Using Visual Symptoms for Debugging Presentation Failures in Web Applications

Sonal Mahajan, Bailan Li, Pooyan Behnamghader, William G. J. Halfond
University of Southern California
Los Angeles, California, USA
{spmahaja, bailanli, pbehnamg, halfond}@usc.edu

***Abstract*—Presentation failures in a website can undermine its success by giving users a negative perception of the trustworthiness of the site and the quality of the services it delivers. Unfortunately, existing techniques for debugging presentation failures do not provide developers with automated and broadly applicable solutions for finding the site's faulty HTML elements and CSS properties. To address this limitation, we propose a novel automated approach for debugging web sites that is based on image processing and probabilistic techniques. Our approach first builds a model that links observable changes in the web site's appearance to faulty elements and styling properties. Then using this model, our approach predicts the elements and styling properties most likely to cause the observed failure for the page under test and reports these to the developer. In evaluation, our approach was more accurate and faster than prior techniques for identifying faulty elements in a website.**

I. Introduction

Ensuring the correct appearance of a website is important for its success. Studies have shown that the aesthetics of a website can significantly impact a user's overall evaluation of a website [36]; particularly, impressions of trustworthiness and usability [18]. *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.

The appearance of modern web applications is defined by a complex mixture of HTML, JavaScript, and Cascading Style Sheets (CSS). This complexity makes it difficult to identify the faulty HTML elements and CSS properties that have caused a presentation failure. For example, typical CSS features, such as floating elements, overlays, and dynamic sizing, make it challenging to debug presentation failures, since elements without any apparent HTML structural relationship can affect the way other elements appear. Traditionally, debugging presentation failures has been a manual effort. Developers use tools, such as Firebug [2], to click on elements in a web page and identify the CSS properties in effect for that element and the source HTML elements of those properties. Unfortunately, this is strictly a manual process, guided by a developer's intuition and experience. Therefore, the accuracy and efficiency of this process can vary significantly by developer.

The research and development communities have proposed techniques for various parts of the process of debugging presentation failures. However, these techniques are generally limited in their applicability. For example, techniques for detecting Cross Browser Issues (XBI) are not applicable to other types of presentation failures (See Section II). Other techniques allow for testing websites' user interfaces, such as Cucumber [1], Sikuli [41], Crawljax [25], Selenium [8], graphical automated oracles [11], and Cornipickle [15], but still require developers to manually enumerate and specify test cases for each correctness property to be tested, and build a debugging process around the running of the test cases in order to accurately identify the faulty element. Finally, techniques such as Fighting Layout Bugs [35] only allow developers to check for application independent failures, such as overlapping text and poorly contrasting colors.

To address these limitations, we propose a new approach to debugging presentation failures. Our technique works by analyzing visual symptoms of a web page using probabilistic techniques. A *visual symptom* is an observable and quantifiable feature of the web page's appearance. Our first key insight is that faulty elements in a page can be automatically identified with a high degree of confidence by observing visual symptoms of the rendered page and using probabilistic correlations to identify the elements most likely to be responsible for the observed failure. Our second key insight is that the probabilistic models can be easily generated from the page under test itself without need for historical or externally provided data samples. Using these insights, our technique analyzes the visual appearance of a page under test and reports a ranked list of faulty HTML element/CSS pairs to developers. In the evaluation, we compared our approach against other techniques, and found that it was able to generate significantly more accurate results in terms of recall and ranking of the faulty elements. It was also fast, able to generate these results for developers in an average of 17 seconds per test case. Overall, these results are very promising and indicate that our approach can be a viable technique for helping developers to debug their web pages.

The rest of this paper is organized as follows. We begin in Section II by presenting background information on common scenarios in which developers must perform presentation failure debugging and motivate our approach by describing some of the limitations of related work. In Section III we describe our approach for debugging presentation failures. Section IV presents the evaluation results. We discuss related work in Section V and conclude in Section VI.

II. Background and Motivation

When a presentation failure occurs, developers must debug their web applications in order to identify the responsible fault. In modern web applications, the logical structure of a web page is defined by HTML tags, and the tags' appearance is defined by CSS properties or HTML attributes that control how each HTML element is rendered in the browser. Therefore, the responsible fault in a page under test (PUT) is actually an HTML tag and either a CSS property or HTML attribute that is set to an incorrect value. We call this combination the *root cause* of the presentation failure. We define the root cause as a tuple $\langle e, p \rangle$ where e is an HTML element in the PUT and p is an HTML attribute or CSS property, whose improperly set value has caused the presentation failure. Although p could be any property or attribute, in practice, only a subset of those actually influence rendering in any way. Through manual analysis of the W3C HTML and CSS specifications, we have identified this subset, which we refer to as an element's *visual properties*. Henceforth, we are only referring to these visual properties when discussing p. Note that in cases where the PUT is dynamically generated, developers can use the root cause to find the responsible JavaScript or server-side code.

There are two main scenarios in which developers may need to debug presentation failures. The first scenario is *Cross Browser Issues (XBI)*, which occurs when a web page is rendered inconsistently across different browsers. XBIs can result in potentially serious usability and appearance failures. Proposed techniques [30], [9], [10] can detect XBIs and assist developers in identifying their root cause. However, the applicability of these techniques is limited with respect to the second scenario described below.

The second scenario is *regression debugging*. Developers often perform maintenance on their web pages in order to introduce new features, correct a bug, or refactor the HTML structure. For example, a refactoring may change a page from a table based layout to one based on <div> tags. During this process, the developers may inadvertently introduce a fault that changes the appearance in an unintended manner and must debug to find the root cause. Existing techniques, such as Cross Browser Testing (XBT) [30], [9], [10], GUI differencing [38], [14], [13], automated oracle comparators [34], [33], or tools based on `diff` may be of limited use in this scenario. The reason for this is that these approaches compare the Document Object Model (DOM) or other tree-based representations of the user interfaces. Small faults that do not change the tree structure can be found by these techniques, but as we show in the evaluation (Section IV), more significant refactorings or changes will cause these techniques to detect a large number of false positive root causes.

