WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications

Shauvik Roy Choudhary, Husayn Versee, Alessandro Orso
Georgia Institute of Technology
shauvik@cc.gatech.edu, hversee3@gatech.edu, orso@cc.gatech.edu

***Abstract*—Cross-browser (and cross-platform) issues are prevalent in modern web based applications and range from minor cosmetic bugs to critical functional failures. In spite of the relevance of these issues, cross-browser testing of web applications is still a fairly immature field. Existing tools and techniques require a considerable manual effort to identify such issues and provide limited support to developers for fixing the underlying cause of the issues. To address these limitations, we propose a technique for automatically detecting cross-browser issues and assisting their diagnosis. Our approach is dynamic and is based on differential testing. It compares the behavior of a web application in different web browsers, identifies differences in behavior as potential issues, and reports them to the developers. Given a page to be analyzed, the comparison is performed by combining a structural analysis of the information in the page's DOM and a visual analysis of the page's appearance, obtained through screen captures. To evaluate the usefulness of our approach, we implemented our technique in a tool, called WEBDIFF, and used WEBDIFF to identify cross-browser issues in nine real web applications. The results of our evaluation are promising, in that WEBDIFF was able to automatically identify 121 issues in the applications, while generating only 21 false positives. Moreover, many of these false positives are due to limitations in the current implementation of WEBDIFF and could be eliminated with suitable engineering.**

I. INTRODUCTION

Web applications are increasingly widespread these days, and we use them daily both for work and personal activities, such as shopping, banking, networking, and email. As the popularity of web applications has increased, they have also become more complex and richer on the client side. A modern web application typically consists of several client-side components in the form of scripts and resources that are linked from the web page and execute in a web browser. Currently, users have the option to use several web browsers, which introduces problems for web-application developers; web applications are expected to behave consistently across all of the popular browsers (and across platforms). However, because the standards for client-side technology are still evolving, there is a great deal of inconsistency in how different web browsers behave. These inconsistencies lead to what we call *cross-browser issues*–differences in the way a web page looks and behaves in different browsers.

Cross-browser issues range from minor cosmetic problems to critical failures that involve the applications' functionality. In both cases, the issue can result in the inability of the users to access one or more services provided by a web application. Current browser usage statistics report that there are seven popular web browsers that are commonly used across various platforms [1]. Maintaining compliance across these browsers is a crucial task for developers. If a feature is broken on one of these top browsers, it could result in (part of) the application malfunctioning or being inaccessible for a class of users. Moreover, these issues occur on the client side (*i.e.,* in the browser), which is out of the developer's direct control. It is therefore often the case that it takes a long time for such issues to get reported and fixed. For this reason, there is much interest in identifying cross-browser issues during in-house testing, before the software is released [2], and companies try to limit the number of browsers they support [3].

Currently, detecting cross-browser issues requires manual inspection of the web pages and their behaviors when they are rendered in different browsers. Existing commercial tools (*e.g.,* [4], [5]) can assist such manual testing by presenting the differential rendering side by side, but they still require a considerable effort from the developer and are limited in functionality. The few research tools that target this problem (*e.g.,* [6], [7]) are also limited by the fact of requiring a considerable amount of manual work, as we further discuss in Section VII.

To address the limitations of existing techniques, we present a novel technique that can (1) detect cross-browser issues automatically, (2) target both visual and functional problems, and (3) guide developers in identifying the causes of the identified problems. Our technique is based on differential testing [8], in that it runs web applications in different environments and compares their behaviors in such environments; intuitively, a difference in behavior indicates a potential problem. More specifically, our approach operates as follows. First, it opens the web page under analysis in different browsers and gathers (1) the DOM generated in each of these browsers and (2) a screenshot of the rendered web page. Second, it compares the information collected and matches the web page elements across browsers. Third, it compares the position and appearance of the matched web page elements and identifies dissimilar elements. Finally, for every issue found, it reports the issue and the specific HTML tag associated with it to help developers understand and solve the issue.

To evaluate the usefulness of our approach, we performed an empirical study in which we assessed the effectiveness and precision of the approach. To perform the study, we built WEBDIFF, a prototype implementation of our approach, applied the tool to nine real web applications, and measured the number of true and false positives generated by WEBDIFF.

26th IEEE International Conference on Software Maintenance in Timișoara, Romania
978-1-4244-8628-1/10/$26.00 ©2010 IEEE

Overall, the results of our study are encouraging: WEBDIFF was able to automatically identify 121 issues in the applications, while generating only 21 false positives. Moreover, many of these false positives could be eliminated with suitable engineering, as they are mainly due to limitations in our current implementation of WEBDIFF.

The main contributions of this paper are:

1) A classification of cross-browser issues for web applications.
2) A novel technique for automatically detecting cross-browser issues by combining the analysis of DOM and visual information.
3) The implementation of the technique in a tool, WEBDIFF, and an evaluation of WEBDIFF on a set of real web applications.

II. BACKGROUND

A. Web Applications and Web Browsers

A web application follows a typical client-server computing model and usually consists of several server and client side components. Server side components get invoked when the web server receives a request (typically, from a remote user through a web browser). As a result of the server side execution, various client side components are dynamically generated and sent back to the web browser in the form of HTML (HyperText Markup Language — http://www.w3.org/html/) pages. These pages, which are rendered by the browser, reference or contain resources such as images, animations, style information (*i.e.,* Cascading Style Sheets (CCS) — http://www.w3.org/Style/CSS/) and scripts (*e.g.,* JavaScript or VBScript).

