Automated Discovery of Cross-browser Incompatibilities in Web Applications

Shauvik Roy Choudhary
Georgia Institute of Technology, Atlanta, GA
shauvik@cc.gatech.edu

ABSTRACT

Web based applications have gained increased popularity over the past decade due to the ubiquity of the web browser platform across different systems. Rapid evolution of web technologies has lead to inconsistencies in web browser implementations, which has resulted in cross-browser incompatibilities (XBIs) – situations in which the behavior of the same web application varies across browsers. Such XBIs can either result in cosmetic defects or completely prevent the user from accessing part of the web application's functionality. Although XBIs are a fairly common problem faced by users of web applications, there hardly exist any tools for discovering them automatically and assisting the fix of such problems. Due to this, developers have to invest considerable manual effort in detecting XBIs, which is both time consuming and error prone. This paper presents our technique and tool – CROSSCHECK to automatically detect and report XBIs to the web developer along with debug information about the affected HTML element, thereby helping the developer to fix the issue. CROSSCHECK is the first technique to practically discover XBIs in real web applications and in doing so, it combines concepts from graph theory, computer vision and machine learning. Our results show that CROSSCHECK [1] is practical and can effectively find both cosmetic and functional XBIs in real world web applications.

1. PROBLEM AND MOTIVATION

Web applications have become increasingly widespread and are used by us for common daily activities. From entertainment to business work-flow and from commerce and banking to social interaction, web applications are rapidly becoming a feasible, when not the dominant option for conducting such activities. Web applications are typically accessed through a web browser. Due to the flexibility of the choice of the web browser, the client-side environment is diverse among users, with an implicit expectation that web applications will always behave consistently across all different browsers. Unfortunately, this is often not the case. Web applications can differ in look, feel, and functionality when run in different browsers. We call such differences, which may range from minor cosmetic differences to crucial functional flaws, *cross-browser incompatibilities (XBIs)*.

The prevalence and growth of XBIs is due to several recent technology trends. The demand for content-rich, interactive web applications has led to web applications that provide significant parts of their functionality on the client-side (i.e., within the web browser). Since the standards for many client-side technologies are still evolving, each browser implements a slightly different version of these technologies and their runtime environments. This is a major source of XBIs, and the recent explosion in the number of browsers [21], computing platforms, and combinations thereof has further exacerbated the problem. The Mozilla broken website reporter contains 1,767,900 websites running on 463,314 hosts, which were reported as broken by the users of the Firefox web browser [12], thereby providing an evidence of the severity of this issue.

In-spite of the prevalence of the issue, current industrial practice for performing cross-browser testing largely consists of manually browsing and visually inspecting a web application under different browsers. The few available industrial tools and services (e.g., [2, 11]) that support this task focus largely on behavior emulation under different browsers or platforms, but do little in terms of automated comparison of these behaviors. That aspect remains largely manual, ad hoc, and hence error prone.

To address the limitations of existing technology, we present our technique and tool – CROSSCHECK to automatically detect such cross-browser issues and help the developer provide a solution. In particular, the technique can (1) identify both visual and functional differences automatically. (2) group these differences into meaningful XBIs. (3) for each XBI, report the differences along with their location to assist the developer to generate a fix.

2. BACKGROUND AND RELATED WORK

2.1 Challenges

Automated detection of XBI needs to overcome some challenges in the domain as explained in this section. Firstly, any manual inspection for identifying such issues is expensive in terms of resources such as time and effort. Hence, the technique should be fully automated and should operate without requiring any manual effort from developers. Secondly, a modern web application consists of multiple dynamically generated screens. To compare the behavior of the web application, a tool would need to extract these screens along with the control flow information between these screens and then compare them across browsers. Such information is often represented as a state graph, where the each node is a dynamic screen and the edges represent actions on these screens (typically a click event), that lead to another screen. A *trace* in this graph is a path along a set of nodes and edges representing a user interaction on the webpage. Differences in corresponding traces across browser state graphs lead to *trace-level XBIs*.

