Root Cause Analysis for HTML Presentation Failures using Search-Based Techniques

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

ABSTRACT

Presentation failures in web applications can negatively impact users' perception of the application's quality and its usability. Such failures are challenging to diagnose and correct since the user interfaces of modern web applications are defined by a complex interaction between HTML tags and their visual properties defined by CSS and HTML attributes. In this paper, we introduce a novel approach for automatically identifying the root cause of presentation failures in web applications that uses image processing and search based techniques. In an experiment conducted for assessing the accuracy of our approach, we found that it was able to identify the correct root cause with 100% accuracy.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Root Cause Analysis

General Terms

Verification, Algorithms, Reliability

Keywords

Search-based software testing, HTML presentation failures

1. INTRODUCTION AND MOTIVATION

Failures in the presentation layer of a web application can negatively impact its usability and end users' perception of the application's quality. In general, these types of failures occur when the rendering of a web application does not match its intended appearance and are caused by faults in the underlying HTML elements or CSS style. Identifying the faulty HTML elements is challenging. Modern web pages can contain several hundred HTML elements, each of which can have dozens of HTML and CSS style attributes that affect its rendering. Furthermore, rendering effects, such as floating elements, overlays, style inheritance, and dynamic sizing, make it difficult to determine which HTML element is responsible for an observed failure. Popular tools, such as Firebug, can help testers by interactively providing CSS information for HTML elements of interest. However, this process still requires testers to manually interact and experiment with HTML elements in order to pinpoint the problematic element and style property; thus making this a potentially labor-intensive and error-prone process.

Existing automated testing techniques for web applications can help to identify faulty HTML elements, but make assumptions that limit their general applicability. Techniques, such as Selenium, Sikuli [1], Cucumber, and Crawljax [6], allow testers to specify correctness invariants that are checked against a web page. The drawback of these techniques is that invariants must be exhaustively specified for every HTML element in order to effectively identify the HTML element responsible for a presentation failure. Other approaches, such as cross-browser testing [2] and GUI differencing [7], avoid the need for specification by assuming the availability of a previous bug-free version of the application. These approaches compare against this version to detect failures and identify the elements responsible for the failure. Although these techniques make assumptions appropriate for their domain, they would not be useful for localizing presentation faults until this "golden" version had been developed and tested.

In this paper, we propose the use of search based techniques to assist testers in identifying the HTML elements and style properties that are responsible for a presentation failure. Our key insight is that image processing techniques can be used to help define fault localization as a search problem and provide fitness functions that can guide the search. Our approach assumes the existence of an image based oracle and then systematically searches the space of possible root causes to identify one that causes the rendering of the faulty web page to match that of the oracle. We performed an experiment on a real world web application to evaluate our approach. The results showed that our approach was able to reliably identify root causes and could outperform a random search based approach.

The rest of this paper is organized as follows. First, in Section 2 we describe our prior work in presentation failure detection that sets the context for and motivates the techniques proposed in this paper. In Section 3 we describe our approach in more detail. We present our accuracy experiment in Section 4 and discuss future work in Section 5.

2. BACKGROUND

In prior work [4], we developed WebSee, an automated ap-

SBST '14, June 2 - June 3, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2852-4/14/06 ...$15.00.

proach for detecting presentation failures in web pages and identifying a set of likely faulty HTML elements. WebSee leverages image processing techniques to compare the web page design against actual screenshots of the rendered web page under test. The current state of the practice is that the web page designs are provided as images by graphic designers. The developers are then expected to implement web pages to match the web page designs [8]. Thus, the assumption of availability of the oracle (web page design) is reasonable in this domain. However, differences in the platform on which the oracle was developed and the testing platform can affect the image comparison. To address this challenge, we engineered a normalization process for capturing the oracle and taking the screenshots, the details of which we omit for space reasons. After comparing the oracle image and the test web page screenshot, the approach builds rendering maps of the web page to identify the HTML elements that are responsible for the regions of detected visual differences. The approach then uses aspects of the visual layout to prioritize these identified elements, in terms of likeliness to be faulty, and reports this list to the developer.

