enclosing element, and the type is denoted as “Text.” This process is repeated for all HTML tags found in the page’s DOM with three exceptions, which we describe below.

The first exception is HTML tags that are not visible in the page. These tags do not affect the layout of the page and therefore do not have a visual relationship with any other tag. Officially, there are specific HTML and CSS properties, such as \texttt{visibility:hidden} and \texttt{display:none}, that can be used to cause a tag to not display. Unofficially, there are a myriad of ways that a developer can use to hide an element. These include setting the \texttt{height} or \texttt{width} CSS properties to zero; using the \texttt{clip} CSS property to cut an element to a zero pixel rectangle; and setting a very high value for the \texttt{text-indent} property to render the element outside the boundary of its container while also setting the \texttt{overflow} property to hidden. Our approach detects these and other mechanisms, and then does not create a node in the LG for the HTML tag. This detection is done by querying the browser’s API after the page is rendered and getting the applicable HTML and CSS properties for each element. The second exception is for HTML tags that do not affect the layout of the page. The tags are not explicitly hidden, as described above, but are nonetheless not visible in the page’s rendering. These types of tags may be used to provide logical structure to the page. For example, a \texttt{<div>} may be used as a container to group other nodes. As with hidden tags, there are many ways to define these tags. Some of the heuristics we employ for this identification process are: (1) container elements that do not
have a border and whose background color is similar to its parent’s background color; (2) tags that have a very small dimension; (3) tags only used for text styling, such as `<font>`, `<strong>`, and `<b>`; and (4) tags representing an unselected option in a select menu.

The third and final exception is for HTML tags embedded in the text of another tag. An example of this is shown in Figure 4 where the link labeled “options” has moved to the second line after the translation and therefore would have a different visual relationship with the link labeled “free trials.” Intuitively, we know that such changes are inevitable due to translated text and should not be considered as IPFs. Therefore, the approach groups such tags together and creates one node in the LG for them with an MBR that surrounds all of the grouped elements and assigns to that node the type “Text.”

Fig. 4: Example of non-failure text changing position after translation

After computing the nodes of the graph, the second step of the approach is to define the $F$ function, which annotates each edge in the graph with a set of visual relationships. Recall that an LG is a complete graph, so this step is computing the visual relationship between each pair of nodes on each edge. To compute the visual relationship between two nodes on an edge, the approach compares the coordinates of each node’s MBR. For example, for an edge $(v,w)$, if $v.y_2 = w.y_1$ then the relationship set would include North. Similarly, if $v.y_2 = w.y_2$ then the set would include Bottom-Aligned and if $(v.x_1 \leq w.x_1) \land (v.y_1 \leq w.y_1) \land (v.x_2 \geq w.x_2) \land (v.y_2 \geq w.y_2)$ then it would include the Contains relationship. The other relationships are computed in an analogous manner.

To illustrate the graph building process, consider the two web pages shown in Figure 1. The MBRs identified for these two web pages are shown in Figure 2. Next, Figure 3 shows the LGs produced for the American and Mexican versions of our example, which hereafter we refer to as $LG$ and $LG'$ respectively. As can be seen, the graph is a complete graph with edges labeled with the spatial relationships between the nodes they represent. For example the edge $(./img, ../div)$ is labeled with the relationships “South, RAlign, LAlign,” which means that the element ./img is to the South of the element ../div and is also Right Aligned and Left Aligned with it.

C. Layout Graph Comparison

In the second phase, the approach compares the two LGs produced by the first phase in order to identify differences between them. The differences that result from the comparison represent potentially faulty tags or text that will be filtered and ranked in the third phase. A naive approach to this comparison would be to pair-wise compare the visual relationships annotating all edges in $LG$ and $LG'$. In experiments we found that the drawback of this approach was that differences were detected for tags that were far away from each other on the page and whose relative change in position was not an IPF. Instead, our approach compares subgraphs of nodes and edges that are spatially close to a given node $n$ in the LG. Our insight, confirmed by the experiments reported in Section IV, is that comparing these more limited subgraphs of $LG$ and $LG'$, which we refer to as neighborhoods, is sufficient to accurately detect IPFs and the responsible faulty elements. In the rest of this section, we explain the details of this comparison, which requires first determining the nodes to be compared and then identifying the neighborhood for each node.