Thirdly for each corresponding pair of screens across browsers, the technique should compare them by leveraging the structural information of that screen. This information can be obtained from the DOM [2] tree, which is maintained by each web browser when it loads any web page. However, This DOM tree differs across

[1]For an extended version of this paper see [17]

[2]Document Object Model http://www.w3.org/DOM/

browsers due to implementation differences and conditional scripts in web applications. Thus, automated comparison of this DOM tree across browsers is non-trivial. Before performing any comparison, our technique needs to match nodes between the different DOM trees obtained by loading the same web page across various web browsers. Our technique comprises of graph theory algorithms to perform both matching as well as structural comparison. Since the DOM tree can be mapped back to a source code element, it is helpful for the developer if the technique can identify differences in the corresponding tree nodes from different browsers. This information can then be translated to the sections of the HTML page and source code that generated it. Providing this information to developers allows them to fix the issue effectively.

An important challenge is to compare the appearance of elements on the web page. The DOM tree contains textual information about an element on the web page. However, it does not contain information about the appearance of the elements on the webpage. Therefore, it is essential to be able to mimic the end users' perception by considering the screen capture data from the web page. For doing so, the technique should first perform dynamic analysis of the DOM tree to extract geometrical information about the web page elements (*e.g.,* dimensions, position etc.) along with a screen capture. These can be used to extract visual features by incorporating geometrical and computer vision distance metrics to find dissimilarities between the elements. Once these features are extracted, the technique should employ automated classification techniques from machine learning to effectively decide if the issue is significant enough to be reported. The point differences identified by such a technique for each element identifies a symptom of a bigger cross- browser incompatibility (XBI). We term such micro-level differences as cross-browser differences (CBDs), which need to be grouped across spatially or geometrically related elements to identify XBIs. This is done by employing a custom clustering algorithm as explained in Section 3.3. All XBIs reported for a pair of screens across browsers are termed as *screen-level XBIs*

Web applications often have variable elements such as advertisements and generated content (*e.g.,* time, date etc.) which are different across multiple requests. If such elements are not ignored, the technique might consider them as changes across browsers thereby resulting in false positives in the results. Hence, the technique needs to find and skip such elements during comparison. A final challenge is due to the inbuilt security measures in web browsers that pose a technical challenge for the technique to transparently extract information for comparison. The technique must overcome such security mechanisms reliably and collect all information needed.

Overcoming the aforementioned challenges allow the technique to identify XBIs in behavior of the web application (and individually its screens) across two browsers. For each XBI, the technique leverages the structural information from the DOM tree to locate the broken element on the web page. This information helps the developer to find related parts of the source code that generated these elements and modify them to fix the issue.

2.2 Related Work

Recent years have seen a spate of tool and web-services that can either visualize a web-page under different client-side platforms (browser-OS combination) or, more generally, provide an emulation environment for browsing a web application under different client platforms. Some representatives of the former include BrowserShots (`http://browsershots.org`) and BrowserCam (`http://www.browsercam.com`), while examples of the latter include Adobe BrowserLab [2], CrossBrowserTesting.com and Microsoft Expression Web [11]. For a more detailed list, the interested reader is referred to [9]. A common drawback of all these tools is that they focus on the relatively easy part of visualizing browser behavior under different environments, while leaving to the user the difficult, manually intensive task of checking consistency. Further, since they do not automatically explore the dynamic state-space of the web application, they potentially leave many errors undetected.

The technique of Eaton and Memon [4] is among the earliest works in the area of cross-platform error detection. Their technique tries to identify potentially problematic HTML tags in a given web page, based on a manual classification of good and faulty pages previously generated by the user. However, XBIs in modern web applications are usually not attributable simply to specific HTML tags, but rather to complex interactions of HTML structure, CSS styles and dynamic DOM changes through client-side code. More recently, Tamm [19] presented a tool to find layout faults in a single page, with respect to a given web browser, by using DOM and visual information. The tool requires the user to manually alter the web page—hide and show elements—while taking screen-shots. This technique is not only too manually intensive to scale to large web applications, but also virtually impossible to apply to dynamically generated pages (i.e., most of the pages in real-world applications). Finally, its focus is not specifically cross-browser testing.