As a result of these limitations, automated support for regression debugging is limited. Therefore, developers employ a primarily manual process for detecting failures and then identifying the root cause. A typical process is that once a presentation failure has been detected, developers use tools, such as Firebug [2], to systematically explore the user interface and find the root cause. However, this is challenging for developers to do accurately. The results of a prior user study showed that given a set of presentation failures, developers were only able to correctly identify the faulty root cause in 36% of the test cases [21]. A similar result was obtained in a user study conducted with Accenture employees [14]. This is not surprising as the complexity of modern web applications means that the relationship between an observed failure and its underlying root cause is not always straightforward or can be obfuscated by the complexity of CSS and HTML code. As we show in the evaluation, our automated technique can provide significant support for developers in terms of identifying an accurate set of likely root causes for a presentation failure.

III. Our Approach

The goal of our approach is to automatically identify the root cause of a presentation failure observed in a PUT. Our approach is analogous to diagnosing a sick patient — we observe "symptoms" that manifest themselves in the visual appearance of a web page and use these to probabilistically infer the underlying root cause in the PUT. Here, a symptom is an observable and quantifiable feature of the PUT's appearance, such as color usage frequency, or text size.

Formally, we define a *visual symptom* as a predicate that can be either true or false in describing a visual difference between the PUT and the oracle. For example, consider a presentation failure where a login button is expected to be red (as specified in the oracle,) but is green in the PUT. In this case, a visual symptom exists as there is a difference between the set of colors present in the oracle and the set of colors present in the rendering of the PUT. This visual symptom would likely be due to the root cause `<button, background-color>`, which controls the color of the button.

The three inputs to our approach are a PUT, an appearance oracle that shows the PUT's correct appearance, and a list of regions of the page to be excluded from the analysis. The form of the PUT is a URL that points to either a location on the network or filesystem where all HTML, CSS, JavaScript, and media files of the PUT can be accessed. The source of the appearance oracle is a screenshot of the previously correct version. A region may be excluded from analysis by specifying its rectangular area location (x-y coordinates of the upper left and bottom right corners) or XPath of the HTML element identifying the region. Exclusion regions allow our approach to handle web pages with dynamic content (e.g., advertisements) that can change every time a page is rendered. These constantly updating parts of the page can cause a PUT to look different than its appearance oracle, which would trigger a false positive detection.

Our approach starts by comparing the visual representations of the PUT and the appearance oracle to detect visual differences. If a visual difference (i.e., presentation failure) is found in the areas other than the listed exclusion regions, the PUT and appearance oracle are analyzed to identify visual symptoms. These visual symptoms then serve as the input to a probabilistic model that maps visual symptoms to likely

root causes. The approach ranks potential root causes by the probabilities derived from the model and returns these to the developer for further investigation.

In the remainder of this section, we provide more details about the approach. In Section III-A we describe the process of detecting presentation failures in a PUT. In Section III-B we present the probabilistic model in more depth, including the formal model definition and how the model is built. We describe the usage of the model in Section III-C. Finally, we define each of the visual symptoms used by our approach in Section III-D.

A. Detecting Presentation Failures

Our approach uses our prior work, WebSee [21], to detect presentation failures in a PUT. WebSee compares the browser rendered screenshot of the PUT with its appearance oracle using Perceptual Image Differencing (PID) [40], a computer vision technique, to find visual differences. PID uses computational models of the human visual system to compare two images, enabling the comparison to be based on a human-like notion of similarity. WebSee also filters out all visual differences that occur in the list of exclusion regions.

B. Building the Probabilistic Model

If a presentation failure is detected, the probabilistic model is built to map visual symptoms to root causes for the given PUT. The model that we use is based on conditional probability, $P(r|S)$. The conditional probability represents the probability that a potential root cause r is faulty given the observed set of symptoms S. We build the model by using Bayes' theorem to decompose the original conditional probability into probabilities that can be easily calculated from prior knowledge. Traditionally, accurately building such a model is a challenging part of a probabilistic based approach. However, two key insights allow us to more easily and effectively build such a model. First, since we are using the probability generated by Bayes' theorem to compare root causes, we can simplify the decomposition in such a way as to minimize the amount of information that needs to be obtained from the data samples. Second, we can obtain this information via a process of injecting faults into the PUT and using the observed visual symptoms to build the model. In the rest of this section, we explain the process for generating the data samples in more detail and how we simplify the decomposition of the conditional probability.

1) Generate Data for Understanding Presentation Failures: Our approach builds the probabilistic model by observing data samples and learning correlations between visual symptoms and root causes for the PUT. A key challenge for our approach is how to obtain useful data samples. One seemingly obvious source is to use a pool of known presentation failures; however, the way in which symptoms manifest themselves is highly dependent on a page's layout and structure, so data is unlikely to generalize across dissimilar pages. To address this generalizability problem, historical data for the page or similar pages on the website could be used; however, unless the page is very mature, there is not likely to be enough data to effectively build the model. Our approach is to build the model using data generated by injecting faults into the PUT and observing the impact of the seeded fault on the symptoms.

This model building process provides several benefits for our approach. First, the process can be completely automated, which eliminates the typical requirement that data samples must be painstakingly extracted from a bug tracking system and labeled. Second, the approach does not have a dependency on the presence of an external data source, since the necessary data can be generated from the PUT itself. Finally, the quality of the data samples is consistent across PUTs, since its generation and completeness are under the control of the approach.

To generate the data samples, our approach first analyzes the PUT to identify all of its potential root causes. For each of these root causes, the approach systematically changes the visual property based on its type. We categorized visual properties into three types: predefined, color and numeric. For predefined properties, we enumerate over all predefined values. For example, we set the value of the CSS property `font-style` to *normal*, *italic*, and *oblique*. For color and numeric properties, since their value range is large, we choose several typical values instead of trying all possible values. For example, we set the CSS property `color` to *red*, *green*, and *blue*. After seeding the fault, the page is rendered and its appearance is compared against the original PUT to see if there was a visible change. This can be done automatically by comparing the images' pixels. If there is a difference, then the approach adds a tuple $\langle p', r \rangle$ to the data sample set, where p' is the version of the PUT with a seeded fault in root cause r.

Note that our approach uses the PUT to generate the data samples and not the original version of the web page. This is done because in regression debugging, the structure of the web page has changed, which means that the set of root causes has changed to those present in the PUT. Therefore, by using the original version of the web page, it would not be possible to build a model for all of the root causes in the PUT. Although using the PUT introduces noise into the data samples, the overall accuracy of the results, as shown in the evaluation, remains high.

