CROSSCHECK: Combining Crawling and Differencing To Better Detect Cross-browser Incompatibilities in Web Applications

Shauvik Roy Choudhary*
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280
Email: shauvik@cc.gatech.edu

Mukul R. Prasad
Software Systems Innovation Group
Fujitsu Laboratories of America
Sunnyvale, California 94085
Email: mukul.prasad@us.fujitsu.com

Alessandro Orso
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280
Email: orso@cc.gatech.edu

***Abstract*—One of the consequences of the continuous and rapid evolution of web technologies is the amount of inconsistencies between web browsers implementations. Such inconsistencies can result in cross-browser incompatibilities (XBIs)—situations in which the same web application can behave differently when run on different browsers. In some cases, XBIs consist of tolerable cosmetic differences. In other cases, however, they may completely prevent users from accessing part of a web application's functionality. Despite the prevalence of XBIs, there are hardly any tools that can help web developers detect and correct such issues. In fact, most existing approaches against XBIs involve a considerable amount of manual effort and are consequently extremely time consuming and error prone. In recent work, we have presented two complementary approaches, WEBDIFF and CROSST, for automatically detecting and reporting XBIs. In this paper, we present CROSSCHECK, a more powerful and comprehensive technique and tool for XBI detection that combines and adapts these two approaches in a way that leverages their respective strengths. The paper also presents an empirical evaluation of CROSSCHECK on a set of real-world web applications. The results of our experiments show that CROSSCHECK is both effective and efficient in detecting XBIs, and that it can outperform existing techniques.**

***Keywords*-web testing, dynamic analysis, machine learning.**

I. INTRODUCTION

Web applications are playing an ever increasing role in our daily lives. From entertainment to business workflow 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. Currently, users have the choice of using several web browsers, with the implicit expectation that web applications will 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)*.

Several recent technology trends have been contributing to the prevalence and growth of XBIs. The demand for content-rich, interactive web applications, in particular, coupled with the development of technologies such as AJAX (Asynchronous JavaScript And XML) [1] and Flash, 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 [2], computing platforms, and combinations thereof has further exacerbated the problem.

The 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., [3], [4]) 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.

In recent work, the authors have proposed two complementary approaches for automatically detecting XBIs: WEBDIFF and CROSST. WEBDIFF [5] operates on single web pages and focuses on finding XBIs that can be detected through visual analysis. In contrast, CROSST [6] can analyze entire web applications, using dynamic crawling, and focuses on finding functional XBIs. In this paper, we present CROSSCHECK, a new technique and tool that combines and adapts these two approaches. CROSSCHECK leverages the strengths of WEBDIFF (precision in detecting visual differences) and CROSST (ability to crawl and detect functional differences) while mitigating their shortcomings. In defining CROSSCHECK, we also improved several core elements of the original techniques to further enhance the effectiveness of the combined approach. In particular, CROSSCHECK leverages machine learning to build a more accurate detector for visual differences, and uses a new and improved metric for image comparison. We also present an evaluation in which we used CROSSCHECK on a set of real-world web applications with real XBIs. The results of the evaluation are promising and show that CROSSCHECK is both efficient and effective in detecting XBIs in web applications.

The key contributions of this work are:

- CROSSCHECK, a new technique and tool for detecting both visual and functional XBIs in web applications.

* This author was an Intern at Fujitsu Laboratories of America at the time this work was performed.

- A new, powerful machine learning-based technique to detect visual XBIs.
- A novel algorithm to cluster related visual and functional differences that can report more meaningful XBIs that are easier to comprehend for the end-user.
- An evaluation of CROSSCHECK on several real-world web applications that shows its effectiveness in detecting real XBIs.

The rest of the paper is organized as follows. Section II introduces some key concepts and terminology to define the scope and nature of XBIs which is then illustrated in an example web application in Section III. Section IV surveys related work on cross-browser testing. We describe our proposed technique in Section V and our tool implementing it in Section VI. Sections VII, VIII present the empirical evaluation of CROSSCHECK along with results, which is followed by concluding remarks in Section IX.

II. BACKGROUND

A. Web Application

Web applications are based on a client-server computing model. In a typical scenario, a human user interacts with the client-side of a web application through a web browser that runs on a computing device (e.g., a desktop PC). Users view web pages, enter data, and perform actions, such as clicks, on widgets (e.g., buttons or hyper-links). These interactions generate requests to the server, and the server responds to such requests with updates to the current web page, encoded in HTML (Hyper-Text Markup language) or XML (eXtensible Markup Language), and to other associated resources, such as style information in CSS (Cascading Style Sheets), client-side code (e.g., JavaScript), images, and so on. These resources are then used to compute and render an updated web page in the web browser. The recent trend is to handle an increasing portion of the user interactions entirely on the client side, using JavaScript code and other components, such as Flash, to compute responses and updates to the current web page. In fact, many of the web pages viewed by the user may have no corresponding REST-based [7] URI. This is typical of several modern web applications based on the AJAX paradigm.

