Detecting and Localizing Visual Inconsistencies in Web Applications

Sonal Mahajan*, Krupa Benhur Gadde†, Anjaneyulu Pasala†, William G. J. Halfond*
*University of Southern California, Los Angeles, USA, {spmahaja, halfond}@usc.edu
†Infosys Ltd., Bangalore, India, {krupa_bg, Anjaneyulu_Pasala}@infosys.com

***Abstract*—Failures in the presentation layer of a web application can negatively impact its usability and end users' perception of the application's quality. The problem of verifying the consistency of a web application's user interface across its different pages is one of the many challenges that software development teams face in testing the presentation layer. In this paper we propose a novel automated approach to detect and localize visual inconsistencies in web applications. To detect visual inconsistencies, our approach uses computer vision techniques to compare a test web page with its reference. Then to localize, our approach analyzes the structure and style of the underlying HTML elements to find the faulty elements responsible for the observed inconsistencies.**

I. Introduction

The increase in usage and popularity of web applications has also led to increased sophistication in the technologies used to define and build their client-side User Interfaces (UIs). This increased sophistication allows the UIs to provide more features and a richer user experience, but also makes the UIs more complex to develop and test. This is problematic for companies as UI related failures can negatively impact a site's usability. And studies have shown that the visual appearance of a web site can impacts users' perception of its trustworthiness and the quality of the services or products it delivers [21], [8]. One aspect in particular, *visual consistency* – having the same styling for the website-wide layout components (e.g., header and footer) across all of the pages in a web application – is important from the perspective of the website's User Experience (UX), usability, and branding. Developers use repeated layout components, such as headers, footers, and navigation menu bars, to make it easy for users to navigate in a consistent manner between pages, and styling, such as colors and logo placement, to reinforce branding efforts.

The use of visual consistency is a widespread development practice and it is easy to assume that its ubiquitous usage means that it is easy to implement. In fact, modern web application architectures, such as model-view-controller, and frameworks that leverage templates help developers ensure visual consistency. However, the success of these techniques is directly tied to the expertise of the developers who employ them. Furthermore, when a Visual Inconsistency (VI) – a discrepancy between an area of a page and its intended (visually consistent) appearance – occurs, developers must still spend enormous time debugging the UIs. Identifying the faulty HTML elements in a UI is challenging. Modern web pages can contain several hundred HTML elements, each of which can have dozens of CSS style properties that affect its rendering. Furthermore, rendering effects, such as floating elements, overlays, style inheritance, and dynamic sizing, makes it difficult to determine which HTML elements are responsible for an observed failure. Although UI inspection tools, such as Firebug, are available, the debugging process that uses these tools is still manually driven. For web applications that contain a large number of pages, verifying visual consistency is a time consuming process and its thoroughness can be affected by time and cost constraints.

To address this problem, we propose an approach to automatically detect and localize VIs in web applications. The inputs to our approach are a reference page that exemplifies the visual theme of the site and a set of areas on that page whose visual appearance should be checked for consistency on other pages of the site. Our approach checks the other pages in a website to detect if they are visually inconsistent with the specified areas. To do this our approach uses computer vision based techniques to compare the specified areas and then analyzes the structure and styles of the underlying HTML elements in those areas to find the faulty HTML elements.

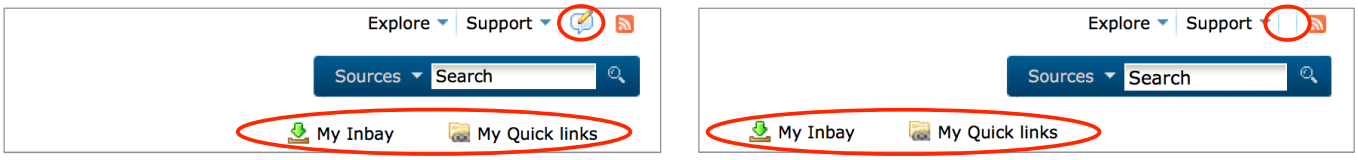
II. Our Approach

The goal of our approach is to automatically detect and localize VIs present in different web pages of a web application. Our approach applies techniques from the field of computer vision [23] to detect VIs, and then analyzes the VI areas to localize them to HTML elements in the page.

Our approach takes three inputs. The first input is the reference web page (R) that specifies the visual correctness properties. The second input is a list of test web pages (T) to be tested for visual consistency against R. The third input is a set of areas (M) of R to be tested for visual consistency across T. The form of R and each $t \in T$ is a URL that points to either a location on the network or file system where all HTML, CSS, JavaScript, and media files referenced by the web page can be accessed. The form of each $m \in M$ is a rectangle that specifies the location of the area to be checked for visual consistency in terms of the x-y coordinates of its upper left hand corner and its width and height.