For testing browsers, there are several test suites developed by practitioners. *Acid Tests* [1], which are part of the *Web Standards Project*, are a set of 100 tests that check a given web browser for enforcement of various W3C and ECMA standards. Similarly, the *test262* suite [20] (formerly called *SputnikTests*) can check a JavaScript engine (of a web browser) against the ECMA-262 specification. It is noteworthy that in an experiment we ran, Mozilla Firefox 7.0.1 failed 187 of the 11016 tests in the *test262* suite, Google Chrome 15.0 failed 416 tests, and Internet Explorer 9 failed 322 tests. In other words, the JavaScript engines of these popular browsers are not standard and differ from one another. These suites reveal some of these differences and justify the development of techniques to identify XBIs.

Recently we presented WEBDIFF [18] to detect CBDs for a single web screen. To do so, it uses a two phase approach. First, it performs a DOM-matching step to find pairs of corresponding screen elements between renderings of the page in two different browsers. Subsequently, it performs visual comparison of these pairs of screen elements to find CBDs. The visual comparison uses a hand-crafted heuristic that considers several visual properties of the screen elements. WEBDIFF was then complemented by our collaborators in another technique called CROSST [9], which focuses on finding trace-level XBIs in the dynamic state-space of a web application. CROSSCHECK combines the concepts from both these techniques and further uses machine learning to perform a automated classification of XBIs as described in the next section.

3. APPROACH AND UNIQUENESS

In this section, we summarize our CROSSCHECK technique [17] for finding both visual and functional XBIs in web applications across multiple browsers. For this purpose, CROSSCHECK considers one browser as "reference" and compares the behavior of the web application in it with that in other browsers.

The algorithm behind the CROSSCHECK technique has three main phases as shown in Figure 1. The CROSSCHECK approach consists of three phases. The **first phase** starts by accepting the URL of a web application along with names of two browsers to be considered for comparison. Then it automatically crawls the target web application within each of the two browser environments. While doing so, it captures and records the observed be-

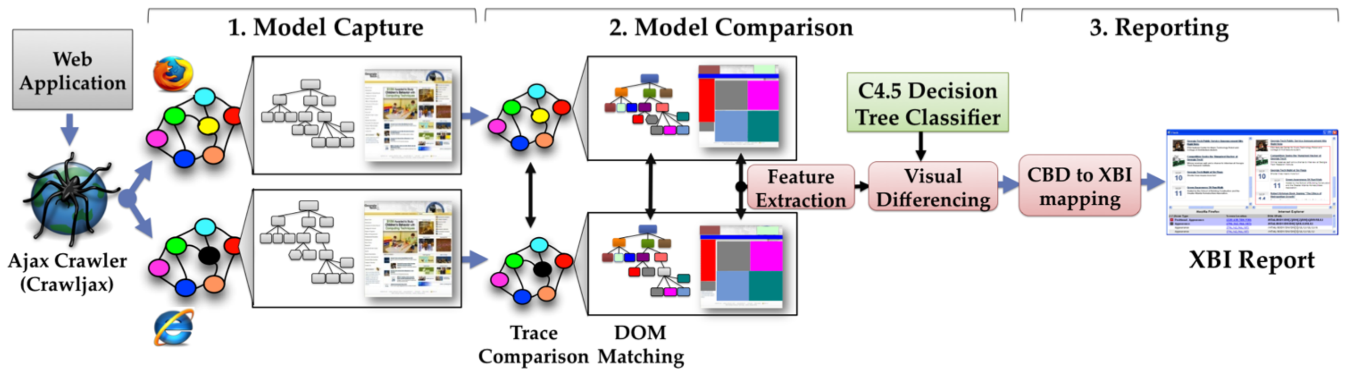


Figure 1: Overview of the CROSSCHECK Approach.

havior in the form of two navigation models, M_1 and M_2, one for each browser. The crawling is performed in an identical fashion under each browser, so as to exercise precisely the same set of user-interaction sequences within the web application in both cases. The **second phase** compares models M_1 and M_2 to check whether they are equivalent and extract a set of model differences, which may represent one or more potential XBIs. The **third phase** analyzes this set of model differences and collates them into a set of XBIs, which are then presented to the end-user.