B. The Web Browser: A Source of Cross-browser Differences

Modern web browsers are fairly sophisticated applications comprised of a number of components. A typical architecture of a web browser is presented in [8]. Of the many functional components at work in a browser, there are three that are of specific interest for understanding the reasons for cross-browser issues. The first and most important among them is the *layout engine*, which is responsible for rendering a web page by combining the structural information in the HTML for the page with the style information in CSS stylesheets. The browser also maintains a *DOM (Document Object Model)* representation of the page in memory to allow client-side scripts (e.g., JavaScript code) to modify the web page dynamically. The layout engine is the primary source of cross-browser differences, as the same HTML/DOM and CSS can produce different-looking pages in different browsers. The second component is the *event-processing engine*, or the *DOM engine*, which couples a user action, such as a mouse click on a specific location, with the execution of specific event-handling client-side code. This engine also performs changes in the DOM based on the DOM-API of the browsers. Browsers also differ in their event-handling algorithms, as well as in the DOM-API they support. This is another source of cross-browser differences. Thus, the same user action can produce a different change to the DOM. A third source of difference is the *JavaScript engine*—the runtime environment for executing JavaScript code within the browser. Subtle but definite differences exist between the JavaScript engines of different browsers, which result in differences in behavior. It is noteworthy that standards do exist for various client-side technologies, such as HTML, CSS, DOM, and ECMA-Script. However, browsers typically implement their own variants of these standards.

C. Cross-browser Incompatibilities

XBIs can manifest in two ways: (1) on individual webpages or (2) in the dynamic behavior of the web application in transitioning from one web page (state) to another in response to a user interaction. We term the former *screen-level differences*, and the latter *trace-level differences*. Screen-level differences can in turn be either visual differences—differences in the visual appearance of one or more widgets on a web page—or DOM-level differences. In order to provide a comprehensive solution, we detect and harness all three kinds of cross-browser differences, that is, visual (screen-level), DOM (screen-level) and trace-level. It is worth noting that a single cross-browser issue can result in several differences of one or more of the kinds we just discussed. For example, a malformed widget on a browser may result in a visual difference of that widget, several differences at the DOM level, and a missing state-transition as result of a click on that widget. We make a distinction between a *cross-browser difference (CBD)* and a *cross-browser incompatibility (XBI)*. A CBD refers to a single difference (of one of the three kinds) of a specific property of a particular element. An XBI refers to a set of related CBDs that can be viewed, analyzed, and debugged by the developer as a single error. Thus, CBDs are symptoms of XBIs. We would like to *detect* CBDs but *report* XBIs to the web application developer.

D. Machine Learning

Machine learning [9] is a branch of artificial intelligence that aims to develop algorithms that allow computers to *learn* behaviors from empirical data. Specifically, learning is interpreted in the sense of *inductive inference*, where one observes data representing incomplete information about a statistical phenomenon and generalizes it into rules to either (1) discover hidden properties of that data or (2) make predictions about future instances of related data. This work makes use of a branch of machine learning called *supervised learning*. Supervised learning deals with the pattern recognition or the *classification* problem. The

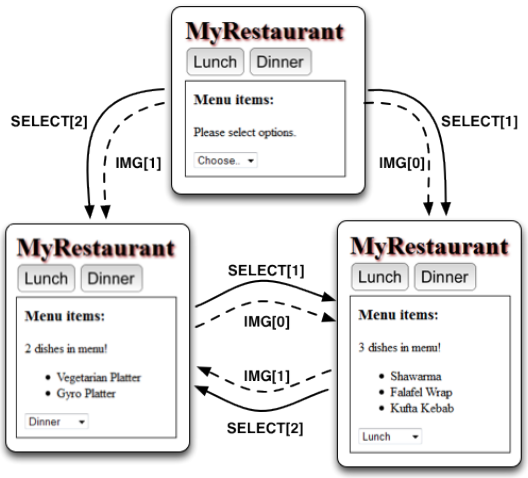
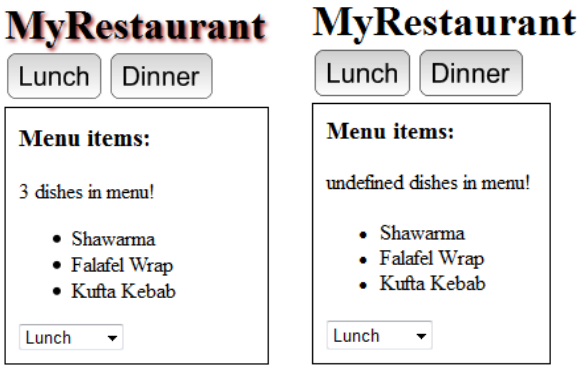


Figure 1. State graph for MyRestaurant in Mozilla Firefox.



