Finding HTML Presentation Failures Using Image Comparison Techniques

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

ABSTRACT

Presentation failures in web applications can negatively affect an application's usability and user experience. To find such failures, testers must visually inspect the output of a web application or exhaustively specify invariants to automatically check a page's correctness. This makes finding presentation failures labor intensive and error prone. In this paper, we present a new automated approach for detecting and localizing presentation failures in web pages. To detect presentation failures, our approach uses image processing techniques to compare a web page and its oracle. Then, to localize the failures, our approach analyzes the page with respect to its visual layout and identifies the HTML elements likely to be responsible for the failure. We evaluated our approach on a set of real-world web applications and found that the approach was able to accurately detect failures and identify the faulty HTML elements.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Algorithms, Reliability

Keywords

HTML presentation failures; detection; localization

1. INTRODUCTION

Presentation failures in web applications occur when the rendering of an HTML web page does not match its expected appearance. These kinds of failures occur 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. Presentation failures can negatively affect the usability of a web application and reduce the end user's perception of the quality of the application and the services it delivers.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642966.

Detecting presentation failures and identifying the corresponding faulty HTML element is challenging. To determine that a failure has occurred, a tester must examine the rendering of each HTML element in a web page and verify its visual properties against an oracle or specification. Once a presentation failure has been detected, it is also challenging to find the faulty HTML element, since an element's visual appearance is controlled by a complex series of interactions defined by the HTML structure and CSS rules. The use of floating elements, overlays, and dynamic sizing also increases the difficulty of identifying the faulty element.

These challenges have led to new processes and techniques for testing web applications; however, these have limitations that reduce their usefulness. Testing tools, such as Selenium [4], CrawlJax [17], Sikuli [10], and Cucumber [2], can find certain types of presentation failures. However, these techniques require testers to exhaustively specify all correctness properties to be checked. These tools are capable of only verifying the syntactical correctness, and not the actual visual appearance of the web page. Another group of techniques, such as cross-browser testing [11] and GUI differencing [25], can also find presentation failures. However, these approaches assume the existence of a bug-free previous version of the web application and detect failures by comparing against this version. Other approaches, such as Google's Fighting Layout Bugs project [23], search for certain kinds of common presentation failures that are application agnostic, such as overlapping DIV tags, but cannot find application specific presentation failures.

In this paper we present a novel approach for detecting presentation failures and then identifying the corresponding faulty HTML element. Our approach leverages image processing techniques to compare layout designs created by graphic designers against actual screenshots of the rendered web page under test. The approach then builds rendering maps of the web page to identify the HTML elements that are responsible for the detected differences. We carried out a case study of our approach's ability to detect and localize presentation failures in real-world web applications, such as Gmail and PayPal. The results of our evaluation show that it is able to detect the occurrence of presentation failures with 100% accuracy and was able to identify result sets that included the faulty HTML element in over 77% of the cases. Overall, our results were positive and indicate that our approach represents a viable new approach for addressing the problem of presentation failures.

2. MOTIVATION

The need to debug presentation failures arises in many scenarios throughout the development process of a web application. We first describe three such scenarios and then discuss the usage of existing techniques with respect to these scenarios.

2.1 Usage Scenarios

During the initial development of a web application's templates or view components, front-end developers are guided by mockups – highly detailed renderings of the intended appearance of the web application created by graphic designers [18, 15, 19]. The developers are generally expected to create "pixel perfect" matches of these mockups [5]. This can be done by hand or by using one of the numerous and popular WYSIWYG editors, such as Adobe Muse, Amaya, or Visual Studio, to generate template HTML and CSS files. Back-end developers also make changes to these templates by adding dynamic content. During this process, the developers perform *presentation development testing* to ensure that the appearance of the developed pages matches the given intended appearance, and if it does not, to determine which HTML elements in the developed pages are responsible for the failure.

Once initial development is completed, developers may inadvertently introduce faults into a web application while making changes to the code. For example, a developer may refactor code to transition a table based layout to one based on the use of the tag. Such changes are not intended to change the pages' appearance, but a developer error can cause a presentation failure in any page that depends on the faulty modification. The developer must then perform *regression debugging* in order to figure out which elements are responsible for the observed failure. In contrast to the first scenario, in regression debugging the developers have a previously correct version (instead of a mockup) that can serve as an oracle.