A web browser consists of different subsystems that handle various functionality, such as processing the client side components and managing the interactions of these components with system resources (*e.g.,* network, display, file system). Grosskurth and Godfrey [9] describe the reference architecture followed in several open source browsers. Among the subsystems of a browser, one of the main components is the layout engine, which is responsible for rendering a web page by parsing the HTML tags in the page and applying to the relevant elements the style information contained in the CSS stylesheets for the page. The browser also maintains a DOM (Document Object Model) representation of the web page in its memory to allow scripts associated with the page to query and modify web page elements. Although there is a standard definition for the DOM format (see http://www.w3.org/DOM/), web browsers often deviate from such standard. Moreover, since most web pages have browser specific code to make them work on different browsers and platforms, the DOM generated by different browsers can be very different. For this reason, simply comparing the DOM information in different web browsers is far from ideal when comparing web pages rendered in such browsers.

B. Image Matching

Image matching is an important problem in the area of Computer Vision. Matching images of real world objects is particularly challenging, as a matching algorithm must account for factors such as scaling, lighting, and rotation. Fortunately, the images that we need to compare in this work are screen captures of web pages rendered in different browsers. In this context, the above issues do not occur, and the main problems are, for instance, the shifting of web page elements or the fact that some elements are not displayed at all (for a complete list of issues, see Section IV.

A basic technique for comparing two images is to compare their histograms, where an *image histogram* represents the distribution of the value of a particular feature in the image [10]. In particular, a *color histogram* of an image represents the distribution of colors in that image, that is, the number of pixels in the image whose color belongs in each of a fixed list of color ranges (bins). Obviously, if two images are the same, their color distributions will also match. Although the converse is not true, and two different images can have the same histogram, this issue is again not particularly relevant in our problem domain.

Basic histogram matching techniques find the difference between corresponding bins across two images, which can result in false positives in the case of small shifts. The use of the Earth Movers' Distance (EMD) [11] can alleviate this issue. *EMD* is a measure of the distance between two distributions and, intuitively, consists of the minimum amount of "work" required to make the two histograms identical by moving around the quantities in the different bins. Because it can ignore small changes in an image, EMD is widely used in computer vision. For the same reason, it is a suitable approach for the problem of comparing the graphical rendering across web browsers, where we want to be able to account for negligible variations while catching larger changes. (In Section V-D, we describe how we defined the threshold for the EMD metric in our technique.

III. MOTIVATING EXAMPLE

Before describing cross-browser issues, we introduce a simple web application that we use as a motivating example. The application, AjaxSearch, displays words from the dictionary that match a given search string provided by a user through a web form. Being a web application, AjaxSearch consists of server and client side components. The client side components of AjaxSearch are its main HTML file, `search.html` (Figure 1), its stylesheet, `style.css` (Figure 3), and the JavaScript file `script.js` (Figure 2). The server side component of AjaxSearch consists of the PHP script `server.php`, which is partly shown in Figure 4. Note that, because each server side component generates a response that gets interpreted on the client side and can contain additional scripts, the output of `server.php` can also be considered a (dynamically generated) client side component, as described Section II-A. In the rest of this section, we will first describe the code and functionality of each of the components we just described, then look at the interactions between them, and finally use the example to present typical cross-browser issues.

```
1  <html>
2    <head>
3      <script src="script.js" type="text/javascript">
4      </script>
5      <link href="style.css" rel="stylesheet"
6                                  type="text/css"/>
7    </head>
8    <body>
9      <h1>Ajax Search:</h1>
10       <input type="text" name="query" id="query" />
11       <input type="button" onclick="search()"
12                                  value="Search" />
13     <h2>Results:</h2>
14       <div id="stats"></div>
15       <ul id="results">
16       </ul>
17   </body>
18 </html>
```

Fig. 1. HTML page: `search.html`.

A. *Client-side Components*

1) `search.html`: An HTML file consists of a hierarchy of tags. The root tag (`<html>`) contains a `<head>` and a `<body>` tags. The `<head>` tag provides meta information about the HTML page and references to external or inline resources. In our example HTML page, `search.html`, the `<head>` tag refers to the the JavaScript file `script.js` and to the CSS stylesheet `style.css` (lines 3–6). The `<body>` tag contains the main web page elements to be rendered. In our example, such tag contains the header "Ajax Search", enclosed in an `<h1>` tag (line 9). It also contains an input text box with id = "$query$" (line 10) and a "Search" button (line 11) with an associated click event, $onclick$ = "$search()$". Finally, it contains a "Results" header, within an `<h2>` tag (line 13), followed by a `<div>` tag with id = "$stats$" and a `<ul>` tag with id = "$results$", which will eventually contain the results returned by the server (lines 14–16).