(a) Mozilla Firefox (b) Internet Explorer

Figure 2. One page of MyRestaurant rendered in two browsers.

aim is to learn a function that maps input properties of data (called *features*) into a *label* of one of a finite set of discrete classes, for each instance of data. This function (termed a *classifier*) is learned from a set of data, where each data instance has been manually labelled to the desired class based on its features. This is called the *training phase*. Subsequently, this classifier can be used to label, new future instances of data. This is called the *testing phase*.

Several different kinds of classifiers exist, which differ in terms of their learning algorithm and the underlying topology of the learnt model. The choice of the classifier is typically driven by the size and nature of the dataset to be analyzed. For this work, we have chosen to use a *Decision Tree* classifier [10]. Decision trees are simple, intuitive, and effective classifiers that work well for relatively small data sets with only a few features. This is indeed the case for our problem, in contrast to other machine learning problems involving datasets with millions of records and/or dozens or even hundreds of features. Decision tree classifiers are typically constructed by repeatedly partitioning the input (feature) space so as to build a tree whose leaf nodes contain only instances of a single class. The C4.5 implementation of Decision Trees by Quinlan [10] is widely used, and our tool uses a version of this implementation.

E. Terminology

In this paper, we use the term *web page*, *web screen*, or *screen* to denote any distinct page that the end-user can view within a web browser. Any visual change to this page would constitute a new web page. We also use the term *state* to indicate an abstraction of a web page computed using its DOM representation (i.e., web pages with different DOMs correspond to different states). Every DOM element has a physical visual representation on a web-page. We use the term *screen element* when referring to a DOM element in conjunction with its visual footprint on the web page.

III. Motivating Example

In this section, we introduce a simple web application that we use as a motivating example to illustrate XBIs and our technique. The application MyRestaurant displays the food items available at a fictitious restaurant. The user can interact with the page elements to view the food items available for lunch or dinner.

For brevity, we do not present the complete source code of the web application but explain its operation and causes of XBIs with short code snippets. The web page shows the name of the restaurant in the header, followed by menu images and a container box. This container is supposed to show the food items available for a particular selection. It also has a drop-down menu that provides an alternative way for the user to make a selection. Figure 1 shows the state graph extracted by a web crawler, such as CRAWLJAX (http://crawljax.org), by navigating and exercising the different pages of a the web application in a web browser (Mozilla Firefox, in this case). The state graph consists of three nodes, each representing a state of the web application, and six edges that represent transitions between these states as a result of user actions on page elements.

However, when the MyRestaurant application is run in Internet Explorer (IE), it results in the following XBIs with respect to the application run in Mozilla Firefox: (1) there is no response when the user clicks on one of the two menu images (**Lunch** and **Dinner**), (2) the header has a brown shadow in Firefox that does not appear in IE, and (3) the count of the food items is shown as `'undefined'` in IE.

XBIs 2 and 3 can be seen in the `'lunch'` state of both browsers presented side by side in Figure 2. The first XBI manifests itself in the four state transitions shown by dotted lines in Figure 1. These transitions are missing from the state graph for IE. This XBI occurs because of the following HTML code:

```
<img src="lunch.png" onclick="listItems(event)" />
```

Here, the developer assigned an `onclick` event to the menu image that calls the JavaScript function `listItems`. This call fails in IE because of the lack of support for the `onclick` attribute of the `<img>` tag. Consequently, the expected state transitions do not happen in IE. This XBI exposes a functional flaw, as the user cannot access the desired content through the two menu images in IE.

The second XBI is attributable to the following CSS:

```
h1{text-shadow: brown 2px 2px 2px;}
```

According to this CSS snippet, the header should have a brown shadow. This shadow is absent in IE because it does not support the `text-shadow` property.

Finally, the third XBI stems from the JavaScript code, which is supposed to show the count of the number of food items.

```
var txt = $("items").childElementCount
                    + " dishes in menu!";
$("stats").innerHTML = txt;
```

This is obtained by querying the number of child nodes of the list element located by id `"items"`. The string is built with this count and assigned to the element with identifier `"stats"`. Because the DOM property `childElementCount` is not supported in IE, the browser returns `'undefined'`, which is the equivalent of `null` in JavaScript, resulting in the XBI. Note that, for conciseness, we use a custom library function `$("id")` to locate an HTML element that would normally be accessed by calling `document.getElementById("id")`.