Once the development has reached a level of stability, developers will make corrective changes to the code based on bug reports. For example, to resolve user-reported failures. We call this scenario *standard debugging*. For this kind of debugging, the developers must generally reproduce and then debug the failure in house.

Techniques developed by both industry and research help developers in one or more of these scenarios; however, the techniques have limitations that reduce their usefulness or practicality. We now discuss each of these techniques in more detail.

2.2 Limitations of Existing Techniques

A group of techniques is based on comparing two working versions of an application against each other and using the differences to localize faults. Examples of these techniques include Cross Browser Testing (XBT) [11, 6], GUI differencing [25], and automated oracle comparators [21]. These techniques compare tree based representations (e.g., GUI tree or HTML DOM) of the current user interface against a reference version, and use differences between these representations to find failures and faults. All of these techniques can be used to debug presentation failures; however, their use imposes limitations on the problem domain that reduces their usefulness.

The first limitation is that these techniques require that the tree based representation of a (previous) correct version must exist so that it can be compared against the version to be debugged. This assumption means that the differencing techniques could only help in regression debugging. Second, to use these techniques for regression debugging requires that the tree based representations be similar enough that a comparison is possible without generating too many false positives. If a refactoring significantly changes the code structure, the techniques would not be able to identify which elements in the tree based representation should be compared. Third, the techniques assume that a difference between the tree based representations implies a failure and, conversely, that a failure implies a difference in the tree based representations. This is not necessarily the case. A difference without a presentation failure can occur for two reasons: (i) there can be different ways to implement the styling of an HTML element and (ii) a page may be refactored in a way that does not translate into a visual difference. Conversely, an example of when a failure may occur without a difference is if an <img> tag does not specify size attributes and the dimensions of the supplied image change on disk.

Several techniques, Selenium [4], Cucumber [2], and Crawljax [17], allow developers to programmatically specify invariants that will then be checked against a web page's DOM. There are three drawbacks to these approaches. First, they require testers to exhaustively specify each visual property to be checked. The size of this set could, at minimum, grow linearly with the number of HTML elements in the page, which could require several thousand invariants for a typical page. Second, the invariants are expressed in terms of an element's CSS or HTML syntax instead of its actual visual appearance. Thus checking an element's invariants does not necessarily ensure that the rendering of the element conforms to the expected visual appearance. For example, it is possible for the appearance of an element to be affected by an HTML ancestor element's CSS properties. Finally, these techniques do not help with localization of a failure unless the fault occurs in the same element(s) checked by an invariant.

Two other approaches, "Fighting Layout Bugs" (FLB) [23] and Sikuli [10], employ a different approach. FLB detects general application-independent correctness properties, such as overlapping text regions and poor color contrast, by looking at a screenshot of the web page. The drawback to this approach is that it cannot detect failures that are application specific and only marks up the screenshot to show where the failure occurred. The testers must still identify the responsible HTML elements. Sikuli is an automation framework that uses sub-image searching to identify and manipulate GUI controls in a web page. Although not intended for verification, one could provide a set of screenshots of each GUI control and use Sikuli to ensure that they match (i.e., there are no presentation failures.) However, since Sikuli uses a sub-image based search of the page, it could match the provided screenshots against any portion of the page, not necessarily the intended region. This means it would be ineffective if there were visually identical elements in the page. Furthermore, Sikuli only provides an element after a positive match; therefore when there is a failure, no match will be made and no element(s) will be provided to the testers to help with localization.

3. APPROACH

The goal of our approach is to automatically detect and localize presentation failures in web pages. To do this, our approach applies techniques from the field of image processing to analyze the visual representation of a web page, identify presentation failures, and then determine which elements in the HTML source of the page could be responsible for the observed failures. We briefly describe the inputs to our approach and its phases at a high-level and then, in the following subsections, discuss the phases in more detail.

The approach takes two inputs, which are provided by the developers or testers. The first input is the web page to be analyzed for presentation failures. The form of the test web page can be a URL that points to either a location on the network or filesystem where all HTML, CSS, JavaScript, and media files of the test web page can be accessed. The second input is the oracle that specifies the visual correctness properties of the test web page. The form of the oracle is an image that can be either a mockup or a screenshot of a previously correct version of the web application.