Before any comparison can take place, the approach must identify nodes in $LG$ and $LG'$ that represent the same HTML element. Although each node contains an XPath, as we described in Section II, certain translation frameworks, such as the Google Translate API, may introduce additional tags. This means that the XPaths will not be an exact match. To address this problem, we adapted a matching approach defined by the WebDiff XBT technique [10]. This approach matches elements
probabilistically using the nodes’ attributes, tag names, and
the Levenshtein distance between XPath IDs. We adapted this
approach to account for common variations introduced by the
translation frameworks. The output of our adapted matching
approach is a map $M$ that matches each HTML tag or text in
the baseline page with a corresponding tag or text in the PUT.
In our experiments, we found that this matching was close to
perfect for our subjects because the translation API introduced
regularized changes for all translated elements.

After computing $M$, the approach then identifies the neigh-
borhood for each $n \in LG$. To do this, the approach first
computes the coordinates of the four corners and center of
$n$’s MBR. Then, for each of these five points, the approach
identifies the $k$-Nearest Neighbors ($k$-NN) nodes in the $LG$.
The neighborhood is defined as the union of the five points’
$k$-NNs. The closeness function in the $k$-NN algorithm is
computed based on the spatial distance from the point to
any area occupied by another node’s MBR. The calculation
for this is based on the classic $k$-NN algorithm [25]. In our
experiments, we have found that the approach works best when
the value of $k$ is set proportionally to the number of nodes in
the $LG$.

The final step is to determine if the relationships assigned
to edges in a neighborhood have changed. To do this, the
approach iterates over each edge $e$ that is part of the neigh-
borhood of any $n$ in $LG$ and finds the corresponding edge
$e'$ in $LG'$, using the previously generated $M$ function. Note
that the corresponding edge always exists since both LGs are
complete graphs. Then the approach computes the symmetric
difference $\delta$ between $F(e)$ and $F(e')$, which identifies the
visual relationships assigned to one edge but not the other. If
the difference is non-empty, then the approach classifies the
data as a potential issue. The output of this step is $I$, a set of
tuples of the form $(e, e', \delta)$

To illustrate the graph comparison, consider the LGs shown
in Figure 3. Consider the node labeled “..div/div/text( ).” The
bold edges connected to this node represent its neighborhood
that will be compared against its counterpart in the other LG.
In the figure, we have underlined and highlighted in red the
labels on the edges for which there is a difference in the
relationships. The edges with underlined red labels will be
reported as potential issues and analyzed further in the third
and final phase of our approach (Section III-D).

D. Ranking the Likely Faulty Elements

In the third and final phase, the approach analyzes the set of
tuples, $I$, identified in the second phase and generates a ranked
list of HTML elements and text that may be responsible for
the observed IPFs. To identify the most likely faulty elements,
the approach applies three heuristics to the tuples in $I$ and then
computes a “suspiciousness” score that it uses to rank, from
most suspicious to least suspicious, the nodes associated with
the edges in $I$.

The first heuristic serves to remove edges from $I$ that were
flagged as a result of to-be-expected expansion and contraction
of text. An example of this is shown in Figure 4. Here a
<div> container element surrounds the icon, title, and text
block. The comparison of the two LGs would detect that the
bottom alignment of the two text blocks has changed and
the comparison would report a potential issue. However, in
this case this should not be counted as an IPF because the
container element has enough space to allow for the text to
expand without disrupting anything outside of the containing
element. To identify this situation, the approach identifies all
dges where the type of the two constituent nodes is either
Text/Element or Text/Text. If the $\delta$ of any of these edges
contains alignment related relationships, then these relation-
ships are removed from $\delta$. If $\delta$ is now empty, then the tuple is
removed from $I$. This heuristic only allows alignment issues
to be taken into account if they affect the visual relationship
between nodes that represent HTML elements.

The second heuristic establishes a method for ruling out low
impact changes in the relative location of two elements. An
eexample of such a change is shown in Figure 5. In the English
version, the edge between the header element “Discover” and
the search box is labeled West. After the translation to Russian,
the element is shifted, and the relationship is changed, so the
label becomes South. Although this is technically a distortion,
it may or may not rise to the level of being considered an
IPF. To calibrate for such situations, our approach allows
testers to provide an $\alpha$ threshold that denotes the degree of
allowed change. For each pair of nodes in an edge in $I$, if the
$\delta$ of that edge contains direction related relationships, then
the approach uses the coordinates of the MBRs to calculate
the change (in degrees) of the angle between the two nodes
forming the edge. If the change is smaller than $\alpha$, then these
direction relationships are removed from $\delta$. If $\delta$ is now empty,
then the tuple is removed from $I$. In our experiments we found
that $\alpha = 45$ provided a reasonable balance in terms of flagging
changes that would be characterized as disruptive and reducing
false positives.