2) Decomposition of the Probabilistic Model: As mentioned above, the probabilistic model is based on conditional probability. Since we cannot calculate $P(r|S)$ directly, we use Bayes' theorem to decompose the conditional probability to the form shown in Equation (1). We now explain how our approach can calculate the components $P(S|r)$, $P(r)$, and $P(S)$ from the data samples.

$$P(r|S) = \frac{P(S|r)P(r)}{P(S)} \tag{1}$$

$\mathbf{P(S|r)}$ represents the probability of the status of visual symptoms S given the root cause r being the faulty root cause. We further expand this joint probability of the status of all visual symptoms to the product of the probability of the status of every single visual symptom $s \in S$, as shown

in Equation (2). This expansion assumes that symptoms are conditionally independent given the root cause. Although this assumption does not always hold for all visual symptoms, it offers our approach several benefits that outweigh the loss in accuracy. First, it is easier to calculate and multiply the independent probabilities than to calculate the joint probability. Second, the model generation task can be easily parallelized, since each independent probability can be calculated without knowing the results of the other probabilities. Although this assumption results in lower fidelity of the visual symptom model, related work has shown that this assumption still allows probabilistic models to perform well enough in most real-world applications [42]. This observation is supported by the high accuracy results shown in the evaluation.

$$P(S|r) = \prod_{s \in S} P(s|r) \tag{2}$$

Based on this assumption, the model can be simplified and we can measure each $P(s|r)$ in the data samples. We calculate the individual probabilities, $P(s|r)$ by iterating over the data samples and observing the visual symptoms that occur for a seeded root cause. For each page in the data samples, we determine whether each visual symptom is true or false for the page. Note that regions of the page that correspond to the exclusion regions are not included for purposes of determining if the visual symptom is true or false. If a visual symptom is true, we increase a counter that relates the occurrence of each visual symptom with the root cause into which the fault was seeded. After analyzing each of the pages in the data samples, we count the occurrence of each visual symptom and divide the number of occurrences by the number of data samples to get $P(s|r)$. Finally, we multiply the probabilities of $P(s|r)$ for each visual symptom to get $P(S|r)$.

$\mathbf{P}(\mathbf{r})$ represents the probability of the root cause r being the faulty root cause of a presentation failure. It shows the relative probability that a given r is the root cause compared to all other possible root causes in the PUT. The probability of a root cause is itself based on the probability that its constituent element e and visual property p are themselves faulty (i.e., $P(r) = P(e)P(p)$). In our approach, we assume that developers cause faults in HTML elements with a uniform probability. Therefore, $P(e)$ is one divided by the total number of HTML tags in the PUT. Assuming this value is one essentially gives our approach no benefit from any kind of historical developer patterns. It is important to note that if we had a more precise model of the distribution of faults (e.g., from historical data), then it would be possible to improve the accuracy of the approach by using that data here. For $P(p)$, the approach counts all occurrences of the visual property represented by p (e.g., all `background-color` properties) and divides this by the total number of visual properties defined in the PUT.

$\mathbf{P}(\mathbf{S})$ represents the probability of the visual symptoms in S being either true or false for a given page. In this case, we are only considering the probability of the visual symptoms within the context of the original PUT, which has a fixed structure, layout, colors, etc. Therefore, the value of $P(S)$ is always constant since we are only considering one page, the PUT.

Using the relationships defined above, we can now rewrite Equation (1). Namely, we know that $P(S)$ and $P(e)$ are constants and therefore we are multiplying the conditional probability for each r by $P(e)/P(S)$. Since we are comparing the conditional probabilities to each other, the constant term can drop out without affecting the relative ranking of root causes for the PUT. This leaves us with the formula shown in Equation (3). Note that in Equation (3) we have substituted $\langle e, p \rangle$ for r to show the relationship of the terms more clearly.

$$P(\langle e, p \rangle | S) = [\prod_{s \in S} P(s|\langle e, p \rangle)]P(p) \tag{3}$$

C. Identifying the Most Likely Root Causes

Once the model has been built, the approach can identify the most likely root cause by ranking all root causes in the PUT according to their probabilities. To do this, the approach first identifies the set R of all root causes in the PUT. Then the approach iterates over each root cause $r \in R$ and does the following: (1) identify $P(p)$ for the visual property in r; (2) determine the symptoms by comparing the oracle and the PUT as described in Section III-D; (3) for each symptom s, look up the value $P(s|r)$ in the model; and (4) use the identified values to calculate the probability of r as specified by Equation (3). Once all $r \in R$ have been assigned a probability, the approach ranks all of the root causes from high to low by their probabilities and outputs this list to the developer.

D. Visual Symptoms of Presentation Failures

A presentation failure manifests itself as a difference between the actual appearance of a PUT and its intended appearance. We refer to these different types of differences as symptoms, which are indications or clues to the underlying root cause of the observed failure. As defined earlier, a *visual symptom* is a Boolean predicate describing the presence of a specific type of difference between the appearance of the PUT and the oracle.

There are two challenges in defining symptoms for our approach. First, a symptom must be independent of any particular web page. Otherwise, the usefulness of the symptom will not generalize to other web pages. To deal with this challenge, we define symptoms that are not based on a web page's structure or content, since these can vary significantly among web pages. Second, the set of symptoms must be comprehensive enough to cover all root causes and also provide distinguishing power among related root causes. This is challenging because there are many visual properties. Fortunately though, many of these properties have similar visual impacts, which allows us to define broad-level groupings of symptoms. Fine-grained symptoms within each broad-level grouping further allow us to accurately distinguish between the root causes. To identify

TABLE I: Mapping between symptoms and visual properties.