IV. Related Work

A. Cross-platform Visualization & Emulation

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 [3], CrossBrowserTesting.com and Microsoft Expression Web [4]. For a more detailed list, the interested reader is referred to [6]. 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.

B. Early Work on Cross-platform Error Detection

The technique of Eaton and Memon [11] is among the earliest works in this area. 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 [12] 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 screenshots. 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.

C. Test Suites for Web Browsers

Acid Tests [13], 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 [14] (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.

D. WebDiff *and* CrossT

Our WebDiff [5] and CrossT [6] techniques provide the foundations of the approach we propose in this paper. WebDiff detects XBIs on a single web page through a two-step process. 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 XBIs. The visual comparison uses a hand-crafted heuristic that considers several visual properties of the screen elements. By contrast, CrossT focuses on finding trace-level XBIs in the dynamic state-space of a web application. It uses automatic crawling to build a graph of the state-space of a web application under two given browsers. It then checks whether the two graphs are isomorphic to detect trace-level XBIs. CrossT also performs a basic DOM-level differencing of matched states to find DOM-level differences (although this differencing can generate a large number of false positives). Our proposed technique combines these two complementary techniques, as we discuss in detail in the next section.

V. Our Approach

The following elements characterize our combined approach for XBI detection:

- WebDiff detects screen-level XBIs, while CrossT's strength is trace-level XBIs. CrossCheck employs these techniques in these respective capacities. A side benefit of this combination is that the crawling enables dynamically generated screens to be checked by visual comparison.
- WebDiff uses DOM matching specifically to detect corresponding screen elements to visually compare, while CrossT uses it as the sole basis for screen-level XBI detection. In doing so, the former can overlook useful symptoms of XBI detection, while the latter can generate too many false positives. We observed that DOM-level differencing can be used as an efficient and reliable detector for textual differences (vs. visual differencing). CrossCheck incorporates this capability.
- CrossCheck uses machine learning to build a classifier for visual comparison of screen elements. This

provides a dramatic improvement over the rather simplistic DOM-differencing used for screen comparison in CROSSST. It is also significantly better than the hand-crafted heuristic for visual comparison used in WEBDIFF, which is difficult to re-target and tends to miss many visual differences. Furthermore, the CROSSCHECK classifier actually combines DOM-differencing features with strict visual-comparison ones to provide a more comprehensive predictor.

Algorithm 1: CROSSCHECK

```
/* CrossCheck -- Overall algorithm. */
Input  : url: URL of target web application
         Br1, Br2: Two browsers
         C: screen-element match classifier
Output: L: List of XBIs
1  begin
2      (M1, M2) ← genCrawlModel(url, Br1, Br2)
       // Compare State Graphs
3      (T, ScreenMatchList) ← traceEquivCheck(M1, M2)
4      foreach (S_i^1, S_i^2) ∈ ScreenMatchList do
           // Compare matched screen pair
5          DomMatchList_i ← matchDOMs(S_i^1, S_i^2)
6          V_i ← ∅
7          foreach (n_j^1, n_j^2) ∈ DomMatchList_i do
               // Compare matched screen-element pair
8              if ¬visualMatch(n_j^1, n_j^2, C) then
9                  V_i ← V_i ∪ (n_j^1, n_j^2)
10             end
11         end
12     end
13     L ← computeXBIs(T, V, ScreenMatchList)
14     return L
15 end
```

Algorithm 1 presents our overall approach. The algorithm's input consists of the URL of the starting page of the target web application, the two browsers for cross-browser testing, and a classifier $\mathcal{C}$ used for visual comparison. It is assumed that this classifier has been created in an offline training phase, using machine learning. The details of the specific training used for CROSSCHECK are discussed in Section VI.

The CROSSCHECK approach consists of three phases. The **first phase** automatically crawls the target web application within each of the two browser environments. While doing so, it captures and records the observed behavior 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. This phase is implemented by function *genCrawlModel()* in Algorithm 1 (line 2).

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 (lines 3-12).

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. This operation is implemented by function *computeXBIs()* (line 13).

The following sections present further details of these three phases.

A. Phase 1: Crawling and Model Capture

This phase explores the state-space of the web application under test using an approach broadly based on CRAWLJAX [15], [16]. 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 [15], [16].)

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.

B. Phase 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.

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 [6], which finds a one-to-one mapping between both nodes and edges of G_1 and G_2. This step is implemented by function *traceEquivCheck*() of Algorithm 1 (line 3). 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.