Although effective at detecting and prioritizing HTML elements, our prior work is limited in terms of performing root cause analysis. WebSee can only use the detection information to prioritize a set of potentially faulty HTML elements, not find the faulty visual property causing the failure. This limits its usefulness for debugging because a developer must still examine the HTML elements and determine if any of their style properties could be responsible for the detected presentation failure. This limitation motivates our work, presented in this paper, which starts with a set of potentially faulty HTML elements and then uses search based techniques to identify the root cause.

3. DEFINING THE SEARCH APPROACH

The goal of our approach is to automatically identify the root cause of a presentation failure in a web page. We define the root cause of a presentation failure as a combination of a faulty HTML element in a web page and a faulty visual property defined for that element. In special cases such as, adding or deleting elements from the HTML page, only the HTML element can also form the root cause. However, such cases are not handled in the current work. A visual property can be a CSS property or an HTML attribute that controls the rendering of the HTML element. A simple approach for root cause analysis is to use a brute force exploration over the possible root cause space. Every potential root cause is substituted with different values with the goal to get the same rendered appearance for the test web page, P, as the oracle, O. But the large universe of possible values for a root cause makes a brute force exploration an expensive approach. This limitation motivated us to explore more sophisticated search based techniques [3, 5], such as genetic algorithms, that could be used to more efficiently search for a root cause.

The key insight of our approach is that we can use image processing techniques to define root cause analysis as a search based technique. Specifically, we use image processing techniques to know when a search has been successful and to guide the search. Our insight is that when the rendered appearance of P and O are the same, it defines a successful search, implying that we have found the correct root cause. Equality of the two renderings is determined by taking a screenshot of the rendered page, P, and then using image processing to compare the screenshot and O on a pixel by pixel basis. The second key insight is that we can also guide the search effectively using fitness functions based on image processing. For example, the number of difference pixels can be used as a fitness function, with a goal to minimize it to zero.

We now explain how our approach works. The approach takes P, O, and a set of potentially faulty HTML elements as inputs. As with our prior work, described in Section 2, the form of O is an image that shows the intended web page design. The set of potentially faulty HTML elements is provided from the output of our prior work. For each potentially faulty HTML element, the approach uses the search based technique to identify potential root causes. For each possible root cause, our approach substitutes a possibly correct value for it into P. Each modified web page, P', is then rendered and compared with O to determine the fitness of the root cause and value. If there is a match, the combination is reported as the most likely root cause for the observed presentation failure. (This combination can also be used to fix the presentation failure in static web pages. However, sophisticated analysis techniques are required in the case of dynamic web pages to locate the source code line for applying the fix. We plan to handle this in our future work.) If no match is found for a potential root cause, the approach tries other possible root causes for the current HTML element and the other potentially faulty HTML elements.

We now discuss the genetic algorithm in detail. The **population** is a list of possible values for the visual property under test. The initial population is generated randomly from a range of legal values. For **selection**, we use linear ranking, selecting the best p chromosomes. For **crossover**, two chromosomes are randomly selected from the population as parents to produce offspring by using a one point crossover with a given crossover rate, C_r. For **mutation**, a random gene in a chromosome is changed using uniform mutation based on a given mutation rate, M_r. In general, the **fitness function** aims to minimize the differences in rendering of P and O. We have defined different fitness functions for the different categories of visual properties, as described in Sections 3.1 and 3.2. The search process is **stopped** after a value for the visual property under test is found that renders P' the same as O.

In our experiments we found that different types of root causes need to be handled differently to make the search technique effective. We identified three such categories based on the impact they have on the rendering of an HTML element. For each of these categories we developed specialized fitness functions. We describe the three categories below.

3.1 Category: Size and Position

This category contains the visual properties, such as margin, padding, height, width, etc. that affect the size or position of a rendered HTML element. The visual properties in this category are assigned numeric values.

The fitness function for this category focuses on minimizing the number of difference pixels between the rendering of P and O. The insight that makes this effective is that the visual properties in this category impact the size and position of an HTML element, which is proportional to the number of difference pixels. As the explored values for a root cause get closer to the correct value, the HTML element in the rendering of P' starts to overlap with that in O, resulting in fewer difference pixels. Conversely, the number of difference pixels increases or stays the same as the explored value moves away from the correct value because the amount of overlap between the two renderings decreases. We compute the number of difference pixels by counting all of the pixels that are different by comparing the screenshot of P' and O.