Impact Area	Visual Properties	Symptoms
Difference pixels	HTML element	AllDPInElement, DPInElement
Color	color, background-color, border-top-color, border-left-color, background-color, border-right-color, border-bottom-color, outline-color	AddedColor, RemovedColor, SingleHue
Size	height, width, padding, padding-bottom, padding-top, padding-right, padding-left, border-top-width, border-bottom-width, border-left-width, border-right-width, max-height, min-height, border-spacing	PageSizeChanged, AllDPInTopOfElement, AllDPInBottomOfElement, AllDPInRightOfElement, AllDPInLeftOfElement
Position	margin, margin-top, margin-bottom, margin-right, margin-left, position, line-height, vertical-align, bottom, top, right, left, float	ExactMatchElement, AlmostMatchElement, NoMatchElement, ShiftLeftElement, ShiftTopElement, ShiftRightElement, ShiftBottomElement
Visibility	display, overflow, visibility	VisibleElement
Text appearance	font-size, direction, font-family, white-space, text-align, letter-spacing, text-decoration, word-spacing, text-indent, font-weight, text-transform, font-variant	TextElement, ContainSpace
Decoration style	outline-style, border-style-left, etc.	DPInBorder, DPOutsideBorder, DPInsideBorder

these groupings, we analyzed the set of visual properties and classified them based on their visual impact on an HTML element. For example, the CSS properties `display` and `visibility` can both make an element appear or disappear, so they are grouped in the visual impact category "Visibility." In Table I under the column labeled "Impact Area" we show the seven areas of visual impact that we identified in this process. For each of the visual impact areas, we list the visual properties that we classified into that area (column "Visual Properties"). In the third column ("Symptoms") we then list the set of symptoms we defined that relate to the group. We will now discuss each of the visual impact areas and their corresponding symptoms in more detail.

Fig. 1: Illustration of difference pixels

1) Difference Pixels Symptoms: It is important for the approach to be able to identify HTML elements that are more likely to be faulty than others. To do this, our approach leverages the idea of using difference pixels to localize potentially faulty elements. The intuition is that an image difference can be computed between the screenshot of the page under test (SPUT) and the appearance oracle. An example of an image difference for two images is shown in Figure 1. The image on the right is the difference image between the left one and the middle one. The areas that are different show up as non-black pixels in the difference image, we call these *difference pixels*. The intuition is that the different pixels will be clustered in or around HTML elements that have a fault in them. Related work [20], [21] has shown that difference pixels can be used to reliably localize faulty HTML elements, therefore we include this insight in our work by encoding it as a symptom. By analyzing the position of the difference pixels, the approach can infer which of the HTML elements are most likely to be part of the root cause.

AllDPInElement: This symptom is true for an element when all of the difference pixels are located in the bounding rectangle box of the rendering of the HTML element. This indicates that the visual difference is completely contained inside the boundary of the HTML element. The approach also defines a similar symptom named **DPInElement**. This symptom is true when the HTML element contains at least one difference pixel.

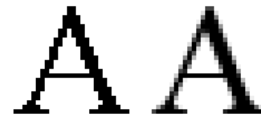

Fig. 2: Illustration of Anti-aliasing

2) Color Symptoms: There are eight visual properties that can manipulate the color of HTML elements. Developers can use these properties to change the color of text, background, border, or outline. To differentiate these CSS properties from others, the approach analyzes the color set differences between the SPUT and the oracle image.

AddedColor: This symptom is true when one or more RGB colors appears in the SPUT that do not appear in the oracle image. This indicates that the presentation failure introduced new colors into the PUT. When developers set a wrong color for an HTML element, it is possible that the color is changed to some other color that did not appear in the original web page. This causes a new color to appear in the color set of the web page, which means the symptom will be true. However, this symptom will not always happen for a color-related fault. For example, if a developer sets the value of the faulty color property to a color already present in the web page, the color set of the web page will remain the same. In this case, the symptom will not be true. This highlights a strength of the probabilistic model. It does not require there to be a perfect symptom/root cause relationship, but can still include symptoms that are simply more likely to be true. In this scenario, proper analysis allows for the model to still assign some non-zero level of probability for this possible situation, which can then be evaluated in context compared to the other possible root causes. Our approach also defines a similar, but opposite, symptom, **RemovedColor**.

SingleHue: This symptom is true when the number of hues (using the HSV color model) that differ between the oracle and PUT is one. Hue is a pure color without any shade. For example, light green and dark green are the same in terms of hue. Focusing on hue can help us identify the side effects of anti-aliasing, which is used when the browser renders text or shapes. Anti-aliasing smooths the color transition of edges by introducing some intermediate colors around the edges. Figure 2 shows an example of anti-aliasing. Normally, one color-related property change should only lead to one color difference between the oracle and SPUT. However, anti-aliasing introduces several new colors in the difference image. The key distinguishing factor is that the hues of these colors are the same since anti-aliasing only introduces intermediate colors in the same hue. Based on this observation, color root causes related to certain kinds of shapes (e.g., text) will mostly lead to one hue difference.

3) Size symptoms: There are 14 visual properties that are able to change the size of an HTML element. They can either directly change the HTML element by modifying its width or height, or indirectly change the element's size by size-related operations, such as border, margin, or padding. Our approach defines the following symptoms for this group.

SamePageSize: This symptom is false when the size of the oracle image is not equal to that of the PUT. This indicates that the presentation failure changed the size of an HTML element that then led to a change in the size of the page.

AllDPInTopOfElement: This symptom is true when all of the pixels that are different between the PUT and oracle are in the top half of an HTML element. This indicates the root cause may have affected visual properties that refer to the top part of the element. Examples of these are `border-top-width` and `padding-top`. These types of faults either directly affect the pixels near the top of the element or indirectly affect them by moving the top either down or up in relation to its intended position. Similarly, we have analogous symptoms. **AllDPInRightOfElement**, **AllDPInLeftOfElement**, and **AllDPInBottomOfElement**.

4) Position Symptoms: In this group, there are 12 visual properties that are able to change the position of the HTML element. The visual impact of these properties are moving the HTML element without changing its appearance. To differentiate these CSS properties, the idea is to try to match the appearance of an HTML element in the PUT with a location in the oracle image. To do this, the approach first retrieves the position and size of the HTML element in the faulty web page. Then, the approach generates the SPUT and crops it to the area of the HTML element, based on the element's position and size. Lastly, the approach determines if the same cropped part of the oracle image matches the cropped image of the HTML element from the SPUT. Based on this matching, the approach derives the following symptoms.

ExactMatchedElement: This symptom is true when an HTML element matches in the oracle at exactly the same position and with the same dimensions. This indicates that the presentation failure does not affect the position of the element itself (but possibly other elements). This happens for `margin-right`, `margin-bottom`, etc. A similar symptom is **AlmostMatchedElement**. This symptom is true when the element matches in the oracle but not at the same position. This indicates that the presentation failure only changed the position of the element, but not its visual appearance. This happens for `margin-top`, `margin-left`, etc. The approach also defines **NotMatchedElement** for when neither of the previous two situations apply.