2) DOM Comparison of Matched Screen Pairs: This step iterates through the pairs of matched screens in *ScreenMatchList* (line 4) and, for each such pair (S_i^1, S_i^2), compares the DOMs of the two screens for equality. This comparison is implemented by function *matchDOMs()* (line 5) 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 in WEBDIFF (see Algorithm 1 in Reference [5] 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. Algorithm *matchDOMs()* uses this metric when matching nodes in the two DOMs. 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 $LevenshteinDistance$ [17] 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) Visual Comparison 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 $DomMatchList_i$ (line 7) 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). The visual comparison is implemented by function *visualMatch()* (line 8) and uses a classifier $\mathcal{C}$ built through machine learning. 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*. Set $\mathcal{V}$ (line 13) is simply the set of all the $\mathcal{V}_i$ computed for individual screen pairs.

C. Machine Learning for Visual Comparison

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 *decision tree* classifier [10] 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 [5], [6].

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 [18] 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. This measure is different from the EMD metric [19] that was used in WEBDIFF; in our experience, the χ^2 metric is much faster to compute and much more accurate than EMD for this specific problem, thus allowing us to detect a larger number of XBIs.

Further details about the training phase of the classifier are presented in Section VI.

D. Phase 3: Mapping Model Differences to XBIs

Algorithm 2: computeXBIs

```
/* Algorithm for computing XBIs from model
   differences. */
Input  : T: Trace-level differences
         V: Visual Differences
         ScreenMatchList: List of matched pairs of screens
Output: L: List of XBIs
1  begin
2      foreach (S_i^1, S_i^2) ∈ ScreenMatchList do
           // Cluster per matched screen pair
3          markVisualDiffs(S_i^1, S_i^2, V_i)
4          markTraceDiffs(S_i^1, S_i^2, T)
5          C_i^1 ← clusterMarkedNodes(S_i^1)
6          C_i^2 ← clusterMarkedNodes(S_i^2)
7          L_i ← mergeMappedClusters(C_i^1, C_i^2)
8      end
9      return L
10 end
```

We compute and report XBIs on a screen-by-screen basis, using the clustering approach outlined in Algorithm 2. To compute the set of XBIs, $\mathcal{L}$, this algorithm uses the list of trace-level differences, $\mathcal{T}$, visual differences, $\mathcal{V}$, and matched screen-pairs, *ScreenMatchList*, computed by Algorithm 1. To do so, it iterates through the pairs of matched screens S_i^1, S_i^2 (line 2). The function *markVisualDiffs()* (line 3) marks those DOM nodes in S_i^1 and S_i^2 that have any visual difference, that is, that appear in the set of visual differences $\mathcal{V}_i$ for this screen-pair. Function *markTraceDiffs()* (line 4) does something similar, but for trace differences. Specifically, let $t \in \mathcal{T}$ be a trace-level difference, which 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 will be marked by *markTraceDiffs()*. This would be done for all trace-level differences (mismatched state transitions) originating from screens S_i^1 or S_i^2. Finally, function *clusterMarkedNodes()* (lines 5,6) clusters all the marked nodes of the given screen. 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, *mergeMappedClusters()* (line 7) merges those clusters from $\mathcal{C}_i^1$ and $\mathcal{C}_i^2$ that contain nodes that are DOM counterparts. 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). The complete set of XBIs, $\mathcal{L}$, is simply a collection of all $\mathcal{L}_i$.

Table I
CONFUSION MATRIX

a	b	← classified as
1959	0	a = false
9	169	b = true

VI. THE TOOL: CROSSCHECK

A. Training the classifier for Visual Comparison

As mentioned earlier, the visual comparison component of our approach uses a machine learning based classifier. We used the public domain WEKA [20] package for building a classifier and applying it within CROSSCHECK for visual comparison.

To train this classifier, we proceeded as follows. We chose a set of 10 web pages from 10 different websites that we knew had various cross-browser problems. Each of these pages was loaded in the two browsers (FF 7.0 and IE 9, in our case) and all relevant screen and DOM data was captured for these pairs of pages. We ran these page pairs through our DOM matching algorithm (Section V-B) to generate a list of matched screen element pairs. Each of these screen element pairs represents a single instance of the visual comparison problem, a typical instance that would be given to the classifier to evaluate in a real-time execution of CROSSCHECK (on line 8 of Algorithm 1). We accumulated these instances over the 10 test pages (each page yielded several instances), which gave us $2,137$ instances to train our classifier.