2) `script.js`: This JavaScript file, shown in Figure 2, contains the main client side logic and defines two functions, `search` and `updateResults`. As we saw earlier, the `search` function is linked to the `click` event of the "Search" button on the `search.html` page. When the button is clicked, this function is therefore invoked by the browser. Inside this function, at line 4, a common DOM API function (`document.getElementById`) is used to get a reference to the input text box using its unique identifier (`query`) and store it in the JavaScript variable `q`. Next, at line 5, the value contained in the input text box is used to build the URL of the server side component. More precisely, the input text value is used as the HTTP Request parameter `q`, which is added to the query string part of the URL. Next, at line 6, the function checks for the availability of the `XMLHttpRequest` (abbreviated as XHR) object. (This check is necessary because this object is not supported on versions of Internet Explorer prior to Version 7.) On lines 8–12, the script creates and submits an asynchronous XHR (popularly known as AJAX—Asynchronous Javascript and XML) request to the server side component `server.php`. The request contains a callback to function `updateResults`, which means that the function will be called when the state of the XHR object changes. (An XHR object transitions through states 0 to 4 during the request process.) Function `updateResults` first checks whether the

```
1  var xhr;
2
3  function search(){
4    var textBox = document.getElementById("query");
5    var url = "server.php?q="+textBox.value;
6    if (window.XMLHttpRequest)
7    {// code for IE7+, Firefox, Chrome, Opera, Safari
8      xhr=new XMLHttpRequest();
9      xhr.onreadystatechange=updateResults;
10     xhr.open("GET", url, true);
11     xhr.send(null);
12   }else
13   {// code for IE6, IE5
14     xhr=new ActiveXObject("Microsoft.XMLHTTP");
15     xhr.onreadystatechange=updateResults;
16     xhr.open("GET", url, true);
17     xhr.send();
18   }
19 }
20
21 function updateResults(){
22   if(xhr.readyState == 4){ // Response is ready
23     if(xhr.status == 200){ // Response status is OK
24       var results = document.getElementById('results');
25       results.innerHTML=xhr.responseText;
26
27       var stats = document.getElementById('stats');
28       stats.innerHTML= results.childElementCount +
29                             " results found !";
30     }else{
31       alert("Error while processing request!");
32     }
33   }
34 }
```

Fig. 2. JavaScript file: `script.js`.

```
1 h1{ text-shadow: #6374AB 2px 2px 2px; }
2 ul{ border:1px solid; }
```

Fig. 3. CSS stylesheet: `style.css`.

state of the XHR object is the final state (4) and the response status is "OK" (lines 22–23). If so, the script assigns the server side response, contained in field `responseText`, to the `results` container (lines 25–26). Next, the script computes the size of the `results` container and updates the `<div>` container identified by id = "$stats$" accordingly.

3) `style.css`: This stylesheet for the example, shown in Figure 3, applies two styles to the web page's elements. The first style adds a shadow to the text inside the main header (`<h1>`). The second style defines a border around the container for the results, which is a `<ul>` tag.

B. *Server-side Components*

1) `server.php`: A server side component obtains input from the client side through an HTTP Request object, whose parameters consist of string pairs `<name, value>` and are accessed by name. In our example, the script `server.php` (Figure 4) is the one invoked when a request is performed. At line 3, the script extracts parameter "q" and checks whether it is alphanumeric. If so, at line 4, it passes the parameter to function `lookupDB`, which finds and returns an array of relevant results. Then, on lines 5–6, iterates over the results, encloses each result in an `<li>` tag to create a list, and sends the list element back to the client component by printing it. (For the sake of space, we do not discuss functions `isAlphaNumeric` and `lookupDB`, whose details are not relevant, and simply assume that the functions are defined in file `common.php`, which is included at line 2.)

```php
<?php
include 'common.php';
if(isAlphaNumeric($_GET['q'])){
  foreach(lookupDB($_GET['q']) as $result){
    echo "<li>".$result."</li>\n";
  }
}
?>
```

Fig. 4. Server side PHP: `server.php`.

```html
<li>computerize</li>
<li>computerized</li>
<li>computerizes</li>
<li>computerizing</li>
<li>computerization</li>
<li>computerizable</li>
```

Fig. 5. Results returned by the server for query *"q=compute"*.

C. Interaction between Components

We now discuss the typical workflow for our example. The web browser initially makes a request for the HTML file `search.html`. When it receives the file from the server, it parses it, discovers the resources linked in the file, and requests them. It then renders the basic page, which contains no results. Assume that the user types the search string "compute" and presses the "Search" button. This action would result in an AJAX request being generated and sent to the server side component `server.php`, which would return the results data back to the browser. Figure 5 shows a possible response generated by `server.php`, whereas Figures 6(a) and 6(b) show the final result of the query as it would be displayed on two browsers.

As the screenshots in the figures show, there are three noticeable differences in the output of the browsers, and thus three potential issues. First, the heading "Ajax Search" does not have a shadow in Figure 6(b); this problem occurs because the `text-shadow` CSS style on line 2 of `style.css` is not supported by Internet Explorer 8. Second, the result count displayed in Figure 6(b) is different; the problem, in this case, is that Internet Explorer 8's DOM does not expose the `childElementCount` field, which is thus "undefined" (see `script.js`, line 28). Finally, the border around the results includes the bullets in Mozilla Firefox 3.5 but excludes them in Internet Explorer 8, due to formatting issues.

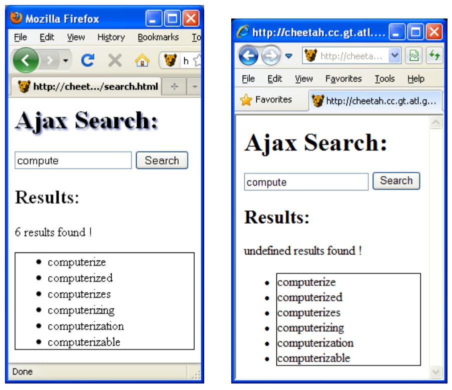
These types of issues are far from rare because developers tend to use mostly one browser during development and then port the code to other browsers. Even in the case where multiple browsers are considered from the beginning, it is difficult to test for all possible browsers and versions. Moreover, such testing is performed in a mostly manual manner, and is thus extremely time consuming (and often neglected). In fact, cross-browser issues are notoriously considered to be a major problem by most web application developers.

