WebSee: A Tool for Debugging HTML Presentation Failures

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

***Abstract*—Presentation failures in a website can negatively impact end users' perception of the quality of the website, the services it delivers, and the branding a company is trying to achieve. Presentation failures can occur easily in modern web applications because of the highly complex and dynamic nature of the HTML, CSS, and JavaScript that define a web page's visual appearance. Debugging such failures manually is time consuming and error-prone, and existing techniques do not provide an automated debugging solution. In this paper, we present our tool, WebSee, that provides a fully automated debugging solution for presentation failures in web applications. When run on real-world web applications, WebSee was able to accurately and quickly identify faulty HTML elements.**

I. Introduction

Ensuring the correct appearance of a website is important for its success. A recent study [18] by Google shows that typically a user forms opinion about a website within the first 50 milliseconds, which is only based on the "look and feel" of the website. Hence, it is crucial to make a good first impression and capture users' interest in the website. Presentation failures in web applications occur when the rendering of an HTML page does not match its expected appearance. Such failures can negatively affect usability, reduce an end user's perception of the quality of the services offered by the website, and undermine a company's branding efforts.

Presentation failures can occur easily in modern web applications because of the highly complex and dynamic nature of the HTML, CSS, and JavaScript that define a web page's visual appearance. To debug presentation failures is both a labor intensive and error prone task. Debugging presentation failures first requires a tester to identify when the layout and style of a web page is incorrect by comparing it with an oracle, such as a design mockup. In this situation, testers can easily miss presentation failures, since the process is based on the testers' visual comparison of a page and an oracle. The second step in the debugging process requires the testers to correctly identify the faulty HTML element responsible for the observed failure. This is challenging, since HTML and CSS interact in complex ways to render the appearance of a web page.

There have been several testing and analysis techniques developed by researchers and industry for debugging presentation failures in web applications. However, these techniques are generally limited in their applicability. For example, testing tools, such as Selenium [3], Crawljax [13], Sikuli [4], Cucumber [1], and graphical automated oracles [6] require testers to exhaustively specify all correctness properties to be checked, which makes the process labor intensive and error prone. Other techniques focus on detecting only one type of presentation failures, such as Cross-Browser Issue (XBI) [5], or maintaining GUI test scripts [8], [15]. Another approach, Google's Fighting Layout Bugs project [17], searches for certain kinds of common presentation failures that are application agnostic, such as overlapping DIV tags. This approach cannot find application specific presentation failures.

To address these limitations we proposed a novel automated technique in prior work [12]. This technique is implemented as a tool, WebSee, for detecting presentation failures and then identifying the corresponding faulty HTML elements. WebSee uses computer vision techniques to detect presentation failures by comparing the visual representation of a browser rendered web page with its intended appearance. WebSee then builds rendering maps of the web page that are used to identify the responsible faulty HTML elements. The identified HTML elements are then ordered in likelihood of being faulty and reported to the tester.

The rest of this paper is organized as follows: In Section II we describe WebSee's underlying technique. In Section III we discuss several scenarios in which WebSee can be used. In Section IV we describe WebSee's architecture and implementation. We discuss the evaluation results of running WebSee on real-world applications in Section V. In Section VI we discuss related work, then conclude in Section VII. Lastly, in Section VIII we describe the location of availability of the tool for public use. Note that we will only summarize the underlying technique and evaluation results in this paper, as a detailed description is available in the paper accepted for publication at ICST 2015 [12].

II. Our Technique

WebSee's technique focuses on automatically detecting and localizing presentation failures in web pages. It leverages computer vision techniques to detect presentation failures by comparing the visual representation of the rendered web page under test and its appearance oracle. The technique then builds rendering maps of the test web page to identify the HTML elements that are responsible for the regions of the web page identified as containing presentation failures.

Our technique takes three inputs: (1) a web page to be tested in the form of a URL pointing to either a location on the network or filesystem, (2) an appearance oracle in the form of an image that can be either a mockup or screenshot of a previously correct version of the test web page, and (3) a set of special regions that define areas of dynamic

978-1-4799-7125-1/15/$31.00 ©2015 IEEE

content, such as ads and news feed, that are common in modern web applications. Our technique processes these inputs through three phases, (1) *detection* – identify presentation failures by comparing test web page rendering and its oracle, (2) *localization* – identify potentially faulty HTML elements responsible for the observed presentation failures in phase 1, and (3) *result set processing* – prioritize the set of potentially faulty HTML elements identified in phase 2. The output of phase 3 is reported to the user.

We now summarize the different phases of our technique in the following subsections. A detailed description of the technique is available in [12].