The third and final heuristic expands the set of edges in
$I$ to include suspicious ancestor elements of nodes whose
relative positions have changed. An example of such a change
is also shown by Figure 5. The elements in the header are $li$
elements. After the translation of the page, the element $li[6]$, which
represents “Troubleshooting” has been pushed down to a new row in the Russian version of the page. Here, we
cannot consider the faulty element to be only $li[6]$. In fact,
be pushed down, so we report an XPath selector that represents
all the text children of the parent $ul$ element. To handle this
situation, when an edge in $I$ is found that has a directional
visual relationship that has changed, the approach traverses the
DOM of the page to find the Lowest Common Ancestor (LCA)
of both nodes and adds an XPath selector that represents all of
its text children to the list of nodes that will be ranked.

After the three heuristics have been applied to $I$, the
approach generates a ranked list of the likely faulty nodes. To
do this, the approach first creates a new set $I'$ that contains
tuples of the form $(n, s)$, where $n$ is any node present in
an edge in $I$ or identified by the third heuristic and $s$ is
a suspiciousness score, initialized to 0 for all nodes. The approach then increments the suspiciousness scores as follows: (1) every time a node $n$ appears in an edge in $I$, the score of $n$ is incremented; and (2) the score of a node $n$ is increased by the cardinality of the difference set (i.e., $|\delta|$). For any XPath selector that was added as a result of the third heuristic, its suspiciousness score is incremented by the number of times it is added to the list. Once the suspiciousness scores have been assigned, the approach sorts $I'$ in order from highest score to lowest score and reports this list to the developer. This list represents a ranking of the elements determined to be the most likely to have caused the detected IPFs.

IV. EVALUATION

To assess the effectiveness of our approach for detecting and localizing IPFs, we implemented it as a prototype tool to evaluate its accuracy, quality of the localization results, and time needed for it to run. Specifically, we addressed the following research questions:

**RQ1:** What is the accuracy of our technique in detecting IPFs?

**RQ2:** If a failure is detected, what is the quality of the localization results provided by our technique?

**RQ3:** How fast is our technique in detecting and localizing IPFs?

To address these questions, we carried out an empirical evaluation of our approach on a set of real-world web applications and compared our results with three well-known approaches for detecting different types of presentation failures in web applications, WebSee [21], X-PERT [11], and Fighting Layout Bugs (FLB) [27]. The following sections, we describe the implementation of our approach and the subject applications used for the experiments. Then we discuss each of the research questions.

A. Implementation

We implemented our approach as a Java prototype tool, GWALI (Global Web Applications’ Layout Inspector). We used several third party libraries to implement some of the functionality required by our approach including the Java Spatial Index library to find the $k$-Nearest Neighbors of the LG’s nodes. Selenium Firefox Webdriver with Firefox version 39.0 was used to load the webpage and execute Javascript to compute the MBRs of text and tag elements in the web page. We ran our experiments on a 64-bit Linux Ubuntu 14.04 machine with a screen resolution of $1920 \times 1080$, 8GB memory, and an Intel Core i7-4790 CPU.

We also compared our approach with three well-known approaches. The first one is WebSee [21], which is a tool designed to find presentation failures in web applications using computer vision techniques. When running WebSee, we also used WebSee’s exclusion region feature to exclude the areas in the pages that contain text from the comparison since WebSee will report a failure once it finds the text in the PUT to be different from the text in the baseline. The second tool we compared GWALI against is FLB [27], which allows developers to automatically detect common layout bugs in the page. The tool provides multiple detectors. Some of the detectors, such as detecting images with invalid URLs or detecting text that is not readable due to low contrast with its background, were not related to IPFs. To reduce false positives, we disabled these detectors and enabled only the ones that were related to internationalization, which were detecting text that is near or overlapping with horizontal or vertical edges. The third tool we compared our approach against is X-PERT [11], which is designed to detect Cross-Browser Issues (XBIs). X-PERT works by loading the same page into two different browsers and comparing the rendering of the two pages. We made a modification to the code, so that we could perform the comparison for two different pages in two windows for the same browser. X-PERT provides multiple modules to perform the comparison. To reduce false positives, we enabled only the structural detection module, since the other modules, such as the visual detection and the text detection modules, were not relevant for detecting IPFs.