To illustrate, we assume there are 300 difference pixels between P and O and that a particular HTML element, <`div style="padding:10px; background-color:#FFFFFF"`>, has been identified as a potentially faulty element. Our approach will consider each possible root cause associated with this element. Our example starts with the *padding* property, which is set to 10px, but should be 20px to be correct. Our approach evaluates the fitness of the initial population for the *padding* property. Inserting 50px into the *padding* property and comparing the rendering of the modified page, P' with O causes there to be 2,100 difference pixels. This guides the search by indicating that it is moving away from the expected value. Then a value of 15px is evaluated, which results in 175 difference pixels, indicating that the search is closer to the correct value. Eventually, the population includes the value of 20px, which results in zero difference pixels and indicates that a correct root cause has been found.

3.2 Category: Color

This category contains the visual properties, such as text color, background-color, border-color, etc. that affect the color of the HTML element. The visual properties in this category can be assigned a color value from the predefined 140 color names, such as Black, Aqua, Beige, etc., or from the normalized color range of 16 million colors ranging from #000000 to #FFFFFF. Note that the predefined color names can be translated into a unique numeric color.

It is not possible to use the difference pixel based fitness function for this category. The reason for this is that changing the color values for a visual property simply results in the same number of difference pixels, albeit of a different color. Nor is it possible to simply look at the oracle and determine the right color to be used. The reason for this is that many of the difference pixels that result from a color root cause are in-between colors used to perform anti-aliasing for curves in text and shapes.

Our approach for this category's fitness function is based on the idea of color distance. Our insight is that one explored color value is better than another if it minimizes the color distance between the difference pixels in O and the rendered image of P'. Color distance is defined using Euclidean distance between the numeric values of a pixel's RGB components. The fitness function is implemented by computing the color distance between the average color of the difference pixels in O and the average color of the difference pixels in the rendered page, P'. Lower numbers indicate that the color is closer to the correct value and higher numbers indicate that the color is farther from the correct value. If the distance between the two average colors is zero, the approach performs a full image comparison of P' and O (i.e., not just comparing the average colors of the difference pixels) to determine if the correct root cause has been found.

To illustrate, consider again the potentially faulty HTML element, <`div style="padding:10px; background-color: #FFFFFF"`>. Our example here starts with the *background-color* property, which should be equal to #FF0000. The approach first calculates the average color of the difference pixels in O and P, which are #FFA000 and #FFFFE0, respectively. (Recall that anti-aliasing will shade the colors in a region where there are curves, thus causing the average to be different from the set color.) The color distance between these two average colors is 369. Our approach evaluates the fitness of the initial population for the *background-color* property. Inserting #000000 into the <div> element and finding the average color of difference pixels in the modified page, P', results in #333300. This makes the color distance with O to be 394, indicating that the search is moving away from the solution. Then, evaluating the color value of #FFF000 results in the average test color value of #FF9000 for which the color distance is 32, indicating that the search is getting closer to the solution. Finally, evaluating #FF0000 results in a color distance of 0, indicating that we have found the correct root cause.

3.3 Category: Predefined Values

This category contains visual properties, such as font-style, display, etc. that accept a value from a set of discrete predefined values. For example, font-style can take a value from the predefined set {italic, oblique, normal}. For these types of values, it is not possible to guide a search as there is no clear notion of "closeness" among the potential values. Therefore, we process this category using an exhaustive exploration of all of the predefined values for a given visual property to find the value that results in the same rendering of O and P. Considering all possible values is expensive, but is mitigated by the fact that, in practice, such HTML properties typically define small sets of predefined values, with the largest we identified containing only 21 elements.

4. EXPERIMENT

We conducted a small experiment to assess the accuracy of our approach by calculating the percentage of the test cases in which the root cause was correctly identified. We compared the results of our approach to random search as a form of "sanity check" [3]. We did not evaluate presentation failures caused by the predefined values category since these were handled using exhaustive exploration and would produce similar results as that of the random search.

4.1 Implementation and Procedure

We implemented our approach in a prototype tool, *RCA*, built in Java. We used the Selenium WebDriver to take screenshots of the test web page and ImageMagick to compare the screenshot with the oracle. We used the Java Genetic Algorithms Package (JGAP) to help implement the genetic algorithm, using its default crossover rate of 0.35 and mutation rate of 0.08.