The following sections present further details of these three phases.

3.1 Crawling and Model Capture

This phase explores the state-space of the web application under test using an approach broadly based on CRAWLJAX [8, 10]. CRAWLJAX is a crawler capable of exercising client-side code, detecting and executing doorways (clickables) to various dynamic states of modern (AJAX -based) web applications. By firing events on the user interface elements, and analyzing the effects on the dynamic DOM tree in a real browser before and after the event, the crawler incrementally builds a state machine capturing the states of the user interface and the possible event-based transitions between them. CRAWLJAX is fully configurable in terms of the type of elements that should be examined or ignored during the crawling process. (For more details about the architecture, algorithms, and capabilities of CRAWLJAX , see Reference [8, 10].)

CROSSCHECK implements an observer module on top of the core crawler. The observer captures and stores a finite-state *navigation model*, M, of the behavior observed during the crawling. The navigation model is comprised of a state graph, G, representing the top-level structure of the navigation performed during the crawling, as well as several pieces of data for each state of the state graph. The *state graph* is a labelled directed graph in which a state corresponds to an actual page that could be observed by an end-user in a browser, and an edge represents an observed transition between two states. Each edge is labelled by the user action (typically, a `click`) and the element that caused the transition. For each observed state s, and corresponding page p, the model records (1) a screenshot of p, as observed in the web browser, (2) the underlying DOM representation for p, and (3) the co-ordinates of the visual representation of each DOM element of p within the browser. This crawling and model capture is performed, on the target web application, for each of the two browsers (Br_1 and Br_2), and the corresponding navigation models (M_1 and M_2) are stored for subsequent analysis.

3.2 Model Comparison

This phase compares models M_1 and M_2, captured in the previous phase, for equality and records the observed differences between them. These differences represent potential manifestations of XBIs in the web application's behavior and are used in the final phase of the algorithm to identify a set of XBIs. The comparison is performed in three steps, representing analyses at three different levels of abstraction of the model.

3.2.1 Trace-level Comparison

This step compares state graphs G_1 and G_2 (of models M_1 and M_2) for equivalence. To do so, it uses the graph isomorphism algorithm for labelled directed graphs proposed in [9], which finds a one-to-one mapping between both nodes and edges of G_1 and G_2. The comparison produces two items: $\mathcal{T}$ and *ScreenMatchList*. $\mathcal{T}$ is a list of those edges from G_1 or G_2 that could not be matched to a corresponding edge in the other state graph—edges that denote trace-level differences. *ScreenMatchList* is a list of matched screen pairs (S_i^1, S_i^2) (from G_1 and G_2, respectively) that is used in the next step.

3.2.2 DOM Comparison of Matched Screen Pairs

This step iterates through the pairs of matched screens in *ScreenMatchList* (previous step) and, for each such pair (S_i^1, S_i^2), compares the DOMs of the two screens for equality and produces $DomMatchList_i$ as its output. $DomMatchList_i$ is a set of pairs of matched DOM nodes, one for each pair of screens. Our DOM matching algorithm largely follows the one proposed by us in WEBDIFF (see Algorithm 1 in Reference [18] for details). The key difference between the two algorithms is the computation of the *match index* metric—a number between 0 and 1 that quantifies the similarity between two DOM nodes. The higher the match index, the higher the likelihood that the two DOM elements represent the same element in the context of their respective screens. Currently, we use the following formula for computing the match index: Given two DOM nodes a and b,

$$MatchIndex = 0.7 \times \Delta X + 0.2 \times \Delta A + 0.1 \times \Delta P$$

where ΔX stands for XPath Distance, ΔA measures the difference in attributes, and ΔP measures the difference in dynamic properties queried from the DOM. The coefficients for this formula were empirically established by us in previous work. The variables are metrics which are defined as follows:

$$\Delta X = 1 - \frac{LevenshteinDistance(a.xpath, b.xpath)}{max(length(a.xpath), length(b.xpath))}$$

$$\Delta A = \frac{countSimilar(a.attibutes, b.attributes)}{count(a.attributes \cup b.attributes)}$$
$$\Delta P = \frac{countSimilar(a.domdata, b.domdata)}{count(a.domdata \cup b.domdata)}$$