A. Phase 1: Detection

The first phase of the technique compares the visual representation of the test web page with its appearance oracle image to detect presentation failures. To capture the visual representation of the test web page, the technique takes a screenshot of the browser window that is rendering the page following a normalization process to make it compatible with the appearance oracle. More details about the engineering step of the normalization process can be found in Section IV. For detecting presentation failures, our technique uses perceptual image differencing (PID), a computer vision based technique for image comparison [19]. PID uses a computational model of the human visual system to determine if two images are visually different, ignoring insignificant imperceptible pixel-level differences, and only detecting those differences that are perceptible to the human eye. The PID algorithm accepts two configurable parameters, a threshold value Δ, a field of view value in degrees F, a luminance value L, and a color factor C. Δ is used to to decide whether the images are below a threshold of perceptible difference, F indicates how much of the observer's view the screen occupies, L indicates brightness of the display the observer is seeing, and C indicates sensitivity to colors. These four parameters together control the strictness of comparing the two images to find differences. The result of the PID algorithm is a set DP that contains all pixels of the two images considered to be perceptually different. The technique then removes all pixels from DP that are within a given special regions area, as these pixels will be processed separately as explained in Section II-D. The technique then applies clustering to identify difference pixels that are likely to be caused by the same fault. The identified clusters and their corresponding difference pixels are stored in a map and returned as an output of the detection phase.

B. Phase 2: Localization

In the localization phase the technique identifies a set of HTML elements in the web page under test that may be responsible for the detected presentation failure. To do this, the approach builds a Rectangle-tree (R-tree) model of the test web page rendered in a browser. An R-tree is a height-balanced tree data structure that is widely used in the spatial database community to store multidimensional information [9]. In this case, the multidimensional data is the bounding rectangle that corresponds to the rendering area of an HTML element. The leaves of the R-tree are the HTML elements of the page and the non-leaf nodes are bounding rectangles that contain groups of nearby elements. For each pixel, $\langle$x, y$\rangle$, in the difference pixels set in the cluster map obtained in phase 1, the containing HTML elements are found by traversing the R-tree's edges. A set of HTML elements obtained for all of the difference pixels in a cluster is the set of potentially faulty HTML elements for the cluster under consideration.

C. Phase 3: Result Set Processing

In the third phase, the HTML elements in the set obtained for every cluster are then ordered based on their likelihood of being the faulty element. The reason for this prioritization is that not all of the elements in the result set are equally likely to contain the fault. In many cases, a presentation fault in one element can impact surrounding elements and cause them to be identified as potentially faulty. For example, if the size of one element grows, then the surrounding elements will also have pixel-level differences due to either an overlap or being moved by the larger element. In experiments, we found that many of these dependent failures could be identified using heuristics. Therefore, we developed a prioritization score for each element based on weighted scores computed by the heuristics. For prioritization, we considered cases where a failure was inherited by all of an element's children, when a failure in a child overlapped its parent, when sizing and positioning caused a cascade of dependent resizing and repositioning, and the percentage of an element's total pixel size that was identified as being in the set of difference pixels.

D. Special Regions Handling

In modern web applications, it is common to have portions of the web pages that are defined by HTML whose content is not known until runtime. For example, in web sites, such as Amazon, there are ads that are loaded at runtime based on a user's browsing history. Comparing this page directly against its appearance oracle would lead to a false positive failure being identified in such areas of dynamic content. Our technique terms such areas as *special regions* – areas of a page where correctness properties other than a perceptual difference apply. For handling special regions, our technique leverages the knowledge that even though the exact contents of such areas are not known ahead of time, the graphic designers or the developers are often able to specify the visual correctness properties that should apply to such regions. Our technique defines two types of special regions, (1) *Exclusion Regions*: allow testers to specify regions for which no correctness properties will apply except size bounding (e.g., areas of advertisements, banners, etc.) and (2) *Dynamic Text Regions*: allow testers to specify regions whose content will be textual and for which the styling properties are known, but not the actual textual content (e.g., areas of user account information from database).

For *exclusion regions*, no check for visual differences is performed, allowing the technique to check the static portions of the web page, but ignore any presentation failures that exist in the areas of an exclusion region. For *dynamic text regions*, first a modified version of the test web page is created by injecting the specified correct styling properties into the HTML elements belonging to the dynamic text region. Then, phases 1 - 3 are run with the screenshot of the original test web page as the oracle and the modified page as the test web page. If a non-empty result set is obtained, then it is concluded that the original test web page had an incorrect style for the text, and the faulty HTML elements are reported to the user.

III. WebSee's Usage Scenarios

In this section we discuss two different scenarios in the lifecycle of a web application's development in which the need for debugging presentation failures arises, and how WebSee can be used effectively in such scenarios.

A. Regression Debugging

Regression debugging is where the developers perform maintenance on existing web pages in order to introduce new features, correct a bug, or refactor their HTML structure. For example, a refactoring may change a page from a table based layout to a table-less layout using <div> tags to improve the accessibility of the web page, without changing its visual appearance. A developer may accidentally introduce a presentation fault causing the refactored page's appearance to be different than the previous version.

Existing techniques, such as Cross-Browser Testing (XBT) [5], GUI differencing [8], automated oracle comparators [15], or tools based on `diff` may be of limited use in this scenario as they compare the Document Object Model (DOM) or other tree-based representations of the user interfaces to detect failures. Small changes that do not change the tree structure much can be found by these techniques; however, more significant refactorings or changes, such as the one described in the example above, will most likely cause these techniques to report a large number of spurious differences.