From a high level, our approach can be described as having two phases. The first phase, detection, compares the visual representations of R and t, in the marked areas, M, to detect

Sonal Mahajan worked as an InStep Intern at Infosys Ltd., during the development of iVCC



(a) Reference page (R) (b) Test page (t)

Fig. 1: Illustrative example

VIs. The second phase, localization, first identifies a set of underlying HTML elements in the detected VI areas using the rendering maps of R and t. It then finds pairs of matching HTML elements from the sets of underlying HTML elements based on heuristics, such as ID and XPath. The pairs of HTML elements are then analyzed to identify the faulty HTML elements responsible for the detected VIs. In the remainder of this section, we provide more details of the two phases.

To illustrate our approach, we use Figure 1 that shows an excerpt from the header portion of a web application. Figure 1a shows the home page of the web application and is used as the reference page (R) specifying the intended appearance of the header. Figure 1b shows the page under test, which is another page in the application. As compared to the reference pages header, there are two differences in the test page: (1) the feedback-icon is missing in the top-right menu links and (2) the bottom menu links are misaligned.

A. Phase 1: Detection

The first phase of the approach detects VIs in different web pages of a given web application. A naive way to detect VIs is to perform a textual comparison of the HTML structure of the web pages. There are two problems with this technique. First, only a textual difference between two pages does not necessarily imply that there is a visual difference (VI). This is because (1) there are often several ways to implement the styling of HTML elements to make them appear the same on rendering (e.g., margin and padding can be used to interchangeably to achieve the same visual effect or setting the CSS property of font-size to 'small' will the have the same visual effect as setting it to 13px) or (2) a page may have been restructured in a way that did not translate into a visual difference (e.g., when a ⟨table⟩ is converted to a table-less layout with ⟨div⟩ tags.)

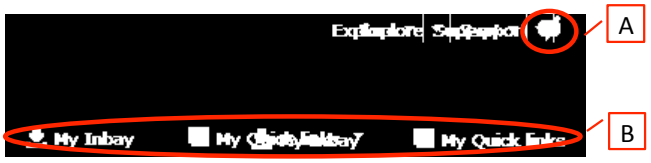


Fig. 2: Visual differences detected between R and t

To address these problems, our approach uses our prior work, WebSee [11], [12], for detecting VIs. WebSee uses Perceptual Image Differencing (PID) [23], a computer vision technique, to compare screenshots of a rendered reference page against screenshots of the rendered web pages under test and detect differences.

For detecting VIs, our approach first captures screenshots of the browser rendered reference page, R, and the test web page $t \in T$. The approach then crops the screenshots of R and t to the given marked areas M, and runs WebSee's PID based detection on the cropped screenshots to get a set of visual difference pixels (*DP*). Using the PID technique of image comparison helps our approach in identifying only the human perceptible visual differences (VIs), ignoring small pixel level differences that many development teams likely consider insignificant.

To illustrate phase 1, consider the pages R and t shown in Figure 1a and Figure 1b, respectively. The approach compares the screenshots of R and t using WebSee and identifies visual differences between them as shown in the two areas in Figure 2. Here the white dots represent difference pixels.

B. Phase 2: Localization

The second phase of the approach localizes the observed VIs to faulty HTML elements in the web page. It finds the faulty HTML elements by first finding the sets of underlying HTML elements in the observed VIs areas from the reference and test web page, respectively, and then analyzing these sets to identify the HTML elements responsible for causing the observed VIs.

Phase 2 takes as input the set of difference pixels (*DP*) identified for areas of visual differences (VIs) by phase 1, the reference page R, and the set of test pages $t \in T$. The localization phase is comprised of three steps: (1) finding sets of underlying HTML elements in the areas of visual differences, (2) mapping corresponding HTML elements from the sets of underlying elements, and (3) identifying faulty HTML elements.