IV. Cross-browser Issues

In this section, we list the type of cross-browser issues and classify them based on their cause.

A. Types of Issues

Cross-browser issues manifest themselves either in the web page layout or in terms of functionality.



(a) Mozilla Firefox 3.5 (b) Internet Explorer 8

Fig. 6. Example web page rendered in two browsers and showing three issues: 1. Header shadow missing in Figure 6(b); 2. Result count `undefined` in Figure 6(b); 3. Bullets placed differently with respect to a bounding box.

1) Layout Issues: Layout issues are very common in web applications and result in differences in rendering the web page across browsers that are visible to the user. These issues can be classified as differences in element position, size, visibility, or appearance. Differences in positions and size are self explanatory. Differences in visibility consist of an element not being visible in one or more browsers. Finally, we define as differences in appearance when the an element's style or content is different across browsers.

2) Functionality Issues: These issues involve the functionality of a web application and are often due to differences in the way the script elements within a web page are executed by different browsers. Functionality issues typically limit the ability of a user to access specific web page elements, such as widgets. Although the users would identify the problem when they try to exercise the affected elements, these issues are sometimes more difficult to identify because they may not have any visible effect (*e.g.,* a button may be displayed correctly even if it does not work).

B. Underlying Causes

Browser compatibility issues are mainly due to one or more of the following reasons.

1) Non Compliant Browsers: Although the client side technologies have associated standards, they continue to evolve. At any point in time, there are thus features that are in the specification but are either not implemented or not implemented correctly in some browsers. These problematic features are often known, and there are web sites (*e.g.,* http://quirksmode.org) that maintain a list of such features to help web developers be aware of them and provide suitable workarounds in their code. The check performed at line 6 by the `script.js` script in Figure 2 is a typical example of one such workaround.

2) Extra Features in Browsers: Many browsers implement extra features that are not a part of the standard to provide more flexibility to developers. While web developers try to avoid using these features as much as possible, they do use them for convenience. For example, Internet Explorer supports conditional comments—comments that allow the commented out code to conditionally run in Internet Explorer while being

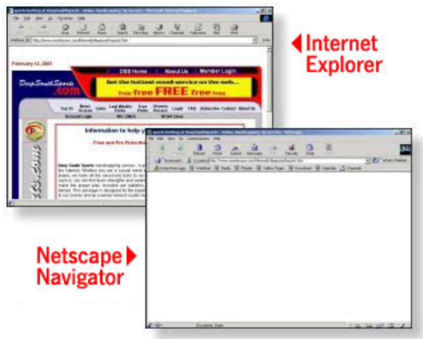


Fig. 7. Issue in Netscape due to an unclosed `table` tag (source: `http://netmechanic.com`).

ignored in other browsers. Conditional comments are often used by developers to include in a page a fix for certain issue present in Internet Explorer but not in other browsers.

3) Browser's Default Style: Each browser has a default stylesheet for HTML elements that defines the appearance of elements that have no specific style associated with them. A typical example of this is the different appearance of buttons and other form elements across browsers. Using the default style for certain HTML elements might cause them to appear differently across browsers.

4) Unavailable Resources: Local resources, such as fonts and browser plugins, may affect the way a page is rendered. In the case of fonts, for instance, if a specific font is unavailable, the browser would choose the best match from the set of available fonts on the system. Since it is left to the browser to find this best match, it is common for different browsers to choose fonts that do not appear similar to each other.

5) Syntactically Incorrect Pages: Most web browsers are lenient in the case of HTML pages that contain missing, misspelled, or misplaced HTML tags. Each browser, however, may handle such pages in a different manner, as there is no standard that describe how to do it. Consider, for instance, the example shown in Figure 7, where an unclosed HTML `table` tag results in a blank page in an old version of Netscape Navigator but is displayed properly in Internet Explorer.

V. Technique

The overall goal of our technique is to find dissimilar web page elements across browsers. To do this, our technique uses a reference browser and compares data from all the other browsers considered against it. For each web page analyzed, the approach works as follows. First, it collects information from each browser's DOM and captures a screenshot of the web page. Second, it identifies and marks variable elements in the reference browser's DOM. Third, it matches each browser's DOM to the DOM of the reference browser. Finally, it compares the matched information across browsers and generates a list of elements that have a mismatch. In the following subsections, we describe these four steps in detail.

A. Data Collection

To collect comparable visual data from the different browsers, each browser should be setup so that the size of the visible area where the web page gets rendered is the same across all browsers. In this way, our technique can capture screenshots of the same size. Once this is done, our technique repeatedly takes a screenshot of the visible area and scrolls down until the end of the page is reached to obtain the combined screenshot of the entire web page. (For pages that are small enough to fit on one page, this is obviously unnecessary.) A client side JavaScript program is then executed on each considered browser to collect partial DOM information that is then sent back to the server for processing. For every DOM node, the script collects the following properties (where we call *DOM element* the web page element associated with the DOM node):

- *tagname*: Name of the tag associated with the DOM element.
- *id*: Unique identifier of the DOM node, if defined.
- *xpath*: X-Path of the node in the DOM structure.
- *coord*: Absolute screen position of the DOM element.
- *clickable*: True if the DOM element has a click handler.
- *visible*: True if the DOM element is visible.
- *zindex*: DOM element's screen stack order, if defined.
- *hash*: Checksum of the node's textual content, if any.