We then computed the features described in Section V for each of these instances. We also manually labelled the instances as either *true*, if they corresponded to an XBI, or *false* otherwise. (We did this by viewing the instances in the context of their respective web pages and performing a side-by-side comparison.) As in any machine learning process, this labeling implicitly codifies the kind of pattern or classifier that will be eventually learnt. We therefore used the following guidelines for the labeling:

- Only *substantial* changes in size, position, or overall appearance (e.g., font, colors, visibility) were marked as errors. Minor differences were ignored.
- All manifestations of a particular XBI were marked as errors. For instance, if the XBI corresponded to the displacement of a particular page element, such as an HTML `<TABLE>` element, instances coming from every DOM node in that `<TABLE>` would need

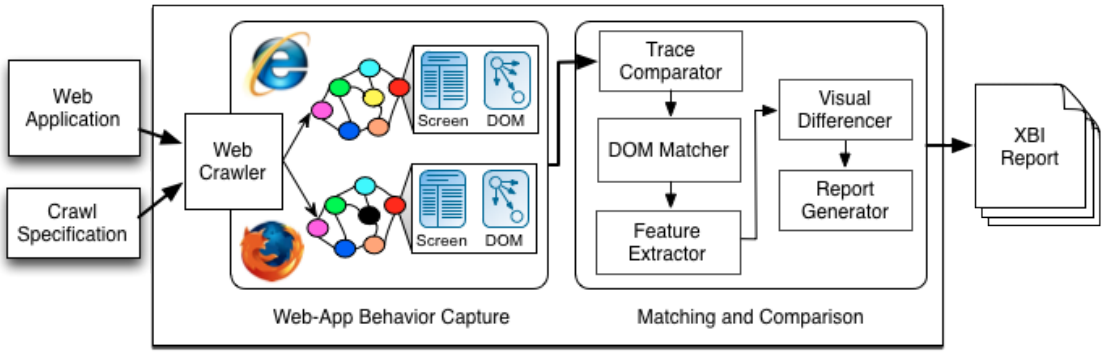


Figure 3. Overview of tool CROSSCHECK.

to be marked as errors, not just the top <TABLE> element. This consistency between the labeling and the feature is crucial for the machine learning algorithm to build a proper classifier. Otherwise, the classifier would be fragmented, non-intuitive, and ultimately of poor quality.

The 2137 instances, of which 178 were labelled *true*, constituted our training data set for a decision tree classifier within WEKA. This gave us a tree of size 11 with 6 leaves. In order to assure ourselves the learnt classifier was of sufficiently high quality, we performed *stratified cross validation* [9] with 10 folds, on the classifier. The *confusion matrix* [9] shown in Table I shows that the learnt classifier was indeed of high quality.

B. Core Tool Implementation

We implemented the combined approach described in the previous section in a tool called CROSSCHECK. Figure 3 shows an overview of CROSSCHECK and its principal components. The tool consists of a crawler, CRAWLJAX, that takes the URL of a deployed web application along with an optional crawl specification and extracts the behavioral model of the web application. The crawl specification contains the instructions to explore a specific behavior of the web application and the browser environments to consider for testing.

The CRAWLJAX crawler is written in Java. We implemented an observer module on top of CRAWLJAX to capture whole-page screenshots and query dynamic DOM data for each crawled web-page, as well as the overall state graph of the crawled behavior. Specifically, we used the Java Win32 API to obtain screenshots, and a custom JavaScript program to query the DOM parameters. After the first phase, CROSSCHECK produces a behavioral model of the web application for each of the different browsers considered.

In the next phase, CROSSCHECK compares these models. The trace comparator performs trace-level comparison as described in Section V and generates a list of unmatched states and transitions along with a set of matching screens for further comparison. The DOM matcher matches DOM nodes across matched screens and also reports unmatched nodes. For each pair of matched DOM nodes, the feature extractor computes the features—SDR, Displacement, Area, LDTD and CID—for the corresponding screen elements, as explained in Section V. These features are used by the machine learnt classifier, comprising the visual differencer, to decide whether this pair of screen elements are different. If they are found to be different, they are added to the set of visual, screen-level differences for this screen pair. The report generator then clusters the trace-level and screen-level differences from all of the screens into XBIs that are presented to the user.

All the components of this phase were also implemented in Java. For computing the CID, we used JavaCV (http://code.google.com/p/javacv), which is a java port of the OpenCV (http://opencv.willowgarage.com) computer vision toolkit. In the next section, we present an empirical evaluation of our approach that we performed using CROSSCHECK.

VII. EMPIRICAL EVALUATION

To assess the usefulness of CROSSCHECK to detect XBIs, we conducted a set of experiments and compared the results against the state of the art. Specifically, we investigated the following research questions:

- RQ1: Can CROSSCHECK identify different kinds of CBDs in real-world web applications and correlate them to identify XBIs?
- RQ2: How effective is CROSSCHECK when compared to CROSST and WEBDIFF?

We first present details of our experimental subjects and procedure. Then, we use our results to address the research questions listed above.

