Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques

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

***Abstract*—An attractive and visually appealing appearance is important for the success of a website. Presentation failures in a site's web pages can negatively impact end users' perception of the quality of the site and the services it delivers. Debugging such failures is challenging because testers must visually inspect large web pages and analyze complex interactions among the HTML elements of a page. In this paper we propose a novel automated approach for debugging web page user interfaces. Our approach uses computer vision techniques to detect failures and can then identify HTML elements that are likely to be responsible for the failure. We evaluated our approach on a set of real-world web applications and found that the approach was able to accurately and quickly identify faulty HTML elements.**

I. Introduction

An attractive and visually appealing appearance is important for the success of a website. A recent study by Google underscores this point by noting that the average visitor forms a first impression of a web page within the first 50 milliseconds of visiting a page [24] — an amount of time that is heavily influenced by a page's aesthetics. Companies put significant effort into the look and feel of their websites, employing graphic designers and illustrators to carefully craft their layout and graphics. Presentation failures – a discrepancy between the actual appearance of a web site and its intended appearance – can undermine this effort and negatively impact end users' perception of the quality of the site and the services it delivers. These types of failures can also impact the usability of a web application's user interface (UI) or be symptomatic of underlying logic or data problems.

The UIs of modern web applications are highly complex and dynamic. Back-end server code dynamically generates content and client-side browsers render this content based on complex HTML, CSS, and JavaScript rules. This makes debugging presentation failures both a labor-intensive and error-prone process. To illustrate, a tester must visually compare the rendering of each page against an oracle, such as a design mockup, to detect that a presentation failure has occurred. This is labor intensive, since even a simple page can contain hundreds of HTML elements each with several dozen CSS properties that must be verified. Once testers detect a presentation failure, the next step in debugging is to identify the faulty HTML element responsible for the failure. This is difficult since an element's visual appearance is controlled by a complex series of interactions defined by the page's HTML structure and CSS rules. The widespread use of HTML rendering features, such as floating elements, overlays, and dynamic sizing, also increases the difficulty of identifying the faulty element as there is often no obvious or direct connection from the rendered appearance to the structure of the underlying HTML.

Testers have several techniques available to them to help debug presentation failures. However, these techniques have limitations that either reduce their effectiveness or make them inappropriate for general usage. For example, many techniques are focused on one type of presentation failure, such as Cross-Browser Issues (XBIs) (e.g., [19], [7], [8]), or a limited and predefined set of application-independent failure types (e.g., Fighting Layout Bugs [23]), and cannot detect other types of presentation failures. Other techniques can only support debugging efforts where there is a prior working version that can be compared against (e.g., [26], [21]). Finally, a group of techniques require testers to exhaustively specify all correctness properties to be checked (e.g., Selenium [4], CrawlJax [16], Cucumber [1], and Sikuli [6]), which is labor-intensive and potentially error-prone.

To address these limitations, we propose a novel approach to assist developers in debugging presentation failures. Our approach uses computer-vision techniques to detect differences between the actual appearance of a web page and its intended appearance, and then analyzes rendering maps of the web page to identify the HTML elements most likely to be responsible for the observed differences. As compared to current techniques, our approach has several distinct advantages. (1) By design, our approach reduces manual effort – it does not require testers to manually specify every correctness property to be checked. (2) The approach can detect a broad range of failures – it does not impose restrictions on the type of underlying fault that can be detected. (3) Our evaluation of the approach shows that it localizes accurately – in over 93% of our test cases it could identify the faulty element. (4) The approach is fast – in experiments on real-world web applications, such as GMail and Craigslist, our approach detected and localized each fault in an average of 87 seconds.

The rest of this paper is organized as follows: In Section II we discuss several scenarios in which web developers need to debug presentation failure and highlight the limitations of existing work with respect to these scenarios. We provide a detailed explanation of our approach in Section III. In Section IV we present the results of our empirical evaluation. We discuss more broadly related work in Section V, then conclude and summarize in Section VI.

II. Motivating Scenarios

In this section we describe several different scenarios in which testers need to debug presentation failures and the limitations of existing techniques with respect to those scenarios. Recent research has focused on one such scenario, XBI. Presentation failures occur in this scenario when a web page is rendered inconsistently across different browsers. XBI can result in potentially serious usability and appearance failures. Proposed techniques [19], [7], [8] have made significant progress in detecting XBI, but are limited in their applicability to other types of presentation failures. The reason for this is that XBI techniques detect failures by comparing the underlying Document Object Model (DOM) of the web pages. As we explain below, such a technique is not applicable for scenarios where DOM (but not visual) changes are intended or where developers are trying to match a page against a mockup provided by a graphical designer.

The second such scenario is what we refer to as *regression debugging*. In this scenario, developers have modified the current version of the web application to correct a bug, introduce a new feature, or refactor the HTML code. For example, a developer may refactor a web page to transition it from using a table-based layout to one based on the use of the <div> tag. During this modification, developers may introduce a fault in the code that results in a presentation failure. Existing techniques, such as Cross-Browser Testing (XBT) [19], [7], [8], GUI differencing [26], automated oracle comparators [21], or tools based on `diff` may be of limited use in this scenario. The reason for this is that these techniques use a tree-based representation (e.g., DOM) to compare the versions of the faulty web page. If a faulty change is small and localized within the tree, it may be straightforward for these techniques to identify the fault. However, if the tree structure has changed significantly (as in the above refactoring example) then a comparison will most likely result in many spurious changes being identified as the fault. To verify this assertion, we conducted a small case study with the most recent implementation of XPERT [8], a well-known tool for performing XBT. In its current state of implementation that is publicly available, XPERT is only able to detect XBIs based on the functionality of the apps and layout/structure of the web page's appearance. Therefore, we ran XPERT on test cases that were handcrafted to have a combination of DOM and visual changes that resulted in layout and structural changes. The results demonstrated that with a change in the underlying DOM structure, the presentation failures were not detected, as no matching DOM node was found in the reference DOM tree. Hence, the changed DOM node was not analyzed, resulting in a false negative. Furthermore, these techniques assume that any difference between the tree-based representation implies a failure. This is not always true as there can be multiple ways to implement the same visual appearance using HTML and CSS properties.