The first step in the localization phase is to identify a set of underlying HTML elements from R and t, respectively, for the areas of visual differences identified by phase 1. To do this, the approach uses our prior work, WebSee [11], [12]. WebSee builds rendering maps of a web page to identify the HTML elements that belong to the detected visual difference areas. WebSee creates rendering maps by first building Rectangle Trees (R-trees) [7] of R and t based on their Document Object Model (DOM) information. An R-tree is a popular data structure from the spatial database community that is used for storing multidimensional data. In the context of HTML pages, WebSee uses the bounding boxes (rectangles) of rendered HTML elements as the multidimensional data to be stored in the R-tree. Thus, an R-tree of an HTML page describes rendering maps between the HTML elements and the pixels defined by them. The R-tree is then traversed for every

pixel $\langle$x, y$\rangle \in$ *DP* to find the HTML elements that contain $\langle$x, y$\rangle$. The obtained HTML elements are added to the set E_R representing a set of underlying HTML elements in the reference page, R. The same procedure is repeated for the test page, t, to obtain the set E_t.

The second step finds matching pairs of elements from the two sets, E_R and E_t, obtained from step 1. A mapping of corresponding elements is required in order to compare the matching elements from R and t to find the faulty HTML elements. This mapping is established based on heuristics, such as XPath, ID, and tagname, that help define similarity between a pair of HTML elements. The approach iterates over every $e_t \in E_t$ and computes a similarity score between e_t and every $e_r \in E_R$. The element e_r with the highest similarity score is then selected as the matching element for e_t and the pair is added to the map of corresponding elements, E_{map}.

We now explain the eight heuristics used for calculating the similarity score between e_t and e_r. **Heuristic 1** is based on the ID of the HTML elements. If the ID of e_t and e_r is exactly the same, then the *similarityScore* is incremented by one. **Heuristic 2** compares the tag names of e_t and e_r. If the tag names are exactly the same, then the *similarityScore* is incremented by one. **Heuristic 3** compares the class names defined in the `class` attribute of e_t and e_r. (These are the names of the CSS classes that define the style rules for rendering the HTML elements.) If the class names match, then the *similarityScore* is incremented by one. **Heuristic 4** compares the XPaths of e_t and e_r using the Levenshtein distance. The Levenshtein distance between two strings, str_1 and str_2, is the minimum number of edits required to transform str_1 into str_2. The distance obtained from *getLevenshteinDistance()* is added to the *similarityScore*. **Heuristic 5** is based on content similarity. This heuristic applies to the HTML elements that contain text content, $\langle$`a`$\rangle$ that contain the target URL in the *href* attribute, and $\langle$`img`$\rangle$ that contain the image file path in the *src* attribute. If the text, href, or src content of e_t and e_r matches, then the *similarityScore* is incremented by one. **Heuristic 6** matches the location or positioning of e_t and e_r when rendered in a browser by comparing the $\langle$x, y$\rangle$ coordinates of the upper left corner of their bounding boxes. If the location coordinates match, then the *similarityScore* is incremented by one. **Heuristic 7** matches the size of e_t and e_r when rendered in a browser by comparing the width and height of their bounding boxes. If the size matches, then the *similarityScore* is incremented by one. **Heuristic 8** checks the alignment of e_t and e_r when rendered in a browser. The approach checks the top alignment by comparing the y_1 values of the bounding boxes of e_t and e_r given by $\langle x_1, y_1, x_2, y_2\rangle$, where $\langle x_1, y_1\rangle$ represents the upper left corner and $\langle x_2, y_2\rangle$ represents the bottom right corner of a rendered element. Similarly, bottom, left, and right alignment is checked by comparing the y_2, x_1, and x_2 values, respectively. If any one of the alignments is found matching, the *similarityScore* is incremented by one.

The final step in the localization phase analyzes the matched element pairs from step 2 to identify the faulty HTML elements responsible for the observed VIs. To do this, the approach compares the structure and style of the paired elements, $\langle e_r, e_t\rangle \in E_{map}$, and if a difference is found adds e_t to the set of faulty HTML elements, E_{faulty}. The approach compares the structure of e_r and e_t based on their *innerHTML*. The *innerHTML* of an element, e, gives the HTML content of all of the descendants of e. In other words, *innerHTML* of e is a sub-tree of the DOM-tree of the HTML page, with e as the root of the sub-tree. For comparing the styles of e_r and e_t, the approach first obtains all of the styling information defined by e_R and e_t. The styling information for an element is given by the CSS properties and HTML attributes along with their values. The CSS properties and HTML attributes control the appearance of an element when rendered in a browser.