WebSee is suitable for this scenario, since it uses the visual representation of the web page to detect presentation failures. A screenshot of the previous working version of the web page, the non-refactored version in the example above, can be used as the oracle and the changed/refactored version as the test web page. WebSee then compares the screenshot of the non-refactored and refactored version using PID to find human perceptible differences.

B. Mockup-driven Development

Mockup-driven development [14], [10] is a typical style of modern web application development in which the developers build a web page to match a high fidelity design mockup created by graphic designers. Here, the general expectation is that developers will create a "pixel-perfect" web page to match the mockup [2].

Since, only a design mockup is available to validate the correct implementation of the web page, the tree-based debugging techniques described above are not applicable in this scenario as there exists no DOM or tree structure to compare the implemented web page against. Another group of techniques based on invariants specification, such as Selenium [3], Cucumber [1], Sikuli [4], Crawljax [13], and graphical automated oracles [6], are also not feasible as they require all the correctness properties to be exhaustively specified, which is labor intensive and error prone. Application agnostic techniques such as "Fighting Layout Bugs" [17] can be used, however, they only facilitate checking of general types of failures, such as overlapping text, and not application specific failures.

WebSee proves to be a good fit for this scenario as well because its detection mechanism relies on visual appearance comparison. The high fidelity design mockups created by graphic designers can be used as the oracle and the implemented web page as the test web page. The areas in the design mockup that may be defined by placeholders for dynamic content, such as advertisements, can be conveniently marked using special regions to avoid any false positives being reported. Also, PID will be able to handle differences, such as screen resolution and color intensity, between the platform on which the oracle was developed and the testing platform, further reducing the false positives.

IV. Tool Description

In this section we describe the architecture and implementation details of WebSee.

WebSee is implemented in Java and is made available to the users in the form of a standalone tool. It can be run on any platform, such as Windows, Linux, and Mac OS X, that has a Firefox browser, which is used for rendering the web page. WebSee analyzes the HTML page rendered in the browser, and hence can be run on web applications built using any server-side technologies, such as PHP, J2EE, or Ruby on Rails.

WebSee is fully automated, requiring only three inputs: the test HTML page, oracle, and a special regions file. The first input, test HTML page, can be provided in the form of a URL pointing to a file on the network or a local file system. If the URL is pointing to a file on the network, then the file is first downloaded locally and then used for further processing. The second input, oracle, can be given in the form of an image file path, simulating the mockup-driven development scenario, or as a URL pointing to an existing "golden version" of the web page on the network or file system, simulating the regression debugging scenario. For the regression debugging scenario, a screenshot of the given oracle HTML page rendered in the Firefox browser is captured using Selenium WebDriver and is used as the oracle image for further processing. The third input, special regions file, is a path to an XML file containing information about the special regions. In the XML file, three pieces of information are to be provided for each special region, (1) identifier: a special region's location and size can be specified in terms of x-y coordinates or the XPath of an HTML element enclosing the intended special region, (2) special region type: exclusion region or dynamic text region, and (3) style properties (only for dynamic text regions): a set of visual properties that should apply to the given special region.

Figure 1 shows a high-level overview of WebSee. It takes three inputs as described above. The inputs are then processed through the three phases, detection, localization, and result set processing, of WebSee to produce the output. In the first phase, detection, the oracle image and the test web page screenshot are visually compared to find differences. The comparison step produces two outputs, an image file with the marked visual differences and a set of visual difference pixels. Clustering is then applied to the visual differences which groups the difference pixels that are likely to be caused by the same presentation fault producing clusters 1–N. The difference image marked with the clusters' information is given as one of the outputs of WebSee. In the second phase, localization,