The third scenario is *mockup-driven development* [17], [14], [18]. In this style of development, front-end developers use *mockups* – highly detailed renderings of the intended appearance of the web application – to guide their development of web application templates. The developers are generally expected to create "pixel perfect" matches of these mockups [2] using web development tools, such as Adobe Muse, Amaya, or Visual Studio. Back-end developers also make changes to these templates by adding dynamic content. Both front-end and back-end developers need to check that their respective changes are consistent with the mockup, and if not, identify the HTML elements that have caused the discrepancy. In this scenario, it is not possible to use any of the tree-based comparison techniques, as there does not yet exist a prior working version of the web page, only the graphical mockup. Using other techniques, such as Selenium [4], Cucumber [1], Crawljax [16], Sikuli [6], or graphical automated oracles [9], is not practical in this scenario for several reasons. First, these techniques require testers to exhaustively specify every correctness property to be checked, which may be very labor intensive. A new tool, "Fighting Layout Bugs" (FLB) [23], does eliminate the need to specify such correctness properties, but it can only detect general types of failures, such as overlapping text regions, and cannot detect application-specific failures. Second, the correctness properties are expressed in terms of HTML syntax, not the visual appearance of an HTML element. Therefore, these techniques may miss presentation failures, such as incorrect inheritance of an ancestor element's CSS properties.

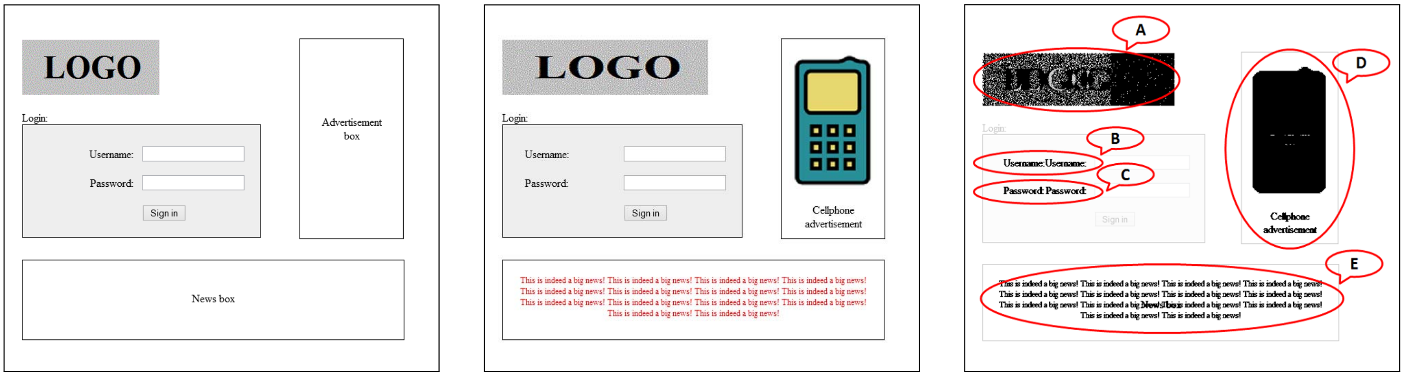
III. Approach

The goal of our approach is to automatically detect and localize presentation failures in web pages. To do this, our approach applies techniques from the field of computer vision to analyze the visual representation of a web page, identify presentation failures, and then determine which elements in the HTML source of the page could be responsible for the observed failures.

The approach takes three inputs. The first input is the web page (P) to be analyzed for presentation failures. The form of P is a URL that points to either a location on the network or filesystem where all HTML, CSS, JavaScript, and media files of P can be accessed. The second input is the oracle (O) that specifies the visual correctness properties of P. The form of O is an image that can be either a mockup or a screenshot of a previously correct version of P. The third input is a set of special regions (SR) defining areas of O that will contain dynamic text, ads, etc., which define dynamic regions common in modern web applications. Special regions provide a mechanism to allow developers to specify such regions that should be handled specially.

From a high-level, our approach can be described as having three phases. The first phase, detection, compares the visual representations of P and O to detect a set of differences in either the special regions or the regular page. The identified set of differences are then clustered into groups that are likely to represent different underlying faults in P. The second phase, localization, analyzes a rendering map of P to identify the set of HTML elements that define the pixels of each set of clustered differences. Finally, the third phase, result set processing, prioritizes the set of elements identified for each cluster and provides this as an output for the developer.

Figure 1 shows an example web application that we will use to illustrate our approach. Figure 1a shows the intended appearance (oracle) of the web page under test. This oracle could be the mockup used by the front-end developers or a



(a) Oracle (O)

(b) Rendered test page screenshot (P_v)

(c) Differences between O and P_v

Fig. 1: Illustrative example

screenshot of a previously correct version. A screenshot of the appearance of the application under development is shown in Figure 1b. As compared to the oracle, there are five visual differences: (1) the size of the logo image has changed, (2,3) the alignment of the "Username" and "Password" labels has shifted, and (4,5) the content of the news box and advertisement box has changed.

A. Phase 1: Detection

The first phase of the approach detects presentation failures by comparing the screenshot of *P*, as rendered in a browser, with its expected appearance, *O*. The approach captures a visual representation (P_v) of *P*. Then, the approach identifies the visual differences (DP) between O and P_v. A Δ parameter is used as a customizable tolerance level to indicate how closely O and P_v must match.

To capture the visual representation, P_v, the approach takes a screenshot of the browser window that is rendering *P*. Since this visual representation will be compared against the oracle, it is necessary to ensure that (1) the browser window size is similar to the oracle image's dimensions (i.e., height and width); (2) the amount of zoom in the browser window is approximately the same as that used while developing the oracle; and (3) the size of the browser viewport is set to ensure that page scrolling does not eliminate visible portions of the page. Note that the testing and oracle platforms do not need to be exactly the same as our comparison technique allows for various levels of tolerance and resizing (see below). The screenshot library of Selenium provides functionality to configure these visual aspects.

The approach compares O and P_v to find differences. In our prior work [15], we used a strict pixel-to-pixel equivalence comparison to identify differences. As we show in Section IV, this type of comparison is impractical for real-world mockups for two reasons. First, the oracle and screenshot may be developed on different platforms and small inconsequential variations may be introduced as a result of scaling or resizing images for comparison. Second, small differences may represent concessions to coding simplicity or be within a level of tolerance that the development team does not consider to be a presentation failure.