B. Subject Applications

Our subject pool contained 54 different web applications. To ensure diversity in our subject pool, we selected interna-
tionalized web applications that were: (1) visited frequently; (2) used different translation technologies; (3) translated into different languages; and (4) designed with different layouts and styles. We identified potential subjects by searching three different sources. The first of these was *builtwith.com*, which is a website that indexes web applications built using various technologies. This site helped us to easily locate websites that were built using different kinds of translation frameworks, such as the Google Website Translator, and ensure that our subjects covered a wide range of translation technologies and frameworks. The second source was the Alexa top 100 most visited web sites list. The third and final source was high-profile websites that targeted international audiences, such as travel-related and telecom company websites and were expected to provide their content in multiple languages. For each of these three sources, we manually inspected the identified websites to find web pages that contained an IPF. We found 30 such pages and used these as the ground truth for true positive IPFs. For true negative IPFs, we selected 24 pages that were internationalized but that did not contain an IPF. Due to space constraints we do not list all of the subjects, but a complete list is available from the project’s website [5]. For each subject application, this process resulted in a baseline page, which we assumed was a correct rendering, and a PUT in another language that contained either zero faults (i.e., a true negative) or at least one IPF (i.e., a true positive). Overall, the faulty web sites identified in this process contained a wide range of IPFs, whose impact ranged from misaligned text to a complete distortion of the page that left it impossible to read. Figure 5 shows an example of the former and Figure 6 shows an example of the latter.

To ensure our experiments were repeatable and not affected by changes in the live websites, we used the Scrapbook Firefox plugin to download local copies of the subject webpages along with all of the corresponding images and styles that were required for the pages to be rendered correctly. We found that some pages used JavaScript to dynamically change content (e.g., rotating main news items) in the DOM, which made it difficult to compare the two versions of the page unless their changes were synchronized. To avoid this, we disabled JavaScript in the page after the pages were loaded and rendered in the browser.

### C. Experiments and Results

To answer RQ1, we measured the precision and recall of GWALI’s detection for the subject applications. We ran GWALI, WebSee, X-PERT, and FLB for every pair of pages (i.e., the baseline and PUT), and compared the output they produce with the manually assigned ground truth. We considered the tools to have a successful detection if they indicated there was a failure of any type in the PUT, regardless of whether the output contained the faulty element. Then we calculated the precision and recall of the tools’ ability to correctly detect if there is a failure. Table I shows the precision and recall of GWALI compared to WebSee, X-PERT, and FLB.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GWALI</td>
<td>91</td>
<td>100</td>
</tr>
<tr>
<td>WebSee</td>
<td>55</td>
<td>100</td>
</tr>
<tr>
<td>X-PERT</td>
<td>55</td>
<td>100</td>
</tr>
<tr>
<td>FLB</td>
<td>73</td>
<td>55</td>
</tr>
</tbody>
</table>

For precision, the results in Table I show that GWALI could detect IPFs with 91% precision, which was much higher than the other approaches. We analyzed the results to understand the reasons for the false positives reported by each of the approaches. For GWALI, we found that there were two general causes of false positives. The first cause was when part of a web page’s content was displayed in multiple layers using the z-index CSS property. The LGs generated by GWALI model the layout of the page in two dimensional space, effectively merging different layers to one. In some cases, web designers use the z-index CSS property in designing sliding content areas. In these cases, the content of every slide has a different z-index value. Our model does not take that into account and merges the slides into one layer which caused false positives. The second cause of false positives was when the change of a direction relationship was large (i.e., greater than $\alpha$ as described in Section III-D), but this change did not distort the layout of page. The $\alpha = 45$ value is a heuristic, and in some cases, such as when a large amount of text shrinks, this can make the change exceed this value. For FLB, the main cause of false positives was the incorrect identification of the horizontal and vertical edges. This was due to inaccuracies in the image processing techniques used to identify the edges. For both WebSee and X-PERT, the underlying root cause of false positives was the same. Both approaches detected any difference with the oracle as a failure. This included any shifting of elements due to expanding or contracting text size.