In the formulas, (a, b) are the two DOM nodes to be matched. XPath is the path of a particular DOM node in the DOM tree from the root, and $LevenshteinDistance$ [7] is the edit distance between the XPaths of the two nodes under consideration. Functions $max(), length()$, and $count()$ return the maximum of two numbers, the length of a string, and the number of elements in a list, respectively. Properties $attributes$ and $domdata$ are maps having $(key, value)$ pairs that represent the DOM node attributes and dynamic DOM properties, respectively. Function $countSimilar$ returns the number of elements between two maps that have equal $(key, value)$ pairs, and operator $\cup$ performs a union of two maps based on keys. Once the DOM nodes are matched using the above formulas, a set of DOM-level differences $\mathcal{D}_i$ is computed for each screen pair (S_i^1, S_i^2) in *ScreenMatchList*.

3.2.3 Visual Differencing of Matched Screen-element Pairs

This is the final and most important step in the model comparison. This step iterates over the pairs of matched DOM elements in $DomMatchList_i$ and, for each pair of matched DOM nodes (n_j^1, n_j^2), compares their corresponding screen elements visually. If the two nodes are found to be different, they are added to the list of screen differences, $\mathcal{V}_i$, for screen pair (S_i^1, S_i^2). For this, a set of features are extracted for each node in the pair (n_j^1, n_j^2) and a machine learning classifier $\mathcal{C}$ is used to decide if the nodes are similar or represent a CBD. Further details of this classifier are presented in the following section. As in the case of DOM-level differences, there is a set of visual differences, $\mathcal{V}_i$, computed for each pair of matched screens (S_i^1, S_i^2) in *ScreenMatchList*.

3.2.4 Machine learning for Visual Differencing

To perform automated classification, CROSSCHECK uses machine learning to build a classifier that is used to decide whether two screen elements being compared are different. For this work, we used a *C4.5 decision tree* classifier [14] because it is simple, yet effective in our problem domain. We chose a set of five features (described below) for the classifier. These features were carefully chosen to encompass the typical manifestations of XBIs found in the wild, based on our experience with cross-browser testing during this and previous work [9, 18].

Let e_1 and e_2 be the screen elements being compared, (x_1, y_1) denote the absolute screen co-ordinates of the top left-hand corner of the bounding box of e_1 (in pixels), w_1, h_1 denote the width and height of this bounding box (also in pixels), and x_2, y_2, w_2, and h_2 denote the corresponding entities for e_2. Using this notation, the features we employed are defined as follows:

- **Size Difference Ratio (SDR):** This feature is designed to detect differences in size between the two elements in question. The feature is computed as a ratio to normalize and remove the effects of minor discrepancies in size, arising from differences in white-space or padding conventions between browsers. Such differences are quite ubiquitous and typically not considered to be XBIs. Specifically, if $a_1 = w_1 * h_1$ and $a_2 = w_2 * h_2$ denote the areas of the bounding boxes of e_1 and e_2, respectively,

$$SDR = \frac{|a_1 - a_2|}{min(a_1, a_2)}$$

  where $min()$ returns the minimum of its two arguments.

- **Displacement:** This feature captures the euclidean distance between the positions of corresponding screen elements:

$$Disp = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- **Area:** This feature is computed as $area = min(a_1, a_2)$ and is included to provide a "thresholding effect" for the other features. In this way, CROSSCHECK can ignore size or position differences in really small elements, which are typically the result of noise in the data capture rather than the manifestation of an actual XBI.

- **Leaf DOM Text Difference (LDTD):** This is a boolean-valued feature that detects differences in the text content of screen elements. In our experience, when such differences are caused by XBIs, they typically pertain to leaf elements in the DOM trees. Thus, this feature evaluates to `true` if and only if the elements being compared are leaf elements of their respective DOM trees, contain text, and the text in the two nodes differ. Otherwise, this feature is assigned the value `false`.