To address these limitations, our approach uses perceptual image differencing (PID), a computer vision based technique for image comparison [27]. PID uses computational models of the human visual system to compare two images. This allows the approach to compare the images based on an idea of "similarity" that corresponds to human's visual concept of similarity. PID models three features of the human visual system: (1) spatial sensitivity, (2) luminance sensitivity, and (3) color sensitivity to compare a given pair of pixels. The PID algorithm also accepts a threshold value Δ as a parameter, which is used to decide whether the images are below a threshold of perceptible difference, a field of view value in degrees F, which indicates how much of the observer's view the screen occupies, a luminance value L, which indicates brightness of the display the observer is seeing, and a color factor C, which indicates the sensitivity to colors. For space considerations, we omit the details of how the PID algorithm functions. The PID technique is particularly well-suited for our problem for two reasons. First, the three modeled features roughly account for the location (or size), contrast, and color of the HTML elements in the two pages, which together cover almost all possible visual rendering effects available via CSS or HTML. Second, the Δ, F, L, and C allow the difference detection to be scaled to reduce false positives (via Δ) and account for screenshot/oracle sizes that are either very small (e.g., smartphone) or large (e.g., desktop web browser).

The approach uses the PID algorithm to compare O and P_v at a tolerance level specified by Δ, F, L, and C. The result of this is a set *DP* that contains all pixels of the two images considered to be perceptually different. All pixels that are within a special regions area, as specified by *SR*, are removed from DP, as these pixels will be processed separately as explained in Section III-D.

Next, the approach identifies difference pixels that are likely to be caused by the same fault. Intuitively, these are difference pixels that are located close to each other, a relationship that can be found by clustering. Therefore, the approach clusters the difference pixels in *DP* and creates a map that consists of tuples with cluster id as the key and a set of difference pixels corresponding to that cluster as the value, ⟨cluster_id, {⟨x_0, y_0⟩, ⟨x_1, y_1⟩, ..., ⟨x_n, y_n⟩}⟩. For clustering, we use a popular density-based clustering algorithm, DBSCAN (Density Based Spatial Clustering of Applications with Noise) [11]. DBSCAN does not require a predefined number of clusters, but rather decides the number of clusters based on the density distribution of the given data points. This feature made DBSCAN more suitable for our approach than other popular clustering algorithms, such as K-means, that require the number of clusters as input, since the potential number of presentation failures existing in a page cannot be known in advance.

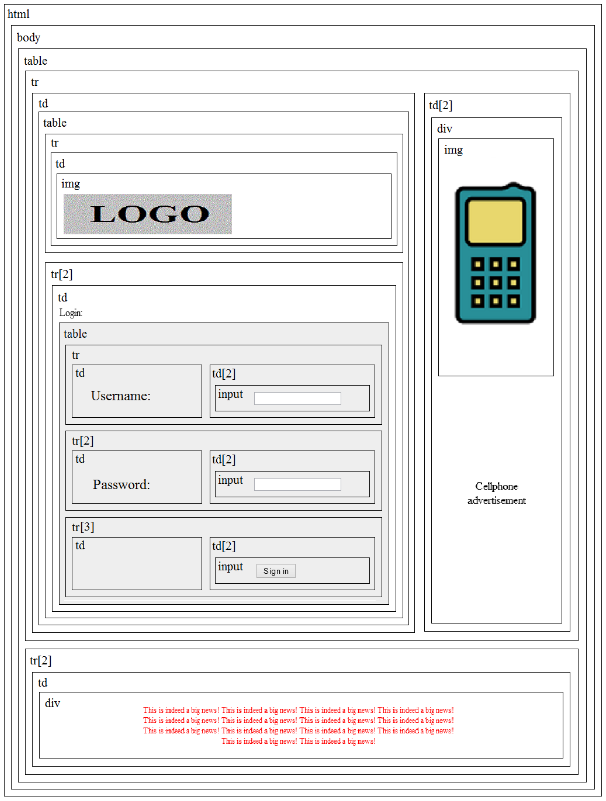
To illustrate the detection phase at a high level, consider the oracle shown in Figure 1a and the screenshot of the page in Figure 1b. The approach compares the two images and identifies the differences between them. Figure 1c shows the difference pixels as black dots. As noted earlier, the logo, text labels of the input boxes, advertisement, and dynamic text area differ from the oracle. Therefore Figure 1c shows difference pixels in the areas of these elements. After removing the difference pixels belonging to areas D and E, as they belong to special regions and will be handled separately (see Section III-D), the approach applies clustering and obtains three clusters shown as A, B, and C in Figure 1c.

B. Phase 2: Localization

The second phase of the approach identifies the set of HTML elements that are most likely to be the cause of the detected presentation failures. To do this, the approach builds a model of *P*, called an R-tree, that describes the pixel-level relationships among elements of an HTML page. For each pixel of the difference set, the approach uses the R-tree to identify the set of HTML elements whose visual representation includes that pixel. The union of all of the identified HTML elements for all difference pixels in the respective cluster is the set of potentially faulty HTML elements (*E*).

The first step is to build an R-tree model of *P* that will be used to map difference pixels to HTML elements. An R-tree is a height-balanced tree data structure that is widely used in the spatial database community to store multidimensional information. In our approach, we use the R-tree to store the bounding rectangle assigned to an element when it is rendered in the browser. The approach extracts the bounding rectangle for each element in *P* via browser-provided APIs. In the R-tree built by our approach, the leaves of an R-tree correspond to rectangles and non-leaf nodes correspond to the tuple ⟨I, child_pointer⟩, where *I* is the identifier for the minimum bounding rectangle that groups nearby rectangles, and *child_pointer* is the pointer to a lower node in the R-tree. The HTML elements corresponding to an $\langle x, y \rangle$ pixel can then be found by traversing the R-tree's edges to find the rectangles containing the pixel. Building and searching an R-tree are standard techniques and can be found in [12].

Other browser APIs, such as the HTML DOM, can be used to map pixels to HTML elements. However, the R-tree representation is more efficient. The reason for this is that the DOM tree models parent-child relationships based on syntax, not layout. Therefore, even when an element is found in the DOM tree that contains the pixel, there may be other elements elsewhere in the tree that also contain the difference pixel. This makes pixel mapping in the DOM an O(n) operation. In contrast, the R-tree search is O(log n) as it is height balanced.