ShiftBottomElement: This symptom is true when the HTML element moves downwards in the SPUT. When the approach is able to match the element in the oracle image, the approach further checks if the element has moved downwards. This happens for `padding-top`, `margin-top`, etc. Similarly, the approach defines **ShiftLeftElement**, **ShiftRightElement**, and **ShiftTopElement**.

5) Visibility Symptoms: There are three CSS properties in this group, and they can make an HTML element invisible (i.e., not appear in the SPUT). To identify faulty root causes that contain these CSS properties, the approach checks if the HTML element is shown in the SPUT.

VisibleElement: This symptom is true when the HTML element visually appears in the SPUT. The approach checks that the height and width are greater than zero in the browser rendering.

6) Text Appearance Symptoms: In this group, there are 12 CSS properties related to the appearance of text. They can change the appearance of text in terms of its size, style, etc. To distinguish this group from others, the idea is to extract the text of the HTML element, then further analyze the content of the text using text processing techniques as follows.

TextElement: This symptom is true when the HTML element in the PUT contains text. Because the 12 CSS properties change the appearance of the text in the HTML element, if the element does not contain any text, none of the 12 CSS properties will have any visual affect on the element, let alone cause a presentation failure. Hence, this symptom equal to true is highly correlated with any of the 12 CSS properties in this group having an effect.

ContainsSpace: This symptom is true when the HTML element in the PUT contains white space. The CSS properties `white-space` and `word-spacing` specify how white space is handled. Similar to the previous symptom, only if the element contains white space can the two CSS properties have a visual impact on the HTML element. Hence, this symptom equal to true is a prerequisite for the two CSS properties to cause presentation failures.

7) Decoration Style Symptoms: In this group, there are six CSS properties. They control the style of the border or outline of the HTML element. One pattern we found for this group is that the visual change is likely to affect a certain area of the HTML element, such as border and outline (a line drawn around the HTML element outside of the border). Hence, the approach first calculates the border of the HTML element by

retrieving rendering information from the DOM of the PUT. Next, the approach calculates the location of the difference pixels. The approach then checks if the difference pixels are on the border, outside the border, or inside the border of the HTML element.

DPOnBorder: The symptom occurs when the difference pixels are located on the border of the HTML element. This indicates the root causes contains CSS properties related to `border-style`. Similarly, the approach defines analogous symptoms, **DPOutsideBorder** and **DPInsideBorder**.

IV. Evaluation

To evaluate our approach, we designed experiments to determine its accuracy and efficiency in supporting regression debugging. The specific research questions we considered are as follows:

RQ1: How accurate is our approach in identifying root causes of presentation failures?

RQ2: What are the computational resources needed to run our approach?

To address these research questions, we carried out an empirical evaluation of our approach on a set of real-world web applications and compared this with the performance of other automated approaches.

A. Implementation of Our Approach

We implemented our approach as a prototype tool, FieryEye, in Java. For building the probabilistic model, we used the `org.w3c.dom` package in Java to extract the DOM structure, `jStyleParser` to extract the CSS properties defined for every HTML element/node in the DOM, and `org.w3c.dom` to inject presentation faults by modifying the values of the different CSS properties and create data samples. To compute the different visual symptoms listed in Table I, we used `Selenium WebDriver` [8] to take screenshots of the web pages and extract bounding rectangle information of the HTML elements, and `OpenCV` [5] to compare the screenshots, extract color information, crop screenshots, and perform sub-image searching. We parallelized the model building using a cloud of 200 Amazon EC2 c4.large instances. The total number of data samples to be created for each subject application was distributed equally over the 200 instances. The output probabilities for the data samples based on the observed visual symptoms from all the instances were then collated to form the probabilistic model for a given subject application.

For analyzing the PUT, the visual symptoms were extracted using the same approach described above. We ran all of the localization experiments on a Ubuntu 14.04 platform with 8GB RAM and a 4th Generation Intel Core i7-4770 Processor.

B. Experiment Protocol

In the evaluation, we focused on one scenario in which regression debugging might be needed, refactoring web pages. The goal of web page refactoring is to change the structure or HTML code of a page without altering its visual appearance. We chose refactoring because it is a topic that has received a lot of attention in the research and development community (e.g., [16]) and as a result there are objective and independently defined processes for how to refactor for certain objectives. Based on a literature review, we chose to focus on three refactoring techniques that were widely mentioned and had clearly defined processes. The first refactoring is to migrate HTML 4 to HTML 5. For example, converting `<div id="header">` into `<header>`. The W3C provides official guidelines for converting a typical HTML 4 web page into an HTML 5 page [4]; The second refactoring is to convert a table based layout to a div based layout. Related work [23] provides a set of best practices to be used in this conversion that will maintain the original appearance of the page. The third refactoring is to replace deprecated tags. For example `<font>` is deprecated in favor of using the CSS `font` property. Here again, the W3C provides a complete list of the deprecated HTML tags [3].

TABLE II: Subject Applications

Name	URL	RC#	T#
Perl	http://dbi.perl.org	1,592	36
GTK	http://www.gtk.org	1,121	30
Konqueror	http://konqueror.org	6,779	39
Amulet	http://www.cs.cmu.edu/~amulet	88	22
UCF	http://www.ucf.edu	2,415	47

Our choice of refactorings guided our selection of subject pages and precluded the use of benchmarks used in prior work (e.g., [21]), as the chosen refactorings were not applicable for those web pages. Our initial starting point was to gather URLs from an online random URL generator (http://www.uroulette.com). We then filtered this set to include only web pages to which all three refactorings could apply. The resulting set of subjects is shown in Table II. For each subject, the number of potential root causes is shown in the column labeled "RC#". For each of these subjects, we created a refactored version with all three refactorings applied. We performed the refactorings manually, but followed the instructions provided for each to make the process objective and repeatable. To confirm that the visual appearance was unchanged, we used an image differencing tool to ensure that there were zero difference pixels between the original and refactored version. On average, refactoring and verifying each subject took 5-10 hours depending on its complexity.