We chose the *Gmail* homepage (`http://www.gmail.com`) as the subject application, because it is a popular real-world web application with a sophisticated layout. However, the homepage's design used internally for implementing it was not available to us to be used as the oracle image. Therefore, we used a screenshot of the *Gmail* homepage rendered in a browser as the oracle.

To generate test cases, we first downloaded all the files required to display the *Gmail* homepage from the web. Then, we created a variant H_i of the original page, H, with a fault seeded for each of the i applicable visual properties. This procedure is analogous to creating mutants in mutation test-

Table 1: Accuracy

Category	RCA	RAND	Test#
Color	100%	37%	7
Numeric	100%	59%	30
Total	100%	55%	37

ing. We identified a total of 45 visual properties (8 color and 37 size and position), that belonged to the two categories. The seeding was done by iterating over each visual property in the list and identifying the set of HTML elements in H where the property could be added or modified. We randomly chose one of the applicable HTML elements, added or modified the inline visual property of the element in H, and saved it as H_i.

We provided H_i and O as inputs to our approach and the random search. We ran both of the approaches five times on each of the 37 test cases and reported the average accuracy of each approach. We limited the search space to allow all 370 test case executions to finish in 24 hours. We time bounded the random approach by terminating its search once the genetic algorithm approach had finished. We did this because exhaustively searching even a reduced search space was time consuming and the goal of our experiment was simply to show the relative benefit of our approach.

Table 1 shows the results of this experiment. The accuracy results of our approach and the random search are shown in the columns labeled "RCA" and "RAND" respectively. The total number of test cases used in the experiment are shown in the column labeled *Test#*. Note that the number of test cases is less than all possible visual properties, because no applicable HTML element was found for some CSS properties, such as max-width, in our subject application.

4.2 Discussion

As the results show, our approach was able to identify the correct root cause in all of the test cases. On the contrary, the random approach was able to identify the correct root cause in only 55% of the test cases. This is a positive result because it shows the feasibility of our approach in terms of being able to successfully identify root causes. The results also show that our approach can outperform a random search that has been given the same amount of time and same sized search space.

Although the results were positive, our experiment has limitations that affect its validity and motivate further experiments in future work. Primarily, our restriction of the search space means that different results may be achieved in terms of time and accuracy when the approach is run on the full range of possible values. For example, we have not fully considered if local maxima exist outside of our restricted search space, which may impact the accuracy and time results. Therefore, we characterize our results as preliminary and only indicating that our proposed approach represents a feasible solution.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented the idea of using search-based techniques to automatically find root causes of presentation failures in web applications. Our approach uses image processing techniques to identify visual differences between the rendering of a web page and its oracle. These differences are then analyzed and search-based techniques are employed to identify the root cause for the observed failure. The correctness of a root cause is determined by its ability to reduce the number of visual differences to zero. The preliminary results from our experiment look promising and validate the feasibility of our approach.

In future work, we plan to explore techniques for improving the performance of our approach. In particular, we plan to improve the search space initialization using heuristics for each of the root cause categories. For example, for size and position related properties, we can use sub-image searching to provide an estimate of the amount of translation an HTML element has undergone and use that to initialize the search space fairly close to the correct value. We also plan to work on prioritization techniques for elements and visual properties. For elements, this will focus on improving the prioritization techniques for the results generated by WebSee. Within each element we will explore heuristics for prioritizing the order in which the properties are explored.

We also want to address the two limitations of our approach. The first is that our approach relies on the assumption that the faulty visual property will be present in the faulty HTML element's domain. This is an inherent limitation as our approach cannot find the root cause if the faulty property is not present in the element. The second is that our technique is currently only defined for the case where there is a single presentation failure in a web page. Handling the multiple fault scenario will be a significant challenge as the fitness functions we have defined only account for one possible change at a time. We will investigate alternate ways of measuring fitness and searching the root causes so that our technique can handle this situation.

6. REFERENCES

[1] 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 2010.

[2] 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)*, May 2013.

[3] M. Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*, FOSE 2007.

[4] S. Mahajan and W. G. Halfond. Finding HTML Presentation Failures Using Image Comparison Techniques. In *submission*, 2014.

[5] P. McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[6] 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 2009.

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

[8] Newman, Mark W. and Landay, James A. 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 2000, pages 263–274.