- χ^2 **Image Distance (CID):** This feature is intended to be the final arbiter of equality in comparing screen elements. For this feature, the images of the respective screen elements are compared by calculating (1) their image histograms and (2) the χ^2 *distance* between them. The χ^2 histogram distance has its origins in the χ^2 test statistic [13] and is computed as

$$\chi^2(H_1, H_2) = \sum_i \frac{(H_1(i) - H_2(i))^2}{H_1(i) + H_2(i)}$$

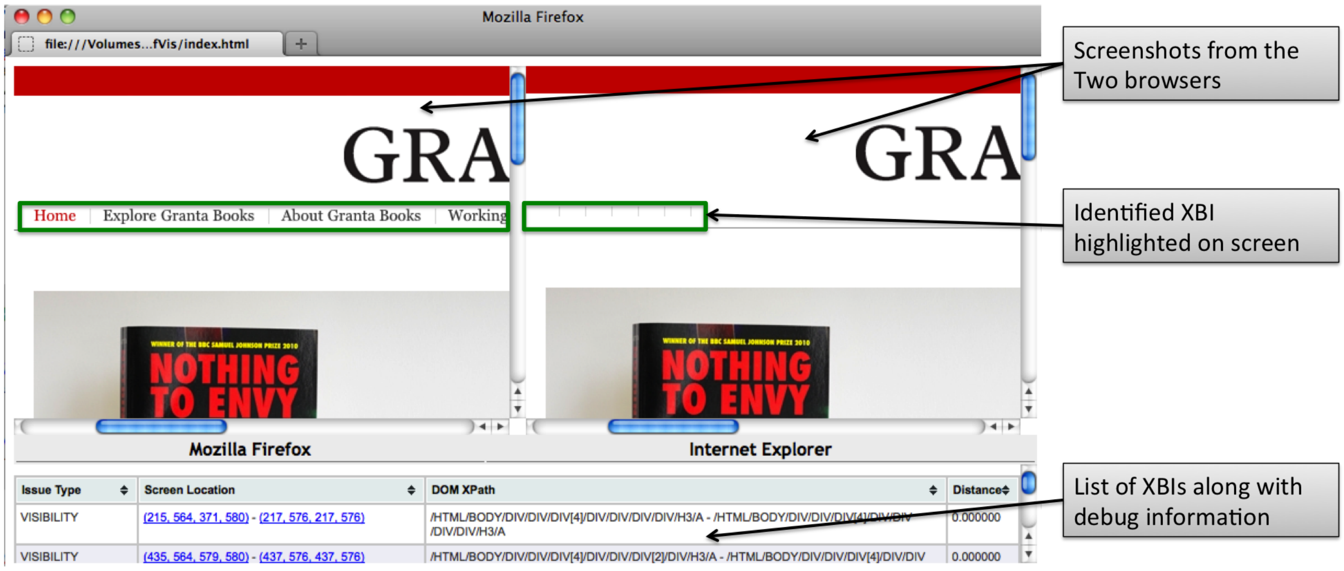
  where H_1 and H_2 are the histograms of the images of the screen elements, and the summation index i indicates a bin of the histogram.