B. Detection of Variable Elements

Variable elements on a web page are generated elements that may change when the page is reloaded, such as ads or statistics. These elements need to be ignored during comparison because they would likely result in false positives. To do so, our technique compares the DOM and screenshot information obtained from the reference browser over two subsequent, identical requests. First, the DOM trees are traversed together in breadth first fashion to check that the nodes are identical in both trees. Then, for matching DOM nodes, our technique compares the part of the screen capture corresponding to such nodes, as identified by the nodes' absolute screen coordinates. (This is basically a simplified version of our comparison algorithm, as here we are not doing a cross-browser check and expect the pages to be exactly identical, except for their variable elements.) All DOM nodes that reveal either a structural or a visual difference in this analysis are marked as variable and ignored in the subsequent steps.

C. Cross-browser Comparison — Structural Analysis

The goal of this phase is to match the DOM nodes obtained from different browsers. To do this, we use the algorithm presented in Algorithm 1, which consists of two procedures: `MatchDOMTrees` and `ComputeMatchIndex`.

`MatchDOMTrees` tries to match DOM tree nodes in every browser Br_i with those in the reference browser Br_0. This algorithm takes as input the DOM information for every browser considered, DOM_i, and generates the mapping Map between the DOM nodes of each browser and the DOM nodes of the reference browser, DOM_0. The `foreach` loop on line 3 iterates over the browsers. DOM_0 is then traversed in a breadth-first fashion using a $WORKLIST$. Initially, all the child nodes of the root node are added to the $WORKLIST$ (line 4). Until the $WORKLIST$ is empty, a node is extracted

Algorithm 1: Structural Analysis.

```
/* MatchDOMTrees */
Input  : $DOM_i$ for browsers $Br_0..Br_n$
         $Br_0$ : reference browser
Output: $Map[i]$ between $Br_0, Br_i$
1  begin
2      $Map \leftarrow$ list of size n, each item initialized to $\emptyset$
3      foreach $DOM_i$ where $i \in [1, n]$ do
4          $insert(root(DOM_0).children, WORKLIST)$
5          while $WORKLIST$ is not empty do
6              $node \leftarrow extract(WORKLIST)$
7              $bestMatch \leftarrow NULL$
8              $bestMI \leftarrow 0$
9              foreach unvisited $node_j$ from $DOM_i$ do
10                 $mI \leftarrow$ ComputeMatchIndex($node, node_j$)
11                 if $mI == 1.0$ then
12                     $Map[i].add(node, node_j)$
13                     $DOM_i.removeNode(node_j)$
14                 else if $bestMI < mI$ then
15                     $bestMatch \leftarrow node_j$
16                     $bestMI \leftarrow mI$
17                 end
18             end
19             if $bestMI > 0.5$ then
20                 $Map[i].add(node, bestMatch)$
21             end
22             $insert(node.children, WORKLIST)$
23         end
24         $removeExtraNodes(Map[i])$
25     end
26     return Map
27 end

/* ComputeMatchIndex */
Input  : $a, b$ where $a \in DOM_i$, $b \in DOM_j$
Output: $\rho$ (Match Index)
1  begin
2      $\alpha \leftarrow 0.9$
3      $\rho, \rho_1, \rho_2 \leftarrow 0$
4      if $(a.id \neq "") \wedge a.id == b.id$ then
5          $\rho \leftarrow 1$
6      end
7      else if $a.tagname == b.tagname$ then
8          $\rho_1 \leftarrow (1 - LevenshteinDistance(a.xpath, b.xpath)/$
                   $max(length(a.xpath), length(b.xpath)))$
9          foreach prop in {"coord", "clickable", "visible", "zindex",
           "hash"} do
10             if $a.prop == b.prop$ then
11                 $\rho_2 \leftarrow \rho_2 + 1$
12             end
13         end
14         $\rho_2 \leftarrow \rho_2/5$
15         $\rho \leftarrow (\rho_1 * \alpha + \rho_2 * (1 - \alpha))$
16     end
17     return $\rho$
18 end
```

from it and processed (lines 5–6). For each such node, the nodes in DOM_i are traversed to find the best match (line 9). To match two nodes, the technique computes a match index by invoking procedure `ComputeMatchIndex`, which is described below. In the case of an exact match (*i.e.,* when the match index value is 1.0), the mapping is added to Map, and the mapped DOM node is removed from DOM_i (lines 11–13). If an exact match is not found, the best matching nodes are added to Map but not removed from DOM_i (in case a better match for these nodes can be found later). Note, however, that the algorithm sets a minimum match index threshold of 0.5 (line 19) to avoid matching nodes that are too dissimilar. After mapping all the nodes, if more than one node from DOM_0 is mapped to the same node, the ones with a lower match index are removed (line 24).