This is indeed a big news! This is indeed a big news! This is indeed a big news! This is indeed a big news!
This is indeed a big news! This is indeed a big news! This is indeed a big news! This is indeed a big news!
This is indeed a big news! This is indeed a big news! This is indeed a big news! This is indeed a big news!
This is indeed a big news! This is indeed a big news!

Fig. 2: *P* with bounding rectangles of HTML elements

The different HTML elements in the example `login.html` are shown by their tag names and bounding rectangles in Figure 2. This figure shows how a browser defines the layout of a web page as a group of rectangles. The bounding rectangle information of the HTML elements is used to construct a page's R-tree, by grouping nearby rectangles together in a virtual minimum bounding rectangle. For example, the four HTML elements, <tr>, <td>, <td[2]>, and <input> that comprise the Username label and text box would be grouped together in one node bounded by a virtual rectangle.

After an R-tree has been constructed for *P*, the approach iterates over all the clusters and identifies the HTML elements that can be responsible for rendering the pixels contained in the cluster's difference set, $DP_{cluster}$. For each pixel $\langle x, y \rangle$ in each cluster's *DP*, the approach finds the containing HTML elements in the R-tree, and adds them to the set *E*. Note that for a given pixel, it is possible to have more than one responsible HTML element, since CSS allows layers of HTML elements with transparent or overlapping areas. Each HTML element is represented in *E* using its XPath identifier, which can uniquely identify an element in an HTML page.

After finding the potentially faulty elements corresponding to the difference pixels, the approach refines the set *E* to include additional likely faulty elements. This refinement is performed based on per-element heuristics. In general, these heuristics are used to analyze the elements present in *E* to check if neighboring (parent, children, or sibling) elements need to be added to the result set. These heuristics are applied to all elements in *E* and elements are added to *E* until there are no new elements to be added. An example of such a heuristic is when an element in *E* has the "hidden" property set. An element with the "hidden" property set would not have a visual area associated with it in the R-tree, so it would never be placed in *E* based on it containing difference pixels. If the "hidden" element's parent, children, or siblings are in *E*, then that makes the "hidden" element more suspicious, so it is added to the set. The set of heuristics was defined based on common patterns of false negatives (i.e., elements that were not placed into *E* but should have been). The addition of this heuristic-based refinement also increases the average size of the result set and introduces false positives. However, as can be seen by comparing the localization results in Section IV with those reported in our prior approach [15], the significantly improved accuracy is achieved with an almost non-existent change in terms of localization quality metrics.

To illustrate the localization phase, consider again the running example. The results from the prior detection phase contain three clusters, A, B and C, shown in Figure 1c. Using the R-tree built from the bounding rectangles information shown in Figure 2, the approach searches for the HTML elements that correspond to each of the pixels in $DP_{cluster}$. For example, $E_{clusterA}$ will contain the XPath of the Logo <img> tag. Similarly, the elements containing the pixels in clusters B and C are identified and added to their respective result set.

C. Phase 3: Result Set Processing

A presentation failure can impact the visual appearance of HTML elements containing, within, or adjacent to the faulty HTML element. For example, if an element increases in size, this can cause other elements to be repositioned and trigger a cascade of dependent presentation failures. Similarly, if a faulty element overlaps another element, then both the overlapping and overlapped element will differ from the oracle. Our approach adds elements, whether their presentation failure is independent or dependent on another element, to the set of potentially faulty HTML elements (*E*) for each cluster. This can be problematic for developers if the size of *E* requires them to inspect many HTML elements.

To address this potential problem, the third phase prioritizes *E*'s elements in order of likelihood to have caused the presentation failure and gives a ranked list, E_l, of potentially faulty elements for each cluster. To do this, our approach analyzes each HTML element $e \in E$ and assigns it a prioritization score. Then the elements of E are ranked according to their prioritization score, with a lower score indicating a higher likelihood of being a faulty element, and the sorted list is added as a value to a map with the cluster ID as the key. The map is then returned to the user. This rank function, which is shown as Equation (1), assigns the prioritization score as a function of four weighted heuristics.

1) Heuristic 1: Contained Elements ($\mathcal{C}$): When a presentation failure occurs in an element, the children of that element are likely to be impacted as well. The reason for this is that style properties of an HTML element are, by CSS rules, inherited from its parent. Therefore, if a parent element and all of its children are identified as potentially faulty, there is a higher likelihood that the faulty element is actually the parent element. Equation (1a) implements this heuristic for an element $e \in E$ by determining if e's parent and all siblings have also been reported as having a difference. If this is the case, then element e is assigned a *C* value of 1. This causes the parent of e to have a lower prioritization score than e and all of its siblings.

2) Heuristic 2: Overlapped Elements ($\mathcal{O}$): The second heuristic deals with the opposite scenario of the first. In cases where a child's visual appearance changes, these changes may cause it to overlap its parents visual area. This will cause the child, parent, and possibly some siblings to be reported as potentially faulty. For example, if an element has a larger than intended border, then this border will overlap with the element's parent's visual area. Therefore, if an element is potentially faulty and at least one, but not all of its children have a visual difference as well, then its more likely that the failure was caused by a presentation fault in a child. Equation (1b) implements this heuristic for an element $e \in E$ by checking if the number of e's children is greater than or equal to one, but less than the total of e's children. If this is satisfied, then e is assigned a value of 1 for *O*. This causes the child(ren) of e to have a lower prioritization score than e.

$$r(e) = w * C(e) + x * O(e) + y * D(e) + z * P(e) \tag{1}$$

$$\mathcal{C}(e) = \begin{cases} 1 & \text{if e.parent} \in \text{E} \wedge \text{e.allSiblings} \subset \text{E} \\ 0 & \text{otherwise} \end{cases} \tag{1a}$$

$$\mathcal{O}(e) = \begin{cases} 1 & \text{if } 1 \leq |\text{e.children} \cap \text{E}| < |\text{e.children}| \\ 0 & \text{otherwise} \end{cases} \tag{1b}$$

$$\mathcal{D}(e) = \begin{cases} 1 & \text{if CI(e) is a sub-image of O} \\ 0 & \text{otherwise} \end{cases} \tag{1c}$$

$$\mathcal{P}(e) = 1 - \frac{|\{\text{d} \mid \text{d} \in \text{D} \wedge \text{d within e.area}\}|}{\text{e.width} \times \text{e.height}} \tag{1d}$$

3) Heuristic 3: Cascading ($\mathcal{D}$): When a sizing related presentation failure occurs in an element, it generally affects the appearance of surrounding elements, leading to a set of dependent failures. This occurs because the position of HTML elements are often specified with relative layouts. When one moves, the others shift to accommodate its new position, causing a cascade of visual differences that are detected by our approach. Our insight is that prioritization can be performed by identifying visual differences that are strictly positional displacement (i.e., a move along the X or Y axis). Any element with only a positional displacement is more likely to have been moved by a faulty element than to be the faulty element. To identify positional displacement, our approach searches the oracle to see if a screenshot of e simply appears at another location in O. If this is the case, a value of 1 is assigned as the D value, meaning that e is more likely to have been identified as the result of a cascade than to be a faulty element. Equation (1c) implements this heuristic with CI referring to the cropped image that contains just the rendering of e.