From a high-level, our approach can be described as having two phases. The first phase, detection (Section 3.1), compares the visual representations of the test web page and the oracle to detect a set of differences. The second phase, localization (Section 3.2), analyzes the rendering of the test web page to identify the set of HTML elements that define the previously detected set of differences. The identified set of HTML elements is provided as an output to the developer.

3.1 Presentation Failure Detection

The first phase of the approach detects presentation failures by comparing the screenshot of a web page under test, as rendered in a browser, with its expected appearance, the oracle. The approach first captures a visual representation of the test web page. Then, the visual differences between the visual representations of the oracle and the test web page are identified. The output is a set of of pixels that represent the differences between the two representations.

To capture the visual representation, the approach takes a screenshot of the browser window that is rendering the page. Since this visual representation will be compared against the oracle, the primary challenge is to ensure that the screenshot is taken under the same conditions as the oracle. Otherwise, differences, such as screen resolution, between the platform on which the oracle was developed and the testing platform can affect the image comparison. To facilitate this we developed a normalization process for taking the screenshots. This was primarily an engineering challenge, so we only briefly summarize this aspect of the approach. We identified three primary visual aspects that had to be controlled for in order to enable an accurate comparison: (1) dimensions of the images (i.e., height and width); (2) amount of zoom in the browser window; and (3) size of the browser viewport to ensure that page scrolling did not eliminate portions of the page. The screenshot library (Selenium [4]) provides functionality to configure these visual aspects, so our approach only tracks and reproduces the oracle's environment.

To identify the visual differences between the oracle and screenshot of the page, our approach uses a pixel level comparison. At a high-level, the approach carries out the comparison by iterating over each pixel in the oracle and comparing it against the corresponding pixel in the test web page screenshot. If the two pixels are not equivalent, then the x and y coordinates of the pixel are added to the difference set. Two pixels are equivalent if they have the same color and saturation values.

3.2 Localization of Faulty Element

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

The first step is to build an R-tree model of the test web page that will be used to map difference pixels to HTML elements. An R-tree is a height-balanced tree data structure that is widely used in the spatial database community to store multidimensional information. In our approach, we use the R-tree to store the bounding rectangle assigned to an element when it is rendered in the browser. The approach extracts the bounding rectangle for each element in the test web page via browser-provided APIs. In the R-tree built by our approach, the leaves of an R-tree correspond to rectangles and non-leaf nodes correspond to the tuple $\langle$I, child_pointer$\rangle$, where I is the identifier for the minimum bounding rectangle that groups nearby rectangles, and *child_pointer* is the pointer to a lower node in the R-tree. The HTML elements corresponding to an $\langle x, y\rangle$ pixel can then be found by traversing the R-tree's edges to find the rectangles containing the pixel. Building and searching an R-tree are standard techniques and can be found in [13].

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

After an R-tree has been constructed for the test web page, the approach identifies the HTML elements that can be responsible for rendering the pixels contained in the difference set identified in the detection phase. For a given pixel, it is possible to have more than one responsible HTML element, since CSS allows layers of HTML elements with transparent or overlapping areas. The approach iterates over each pixel $\langle x, y\rangle \in$ the difference set, finds the corresponding HTML elements in the R-tree, and adds them to the result set of potentially faulty HTML elements. Each HTML element in the result set is represented using its XPath identifier, which can uniquely identify an element in an HTML page.

4. EVALUATION

We conducted a case study to evaluate the accuracy of our approach in detecting presentation failures and reporting faulty HTML elements. For this, we utilized the subject

Table 1: Overview of subjects and accuracy of the approach

App	URL	Size	#T	Localization (%)
Gmail	http://www.gmail.com	161	53	79
Craigslist Autos	http://losangeles.craigslist.org/i/autos	70	41	66
Virgin America	http://www.virginamerica.com	1,016	41	78
PayPal	http://www.paypal.com	317	51	84

applications shown in Table 1. We chose these web pages because they are popular real-world web applications, and represent a mix of different implementation technologies and layouts that are widely used across modern web applications. The size of each page (in terms of the total number of HTML elements) is also shown for each subject under the column labeled "size".