For recall, the results in Table I show that GWALI, WebSee, and X-PERT had 100% recall. We investigated the results of FLB in order to better understand the reason for its false negatives. We found that this was due to the fact that the detectors in FLB could only detect a subset of the types of visible failures that could be associated with an IPF. In particular, the only detection rule in FLB that could detect symptoms related to IPFs was the one we had enabled, the check for overlapping text. Since not all IPFs had overlapping text, many IPFs were missed by the FLB approach.

![Fig. 6: Text overlapping with a button after translation](image-url)
Overall these results show that GWALI is very accurate in detecting IPFs. Although other approaches had the same level of recall, GWALI also had much higher precision. It is important to note that these results do not indicate that the other techniques are, in general, inaccurate. Only that their detection mechanisms are not as effective as GWALI’s when trying to detect IPFs.

To answer RQ2, we calculated the number of elements that a developer would be expected to examine when using the output of the evaluated approaches. To determine this, we first ran all of the approaches, except for FLB, on the 30 subjects that contained one or more IPFs. We did not evaluate the localization for FLB because it provides its output as a marked up screenshot of the page, with areas of the observed failures highlighted. It was not clear how this result could be quantified and compared with the other approaches.

Both GWALI and WebSee report a ranked list of potentially faulty elements, which we used as a proxy measure for expected effort. Although an imperfect metric, rank is widely used and allows us to quantify and compare results without the expense of a field study with real developers. For subjects with only a single fault, we simply reported the rank of the faulty element after running the tool. For subjects with multiple faults, we calculated the rank of each fault using a methodology proposed by Jones and colleagues [17]. The general idea of this methodology is to report the rank of the first faulty element that appears in the result set, simulate the fix of that fault, and then rerun the analysis to get the ranking of the next highest fault. This is repeated until all faults are “fixed.” The general intuition of the motivation behind using this methodology is that it approximates the workflow of a developer who scans the results, fixes a fault, and then reruns the analysis to see if any more faults remain.

Calculating expected effort for X-PERT is a little more complicated because it returns an unordered set. If the faulty element is present in this set, then its location follows a uniformly random distribution. In the case of a single fault, the expected effort is therefore \((n+1)/2\), where \(n\) is the size of the set, since the developer would, on average, have to examine half the elements in the set before finding the faulty element. Calculating this metric for the case when there are multiple faults generalizes to the problem of calculating the average number of comparisons for a linear search for \(k\) number of items in an unordered set of size \(n\) where the distribution of the \(k\) items is uniformly random. Equation (1) shows the standard equation for calculating this metric.

\[
\frac{n + 1}{k + 1}
\]

An additional complication of calculating the expected number of comparisons for X-PERT is that its output may not contain the faulty element. In this case, we need to approximate the amount of additional effort that would be required to find the element in the remaining set of elements contained in the page. The challenge in this approximation is knowing exactly how developers will search the remaining elements. Instead of directly addressing this question, we calculate an upper and lower bound on the best and worst case scenarios. The best case scenario is that the developer finds the fault as they examine the first element in the remaining set of elements. The expected effort for this would then be the size of the set returned by X-PERT plus one. In the worst case, the developer performs a linear search of the remaining elements. To calculate this, we reuse Equation (1) to determine how many checks would be needed for this scenario. Equation (2) shows the expected number of checks for the scenario where the faulty element has not been found in the output set returned by X-PERT. In this equation, \(n\) is the size of the result set returned by X-PERT, \(m\) is the size of the PUT, and \(k\) is the number of faults.

\[
\begin{align*}
\left\{ \begin{array}{ll}
\frac{n + 1}{k + 1} & \text{best case} \\
\frac{n + \frac{n-k+1}{k+1}}{k+1} & \text{avg case}
\end{array} \right.
\end{align*}
\]

Our results are as follows. For GWALI, the median number of expected comparisons was three; for WebSee, the median was 198; and for X-PERT, the median calculated with the best case assumption was 38 and with the worst case assumption was 157. It is important to note that these numbers represent a lower bound on the expected number of comparisons because we only calculated these numbers for the subject applications that contained IPFs. If we were to compute this number over all subjects that were reported as having IPFs, then the numbers for all subjects would increase. If we assume a linear search for cases where there would be no faulty element, then the median numbers for WebSee and X-PERT would rise significantly because they had a lower precision for detecting IPFs.