`ComputeMatchIndex` computes the match index for each pair of nodes passed to it as parameters using the properties of the nodes that we discussed in Section V-A. First, it checks whether (1) the id is defined and (2) the two nodes have the same id. If so, it identifies the nodes as a perfect match and assigns 1 to the match index ρ (line 5), which is then returned. (Because ids are manually assigned by developers and are unique, two nodes with the same id are necessarily corresponding nodes in the two DOM trees.) If the ids do not match or are not defined, the algorithm compares the tagnames of the nodes (line 7). Although different nodes can have the same tagname, corresponding nodes cannot have different tagnames. Therefore, if the tagnames are not equal, the default value of ρ, zero, is returned. Otherwise, the algorithms computes the matching index using some of the other properties of the nodes. First, the algorithm computes the normalized Levenshtein distance between the xpaths for the two nodes and assigns its ones complement to ρ_1 (line 8).[1] Then, the algorithm computes the fraction of properties coord, clickable, visible, zindex, and hash that match between the two nodes, and assign the computed value to ρ_2 (lines 9–14). Finally, the algorithm computes the matching index by adding ρ_1 and ρ_2, suitably weighted (line 15). Because both ρ_1 and ρ_2 are values between zero and one, the value of the resulting matching index, ρ, is also between zero and one. The reason why ρ_1 is weighted considerably more than ρ_2 in the computation of ρ is because two corresponding nodes should have the same, or at least a very similar, xpath. Intuitively, the other properties are only used to break a "tie" between two very likely matches. It is worth noting that we verified experimentally that there are only very rare cases in which ρ_2 plays any role in deciding a match. In other words, in the absence of developer-defined ids for the nodes, the nodes' xpaths are a reliable indication of whether two nodes match.

To illustrate with an example, let us consider the DOM information for nodes n1 and n2 shown in Figure 8 using the JSON (Javascript Object Notation) format, and assume that there are 50 additional matching elements in the omitted part of the xpath properties for the two nodes. Because there is no developer-defined id, but the nodes have the same tagname, the algorithm would compute the match index for n1 and n2 using the other properties of the nodes. In this case, ρ_1 would be 0.98 (the edit distance between the two xpaths is 1, and the number of element in both sequences is 54), and ρ_2 would be 0.8 (four of the five properties match). The overal match index for the two nodes would therefore be approximately 0.96, which indicates a very likely match.

D. Cross-browser Comparison — Visual Analysis

DOM information can help us identify which page elements may be causing a problem. However, the DOM does not have information on how the element exactly appears on screen. Therefore, our technique also performs a visual analysis of the

[1]The Levenshtein distance between two sequences is defined as the minimum number of edits (*i.e.,* insertion, deletion, and substitution) needed to transform one sequence into the other [12].

```
// Node n1, from Mozilla Firefox's DOM
{ "tagname":"LI", "id":"", "xpath":"/HTML/BODY/.../UL/LI",
  "coord":"140,60", "hash":"0xA56D3A1C", "clickable":"0",
  "visible":"1", "zindex":"0" }

// Node n2, from Internet Explorer's DOM
{ "tagname":"LI", "id":"", "xpath":"/HTML/BODY/.../OL/LI",
  "coord":"140,60", "hash":"0x1C394C1B", "clickable":"0",
  "visible":"1", "zindex":"0" }
```

Fig. 8. Example DOM information.

page. The visual analysis, described in Algorithm 2, leverages the structural information computed in the previous step to (1) identify corresponding elements in the browser screenshots and (2) perform a graphical matching of such elements. In the rest of this section, we describe the different parts of the algorithm in greater detail.

`CrossBrowserTest` is the main function in the algorithm. It takes as input, for each browser Br_i, its mapping from the reference (Map_i), its screen capture information (SC_i), and the set of variable nodes in the reference browser (VN_0). As a preliminary step, the algorithm "grays out" the areas corresponding to variable nodes in the screenshots for the reference browser to eliminate possible false positives caused by them (line 2). It then processes the data from the different browsers. For each browser, the algorithm first finds the mapped variable nodes in it and grays out the corresponding areas as well (lines 5–6). It then visits the DOM tree nodes from leaves to root (*i.e.,* in a bottom-up fashion). For each DOM node ($node_j$), it gets the corresponding node $node$ in the reference browser, according to the mapping (line 8). The algorithm checks for positional shifts in the DOM node by comparing the relative distances of both $node$ and $node_j$ from their DOM containers (lines 9–10). If it finds such a shift, the technique verifies it in the areas of the screenshots that correspond to the DOM containers by invoking procedure `VChk`, described below (lines 11–16). If the difference is confirmed by `VChk`, the relevant DOM node is added to set $Mismatch_i$ (line 17).

Next, the algorithm compares the actual nodes, without considering their containers, for visibility, size, and appearance differences. Visibility and size differences are checked first, so as to limit the number of (more expensive) graphical comparisons. First, the algorithm checks for visibility differences by invoking procedure `visibilityDiff` and passing $node$ and $node_j$ as parameters (line 23). Procedure `visibilityDiff`, not shown here for space reasons, mainly compares the visibility attributes of $node$ and $node_j$ and returns a value that indicates whether they match. If they do not match, the algorithm adds $node_j$ in the $Mismatch_i$ set and specifies that the difference relates to visibility. If there is no visibility difference, the algorithm checks for size differences by invoking procedure `sizeDiff` on $node$ and $node_j$ (line 26). Similarly to `visibilityDiff`, procedure `sizeDiff` compares the dimensional attributes of the two nodes and returns a value that indicates whether they match. In case of a mismatch, the algorithm records the difference and its reason (line 27). If both visibility and position match, the algorithm compares the screenshots' graphical areas corresponding to $node$ and $node_j$ using again procedure `VChk` (lines 28–31).