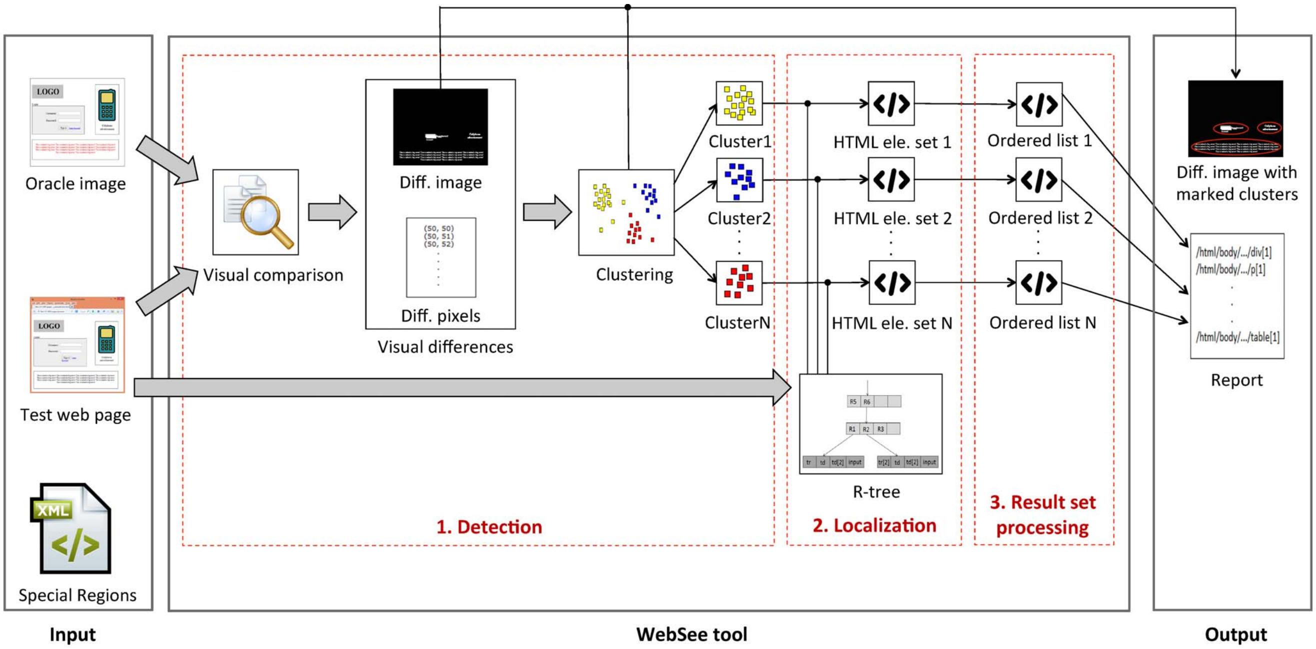


Fig. 1: High-level Overview of WebSee

the rendering model, R-tree, of the test web page is built and is used to identify a set of potentially faulty HTML elements for each cluster of difference pixels. In the third phase, result set processing, the HTML elements in the set obtained in the localization phase are prioritized in order of likelihood to have caused the observed presentation failure producing ordered lists 1–N corresponding to every cluster. The ordered lists of faulty HTML elements for all the clusters are saved in a report file. The report and the difference image marked with the clusters is then given as output to the user. For each special region in the input special regions XML file, the three phases, detection, localization, and result set processing, are run as described in Section II-D and the results are appended to the report file and the visual differences image file.

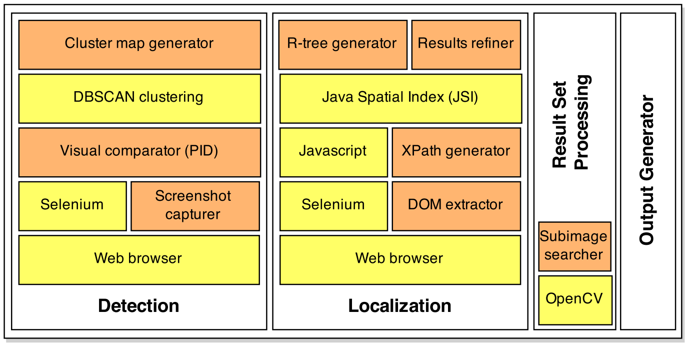
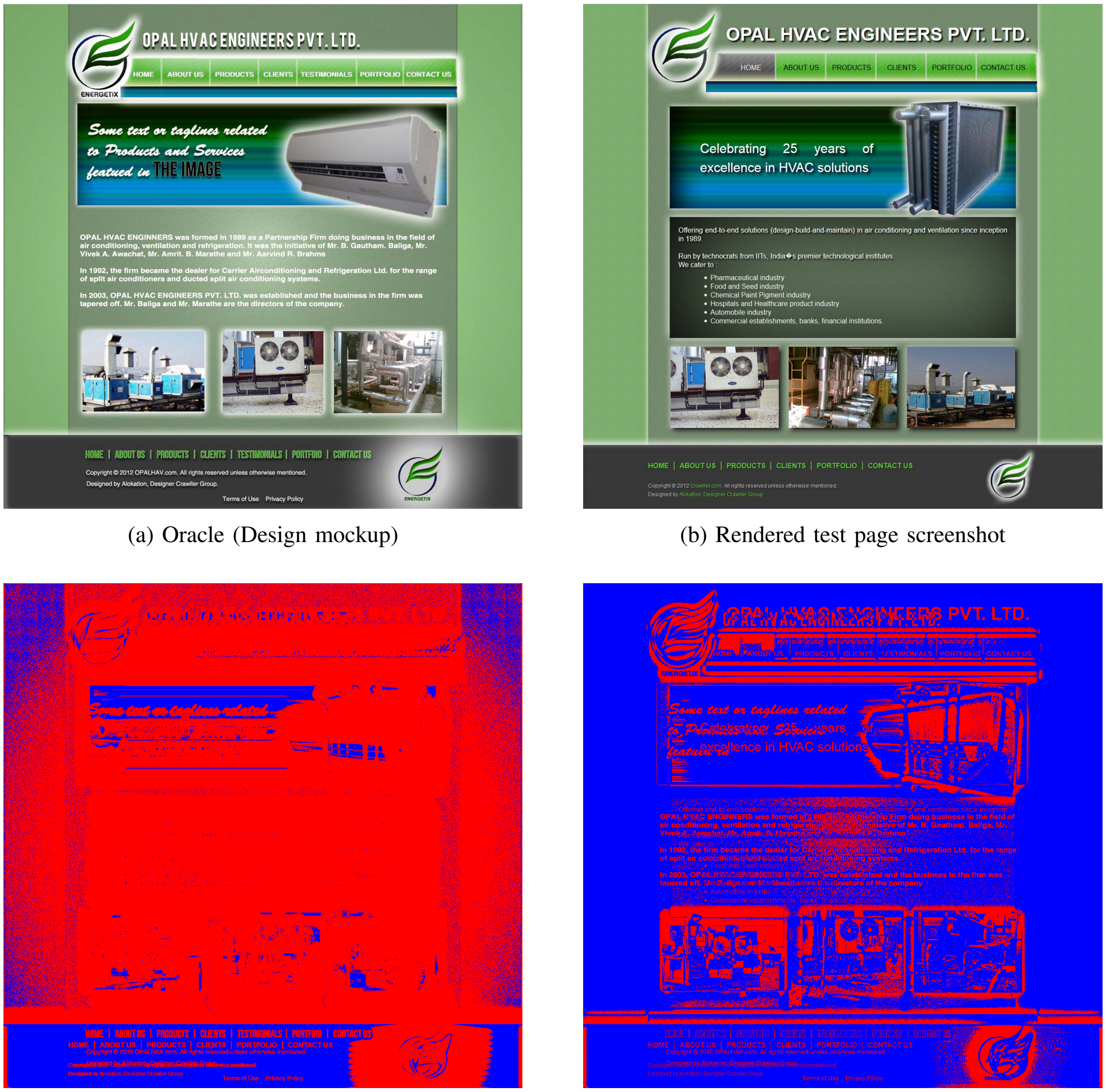