To create test cases for the evaluation, we seeded presentation failures into the refactored versions of each subject. We used the following process for each subject p: (1) download p and all of the files required to display it, (2) take a screenshot of p (which represents the previously correct version of the subject application before refactoring) rendered in a browser to serve as the oracle, O, (3) generate a set P' for the refactored page p' that contains variants of p', each with a unique randomly seeded presentation fault. We created these variants by introducing a fault in p' that changed the original value of each unique visual property in P'. We eliminated any

TABLE III: Accuracy and timing results.

Application	FieryEye				XPERT				TDT				WebSee			
	Rnk.	Rcl.	Mdl. (s)	Prd. (s)	Rnk. A	Rnk. B	Rcl.	Prd. (s)	Rnk. A	Rnk. B	Rcl.	Prd. (s)	Rnk. A	Rnk. B	Rcl.	Prd. (s)
Perl	6	100	149	7	55.8	9	61.1	13.7	357.8	55.3	100	0.2	51	7.5	61	14
GTK	1.5	100	57	8	496	46	96.7	7.5	288	25.5	100	0.1	22	11.5	83	14
Konqueror	13.5	100	175	19	281	17	51.3	13.4	1350	98.5	100	0.2	93	16	59	21
Amulet	12.5	100	20	11	19	19	0	6.1	36.8	24	100	0.1	10.5	7	68	15
UCF	6	100	420	40	369.5	57.5	78.7	19.5	713	118.3	100	0.2	18	9	57	46
Average	7.9	100	164	17	244.3	29.7	57.6	12	549.1	64.3	100	0.2	38.9	10.2	65.6	22

variant in P' where the seeded fault did not actually produce a presentation failure. To do this we used image comparison to determine if there was a pixel level difference between the rendering of the variant and O.

Each test case was comprised of an appearance oracle (O), which was the screenshot of the original website, the HTML corresponding to the appearance oracle (p), the refactored page with a seeded fault (variant in P'), and the targeted root cause (the element-visual property combination in which the presentation failure was seeded to create a variant). The number of test cases generated for each app is shown under the column *T#* in Table II.

We applied a similar seeding strategy for building the model for each subject application. As described in Section III-B, a list of all possible root causes was computed for a given PUT by enumerating all the HTML elements in the page and their corresponding visual properties. The data samples were then generated by iterating over these potential root causes and injecting a unique fault in the PUT by changing the value of the root cause, and then identifying the resulting visual symptoms.

To provide a reference point for our approach, we compared FieryEye against three automated approaches that can detect presentation failures and find root causes at the level of faulty HTML elements. The first of these, WebSee [21], is a prior technique for detecting and localizing presentation failures. We were able to use this technique without any modification. The second, XPERT [10], is a well-known tool for performing XBT. Although XPERT targets XBI, we included it to provide a reference point for how well state of the art XBT techniques could detect more general presentation failures. The publicly available version of XPERT lacked the "visual analysis," which compares two elements in different pages using a color histogram. So we implemented this portion as described in the XPERT paper and used the same image processing libraries that we used in our implementation of FieryEye. The third technique, Textual Differencing Tool (TDT), is a tool based on `diff`. We built TDT to model the type of approach a developer might employ using off the shelf libraries and popular command line tools. The TDT first used `jStyleParser` to inline all CSS properties in the two HTML pages. Then the source of both pages was normalized using `jsoup`. To compare the pages, TDT used `diff` to identify lines that were different. Finally, TDT parsed the difference lines to extract the HTML tags and visual properties they contained and used this to generate a set of candidate faulty root causes.

C. Metrics

To compare the accuracy of the four approaches, we calculated the ranking of the correct root cause in the result set returned for each test case. For FieryEye, we could directly use the output as provided by the tool. However, for the other three techniques, we had to calculate the rankings differently, since they were not originally designed to rank root causes.

WebSee returns a ranked list of HTML elements, not root causes. So we processed the ordered list returned by WebSee and computed the rank in two ways:

Ranking A: Iterate over the ranked element list and add the number of visual properties associated with each incorrect faulty element until the correct faulty element is found. When the expected faulty element is found in the list, add half the number of visual properties associated with the faulty element to the total number of incorrect root causes computed. (This amount represents the average number of root causes to be explored for the element.) This number is returned as the rank of the correct root cause.

Ranking B: Iterate over the list of potentially faulty elements and increment the rank counter by one for every incorrect faulty element until the correct faulty element is found. Then, similar to ranking A, when the expected faulty element is found in the list, add half the number of visual properties associated with the faulty element to the rank counter and return this as the rank of the root cause.

The goal with using both rankings is to quantify a range in the way developers may use the results returned by WebSee. In this case, Ranking A represents an upper bound on effort, a developer who will systematically inspect each potentially faulty root cause until finding the faulty one. Ranking B represents a lower bound, a case where a developer can quickly rule out possible root causes based only on the element.

For XPERT we faced two challenges in calculating rank, the technique targets HTML elements and the results are returned unsorted. We addressed the first challenge in the same way we did for WebSee, by counting each HTML element as representing n root causes, where n is the number of visual properties defined by the element. To address the second challenge, we assumed that, on average, the developer would have to examine half of the identified set of root causes before they found the faulty one. We calculated rank for TDT in this manner as well. The median rankings (*A* and *B* for Ranking A

and Ranking B) for each app and the four approach are shown under the columns named *Rnk.* in Table III.

We also calculated the recall of the four approaches. To do this we measured the percentage of the result sets that contained the correct root cause, regardless of its rank. We show the recall as a percentage under the column labeled *Rcl.* in Table III.

For RQ2, we measured the average running time for each of the test cases as they were executed by the four approaches. For all four approaches, this number is reported in seconds under the column labeled *Prd.* in Table III. We also report the model building time, in seconds, for FieryEye under the column labeled *Mdl.*

D. Discussion of Results

Our results show that FieryEye performed efficiently and was able to achieve better rankings than the other considered techniques. As can be seen from Table III, FieryEye reports an average median rank of 7.9, which means that in 50% of the cases the developer will have to inspect less than eight root causes. Depending on the approach compared against, this resulted in 31–541 fewer root causes for Ranking A and 2–56 fewer root causes for Ranking B that would have to be evaluated by the developer. For the next best technique, WebSee, the average median ranking was over 38 for Ranking A, which is likely to be too high of a number to be feasible for developers to inspect. For Ranking B, the average median was reported to be 10. Although this number is close to FieryEye's median rank, it should be noted that Ranking B represents a very optimistic estimate of the debugging effort and that WebSee's ranking results come with a much lower recall of 65%. WebSee did better for only one subject, Amulet. We investigated this app further and found that it contained only a small number of HTML elements with very few visual properties. This resulted in small average result set sizes and in some cases, zero size result sets, which artificially lowered the median.