Algorithm 2: Visual Analysis.

```
/* CrossBrowserTest */
Input  : $Map_i, SC_i$ for browsers $Br_0..Br_n$
         $Br_0$ : reference browser
         $VN_0$ : set of variable nodes in $Br_0$
Output: $Mismatch_i$ between $Br_0, Br_i$
1  begin
2      $SC_0 \leftarrow grayOutNodes(SC_0, VN_0)$
3      for $i \rightarrow 1$ to $n$ do /*for all browsers*/
4          $Mismatch_i \leftarrow \emptyset$
5          $VN_i \leftarrow getAllMappedNodes(VN_0, Map_i)$
6          $SC_i \leftarrow grayOutNodes(SC_i, VN_i)$
7          foreach $node_j \in Map_i, \notin VN_i$ do /*Bottom-Up*/
8              $node \leftarrow getMappedNode(node_j, Map_i)$
               // Check for positional shifts
9              $relDist1 \leftarrow relDistToContainer(node)$
10             $relDist2 \leftarrow relDistToContainer(node_j)$
11             if $diff(relDist1, relDist2)$ then
12                 $parent1 = parent(node)$
13                 $parent2 = parent(node_j)$
14                 $box1 \leftarrow parent1.box$
15                 $box2 \leftarrow parent2.box$
16                 if not $VChk(SC_0, SC_i, box1, box2)$ then
17                     $insert(Mismatch_i, parent2,$
18                                      $\text{"}content\ shifted\text{"})$
19                 end
20             end
21             $box1 \leftarrow node.box$
22             $box2 \leftarrow node_j.box$
23             if $visibilityDiff(node, node_j)$ then
24                 $insert(Mismatch_i, node_j,$
25                              $\text{"}visibility\ changed\text{"})$
26             else if $sizeDiff(node, node_j)$ then
27                 $insert(Mismatch_i, node_j, \text{"}changed\ size\text{"})$
28             else if not $VChk(SC_0, SC_i, box1, box2)$ then
29                 $insert(Mismatch_i, node_j,$
30                              $\text{"}changed\ appearance\text{"})$
31             end
32         end
           // Group Issue List
33         $Mismatch_i \leftarrow clusterNodes(Mismatch_i)$
34     end
35 end
```

```
/* VChk -- Visual Check */
Input  : $SC_i, SC_j$ for browsers $Br_i, Br_j$
         $Box_i, Box_j$: Image co-ordinates to compare
Output: Matches : Boolean value indicating a match or a mismatch
1  begin
2      $image_i \leftarrow crop(SC_i, Box_i)$
3      $image_j \leftarrow crop(SC_j, Box_j)$
       // average colors per $100px^2$ image area
4      $\alpha \leftarrow (numColors(image_i) \times 100)/area(Box_i)$
       // threshold based on color density and size
5      $threshold \leftarrow chooseThreshold(\alpha, Box_i)$
       // Earth Movers' Distance (EMD)
6      $emd \leftarrow getEMD(image_i, image_j)$
7      if $emd \leq threshold$ then
8          return true
9      end
10     return false
11 end
```

Also in this case, a mismatch is recorded by adding $node_j$ to set $Mismatch_i$ and specifying that the difference relates to appearance. Finally, the algorithm clusters the information in the $Mismatch_i$ set by aggregating differences that occur in neighboring areas on the screen, so that they can be reported together to developers (line 33). The clustering is performed in procedure `clusterNodes` using the visual coordinates of the nodes in the $Mismatch_i$ set. Other details about `clusterNodes` are of limited relevance and are not reported for space reasons.

TABLE I
THRESHOLD VALUES CHOSEN BY CHOOSETHRESHOLD.

	Small CD (< 1)	Large CD (≥ 1)
Small Image ($< 10^4$ square pixels)	1	0.5
Large Image ($\geq 10^4$ square pixels)	2	1

Procedure VChk performs the image comparison part of our technique. VChk takes as inputs two screen captures and two bounding boxes—set of coordinates of the graphical areas that must be compared. First, the algorithm extracts the relevant parts of the images from the screen captures (lines 2–3). It then compares the extracted images using the EMD metric (see Section II-B). As a threshold for EMD, the algorithm uses different values based on size and color density of the images, where the color density (CD) is computed as the average number of colors per 100 square pixels. In our early experiments, where we ran this part of the algorithm on a large number of screenshots coming from a wide range of pages, we found that the use of a single threshold was not ideal, as images with different characteristics tend to behave differently when rendered. Based on such findings, we defined the four values of the threshold that are shown in Table I.

VI. Empirical Evaluation

To assess the usefulness of our technique, we implemented it in a tool, WebDiff, and performed an empirical evaluation on a set of real web applications. More precisely, we investigated the following two research questions:

RQ1: Can WebDiff identify cross-browser issues in web applications?

RQ2: Can WebDiff identify such issues without generating too many false positives?

In the rest of this section, we describe our experimental setup, present the study performed to investigate RQ1 and RQ2, and discuss the results of the study.

A. Experimental Setup and Procedure

For our evaluation, we selected three widely used web browsers—Mozilla Firefox (Version 3.6), Google Chrome (Version 4.1), and Internet Explorer (Version 8.0)—and ran them on Microsoft Windows XP. First, we resized the three browsers so that they had the same viewport size (1000×800 pixels), that is, the same size for the portion of the browser containing the rendered web page (see Section V-A). Note that, although we performed this step manually for the study, the current version of WebDiff supports automated resizing as well.

The module of WebDiff that performs the screen capture and the collection of DOM information consists of a Python script. The script uses the win32api and the python image library to automate image capturing, page scrolling, and combination of partial visual information. To collect the information from the DOM, the script runs, within each web browser, a JavaScript URL program that queries and collects the information. For image processing, WebDiff leverages the OpenCV (http://opencv.willowgarage.com) computer vision library.