4.1 Implementation and Procedure

We implemented a prototype tool built in Java to demonstrate our proof of concept. In the detection (Phase 1) module, Selenium WebDriver is used to take screenshots of the HTML page under test and ImageMagick to compare images and calculate differences. In the localization (Phase 2) module, we use the Java Spatial Index library to help implement the R-trees for the HTML pages and Selenium WebDriver to extract bounding rectangle information.

For our experiment, we created test cases by using an automated fault-seeding technique to mutate the original web page of each subject application with a random presentation failure. For seeding faults, we first manually analyzed the W3C's HTML and CSS specification [3, 1] to identify HTML elements with *visual properties* — HTML attributes or CSS properties that could impact the visual appearance of an HTML element. For example, the CSS property "font-size" applies to elements, such as or <p> that contain text, but not to <img>. Then, the basic process for each subject application web page P was as follows: (1) download from the web all files required to display P in a browser; (2) take an image capture of P to serve as the oracle O; (3) create a set P' that contains variants of P, each with a different randomly seeded presentation fault; (4) run our tool on O and each member of P'; (5) manually verify the detection and localization results generated by our tool.

Our experiment was designed to evaluate the accuracy of performing detection and localization based on our image detection and localization techniques. In particular, we focused on the presentation development testing and regression debugging scenarios since these could be completely automated without developer intervention. Since we did not have access to mockups, templates, or multiple versions of our subject applications, we used the screenshots taken in step 2 of the protocol as a stand-in for the mockup or screenshot of the (previous) correct version. This allowed us to simulate the presentation development testing and regression debugging scenarios for evaluation purposes.

The results of this experiment are shown in Table 1. The column "#T" shows the total number of test cases used per application. Accuracy for localization is shown under the column labeled "Localization." For detection accuracy, we calculated the percentage of test cases in which our approach could detect that a presentation failure had occurred. For localization accuracy, we calculated the percentage of test cases in which the expected faulty element was reported in the result set.

4.2 Discussion

Our tool was able to detect the presentation failures in all testcases. Accuracy for localization was also high – the tool was able to identify a result set that contained the faulty element for, on average, 77% of the test cases. We investigated the result sets in order to determine the reason that 23% of the result sets did not contain the faulty element. We found that two reasons dominated. The first of these is that detecting the results of deletion or addition of an HTML element was not possible with our approach. When an element is deleted, it is no longer present in the web page's R-tree; therefore it cannot be reported in the result set. When an element is added, the new element is present in the R-tree and reported as faulty. However, it is not part of the expected element set, since it was not present in the original web page.

The second reason the tool did not identify faulty elements in the result set was that some seeded faults only changed the appearance of *other* elements. That is to say, they did not change the appearance of the element into which they were seeded. For example, a seeded fault that set the CSS position property to $fixed$ caused the surrounding elements to be re-positioned. This made the HTML elements surrounding the seeded element, but not the seeded element, appear to be a difference.

5. RELATED WORK

The most closely related work is discussed in Section 2. In this section we elaborate on the contrast with cross-browser testing (XBT) and discuss more broadly related work.

XBT techniques by Roy Choudhary and colleagues [11] compare the rendering of a reference version of a web page in one browser against its rendering in another browser to detect cross browser issues (XBI). As discussed in Section 2.2, this approach is not applicable for presentation development testing and standard debugging. For regression debugging its use would be limited to scenarios where the DOM had not changed significantly and matching elements could still be matched using probabilistic techniques. A visual comparison of certain elements is done using color histograms instead of the pixel by pixel comparison employed by our approach.

Browser plug-ins, such as "PerfectPixel" [7] for Chrome and "Pixel Perfect" [8] for Firefox help developers to detect pixel-level differences with an image based oracle. They overlay a semi-transparent version of the oracle over the HTML page under test, enabling developers to do a per pixel comparison to detect presentation failures. However, they require the developer to manually locate the faulty el-

ements. In contrast, our approach is fully automated for detection and localization.

There has been active research in the field of user interface testing. Memon and colleagues [22] have done substantial work in the area of model-based automated GUI testing. These techniques differ from our approach in that they are not focused on testing the appearance of the user interface, but instead focus on testing the behavior of the system based on event sequences on the user interface. Another work by Eaton and Memon [12] in the field of web applications focuses on reporting HTML tags present in the test web page that are not supported by a specific browser. This can detect presentation failures caused by unsupported tags, however it cannot detect failures related to application specific appearance properties.