Fig. 2: System Architecture of WebSee

The system architecture of WebSee is shown in Figure 2. It consists of four modules, representing the three phases of the technique: detection, localization, and result set processing, and an output generator module. The four modules are shown as white boxes in the figure. The boxes in yellow color show the different external libraries or components used by WebSee. The boxes in orange show the different components we implemented in Java. The different boxes in the modules are to be read bottom-up, showing the dependency between different components. For example, in the detection module, the external component "Web browser" is required by the "Selenium" component. The horizontally placed boxes are to be read left-right, indicating the "used by" relationship. For example, in the detection module, component "Selenium" is used by the component "Screenshot capturer" to take a screenshot of a browser rendered web page. We now explain the implementation details of each of the four modules.

The *detection* module uses the Selenium WebDriver to capture a screenshot of the test web page rendered in the Firefox web browser and then uses perceptual image differencing to visually compare the oracle image and the test web page screenshot. We rewrote the PID library in Java based on the native "pdiff" library in C++. WebSee uses a popular density-based clustering algorithm, DBSCAN (Density Based Spatial Clustering of Applications with Noise) implemented in the Apache Commons Math3 library, to cluster the difference pixels into different groups. We used DBSCAN for clustering as it 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. The clusters are stored in a cluster map with the key as the cluster ID and value as the set of difference pixels belonging to that cluster.

The *localization* module uses Selenium WebDriver to extract the DOM information for each element in a browser rendered web page, such as the bounding rectangle, parent, and children elements, and runs a JavaScript routine to compute the element's XPath. The Java Spatial Index (JSI) library is used to help implement the R-trees for the HTML pages based on the extracted bounding rectangles information. The extracted element information is stored in the node of the R-tree. R-tree edge traversal is used to find the HTML elements containing a


OPAL HVAC ENGINEERS PVT. LTD.
HOME
ABOUT US
PRODUCTS
CLIENTS
TESTIMONIALS
PORTFOLIO
CONTACT US
ENERGETIX
Some text or taglines related to Products and Services featued in THE IMAGE
HOME | ABOUT US | PRODUCTS | CLIENTS | TESTIMONIALS | PORTFOIO | CONTACT US
Terms of Use
Privacy Policy
ENERGETIX

OPAL HVAC ENGINNERS was formed in 1989 as a Partnership Firm doing business in the field of air conditioning, ventilation and refrigeration. It was the initiative of Mr. B. Gautham. Baliga, Mr. Vivek A. Awachat, Mr. Amrit. B. Marathe and Mr. Aarvind R. Brahme

In 1992, the firm became the dealer for Carrier Airconditioning and Refrigeration Ltd. for the range of split air conditioners and ducted split air conditioning systems.

In 2003, OPAL HVAC ENGINEERS PVT. LTD. was established and the business in the firm was tapered off. Mr. Baliga and Mr. Marathe are the directors of the company.

Copyright © 2012 OPALHAV.com. All rights reserved unless otherwise mentioned.
Designed by Alokation, Designer Crawller Group.