A. Subjects

For our evaluation, we use the seven web applications listed in Table II. For each application, the table shows its *name*, URL, type, number of states and transitions in the crawled state flow graph for that application, and the max, min, and average number of DOM nodes per screen that were extracted from the web application and analyzed by CROSSCHECK. (This latter metric serves as a measure of the complexity of the pages being analyzed.) The first application, Restaurant, is the web application we presented in the motivating example; it was developed by us to demonstrate different kinds of cross-browser issues. Organizer is an open source personal organizer and task management application [21]. 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

Table II
INFORMATION ABOUT THE PROGRAMS USED IN THE EMPIRICAL EVALUATION.

Name	*URL*	*Type*	*States*	*Transitions*	DOM Nodes (per screen)		
					max	*min*	*average*
Restaurant	http://localhost/restaurant	Information	3	8	785	846	821
Organizer	http://localhost/organizer	Productivity	13	99	10001	27482	13051
GrantaBooks	http://grantabooks.com	Publisher	9	8	15625	37800	25852
DesignTrust	http://designtrust.org	Business	10	20	7772	26437	18694
DivineLife	http://sivanandaonline.org	Spiritual	10	9	9082	140611	49886
SaiBaba	http://shrisaibabasansthan.org	Religious	13	20	524	42606	12162
Breakaway	http://breakaway-adventures.com	Sport	19	18	8191	45148	13059

chose them from a set of ten random web applications by selecting those with visible XBIs in their entry page.

B. Experimental Setup

For conducting our experiments, we ran CROSSCHECK on the subject web applications. The experiments were performed on a 64-bit Windows 7 computer with 4GB of RAM. We used two browsers for experimentation: the latest stable versions of Mozilla Firefox (v7.0.1) and Internet Explorer (v9.0.3). Although CROSSCHECK could be run on more browsers, we believe that two browsers are sufficient to evaluate our technique. The first two web applications used for experiments were setup locally on an Apache web server, while the remaining ones were crawled from their live URLs.

C. Results

Table III(a) presents the results of our experiments. The first column shows the name of the subject. The next six columns show the cross-browser differences (CBD) reported by CROSSCHECK, categorized by issue type: trace level differences (Tr), positional shifts (Po), size differences (Sz), visibility issues (Vs), text-level mismatches (Tx), and visual appearance issues (Ap). The last two columns present the total numbers of CBDs identified and the number of clustered XBIs corresponding to these differences. As the table shows, CROSSCHECK identified CBDs of all types in the selected web applications. In addition, it was able to cluster related issues into the same XBI, as shown in the last column. For instance, in the case of GrantaBooks, the tool reported 16 trace, 11 visibility and 1 visual appearance difference–these 28 CBDs were then clustered into 16 XBIs to be reported to the user.

For comparing CROSSCHECK against CROSST and WEBDIFF, we exercised all three tools on the same web applications, manually confirmed the differences reported by the tools, and compared the different tools' results. Table III(b) presents the results of this comparison. The table shows the differences reported (and confirmed) for each of the tools in columns *Rep* and *Conf*. The table also reports the confirmed trace level (TL) and screen level (SL) differences. As shown in the table, CROSSCHECK reported 314 errors with 64% false positives (FPs), as compared against 49 (98% FPs) for CROSST and 119 (79% FPs) for WEBDIFF. Moreover, CROSSCHECK outperformed CROSST and WEBDIFF in their respective domains: $(TL + SL) >$ CROSST$_{conf}$ and $SL >$ WEBDIFF$_{conf}$, in most cases. The improvement over CROSST can be attributed to a better screen level matching. whereas the improvement over WEBDIFF is due to the use of the machine learnt classifier for visual comparison.

VIII. DISCUSSION

As shown in the previous section, CROSSCHECK was able to find different types of cross browser differences in the web applications we considered and to cluster them into a smaller number of XBIs to be reported to developers. This result addresses RQ1, which relates to the effectiveness of the technique. CROSSCHECK was also able to outperform the state-of-the-art tools in terms of efficacy, thereby addressing RQ2. Next, we discuss the main limitations of CROSSCHECK.

Limitations of the Crawler: Our current implementation uses the state-of-the-art crawler CRAWLJAX to explore a web application state space, and CRAWLJAX has some limitations. Due to its black-box view of the application, CRAWLJAX needs a specification to guide its exploration of the application's behavior and decide which part of it to ignore. This requires some manual effort and, in some cases, can limit the behavior coverage that can be achieved through the crawler. Also, the crawler currently does not support all kinds of user interactions with all kinds of widgets on all browsers. For example, the current version of CRAWLJAX does not support clicks on tags `<area>` in Internet Explorer. It is important to note that our core technique for XBI detection is orthogonal to the specific technique used for web application crawling. Therefore, CROSSCHECK can benefit from any improvement to the crawling tool. For example, we could use a suitable white box technique to automatically extract the crawl specification through program analysis (e.g., [22]).