4) Heuristic 4: Pixels Ratio ($\mathcal{P}$): When a presentation failure occurs in an element, it typically causes a significant portion of the element to change appearance. This causes a relatively high number of its corresponding pixels to be identified as difference pixels. Our fourth heuristic captures this intuition by assigning a lower prioritization score to elements that have a high percentage of their pixels reported in difference than those with a lower percentage. Equation (1d) calculates this pixel ratio for an element $e \in E$. The numerator is the number of pixels that are in the difference pixel set and within the rendering rectangle of e. This is divided by the total number of pixels in e's rendering rectangle, and then subtracted from one.

D. Special Regions Handling

In modern web applications, it is common to have parts of a web page that are dynamically defined. For example, part of a page could contain an advertisement, user account information, or text from a database. Even though the actual contents of these regions of the page are not known, designers and developers are often able to specify other correctness properties that should apply. For example, that all text within the region should have a certain color, size, or font face. Our approach provides a mechanism, called special regions, for denoting these areas and specifying the correctness properties that should apply to them.

The set of special regions is specified by the SR input to the approach. Each special region is represented as a tuple $\langle area, type, style \rangle$. $area$ specifies the location of the special region in terms of the x-y coordinates of its upper left hand corner and its width and height. Developers may provide these coordinates directly, or provide the XPath of an HTML element whose bounding rectangle will then be used to compute the $area$ coordinates. $type$ denotes the type of special region that is represented by the area. Our approach defines two types of special regions, Exclusion (SR_1) and Dynamic Text (SR_2). Together, these two types of special regions can be used to handle web pages with dynamic content, such as Amazon.com and CNN.com. We explain each of these in more detail below in Section III-D1 and Section III-D2. $style$ contains the set of visual properties that should apply to a region. Referring back to the example, there are two special regions that would be identified by the developers and included in the *SR* input to the approach. The first special region is $\langle D, \mathrm{SR}_1, \emptyset \rangle$, which corresponds to the advertisement box. Here SR_1 denotes that D is an Exclusion area and the empty set denotes that there are no style properties to verify. The second special region is $\langle$E, SR_2, {font-size: 12px, color: red, font-weight: bold}$\rangle$, which corresponds to the news box. Here SR_2 denotes a dynamic text region and the $style$ set contains the font properties that will be verified.

For each special region, the approach calls a predefined processing function that implements the difference semantics for a given special region type. A processing function takes two inputs; the web page, *P*, and the special region tuple, *sr*. The processing function then analyzes the area to see if there is a difference with respect to the region's type semantics, and if so, returns the set of difference pixels, DP_{sr}. Then, the approach again goes through phases 1 to 3 for finding faulty elements in the special regions, but starting with the set DP_{sr} and the R-tree that has already been created in the localization phase.

1) SR_1: Exclusion Regions: Exclusion Regions allow testers to specify regions of the web page for which no correctness properties will apply except size bounding. Generally, these regions are used for dynamic content for which a designer cannot assign correctness properties other than that the content should be bounded within the region's area, such as advertisements, banners, and media, where the content is provided at runtime by a third party or content that should be ignored, such as intentional changes to a web page between versions. The processing function associated with Exclusion Regions simply returns an empty difference pixel set. Referring back to the example, the dynamically loaded advertisement (D) corresponds to an Exclusion Region. Therefore the invocation of the special region's processing function returns the empty set and no difference pixels are added to DP_{sr}.

2) SR_2: Dynamic Text Regions: Dynamic Text Regions indicate areas of a web page whose content will be textual and for which the styling of the text is known but the exact text is unknown at design time. Examples of Dynamic Text Regions are parts of a web page provided by a database or by a call to a web service. Note that static text, such as titles and labels, does not need to be classified as a Dynamic Text Region, since the normal difference semantics would detect any presentation failures in static text that is present in the oracle.

A simple, but naive, way to check dynamic text regions is to obtain the style properties for the HTML elements that are to be checked, and match them directly with the expected style properties. However, this simple textual differencing may not give correct results, as there are several ways in which the appearance of an HTML element can be specified. For example, font size could specified as "10px" or "x-small," or a font property could be inherited from an ancestor.

To address this problem, we employ a technique that checks the actual visual appearance of the text. The basic intuition is that the approach first identifies all HTML elements within the special region's area. This is done using the same mechanism as in phase 1, detection, and phase 2, localization. Then the style properties to be enforced are injected into the identified HTML elements. The approach then compares the original page (P) and the page (P') with the injected style properties. If a difference arises because of the insertion of the correct style properties, then the approach infers that the original page had a different (incorrect) style for the text and the identified difference pixels are added to the set DP_{sr}.

Referring back to the example, the news box (E) is a Dynamic Text Region. The approach applies the styling, {font-size: 12px, color: red, font-weight: bold}, specified in the special region tuple, and assigns this to the HTML elements that comprise the area of the news box. The approach takes a screenshot of the original HTML page (P) under test and compares this against P'_v. Phases 1 is then run on the two versions to compare them and get the set DP_{sr}. Since the styling properties in our example are indeed different, the difference pixels associated with the news box area highlighted by area E are returned as the output of the processing function and added to DP_{sr}. Upon invocation of phases 1 through 3,

TABLE I: Subject applications

	App	URL	Size
Set 1	OPAL	http://www.opalhvac.com	83
	Crawller	http://www.crawller.com	266
	Inno crawll	http://inno.crawller.com	232
Set 2	Gmail	http://www.gmail.com	72
	USC CS Research	http://www.cs.usc.edu/research	322
	Craigslist	http://losangeles.craigslist.org	1100
	Virgin America	http://www.virginamerica.com	998
	Java Tutorial	http://docs.oracle.com/javase/tutorial/essential/io/summary.html	159

the faulty <div> element corresponding to the cluster E is then reported to the user.