(a) Oracle (Design mockup)

OPAL HVAC ENGINEERS PVT. LTD.
HOME
ABOUT US
PRODUCTS
CLIENTS
PORTFOLIO
CONTACT US
Celebrating 25 years of excellence in HVAC solutions
HOME | ABOUT US | PRODUCTS | CLIENTS | PORTFOLIO | CONTACT US

Offering end-to-end solutions (design-build-and-maintain) in air conditioning and ventilation since inception in 1989.

Run by technocrats from IITs, India�s premier technological institutes.
We cater to :

- Pharmaceutical industry
- Food and Seed industry
- Chemical Paint Pigment industry
- Hospitals and Healthcare product industry
- Automobile industry
- Commercial establishments, banks, financial institutions.

Copyright © 2012 Crawller.com. All rights reserved unless otherwise mentioned.
Designed by Alokation, Designer Crawller Group.

(b) Rendered test page screenshot

(c) F = 89.9, L = 800, C = 1.0, Δ = 0

(d) F = 27.0, L = 20, C = 0.0, Δ = 10%

Fig. 3: Illustrative example for impact of PID parameters

given difference pixel, ⟨x, y⟩. The "results refiner" component refines the set, E, of HTML elements obtained from R-tree traversal to include additional likely faulty elements. This analysis is based on heuristics, such as (1) for $e \in E$, if all of its children are hidden, then add the children to E, (2) if $e \in E$ has a hidden sibling and all of the siblings after that are in E, then add the hidden sibling to E, (3) if $e \in E$ has all of its siblings in E but not the parent, then add the parent in E, (4) if $e \in E$ has only one child that is not in E, then add it to E, (5) if $e \in E$ is a <select> tag, then add all of its children, the <option> tags, to E. Similarly, if $e \in E$ is an <option> tag, then add its parent, the <select> tag, to E. This refinement analysis follows the idea of a worklist algorithm and terminates when E stabilizes.

The *result set processing* module uses OpenCV for providing sub-image searching capability in the cascading heuristic and uses the element's DOM information stored in the R-tree node for computing the other heuristics.

The *output generator* module writes a text file of the ordered list of HTML elements grouped by clusters and marks the difference pixels belonging to different clusters in different colors on the visual differences image for easy identification.

A. Challenges Faced

The first challenge we faced was in normalizing the oracle image and the test web page screenshot for visual comparison. The normalization process is important to ensure a fair and accurate comparison of the two images. For this, the first step was to ensure that page scrolling does not eliminate visible portions of the page. The screenshots library of Selenium provides us with this facility. The second step in the normalization process is to ensure that the browser viewport size is set to match the oracle image's dimensions (i.e., width and height). Viewport is the rectangular visible portion of the canvas seen on the browser where the HTML page is rendered. This poses difficulties as Selenium only allows size to be set for the browser window, not for the viewport directly. Setting the browser window size to the oracle image's dimensions is not useful, as it includes the viewport plus the browser border containing the browser header, footer, and vertical scroll bar. Selenium captures the viewport content when taking a screenshot of a rendered web page, which in this case would be smaller than the oracle image. To add to the difficulty, the browser border width is platform and version dependent. For finding the correct browser window size dynamically, we used the following approach: (1) resize the browser window to the oracle image's size and take a screenshot, (2) if screenshot width == oracle width, then return the browser window size, (3) else increment the browser window size by 1, take a screenshot and go to (2). Note that we are only matching the viewport's width to the oracle image's width as the browser height does not play a role while capturing a screenshot, since Selenium takes screenshot by removing scrolling.

The second challenge we faced was in finding the optimum values for the configurable parameters, field of view (F), luminance value (L), and color factor (C), as well as the appropriate threshold value (Δ) for the PID technique. We had to do this in order to find only the human perceptible presentation failures, and not employ a strict image comparison in the detection phase. To address this challenge, we ran WebSee multiple times adjusting the different parameters and observed the set of resultant difference pixels and faulty HTML elements. We observed that with F set to 89.9, L set to 800, C set to 1.0, and Δ set to zero, PID required a pixel-perfect match. Consider the example shown in Figure 3 for a real-world mockup used as the oracle and a screenshot of the rendered test web page. Comparing the two images using the above mentioned parameter values of PID produced a difference image shown in Figure 3c, where the red color marks a difference pixel and the blue color marks a matched pixel. As can be seen, almost the entire image, precisely 84%, is shown as a difference, leading to a lot of false positives. After multiple experiments, we found the optimum values of F to be 27.0, L to be 20, C to be 0.0, and Δ to be 10% of the oracle image area. The PID result obtained with these values is shown in Figure 3d, which as can be seen now contains only the human perceptible differences.