3.3 Reporting

In this phase, the differences identified in the previous phase are grouped into meaningful XBIs and presented to the developer. To compute the set of XBIs, CROSSCHECK uses the list of trace-level differences, $\mathcal{T}$, visual differences, $\mathcal{V}$, and matched screen-pairs, *ScreenMatchList*, computed by step 3.2.1. To do so, it iterates through the pairs of matched screens $S_i^1, S_i^2$and marks those DOM nodes in that have any visual difference, that is, that appear in the set of visual differences $\mathcal{V}_i$ for this screen-pair. Similarly each trace-level difference, $t \in \mathcal{T}$ denotes a missing edge originating from state S_i^1 to some other state S_j^1. If this transition was created by a click on some DOM element e in S_i^1, then e is marked. This is done for all trace-level differences (mismatched state transitions) originating from screens S_i^1 or S_i^2. Finally, all the marked nodes of the given screen are clustered using the following logic: Two nodes are clustered together if and only if they have an ancestor-descendent relationship in the DOM tree. This operation gives a set of clusters $\mathcal{C}_i^1$ (respectively, $\mathcal{C}_i^2$) for S_i^1 (respectively, S_i^2). Finally, these clusters from $\mathcal{C}_i^1$ and $\mathcal{C}_i^2$ are matched if they contain nodes that were found to be matching DOM counterparts during matching performed in step 3.2.2. Each merged cluster denotes an XBI. A set $\mathcal{L}_i$ of XBIs is computed for each screen pair (S_i^1, S_i^2) and included in the report to be presented to the developer.

An example report generated for a real web application[3] has been presented in Figure 2. The top section shows the screenshot from two browsers (reference browser on the left and target browser on the right). The bottom section shows the XBIs found by CROSSCHECK and along with each item in the issue list is indicated the type of the issue, the location on the screen screen where it was identified, the DOM xpath of the element that is involved in this issue and possibly additional debug data by the developer.

[3]GrantaBooks - http://www.grantabooks.com


Mozilla Firefox
file:///Volumes...fVis/index.html
Screenshots from the Two browsers
GRA
GRA
Home
Explore Granta Books
About Granta Books
Working
Identified XBI highlighted on screen
Mozilla Firefox
Internet Explorer
List of XBIs along with debug information

Issue Type	Screen Location	DOM XPath	Distance
VISIBILITY	(215, 564, 371, 580) - (217, 576, 217, 576)	/HTML/BODY/DIV/DIV/DIV[4]/DIV/DIV/DIV/DIV/H3/A - /HTML/BODY/DIV/DIV/DIV[4]/DIV/DIV /DIV/DIV/H3/A	0.000000
VISIBILITY	(435, 564, 579, 580) - (437, 576, 437, 576)	/HTML/BODY/DIV/DIV/DIV[4]/DIV/DIV/DIV[2]/DIV/H3/A - /HTML/BODY/DIV/DIV/DIV[4]/DIV/DIV	0.000000

Figure 2: Example Report Produced by CROSSCHECK for GrantaBooks.com

4. RESULTS AND CONTRIBUTIONS

To assess the usefulness of CROSSCHECK , we implemented it in a tool and performed experiments to assess the effectiveness and efficiency of our technique. In this section, we summarize the procedure and results from our empirical evaluation. For details, an interested reader can refer to [17, 18].

For our experiments, we chose the latest stable versions of Mozilla Firefox and Internet Explorer browsers. Our experimental subjects consisted of seven web applications as shown in Table 4. The first subject, Restaurant was developed by us to demonstrate different kinds of cross-browser issues. The second one, Organizer is an open source personal organizer and task management application [22]. The remaining subjects are real-world web applications we selected using Yahoo!'s random URL service (`http://random.yahoo.com/bin/ryl`). More precisely, we chose them from a set of ten random web applications by selecting those with visible XBIs in their entry page.

For each of these subjects, CROSSCHECK captured the navigation model of the website and then used it to compare and report XBIs as described in the technique. Each subject was processed in under half hour and resulted in the collection of between 3–19 screens and 8–99 transitions in each state graph of application. We found these numbers to meet the time and memory requirements of such a tool, thereby confirming its practicality.

NAME	Tr	Po	Sz	Vs	Tx	Ap	**CBD**	**XBI**
Restaurant	4	0	2	0	2	3	11	9
Organizer	14	0	42	5	0	1	62	18
GrantaBooks	16	0	0	11	0	1	28	16
DesignTrust	4	2	39	2	0	146	189	130
DivineLife	7	0	0	3	1	70	81	73
SaiBaba	2	5	31	7	3	55	103	89
Breakaway	0	13	132	0	0	246	391	268