![Fig. 7: Histogram of the ranks reported by GWALI](image-url)
We further analyzed the ranking results of GWALI by compiling them into a histogram, which is shown in Figure 7. This diagram shows that over 81% of the correct faulty elements were ranked in the top six returned results. We consider this to be a very strong result. Using a tool, such as Firebug [3], it would take a developer only a few minutes to use the reported XPath IDs to inspect the rendering of six elements.

Overall, we consider these results to be a strong indication that GWALI can accurately localize the faulty element. As with RQ1, we want to emphasize that the results do not show that X-PERT and WebSee are not accurate techniques. Instead, we interpret our results as showing that the localization techniques defined in those approaches are not appropriate for IPFs and that the techniques proposed in our approach are both necessary and more accurate for the IPF localization problem.

To answer RQ3, we measured the running time of GWALI, WebSee, X-PERT, and FLB on the subject applications. For each tool, the total running time included the total time required to start the tool from loading the browser until the tool shut down and produced its output. For GWALI, we also measured the time required for each part of the approach.

For each tool, Table II shows the average, minimum, and maximum measured running time. As the results show, GWALI needed 9.75 seconds on average to run, which is slightly slower than X-PERT (8.97 seconds) and FLB (9.19 seconds). WebSee was significantly slower than the other approaches because its ranking heuristics used sub-image comparison. We further analyzed the time required for the different operations carried out by GWALI. The average distribution is shown in Figure 8. As can be seen from the graph, the most expensive part of the approach is interacting with the browser to collect MBRs and to load the page in the browser. This represents the cost to load and analyze the baseline page and the PUT. In a real-world scenario, we could expect half of this cost (the part associated with the baseline page loading and analysis) to be amortized over the total number of PUTs analyzed (i.e., the number of translated pages compared against.) Also, note that 36% of GWALI’s running time (3.5 seconds) was consumed by the process of loading the browser. This means that the performance of our approach can be significantly improved when the testers use it to check a large number of test cases. This improvement can be achieved by loading the browser once and running all the test cases one after another in the same browser window.

Although the results for this RQ showed that our approach was slower than two of the other approaches, we still consider this to be a positive result. Namely because GWALI’s runtime was not that much slower than the other approaches (less than a second from the fastest) and the average time was under ten seconds, which in absolute terms is not a long time.

### D. Threats to Validity

There are threats to validity which we have tried to address during our evaluation. To address external threats to validity, we made sure that we have a variety of different translation technologies, different languages, and different styles of websites so we have a sample that is representative of the actual web. However, one limitation of our approach is handling right-to-left languages, such as Arabic, Farsi, and Hebrew. In these languages the layout of the page is mirrored, which needs special handling when building and comparing the LGs. We have also made our subjects publicly available via the project website [5] to enable independent inspection of the subjects.

To address threats to internal validity, we checked our results manually to verify correctness. A possible threat to validity is that we fixed the screen resolution for our experiments. In the subject applications, some IPFs could appear or disappear depending on the screen resolution. However, it is important to note that the ground truth of IPFs (i.e., true positives and true negatives) would therefore also vary. Furthermore, these IPFs could be easily detected by systematically running GWALI at different screen resolutions, as is done in related techniques [28]. Another threat is our assumption of how the developers will use the result set produced by our tool. We used rank, which is a commonly used metric to measure the effectiveness of localization. When using rank, we assume that the developers will examine the first ranked element, and check if it is faulty or not, if it is not faulty, then they will examine the second reported element, and so on, until they find the actual faulty element. Our claim of the effectiveness of the localization in our approach relies on this assumption. Also, in case there are multiple failures in the page, our assumption is that the developers use the process described above until they find the first fault, once the first fault is found, it will be fixed first, then they will rerun the tool again on the new version

### TABLE II: Average, min, and max execution time (in seconds) for GWALI compared to other well-known tools

<table>
<thead>
<tr>
<th></th>
<th>GWALI</th>
<th>WebSee</th>
<th>X-PERT</th>
<th>FLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>9.75</td>
<td>519.66</td>
<td>8.97</td>
<td>9.19</td>
</tr>
<tr>
<td>Min</td>
<td>6.08</td>
<td>115.80</td>
<td>6.17</td>
<td>6.56</td>
</tr>
<tr>
<td>Max</td>
<td>36.21</td>
<td>1640.28</td>
<td>17.04</td>
<td>23.84</td>
</tr>
</tbody>
</table>

Fig. 8: Break down of the running time of the tool
of the page, and they will repeat this process until all faults are fixed or the result set contains only non-faulty elements. This is a commonly used assumption in fault localization to measure the effort needed to localize faults [17].