The third challenge we faced was in finding an efficient pixel to HTML element mapping technique for the localization phase. Finding an efficient technique is important since this forms the core piece of WebSee. Browser accessible APIs, such as JavaScript and DOM can be used to track a given pixel, $\langle$x, y$\rangle$, to a rendered HTML element. Javascript provides a function elementFromPoint(x, y) for this. However, there are two problems with this approach. The first problem is that the coordinates, $\langle$x, y$\rangle$, need to be provided as relative values with respect to the upper-left corner of the browser's viewport, whereas the difference pixels obtained from the detection phase after comparing the images of the visual renderings of the oracle and the test web page are in absolute coordinates. The second problem is that elementFromPoint(x, y) gives inaccurate results in certain scenarios as it only returns the topmost element which lies under the given pixel coordinates. For example, this causes a problem when the faulty element is not the topmost element, but is visible through the transparent background of the topmost element. The second approach of using the DOM for tracking a pixel to HTML element using elements' bounding rectangle information has a time complexity of O(n), as the DOM tree models parent-child relationships based on syntax, not visual 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. To address the above problems, we found the R-tree data structure to be suitable for mapping a pixel to HTML elements as it can be used to model the layout structure of the rendered web page, allowing tracking of absolute coordinates to HTML elements. An R-tree also returns all of the HTML elements containing a given pixel, giving more complete results. Since an R-tree is height balanced, each traversal has a complexity of only O(log n). Since browser windows can contain tens of millions of pixels, this difference is quite significant, precisely R-tree provided a speedup of 12x.

The fourth challenge we faced was in handling multiple failures in a web page. Though the detection and localization phases were able to identify all of the faulty elements, the result set processing phase could not group and order the faulty elements accurately. To address this challenge, we implemented a divide and conquer approach for multiple faults, using a clustering technique to group difference pixels that are near to each other, then finding the HTML elements corresponding to the difference pixels in every cluster and ordering them within the cluster. The intuition behind this was that difference pixels that are located close to each other are likely to be caused by the same fault. Using popular clustering techniques, such as K-means, was not suitable as they required the number of clusters as input, and the potential number of presentation failures existing in a page cannot be known in advance. To solve this problem, we used DBSCAN, as it decides the number of clusters based on the density distribution of the given data points.

V. Evaluation

In this section we summarize the results of our empirical evaluation obtained by running WebSee. A formal discussion of the results can be found in [12].

For evaluating WebSee, we measured its accuracy, quality of localization, and runtime by running it on several hundred faults seeded into well-known web apps, including Gmail, Craigslist, Oracle, Virgin America, and USC CS Research. We also evaluated WebSee in the context of a study, where users had to manually perform the detection and identification of the potentially faulty elements. We performed another case study using real-world mockups provided by an industrial partner to gauge WebSee's performance in a realistic mockup development scenario.

The complete list of web pages used as subject applications is shown in Table I. We selected these subjects because they are representative of the different implementation technologies and layouts that are popular across all modern web applications. The size of each page (in terms of the total number of HTML elements) is shown for each subject under the column "size". Subject applications in set 1 were used for the real mockups

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

TABLE II: Empirical Evaluation Results

		Accuracy	Quality			Timing
App	#T	Localize	Result	Rank	Dist	Total
		(%)	Set Size			Time (s)
Gmail	52	92	12 (16%)	2	4	7.2
USC CS Research	59	92	17 (5%)	5	6	149.2
Craigslist	53	90	32 (3%)	7	3	83.6
Virgin America	39	97	49 (5%)	8	12	180.9
Java Tutorial	50	94	8 (5%)	2	5	14.5

case study and set 2 was used to seed faults to create artificial test cases.

The results of WebSee's empirical evaluation for the experiment with seeded presentation faults are shown in Table II. The number under column $\#T$ shows the test cases generated for each subject application. To create the test cases, we used a random seeding based approach that injected presentation failures into the subject applications. We used this approach of creating synthetic test cases since obtaining a large number of real-world mockups of web applications that cover a wide variety of presentation failures was difficult. However, we were able to obtain a limited set of real-world mockups from our industrial partner, the evaluation of which we will explain subsequently. To seed the presentation failures and generate artificial mockups for synthetic test cases, we used the following process for each subject application p: (1) download p and all of the files required to display it, (2) take a screenshot of p rendered in a browser to serve as the oracle, (3) generate a set P' that contains variants of p, where each variant contains a unique randomly seeded presentation fault. We discarded those variants of the subject application that did not show a perceptible visual difference. Thus, each test case of a subject application comprised of the oracle image and a presentation failure injected variant with a perceptible visual difference. We then ran WebSee on all of the test cases and obtained results for $Accuracy$, $Quality$ of localization, and $Timing$.