IV. Evaluation

To evaluate our approach, we designed experiments to determine its accuracy, localization quality, and time needed to perform the analysis. The specific research questions we considered are:

RQ1: What is the accuracy of our approach for detecting and localizing presentation failures?

RQ2: What is the quality of the localization results?

RQ3: How long does it take to detect and localize presentation failures with our approach?

To address these research questions, we carried out a large scale empirical evaluation of our approach on a set of real-world web applications and also compared the results with the debugging performance of graduate-level software engineering students.

A. Implementation

We implemented our approach in a prototype tool, WebSee. The implementation of WebSee is in Java and leverages several third party libraries to implement some of the specialized, but standardized, algorithms. In the detection (Phase 1) module, WebSee leverages the Selenium WebDriver to take screenshots and the perceptual image differencing library, "pdiff", to compare images and calculate differences. The clustering algorithm, DBSCAN, as implemented in the Apache Commons Math3 library, is used to cluster the difference pixels. In the localization (Phase 2) module, we leverage the Java Spatial Index library's implementation of the R-trees and the Selenium WebDriver to extract bounding rectangle information. For the prioritization (Phase 3) module, the sub image searching capability for the cascading heuristic is supplied by OpenCV.

B. Subject Applications

For our experiments, we utilize the eight subject applications shown in Table I. We chose these web pages because they represent a mix of different implementation technologies and layouts that are widely used across all web applications. In particular, we chose our set of test subjects to include web pages that were defined by statically generated HTML, CSS, and JavaScript and pages defined by dynamically generated HTML. The size of each page (in terms of the total number of HTML elements) is also shown for each subject.

C. Empirical Evaluation

In our first experiment, we measured WebSee's accuracy, answer quality, and analysis time for presentation failures in the subject application. To provide a measurement reference point, we also asked graduate-level software engineering students to detect and localize a subset of these test cases and compared their performance against WebSee. To create test cases for the evaluation, we used a random seeding based approach that inserted presentation failures into the subject applications. We used this approach because we were not able to obtain a sufficiently large enough set of mockups of real-world web applications in order to ensure that our approach could be validated against a wide range of presentation failures. (See Section IV-D for evaluation on the limited set of mockups.)

To seed the presentation failures and generate mockups, we used the following process for each subject page p. (1) download from the web all files required to display p; (2) take an image capture of p to serve as the oracle O; and (3) create a set P' that contains variants of p, each variant created by randomly seeding a unique presentation fault. To identify the types of faults to seed, we first manually analyzed the W3C HTML and CSS specifications to identify *visual properties* — HTML attributes or CSS properties that could change the visual appearance of an element. We seeded faults by changing the original value of each unique visual property present in p. We eliminated any variant of p with a seeded presentation fault that did not actually produce a presentation failure. To identify these, we computed the set of pixel differences between the rendering of p and O, and only included the variant if this set's size was non-zero. The visual impact of a seeded fault varied, making the test cases vary in complexity for WebSee. In some cases, the seeded fault caused almost all of a page to be shown as having a pixel-level difference. For example, changing the value of the *padding* CSS property in the <body> tag. In other cases, the seeded fault impacted only a small area (e.g., changing the text color of a <span> tag.)

Each test for WebSee included the oracle O and a $p' \in P'$ as inputs. The expected output was the HTML element in which we had modified a visual property. The number of test cases generated for each application is shown in Table II in the column labeled "#T". We ran WebSee on each of these test cases and manually verified the reported results. The test machine was a Windows 8.1 platform with 8GB RAM and a 3rd Generation Intel Core i7-3770 processor.

We considered two types of accuracy: detection and localization. Detection accuracy was computed as a sanity check to measure the correctness of the PID technique, as we had only included test cases for which the rendering contained a perceptible difference. Thus, as expected, WebSee was able to detect the presentation failures in all of the test cases. For localization accuracy, we calculated the percentage of test cases in which the expected faulty element was reported in the result set. These results are shown under the column labeled "Accuracy" in Table II. Localization accuracy was high – WebSee was able to identify a result set that contained the faulty element for 93% of the test cases. We investigated the results to determine the reason why some elements did not appear in the localization set. We found that the dominant reason was that some seeded faults only changed the appearance of *other* elements. They did not change the appearance of the element into which they

TABLE II: RQ1: Effectiveness of the approach

		Accuracy	Quality			Timing			
App	#T	Localize	Median Result	Median	Median	P1	P2	P3	Total
		(%)	Set Size	Size	Distance	(s)	(s)	(s)	Time (s)
Gmail	52	92	12 (16%)	2	4	2.7	2.5	1.4	7.2
USC CS Research	59	92	17 (5%)	5	6	48.2	12.9	77.0	149.2
Craigslist	53	90	32 (3%)	7	3	5.3	43.1	32.1	83.6
Virgin America	39	97	49 (5%)	8	12	22.6	38.5	103.0	180.9
Java Tutorial	50	94	8 (5%)	2	5	4.9	5.3	3.6	14.5

were seeded. For example, a seeded fault that set the CSS position property to *fixed* caused the surrounding elements to be re-positioned. This made the HTML elements surrounding the seeded element, but not the seeded element, appear as a difference.

To measure the quality of the responses provided by the localization process we calculated three metrics, "Size," "Rank", and "Distance," which are shown under the column labeled "Quality" in Table II. The first metric, "Size," represents the median size of the result set returned by WebSee. In parenthesis, this number is shown as a percentage of the subject's total number of HTML elements. Across all apps, the average median result set size was 23 elements, which was, on average, about 10% of each app's total element count. The second metric, "Rank," represents the average rank of the actually faulty element within the result set. In cases where the faulty element was not present, we counted the faulty element rank as the size of the result set. Overall, the average median rank was 4.8, which means that the tester, on average, would have to examine 2% of the elements in a page before encountering the faulty element. The third metric, "Distance," represents a measure of how "close" the elements in the result set are to the actual faulty element. We calculated this metric for only the result sets where the faulty element was not present. The intuition of this metric is that reporting an element in the close vicinity of the faulty element may still be useful for the developer in debugging. We calculated distance based on the DOM tree structure of the web page. For example, if a reported element is at a distance of one from the faulty element, then the tester will have to additionally inspect the parent and/or children of the reported element to find the fault. We investigated the results to determine the reason the fault elements were not receiving a top rank in the results set. We found that the dominant cause of this was faults that caused a change in the size or position of the element. This type of fault had a cascading effect on other elements in the page. For example, increasing the padding around a <div> element would grow the parent elements and could possibly move sibling elements.