There has also been active research in the field of validating HTML for syntax [20]. These techniques check for malformed HTML code that can cause presentation failures if the rendering browser does not handle them properly. However, these techniques can only detect presentation failures related to HTML syntax errors and not failures related to application specific appearance properties.

Recently, techniques to test JavaScript [9] and analyze CSS [16] have been proposed. These techniques deal with specific components of the client side and as such are not meant for detecting presentation failures in a web application. Another technique based on impact analysis of CSS changes across a web site [14] notifies the developer if changes made in a CSS file are introducing new presentation failures in other web pages of the web site. However, this technique relies on the availability of the golden version of all the web pages in the web site and is not able to handle presentation failures caused by changes in the DOM structure of the web page.

Wang and colleagues [24] propose a technique to automatically perform presentation changes in dynamic web applications. Their technique facilitates automatic changes to the server side code generating web pages based on a given presentation change. Though this produces a new looking web page, its appearance is not verified for correctness.

6. CONCLUSION AND FUTURE WORK

In this paper we introduced a new technique for detecting and localizing presentation failures in web applications. Our approach uses image processing techniques to compare a web page rendered in a browser with its oracle and identify difference pixels. Then the approach builds a rendering map of the page and uses the difference pixels to identify and rank a set of likely faulty HTML elements. In the evaluation our approach detected 100% of the presentation failures and was able to locate the faulty element in over 77% of the cases. Overall, these preliminary results are promising and indicate the feasibility of our approach in being able to assist developers by automating the detection and localization of presentation failures.

In future work, we first plan to explore techniques to handle the dynamic nature of modern web applications. Example of dynamic portions of a web application are advertisements, user account information, text from a database, etc. For this, we plan to provide a mechanism by which the users can mark the dynamic portions of the web page and specify the correctness properties that should apply to the respective portions. The idea is to check certain visual properties, such as font, while ignoring the content differences.

Secondly, we plan to relax the expectation of a "pixel-perfect" match of the test web page to the oracle image, as it may not always be desirable. In many cases, small differences may represent concessions to coding simplicity or be within a level of tolerance that the development team does not consider to represent a presentation failure. In these cases, a more relaxed comparison that expresses a "close enough" match is more appropriate. For this, we plan to introduce the notion of a tolerance level while deciding the equivalence of two image pixels.

Thirdly, we plan to prioritize the result set produced as the output of the localization phase. Our approach adds HTML elements to the result set, regardless of whether their presentation failure is independent or dependent on another element. This can be problematic for developers if the size of the result set requires them to inspect many HTML elements. To address this problem, we plan to use heuristics based on the parent-child relationships of the elements and rank them with respect to their likelihood of being faulty.

7. REFERENCES

[1] CSS Reference. `http://www.w3schools.com/cssref/`, Dec. 2013.

[2] Cucumber. `http://cukes.info/`, Dec. 2013.

[3] HTML 5. `http://www.w3schools.com/tags/`, Dec. 2013.

[4] Selenium. `http://docs.seleniumhq.org/`, Dec. 2013.

[5] Front-end Developers Job Postings. `http://www-scf.usc.edu/~spmahaja/front-end-job-postings/`, Apr. 2014.

[6] Mogotest. `http://mogotest.com/`, Apr. 2014.

[7] Perfect Pixel Chrome. `https://chrome.google.com/webstore/detail/perfectpixel-by-welldonec/dkaagdgjmgdmbnecmcefdhjekcoceebi?hl=en`, Apr. 2014.

[8] Pixel Perfect Firefox. `https://addons.mozilla.org/en-us/firefox/addon/pixel-perfect/`, Apr. 2014.

[9] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[10] 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, pages 1535–1544, New York, NY, USA, 2010. ACM.

[11] 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.

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

[14] 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.

[15] 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.

[16] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 408–418, Piscataway, NJ, USA, 2012. IEEE Press.

[17] 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, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[18] 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, pages 263–274, New York, NY, USA, 2000. ACM.

[19] F. K. Ozenc, M. Kim, J. Zimmerman, S. Oney, and B. Myers. How to Support Designers in Getting Hold of the Immaterial Material of Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2513–2522, New York, NY, USA, 2010. ACM.

[20] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[21] 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.

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

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

[24] 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.

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