Inconsistency in Browser Support: Our core technique relies on the DOM information obtained from the browser to detect XBIs. Sometimes, the DOM node's information provided by the browser is inaccurate, which can lead to false positives. For instance, an inaccuracy in the geometric information of a node would result in the computation of inaccurate feature values for the visual comparison, potentially resulting in identical screen elements being flagged as different. This issue can be mitigated by performing some noise removal on the extracted geometries and also by evolving the feature set to make the approach more resilient to such noisy sources. We are currently working along this direction.

Limitations of Visual Analysis: 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 that

Table III
CROSSCHECK EMPIRICAL RESULTS

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

(a) CROSSCHECK detailed results on subjects

NAME	CROSSCHECK				CROSST		WEBDIFF	
	Rep.	Conf.	TL	SL	Rep.	Conf.	Rep.	Conf.
Restaurant	11	11	4	7	11	6	11	5
Organizer	62	50	14	36	202	14	28	8
GrantaBooks	28	27	16	11	348	16	10	9
DesignTrust	189	27	4	23	68	0	98	21
DivineLife	81	13	7	6	1741	10	67	8
SaiBaba	103	36	2	34	188	3	42	5
Breakaway	391	150	0	150	306	0	291	63
TOTAL	865	314	47	267	2864	49	547	119

(b) CROSSCHECK results compared to CROSST and WEBDIFF

this limitation can also be a source of false positives—small differences that would be ignored by a human eye may be identified by the vision algorithm. In future work, we will investigate how to improve visual analysis algorithms to eliminate, or at least mitigate, this issue.

IX. CONCLUSION

Cross-browser Incompatibilities are common and represent a serious problem for web application developers. In this paper, we presented CROSSCHECK—a technique and tool for detecting XBIs in web applications. We described how CROSSCHECK combines two recent complementary approaches developed by the authors to perform this task more effectively and completely. CROSSCHECK not only reports both functional and visual differences, but also correlates and clusters different (but related) issues to present fewer XBIs to developers. Our experiments show that CROSSCHECK can detect XBIs both effectively and efficiently, and that it can outperform current state-of-the-art techniques.

In future work, we will improve CROSSCHECK to address the limitations discussed in Section VIII. Another logical next step for this work is to investigate techniques to assist the developer in diagnosing and fixing the XBIs detected by our technique. As a first step in this direction, we plan to perform a user study that can let us better understand developer needs for this task and build automated techniques around those.

ACKNOWLEDGMENT

The authors would like to thank Ali Mesbah for his initial work behind CRAWLJAX and CROSST which were instrumental for this work. This work was partially supported by NSF grants CNS-1117167, CCF-0964647, and CCF-0725202 to Georgia Tech.

REFERENCES

[1] J. J. Garrett, "Ajax: A new approach to web applications." http://www.adaptivepath.com/publications/essays/archives/000385.php, February 2005.

[2] Wikipedia, "List of web browsers," http://en.wikipedia.org/wiki/List_of_web_browsers.

[3] Adobe, "Browser lab," https://browserlab.adobe.com/, May 2010.

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

[5] 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.

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

[7] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, pp. 115–150, May 2002.

[8] A. Grosskurth and M. W. Godfrey, "A reference architecture for web browsers," *21st IEEE International Conference on Software Maintenance*, pp. 661–664, September 2005.

[9] T. M. Mitchell, *Machine learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997.

[10] J. Quinlan, *C4.5: programs for machine learning*, ser. Morgan Kaufmann series in machine learning. Morgan Kaufmann Publishers, 1993. [Online]. Available: http://books.google.com/books?id=HExncpjbYroC

[11] C. Eaton and A. M. Memon, "An empirical approach to evaluating web application compliance across diverse client platform configurations," *Int. J. Web Eng. Technol.*, vol. 3, no. 3, pp. 227–253, 2007.

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

[13] "Acid Tests - The Web Standards Project," http://www.acidtests.org.

[14] "test262 - ECMAScript," http://test262.ecmascript.org/.

[15] 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)*. IEEE Computer Society, 2008, pp. 122–134.

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

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

[18] 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*, vol. 50, no. 302, pp. 157–175, 1900.

[19] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International Journal of Computer Vision*, vol. 40, pp. 99–121, 2000.

[20] "Weka 3: Data Mining Software in Java," http://www.cs.waikato.ac.nz/ml/weka/.

[21] F. Zammetti, *Practical Ajax Projects with Java Technology*, ser. Apress Series. Apress, 2006. [Online]. Available: http://books.google.com/books?id=UPTjGFA5NugC

[22] W. 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*. ACM, 2007, pp. 145–154.