As the results in Table II show, WebSee was able to return a set of potentially faulty HTML elements that contained the original seeded fault 93% of the time, indicating a strong localization accuracy. We also computed the detection accuracy of WebSee in the form of a sanity check to validate the correct implementation of the PID technique, and as expected WebSee was able to detect all of the presentation failures. The quality of the localized result set of HTML elements was computed using the median values of the three metrics, *Result Set Size*, *Rank*, and *Distance*. As can be seen from Table II, WebSee reported an average median result set size of 23 HTML elements, which was, on average, about 10% of each app's total element count. The $rank$ metric indicates the effectiveness of the result set processing phase. WebSee reports an average median rank of 4.8, which means that in 50% of the cases the developer will have to inspect less than five HTML elements, which is only 2% of the total elements present in the average subject page. The $distance$ metric is used to show the effectiveness of WebSee for the 7% of the test cases where the correct faulty HTML element was not present in the result set. WebSee reports an average median distance of 6, which is the DOM distance of the reported HTML elements in the result set from the correct faulty HTML element, implying that the correct faulty element is in close vicinity of the reported elements. WebSee's total running time shown under the column *Total Time*, on average, was reported to be 87 seconds.

We also evaluated WebSee in the context of a user study, where the users had to manually detect and localize presentation failures. The users were only able to visually detect 76% of the failures and could generate a set containing the original seeded fault only 36% of the time. WebSee was also much faster, and only needed an average of 87 seconds to perform this analysis versus an average of 7 minutes for the users.

To validate the reliability of the results obtained from seeded presentation faults, we performed a case study with three real-world mockups obtained from an industrial partner to detect and localize differences between the mockups and the deployed versions of the web pages. We observed that WebSee was able to detect all the presentation failures and return sets that included all of the expected HTML elements. For each of the subject applications, 45% of the faulty elements were listed in the top 5 and 70% were within the top 10.

VI. Related Work

Preliminary work [11] by the authors in the field of debugging presentation failures showed the feasibility of using image processing for detecting presentation failures automatically. However, it used a simplistic pixel-to-pixel image comparison approach for detection, which as seen in Section IV is likely to produce a high number of false positives. In contrast to this, WebSee used a more sophisticated computer vision based technique, PID. The preliminary work also did not have any support for handling dynamic portions of modern web pages, which was addressed in WebSee with the Special Regions Processing phase.

Work in the field of XBT focuses on finding XBI, which occurs when a web page is rendered inconsistently in different browsers causing serious usability and appearance failures. A proposed technique [5] can detect XBIs effectively by comparing the rendered DOMs of a web page in a reference browser and a test browser. However, as discussed in Section III, this technique is not useful in the mockup-driven development scenario, as a reference DOM structure of the web page is not available. Use in regression debugging is also limited to scenarios where the DOM structure has not changed significantly. A commercial tool, Browserbite [20], uses image comparison on screenshots of a test web page that is rendered in different browsers for finding XBIs. Although this technique is similar to our approach, in terms of using image comparison, Browserbite cannot localize the faulty HTML elements. Also, it uses a human-based classification to find true positives from the observed visual XBIs, which can potentially lead to false positives.

Techniques based on textual differencing (e.g., `diff`) of the HTML source or DOM comparisons [5], [15], [8] are not applicable in the mockup driven development scenario and are of limited use in the regression debugging scenario, as explained in Section III. There are two problems with these techniques. First, only a textual difference between the two DOM trees does not necessarily imply that there is a visual difference, as there are often several ways to implement the styling of HTML elements to make them appear the same when rendered (e.g., when a <table> is converted to a table-less layout with <div> tags.) Second, the lack of a textual difference does not imply that there are no presentation failures. For example, a failure can occur without any textual change to an <img> tag if the tag does not specify size attributes and the dimensions of the image file change on disk.

Browser plug-ins, such as "PerfectPixel" for Chrome and "Pixel Perfect" for Firefox assist developers to manually detect pixel-level differences by overlaying a semitransparent oracle image. In contrast, WebSee is fully automated for detection and localization.

Memon and colleagues [16] have done substantial work in the area of GUI testing. These techniques differ from WebSee 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 [7] can detect presentation failures caused by unsupported tags, but not failures related to application specific appearance properties.

Lastly, recent work in the field of program slicing of picture description languages (e.g., postscript) also uses a visual-based analysis [21], [22]. However, this work is not focused on detecting presentation failures in web applications.

VII. Conclusion

In this paper, we presented our tool, WebSee, a fully automated solution for debugging presentation failures. WebSee uses computer vision algorithms for comparing a browser rendered test web page and its oracle image to find visual differences. It then builds a rendering map of the test web page and uses the identified visual difference pixels to find the potentially faulty HTML elements, which are then ranked in order of likelihood of being faulty. WebSee also facilitates the handling of the dynamic nature of modern web applications, by allowing testers to specify regions that will contain dynamic text or images. WebSee's architecture, implementation details, and usage scenarios were also presented.

VIII. Tool Availability

The source code of WebSee is available at http://github.com/sonalmahajan/websee for public download and use. We have also uploaded a small two minute demo video to YouTube (http://youtu.be/VY3s0LeDC8o), which shows an early implementation of WebSee.

References

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

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

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

[4] 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*, CHI '10.

[5] 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)*.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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. To Appear.

[13] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09.

[14] 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*, DIS '00.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] S. Yoo, D. Binkley, and R. Eastman. Seeing Is Slicing: Observation Based Slicing of Picture Description Languages. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14.

[22] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent Program Slicing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014.