To show the accuracy results from a broader perspective, we plotted a cumulative frequency distribution graph, shown in Figure 3, of the correct root cause rank reported by all the four techniques. In this graph, a point (X, Y) on any trend line of the four techniques means that in Y% of the cases, the technique ranked the correct root cause within the top X. From this graph, it can be seen that FieryEye more reliably ranks the correct root cause higher in the result set than WebSee, XPERT, and TDT, making it more useful for developers in debugging presentation failures. For example, FieryEye ranked the correct faulty root cause in the top five in almost 45% of the cases, whereas WebSee reported the correct root cause in the top five in only 5% of the cases for Ranking A and in 10% for Ranking B. XPERT and TDT reported the correct root cause in the top five in less than 1% of the cases for Ranking A as well as Ranking B.

Overall, we consider these ranking results to be strong indicators of the potential usefulness of our approach. In relative terms, the results are significantly better than alternative techniques. In absolute terms, the rankings are generally low and well within the "first few" guidelines considered useful for developers by Parnin and Orso [29], and within accuracy levels that have been shown in related work to be considered useful by practitioners debugging user interfaces [14].

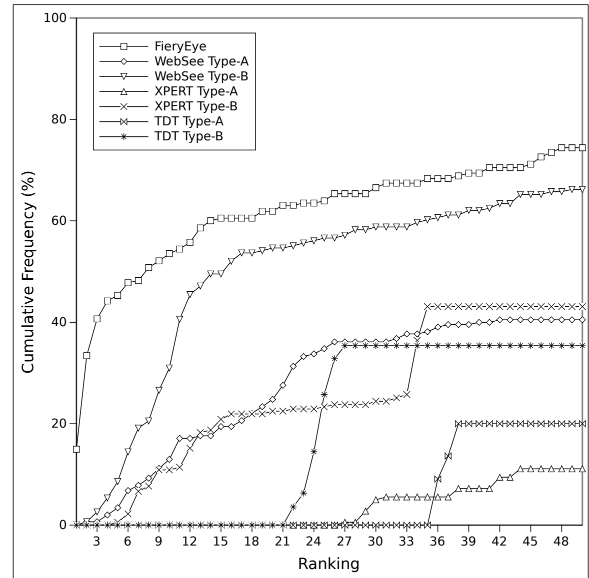

Fig. 3: Cumulative frequency distribution of root cause rank

It is important to note the implications of these results is limited to the regression debugging scenario. In particular, XPERT is designed for XBT, not regression debugging. Therefore, we see these results, with respect to XPERT, only as a confirmation of our assertion in Section II that techniques for XBT are not suitable for regression debugging.

Prediction time was more or less comparable for all of the approaches, except TDT. However, TDT was hopelessly imprecise. The remaining approaches ranged from 12–22 seconds per test case, which is likely to be an acceptable amount of time for the developer. For FieryEye's model building, the average computation time for each of the 200 instances running in parallel was a little less than 3 minutes. This time can be decreased further by using a higher number of Amazon EC2 instances, thus reducing the number of data samples that are processed on each instance. The cost required for building the model is also low. As of October 2015, the price for on-demand c4.large instances was $0.11 per hour. Thus, with an average of less than 3 minutes for model building computation across 200 instances, on average, only $1 of computational resources was required to support the automated debugging efforts.

E. Threats to Validity

In this section we discuss threats to the validity of our conclusions. The first potential threat is that we used a uniform distribution of faults for generating the test data and training the model. In other words, we assumed that each type of presentation failure is equally likely to occur. It is possible that real root causes may have a different distribution. However, we

do not have such information available to us, and if we did, we could also use it to help train the model by more appropriately setting the $\mathbf{P}(\mathbf{r})$ in Equation (1). The second threat is that we controlled the changes introduced for simulating regression debugging. To mitigate this, we chose to carry out HTML page refactoring because there were externally defined standards for how to perform the three different types of refactorings and we could follow these approaches to minimize the unintentional introduction of bias. Our third threat is that we used a fault seeding mechanism for generating the test cases instead of real-world faults. We adopted this mechanism as we did not have access to real refactored web pages. However, to minimize this threat we randomized the fault seeding and used a completeness criteria of seeding into every unique CSS property of each page. Randomized fault seeding also minimizes the potential bias of using fault injection for both model building and generating test cases. A fourth threat is that our subject selection was biased towards applications that needed significant refactorings. We introduced this bias because, as noted in Section II, it is clear that all techniques would perform similarly for small localized changes in the DOM. Therefore, we wanted to focus on the unknown area, which is larger DOM changes. The generalizability of these results would depend on the real-world distribution of types of refactoring, which is unknown to us at this time. Our final threat is that for many techniques we introduced a ranking metric for them. Unfortunately, we could not avoid this since we needed a quantifiable measure to compare the approaches, but for each of the tools, we tried to define a ranking that reflected the way the tool would likely be used if applied to the scenario and provided reasonable upper and lower bounds.

V. Related Work

Work by the authors [20], [21], [22] in the field of debugging presentation failures in web applications facilitated automatic detection and localization of presentation failures to HTML elements. However, the applicability of this work is limited in root cause analysis, as it does not report the faulty visual property in the faulty HTML element, that collectively form the root cause of a presentation failure.

Another work [19] handles root cause analysis of web application presentation failures using search-based techniques. This work appeared as a new ideas short paper and performed a small case study to evaluate the viability of the approach. However, the key limitations such as low performance and accuracy owing to the large root cause search space were reported. In contrast to this work, our approach uses a probabilistic model to find root causes and is more accurate and performance efficient.