Table 1: CROSSCHECK Results

The results reported by CROSSCHECK have been presented in Table 4, where the first column lists the subject name followed by the number of trace-level differences reported. The next five columns show the different screen-level differences; namely in position, size, visibility, text and visual appearance. These difference are added to report the CBDs followed by the results from grouping them into fewer, more meaningful XBIs. For each of the issues reported, we manually inspected it to report false positives and negatives. For our subjects, CROSSCHECK did not report any false negatives. The tool found 865 differences out of which 314 were true issues (63% FP). We find these results to be encouraging, considering that CROSSCHECK is the first tool to find XBIs automatically and because we see a potential of eliminating the false positives in the future. In particular, the main limitations of CROSSCHECK are: (1) Computer vision algorithms are effective for comparing the underlying binary data of images, but they are not as good at accurately mimicking a human perception of visual differences. We found them to be the main source of false positives and would like to investigate on better algorithms to eliminate or at-least mitigate this issue. (2) The crawler has a black box view of the application and needs to be instructed on how to interact with the web application. For greater degree of automation and server side coverage, a white-box assisted crawler should be developed. (3) Our core algorithm relies on a basic level of browser support to obtain DOM information from each screen. This information can be sometimes inaccurate, leading to false positives. This issue can be mitigated by performing noise removal and by collecting or deriving more data to make it resilient to noise.

We would like to address these problems in the near future. CROSSCHECK is being used by developers at Fujitsu and our current goal is to perform user studies with them to make improvements based on feedback from users. Finally, we want to work on computing the behavioral equivalence of web applications running on different platforms (eg. desktop and mobile).

5. CONCLUSION

XBIs are relevant and a serious concern for web application developers. Current techniques are limited in finding these issues and require considerable manual effort. Our technique CROSSCHECK identifies XBIs automatically and helps developer fix the issue by providing information about the broken element. For XBI identification, CROSSCHECK uses concepts from graph theory (state graph isomorphism, DOM tree matching), computer vision (visual analysis) and machine learning (automated classification). Our results suggest that CROSSCHECK is practical and can find both visual and functional XBIs in real world applications.

6. REFERENCES

[1] Acid Tests - The Web Standards Project. `http://www.acidtests.org`.

[2] Adobe. Browser lab. `https://browserlab.adobe.com/`, May 2010.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272, 2008.

[4] C. Eaton and A. M. Memon. An empirical approach to evaluating web application compliance across diverse client platform configurations. *Int. J. Web Eng. Technol.*, 3(3):227–253, 2007.

[5] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 145–154. ACM, 2007.

[6] X. Jia and H. Liu. Rigorous and automatic testing of web applications. In *In 6th IASTED International Conference on Software Engineering and Applications (SEA 2002*, pages 280–285, 2002.

[7] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[8] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proc. 8th Int. Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.

[9] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 561–570, New York, NY, USA, 2011. ACM.

[10] A. Mesbah and A. van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *Proc. 31st Int. Conference on Software Engineering (ICSE'09)*, pages 210–220. IEEE Computer Society, 2009.

[11] Microsoft. Expression web. `http://www.microsoft.com/expression/products/Web_Overview.aspx`, May 2010.

[12] Mozilla. Firefox broken website reporter. `http://reporter.mozilla.org/app/stats/`, July 2010.

[13] K. Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine Series 5*, 50(302):157–175, 1900.

[14] J. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann Publishers, 1993.

[15] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[16] D. Roest, A. Mesbah, and A. v. Deursen. Regression testing ajax applications: Coping with dynamism. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 127 –136, 6-10 2010.

[17] S. Roy Choudhary, M. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *ICST '12: Proceedings of the International Conference on Software Testing*. IEEE, April 2012.

[18] S. Roy Choudhary, H. Versee, and A. Orso. WebDiff: Automated identification of cross-browser issues in web applications. In *Proceeding of the 2010 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, September 2010.

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

[20] test262 - ECMAScript. `http://test262.ecmascript.org/`.

[21] Wikipedia. List of web browsers. `http://en.wikipedia.org/wiki/List_of_web_browsers`.

[22] F. Zammetti. *Practical Ajax Projects with Java Technology*. Apress Series. Apress, 2006.