V. RELATED WORK

There has been a lot of research work focusing on testing the GUI of web applications. Although these have not focused on automating the detection of IPFs, they have covered many closely related topics, which we describe below.

WEBDIFF [10], X-PERT [11], and Browserbrite [26] are well-known approaches in the field of Cross-Browser Testing (XBT). These approaches work by comparing two versions of the page that are rendered in two different browsers to detect XBIs. These approaches, and other cross browser testing approaches, suffer from over-sensitivity when used to detect IPFs. In XBT, the two versions of the page are expected to match exactly, and a small change in the page that occurs due to the translation of the text is detected as a failure by these techniques.

ReDeCheck [28] is a responsive web design testing tool. The tool uses a model that is similar to the alignment graph used in X-PERT, and thus suffers from the same problems. Although the tool is effective in detecting layout faults in responsive web design, it does not address the problem of internationalization.

Fighting-Layout-Bugs [27] is a technique that allows developers to detect general correctness properties, such as text that is overlapping with horizontal or vertical edges or text with too low contrast with the background. A drawback of this approach is that it can only detect a limited set of possible IPFs. Another drawback is that it only outputs a screenshot of the page with the failures marked on it, so the developer has to inspect the screenshot and manually search through the page to identify the faulty elements.

WebSee [21], [20], [19] is a tool that uses computer vision techniques to compare a browser rendered test web page with an oracle image to find visual differences. Such a tool is useful in some usage scenarios where the test web page is expected to exactly match the oracle (e.g., Mock-up driven development and regression debugging). However, WebSee has limitations similar to those of XBT techniques with respect to IPFs and does not perform well on detecting or localizing IPFs.

Apple offers to the developers who use Xcode IDE the ability to test the readiness of the layout of their applications to adapt to international text [1]. This is done using pseudo-localization, where all the text in the application is replaced with other dummy text that has some problematic characteristics. The new text is typically longer and contains non-Latin characters. This helps in revealing internationalization faults in the applications. The tool also changes the direction of the text to test right-to-left languages. Then the developer has to verify that elements in the GUI reposition and resize appropriately after the pseudo-localization and the direction change. This technique suffers from two issues. First, it requires manual effort from the developers to inspect every page of the application, which is impractical for applications that have large number of pages. Second, the technique uses an estimate for the amount of possible expansion of text, and developers will not be able to verify the behavior of their GUI if the text expands more.

A tool provided by the World Wide Web Consortium (W3C) called i18n checker [9], when given a webpage, checks for a group of internationalization related properties to make sure they are set properly. For example, it checks if the character encoding is specified in the HTML document, and the language attribute is added in the HTML tag. The tool is effective in providing similar non-layout related checks. However, it does not perform a test on the layout of the page to make sure that it can adapt international text.

Several techniques, Cornipickle [16], Cucumber [2], Crawljax [22], and Selenium WebDriver [8] allow developers to specify assertions on the page’s HTML and CSS data. These assertions will then be checked against the web page’s DOM. There are multiple drawbacks to these approaches. First, these approaches need developers’ guidance by manually specifying desirable properties they want to check in the page, and there could be hundreds of assertions for a single page.

An extensive work in the area of automated GUI testing [24], [23] 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.

VI. CONCLUSION

Many companies internationalize their websites to target customers who speak different languages. This process can introduce IPFs in the layouts of the websites. Manually inspecting every page with each language option to detect such failures is a labor intensive and error prone activity. In this paper, we presented an automated novel approach that can detect and localize IPFs. This is done by building Layout Graphs of the pages and comparing these Layout Graphs to identify the failures. To evaluate our approach, we implemented it as a prototype tool, GWALI. The results of the evaluation show that our approach can detect IPFs with 91% precision and 100% recall. Our approach can identify the faulty element with a median rank of three and with an average running time of 9.75 seconds per web page. We believe these results are positive and show that our approach could help developers to address IPFs in web applications.

ACKNOWLEDGMENT

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

REFERENCES