Referring back to the example, WebSee reports the HTML elements containing the difference pixels in the areas A and B in sets E_R and E_t for the pages R and t, respectively. Then the corresponding HTML elements in the visual difference areas of A and B (Figure 2) of E_R and E_t are mapped and added to E_{map}. For instance, referring to area B, the bottom menu links defined by the $\langle$`ul`$\rangle$ and $\langle$`li`$\rangle$ elements in E_R and E_t are mapped based on similarity score and added to E_{map}. Similarly, for area A, the $\langle$`li`$\rangle$ elements and their corresponding children $\langle$`a`$\rangle$ elements are mapped. Then the VI in the bottom menu links contained by the $\langle$ul$\rangle$ element is reported as a faulty element and added to the set E_{faulty}, since e_t's CSS property *width* has a different value compared to e_r. For the VI of missing feedback-icon in the top-right menu links, the $\langle$`li`$\rangle$ element reported as the faulty HTML elements, as a structural discrepancy is reported — its child element $\langle$`a`$\rangle$ is missing from e_t as compared with e_r.

III. Related Work

Prior work [11], [10], [12], [9], [13] by the authors in the field of visual testing of web applications focuses on handling presentation failures — a discrepancy between the intended appearance of a website and its actual appearance. In our approach, we build on this prior work and extend it by introducing new techniques to enable it to detect and localize VIs. The localization technique of WebSee identifies all of the underlying HTML elements defined by the areas of visual differences. However, this is likely an over-approximated set and needs to be further analyzed to identify the precise set of faulty HTML elements responsible for causing VIs (steps 2 and 3 of phase 2).

Another prior work [3] by the authors detects and localizes presentation failures by comparing visual relationships of HTML elements in internationalized web pages. However, this technique could lead to false negatives as it cannot detect inconsistencies related to styling of the page.

Another group of techniques detect Cross Browser Issues (XBIs) — a type of presentation failure that occurs when a web page is rendered inconsistently across different browsers. Roy Choudhary and colleagues [18], [4], [5] find XBIs by comparing the DOM structure of the same web page rendered in a reference browser and a test browser, then report a set of HTML elements likely to have caused the XBIs. Although

a similar technique of comparing DOM structures of the reference and test web page can be used for detecting VIs, it could lead to false positives as there are often several ways to achieve the same rendering effect by using different HTML elements (DOM nodes).

Browser plug-ins, such as "PerfectPixel" [1] for Chrome and "Pixel Perfect" [2] for Firefox, assist developers in finding visual differences by overlaying a screenshot of the reference web page on the rendered test web page. These plugins can help developers find VIs, however this is a manual process. Also, after detecting a VI, developers have to manually identify the faulty HTML elements. In contrast, our approach is fully automated. Another tool called, "Fighting Layout Bugs" [20], automatically detects application independent presentation problems, such as text which is very near or overlaps a horizontal/vertical edge of an element, text which is not readable because of too low contrast, and invisible elements. However, the Fighting Layout Bugs tool cannot be used to find VIs as they are not application independent.

Work in the area of GUI testing by Memon and colleagues [14], [22], [15], [16] tests the behavior of a software system based on event sequences triggered from the GUI. Unlike our work, their work is not focused on finding visual problems with the GUI, but rather using the GUI as a driver to find behavioral problems with the system under test.

Another group of techniques validate syntax of HTML [19], [17], [6] by checking for malformed HTML code that can cause presentation failures if the rendering browser does not handle them properly. However, these techniques can only detect VIs related to HTML syntax errors and not failures related to application specific appearance properties.

IV. Conclusion

In this paper we introduced a new idea for detecting and localizing visual inconsistencies in web applications. Our approach uses computer vision techniques for detecting the inconsistencies. It then finds the underlying HTML elements in the areas of visual differences using rendering maps built for the web pages and analyzes the underlying elements to find the faulty HTML elements responsible for the observed inconsistencies. Overall, we believe this is a promising technique that can effectively assist developers in debugging visual inconsistencies in web applications.

Acknowledgment

This work was supported by National Science Foundation grant CCF-1528163. We also thank Sri Hari Tulasi, Visual Design Group, Infosys Limited, for his valuable inputs.

References

[1] Perfect Pixel Chrome. http://www.welldonecode.com/perfectpixel/.

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

[3] A. Alameer, S. Mahajan, and W. G. J. Halfond. Detecting and localizing internationalization presentation failures in web applications. In *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016.

[4] 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. IEEE Computer Society, 2012.

[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)*, pages 702–711, May 2013.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] S. Mahajan, B. Li, P. Behnamghader, and W. G. J. Halfond. Using Visual Symptoms for Debugging Presentation Failures in Web Applications. In *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016.

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

[15] 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)*.

[16] 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.

[17] 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.

[18] 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, 2010.

[19] 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, 2012.

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

[21] 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.

[22] 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.

[23] 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.