We measured WebSee's running time for each of the five phases. The average for each phase is shown in seconds in the columns labeled "P1" – "P3" of Table II, and the average total time is shown for the entire process is shown as well. At a high-level, WebSee's runtime ranged from 7 seconds to about 3 minutes, with an average of 87 seconds. Within each test case, Phases 2 and 3 dominated the running time of the analysis. We determined that most of Phase 2's time was consumed by process of building the R-tree. In particular, it was necessary to access the Selenium WebDriver interface repeatedly and this was a comparatively slow API. Phase 3's time was consumed by the cascading analysis involving sub-image matching.

We also evaluated the performance of WebSee in the context of a user study. Our users were students drawn from a graduate level class on web application software testing and analysis at the University of Southern California. All of the students were experienced web developers and had received training in using the Firebug tool to debug web application presentation failures. Each student received a unique set of nine test cases with manually seeded perceptually visible faults plus a test case that did not contain a presentation failure. Each test case was comprised of a test page and oracle image. We asked the students to visually inspect the rendering of each page in a browser to determine if there was a presentation failure, and if so, use Firebug to help them identify the faulty element. The students had 70 minutes to carry out the task. On average, the students were able to correctly determine if there was a visual difference in 76% of the cases and identify the correct faulty HTML element for 36% of the test cases. Despite a request to list all potentially faulty elements, all students only provided one element, so it was not possible to calculate the additional quality metrics for their localization answers.

Overall, the results of this experiment were very positive. The results showed that our approach was able to detect all of the presentation failures versus developers detection rate of 76%. Localization accuracy was also much higher for our approach (93%) as compared to the developers (36%). It was not possible to compare answer quality; however, our approach required developers to inspect a relatively small number of the total elements and the set sizes with the highest modalities were of size one and two, which indicates that the approach was generally returning very small result sets. Due to constraints of the user study setup, we could not compare runtime. However, by itself, 87 seconds represents a small amount of time. In comparison, none of the users in the study finished within the allotted 70 minutes, indicating, at best, an average test case time of seven minutes.

D. Case Study with Real Mockups

A potential threat to the validity of our results is that our mockups were artificially generated instead of from actual software engineering projects. The use of real mockups may result in different levels of accuracy and quality of result sets because the real mockups may not reflect the actual implemented design as faithfully as the simulated mockups (i.e., there may be observed differences for reasons other than the seeded fault). This can occur for several reasons: (1) variations may be introduced due to differences between the oracle and testing platform; (2) some differences may be too minor to be considered presentation failures; or (3) there may be design changes made due to implementation constraints or as concessions to coding simplicity. To improve the validity of the results from our first experiment, we obtained three real-world mockups (Set 1 shown in Table I) from an industrial partner and performed a case study to detect and localize differences between the mockups and the deployed versions of the web pages.

We performed our case study as follows. First we analyzed the pages to identify visual differences. Second, we classified the differences in one of the three categories listed above or as special regions, which were marked with placeholder text or images. Note that we did not have access to the project managers to determine if the differences we placed into category (3) were real presentation failures or intended deviations. All of the case study subjects had multiple differences in each of the four categories. Third, we provided the mockup, deployed web page, and special regions as inputs to WebSee. We ran WebSee multiple times adjusting the different configurable parameters and tolerance levels for the PID algorithm and observing the set of reported faulty elements.

The results of our case study were informative. Our first finding was that the use of the PID algorithm with its different configurable parameters and tolerance level significantly improved the accuracy of the approach versus our prior approach, which required a pixel-perfect match [15]. To simulate the pixel-perfect match, we set the PID's Δ to zero, F value to 89.9, L to 800, and C to 1. For several of the subjects, this resulted in over 84% of the page being reported as a difference [3]. In contrast, by taking advantage of the PID's customizability, we were able to find a setting (Δ of 10%, F value of 27, L of 20, and C of 0) at which all of the differences in categories (1) and (2) were not counted as failures, but those in category (3) were, indicating that detection could be done with high accuracy. Our second finding was that WebSee was able to return sets that included all of the expected HTML elements for the detected failures. It was not possible to use the metrics from experiment one to rate the quality of the answers, since there were multiple failures in the set. However, we found that WebSee returned what we considered to be useful results. For each of the subject applications, 45% of the faulty elements were listed within the top five and 70% were within the top ten ranked elements. The analysis time for the real and artificial mockups was similar. Overall, we feel that the results of the second experiment confirm that WebSee can perform accurate and useful detection and localization for real mockups as well.

E. Threats to Validity

In our studies, we used externally developed commercial web pages that covered a wide range of modern design styles and frameworks. We also systematically tested the ability of our approach to detect a wide variety of presentation faults by using the W3C specification to identify all possible visual properties. In Experiment 1, to eliminate the bias in the seeding of faults, the process was randomized via automation and an objective completeness criteria of seeding for each applicable visual property was used. All results were checked manually.

V. Related Work

Preliminary work by the authors appeared as a new ideas shortpaper [15]. In that paper, the authors introduced the idea of automatic detection and localization of presentation failures using image processing techniques. The paper performed a small case study on a small set of test cases to show the viability of the approach. This paper has several significant differences: (1) we replaced the pixel-to-pixel image comparison used in the detection phase by a more sophisticated computer vision based technique, PID; (2) we introduced handling of dynamic portions of a web page with special regions processing; (3) we introduced handling of multiple presentation failures with the notion of clustering; (4) we refined the set of potentially faulty elements generated by the localization to make it more comprehensive, resulting in improved localization accuracy; (5) we prioritized the result set produced by the localization phase with the most likely faulty elements placed at the top to offer ease of use to the developers; and (6) we performed an extensive empirical evaluation on a set of real-world web applications.