Work by Roy Choudhary and colleagues [30], [9], [10] in the field of XBT compares the rendering of a given web page in a reference browser against its rendering in another browser to detect cross browser issues and report the potentially faulty HTML elements. As discussed in Section II, use of this approach in regression debugging is limited to scenarios where the DOM had not changed significantly and matching elements could still be matched using probabilistic techniques. As discussed in Section IV, the use of the XBT techniques is further limited in the context of debugging presentation failures as it does not provide the faulty visual property in the output. The potentially faulty HTML elements given as output are also not ordered, posing difficulty for the developer in reaching the correct faulty element quickly. A commercial tool, Browserbite [32], uses image comparison on screenshots of a test web page that is rendered in different browsers for finding XBI. However, Browserbite cannot find the root cause of an observed XBI. Also, it uses a human-based classification to find true positives from the observed visual XBI, which can potentially lead to false positives.

Browser plug-ins, such as "PerfectPixel" [6] for Chrome and "Pixel Perfect" [7] for Firefox help developers to visualize pixel-level differences with an image based oracle. They overlay a semitransparent version of the oracle over the HTML page under test, enabling developers to do a per pixel comparison to detect presentation failures. However, the developers then have to manually identify the root causes of the observed presentation failures. In contrast, our approach is fully automated for root cause analysis.

Memon and colleagues [24], [39], [26], [27] 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 and analyzing the presentation of the GUI, but instead focus on testing the behavior of the system based on event sequences triggered from the GUIs.

A group of techniques, such as validating for malformed HTML [31], [28], [12], automatically performing presentation changes [37], and impact analysis of CSS changes [17], focus on different aspects of web applications' client side that can impact their appearance. Though these works can help debug certain specific types of presentation failures, they cannot handle all types of presentation failures in a generic fashion.

VI. Conclusion

In this paper, we introduced a new approach for finding root causes of presentation failures in web applications. Our approach uses image processing techniques to find visual symptoms and probabilistic models based on Bayes' theorem to predict the root causes. The approach builds the models by automatically seeding faults in the PUT and observing the visual symptoms. The probabilistic models are then used to rank all of the root causes in the PUT according to their likelihood of being faulty. In the evaluation, our approach reported the correct root cause at an average median rank of 7.9 and outperformed other related techniques. Overall, these are strong results and indicate that our approach could effectively assist developers in debugging presentation failures. In future work we plan to investigate techniques for handling multiple presentation failures in a web page.

Acknowledgment

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

REFERENCES

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

[2] Firebug. https://addons.mozilla.org/en-US/firefox/addon/firebug/.

[3] HTML 5. http://www.w3schools.com/tags/.

[4] HTML 5 Migration. http://www.w3schools.com/html/html5_migration.asp.

[5] OpenCV. http://opencv.org.

[6] Perfect Pixel Chrome. https://chrome.google.com/webstore/detail/perfectpixel-by-welldonec/dkaagdgjmgdmbnecmcefdhjekcoceebi?hl=en.

[7] Pixel Perfect Firefox. https://addons.mozilla.org/en-us/firefox/addon/pixel-perfect/.

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

[9] S. R. Choudhary, M. R. Prasad, and A. Orso. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 171–180, Washington, DC, USA, 2012. IEEE Computer Society.

[10] S. R. Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications. In *Proceedings of the 35th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2013)*, pages 702–711, San Francisco, USA, May 2013.

[11] 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.

[12] 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.

[13] M. Grechanik, Q. Xie, and C. Fu. Creating GUI Testing Tools Using Accessibility Technologies. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '09, pages 243–250, Washington, DC, USA, 2009. IEEE Computer Society.

[14] M. Grechanik, Q. Xie, and C. Fu. Maintaining and Evolving GUI-directed Test Scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[15] S. Hallé, N. Bergeron, F. Guerin, and G. L. Breton. Testing Web Applications Through Layout Constraints. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–8, 2015.

[16] E. R. Harold. *Refactoring HTML: Improving the Design of Existing Web Applications*. Addison-Wesley Professional, 1st edition, Dec 2012.

[17] 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*, UIST '13, pages 353–356, New York, NY, USA, 2013. ACM.

[18] G. Lindgaard, C. Dudek, D. Sen, L. Sumegi, and P. Noonan. An Exploration of Relations Between Visual Appeal, Trustworthiness and Perceived Usability of Homepages. *ACM Trans. Comput.-Hum. Interact.*, 18(1):1:1–1:30, May 2011.

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

[20] 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.

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

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

[23] A. Y. Mao, J. R. Cordy, and T. R. Dean. Automated Conversion of Table-based Websites to Structured Stylesheets Using Table Recognition and Clone Detection. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 12–26, Riverton, NJ, USA, 2007. IBM Corp.

[24] A. M. Memon, I. Banerjee, and A. Nagarajan. What Test Oracle Should I Use for Effective GUI Testing? In *ASE*, pages 164–173, 2003.

[25] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1):3:1–3:30, Mar. 2012.

[26] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon. A Pattern-Based Approach for GUI Modeling and Testing. In *Proceedings of the 2013 International Symposium on Software Reliability Engineering*, ISSRE '13, pages 288 – 297, 2013.

[27] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software. *Automated Software Engg.*, 21(1):65–105, Mar. 2014.

[28] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and Fix-propagating for HTML Validation Errors to PHP Server-side Code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

[29] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[30] S. Roy Choudhary, H. Versee, and A. Orso. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[31] 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*, ICSE 2012, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[32] N. Semenenko, M. Dumas, and T. Saar. Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 528–531, Washington, DC, USA, 2013. IEEE Computer Society.

[33] S. Sprenkle, H. Esquivel, B. Hazelwood, and L. Pollock. WEBVIZOR: A Visualization Tool for Applying Automated Oracles and Analyzing Test Results of Web Applications. In *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic & Industrial Conference*, 2008.

[34] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated Oracle Comparators for Testing Web Applications. Technical report, In the Intl. Symp. on Software Reliability Engineering, 2007.

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

[36] N. Tractinsky, A. Cokhavi, M. Kirschenbaum, and T. Sharfi. Evaluating the Consistency of Immediate Aesthetic Perceptions of Web Pages. *International Journal of Human-Computer Studies*, 64(11):1071 – 1083, 2006.

[37] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 16:1–16:11, New York, NY, USA, 2012. ACM.

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

[39] Q. Xie and A. M. Memon. Studying the Characteristics of a "Good" GUI Test Suite. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, pages 159–168, Washington, DC, USA, 2006. IEEE Computer Society.

[40] 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.

[41] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM.

[42] H. Zhang. The Optimality of Naive Bayes. *AA*, 1(2):3, 2004.