Work by Roy Choudhary and colleagues [19], [7], [8] in the field of XBT compares the rendering of a reference version of a web page in one browser against its rendering in another browser to detect XBI. As discussed in Section II, this approach is not applicable for mockup driven development as an existing golden version of the page is not available. Use in regression debugging is also limited to scenarios where the DOM had not changed significantly and matching elements could still be matched using probabilistic techniques. Their techniques use color histograms for visual comparison of certain elements instead of the perceptual image differencing employed by our approach.

Browser plug-ins, such as "PerfectPixel" for Chrome and "Pixel Perfect" for Firefox help developers to detect pixel-level differences with an image based oracle. They overlay a semi-transparent version of the oracle over the HTML page under test, enabling developers to do a per pixel comparison to detect presentation failures. However, they require the developer to manually locate the faulty elements. In contrast, our approach is fully automated for detection and localization.

Sikuli [6] is an automation framework based on computer vision techniques that uses sub-image searching to identify and manipulate GUI controls in a web page. Although not intended for verification, one could provide a set of screenshots of each GUI control and use Sikuli to ensure that they match (i.e., there are no presentation failures.) However, since Sikuli uses a sub-image based search of the page, it could match the provided screenshots against any portion of the page, not necessarily the intended region. This means it would be ineffective if there were visually identical elements in the page. Furthermore, Sikuli only provides an element after a positive match; therefore when there is a failure, no match will be made and no element(s) will be provided to the testers to help with localization.

Memon and colleagues [22] have done extensive work in the area of model-based automated GUI testing. These techniques differ from our approach in that they are not focused on testing the appearance of the user interface, but instead focus on testing the behavior of the system based on event sequences triggered from the user interfaces. Another work by Eaton and Memon [10] in the field of web applications focuses on reporting HTML tags present in the test web page that are not supported by a specific browser. This can detect presentation failures caused by unsupported tags, however it cannot detect failures related to application specific appearance properties.

Another group of techniques validate HTML for syntax [20] by checking for malformed HTML code that can cause presentation failures if the rendering browser does not handle them properly. However, these techniques can only detect presentation failures related to HTML syntax errors and not failures related to application specific appearance properties.

Recently, techniques to test JavaScript [5] have been proposed. These techniques deal with specific components of the client side and as such are not meant for detecting presentation failures in a web application. Another technique based on impact analysis of CSS changes across a web site [13] notifies the developer if changes made in a CSS file are introducing new presentation failures in other web pages of the web site. However, this technique relies on the availability of the golden version of all the web pages in the web site and is not able to handle presentation failures caused by changes in the DOM structure of the web page.

Wang and colleagues [25] propose a technique to automatically perform presentation changes in dynamic web applications. Their technique facilitates automatic changes to the server side code generating web pages based on a given presentation change. Though this produces a new looking web page, its appearance is not verified for correctness.

VI. Conclusion

In this paper we introduced a new technique for detecting and localizing presentation failures in web applications. Our approach uses computer vision based techniques to compare a web page rendered in a browser with its oracle and identify difference pixels. To handle the dynamic nature of modern web applications, our approach allows developers to specify regions that will contain dynamic text or images that should be specially handled. The approach builds a rendering map of the page and uses the difference pixels to identify and rank a set of likely faulty HTML elements. In the evaluation our approach detected 100% of the presentation failures and was able to locate the faulty element in over 93% of the cases. Overall, these are strong results and indicate that our approach is able to assist developers by automating the detection and localization of presentation failures.

References

[1] Cucumber. http://cukes.info/.

[2] Front-end Developers Job Postings. http://www-scf.usc.edu/~spmahaja/front-end-job-postings/.

[3] Real Mockups Experiment. http://www-scf.usc.edu/~spmahaja/real-mockups-experiment/.

[4] Selenium. http://docs.seleniumhq.org/.

[5] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*.

[6] T.-H. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

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

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

[9] M. E. Delamaro, F. de Lourdes dos Santos Nunes, and R. A. P. de Oliveira. Using concepts of content-based image retrieval to implement graphical testing oracles. *Softw. Test. Verif. Reliab.*, 23:171–198, 2013.

[10] C. Eaton and A. M. Memon. An Empirical Approach to Testing Web Applications Across Diverse Client Platform Configurations. *International Journal on Web Engineering and Technology (IJWET), Special Issue on Empirical Studies in Web Engineering*, 3(3):227–253, 2007.

[11] M. Ester, H. peter Kriegel, J. S, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. 1996.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[13] H.-S. Liang, K.-H. Kuo, P.-W. Lee, Y.-C. Chan, Y.-C. Lin, and M. Y. Chen. SeeSS: Seeing What I Broke – Visualizing Change Impact of Cascading Style Sheets (Css). In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*.

[14] P. J. Lynch and S. Horton. *Web Style Guide, 3rd Edition: Basic Design Principles for Creating Web Sites*. Yale University Press, New Haven, CT, USA, 3rd edition, 2009.

[15] S. Mahajan and W. G. J. Halfond. Finding html presentation failures using image comparison techniques. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE) – New Ideas track*, September 2014.

[16] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*.

[17] M. W. Newman and J. A. Landay. Sitemaps, Storyboards, and Specifications: A Sketch of Web Site Design Practice. In *Proceedings of the 3rd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*.

[18] F. K. Ozenc, M. Kim, J. Zimmerman, S. Oney, and B. Myers. How to Support Designers in Getting Hold of the Immaterial Material of Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

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

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

[21] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In the Intl. Symp. on Software Reliability Engineering, 2007.

[22] J. Strecker and A. M. Memon. Testing Graphical User Interfaces. In *Encyclopedia of Information Science and Technology, Second ed.* IGI Global, 2009.

[23] M. Tamm. Fighting layout bugs. https://code.google.com/p/fighting-layout-bugs/, October 2009.

[24] A. N. Tuch, E. E. Presslaber, M. StöCklin, K. Opwis, and J. A. Bargas-Avila. The Role of Visual Complexity and Prototypicality Regarding First Impression of Websites: Working Towards Understanding Aesthetic Judgments. *Int. J. Hum.-Comput. Stud.*, 70(11), Nov. 2012.

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

[26] Q. Xie, M. Grechanik, C. Fu, and C. M. Cumby. Guide: A GUI differentiator. In *ICSM*, pages 395–396, 2009.

[27] H. Yee, S. Pattanaik, and D. P. Greenberg. Spatiotemporal Sensitivity and Visual Attention for Efficient Rendering of Dynamic Environments. *ACM Trans. Graph.*, 20(1), Jan. 2001.