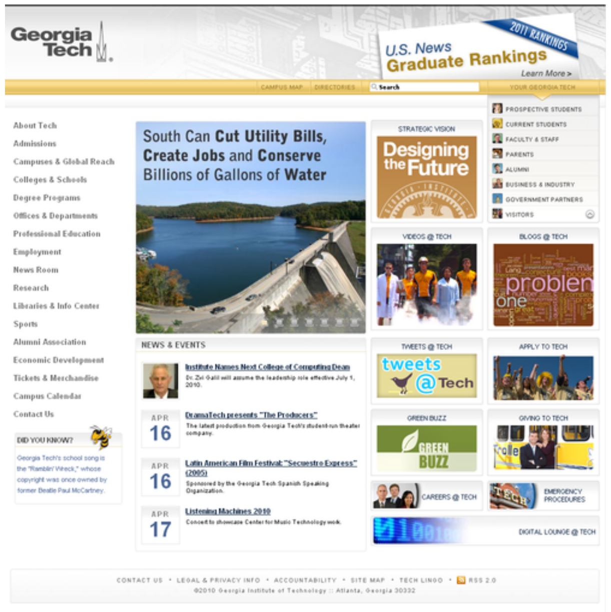
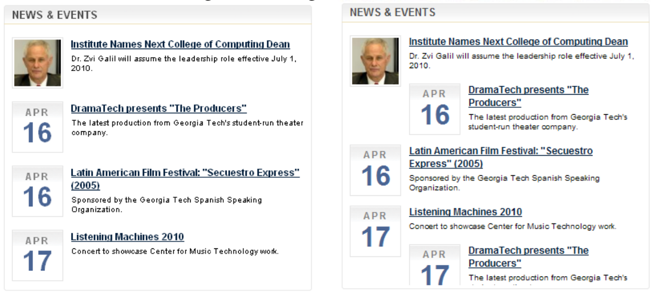


Fig. 9. Georgia Tech web site.


NEWS & EVENTS

Institute Names Next College of Computing Dean
Dr. Zvi Galil will assume the leadership role effective July 1, 2010.

APR 16 DramaTech presents "The Producers"
The latest production from Georgia Tech's student-run theater company.

APR 16 Latin American Film Festival: "Secuestro Express" (2005)
Sponsored by the Georgia Tech Spanish Speaking Organization.

APR 17 Listening Machines 2010
Concert to showcase Center for Music Technology work.

NEWS & EVENTS

Institute Names Next College of Computing Dean
Dr. Zvi Galil will assume the leadership role effective July 1, 2010.

APR 16 DramaTech presents "The Producers"
The latest production from Georgia Tech's student-run theater company.

APR 16 Latin American Film Festival: "Secuestro Express" (2005)
Sponsored by the Georgia Tech Spanish Speaking Organization.

APR 17 Listening Machines 2010
Concert to showcase Center for Music Technology work.

APR 17 DramaTech presents "The Producers"
The latest production from Georgia Tech's

Fig. 10. Issue with Georgia Tech's web site: shifted elements.

As subjects for our study, we selected nine web pages. The first page is the front page of our institutional web site, http://www.gatech.edu, shown in Figure 9. As the figure shows, the web site uses a professional template and leverages a variety of HTML features. We selected this page because we are aware of several cross-browser issues that affect it. Some of these issues are clearly visible in Figure 10: when rendered in Internet Explorer, certain elements on the web page are shifted. The remaining eight pages were chosen randomly. To perform a fair sampling of subjects, we picked them using a random link generator service provided by Yahoo! (http://random.yahoo.com/bin/ryl). The complete list of web pages considered is shown in Table II. As the table shows, the web pages considered cover a range of web sites, including company, university, commercial, and private sites.

To answer RQ1 and RQ2, we ran WebDiff on the nine web pages we selected and collected the reports generated by the tool. We then checked by hand each report and classified it as either a true or false positive.

B. Experimental Results and Discussion

Table III reports the results of the study. For each subject, the table reports the different types of cross-browser issues identified by WebDiff and manually confirmed as true positives: positional differences (*Pos.*), changes in size (*Size*),

TABLE II
SUBJECTS USED FOR THE STUDY.

Subject	URL	Type
GATECH	http://www.gatech.edu	university
BECKER	http://www.beckerelectric.com	company
CHESNUT	http://www.chestnutridgecabin.com	lodge
CRSTI	http://www.crsti.org	hospital
DUICTRL	http://www.duicentral.com	laywer
JTWED	http://www.jtweddings.com	photography
ORTHO	http://www.otorohanga.co.nz	informational
PROTOOLS	http://www.protoolsexpress.com	company
SPEED	http://www.speedsound.com	e-commerce

TABLE III
NUMBER OF CROSS-BROWSER ISSUES IDENTIFIED.

	# Faults identified					
Subject	*Pos.*	*Size*	*Vis.*	*Gen.*	*Tot.*	*FP*
GATECH	2	3	0	1	6	0
BECKER	2	12	0	2	16	1
CHESNUT	8	4	0	2	14	2
CRSTI	4	4	0	2	9	0
DUICTRL	9	8	0	2	19	4
JTWED	3	9	0	1	14	0
ORTHO	0	0	0	2	2	2
PROTOOLS	4	5	0	2	11	9
SPEED	23	5	0	2	30	3
Total	55	50	0	16	121	21

visual differences (*Vis.*), and general appearance differences (*Gen.*). In addition, the table shows the total number of true positives (*Tot.*) and false positive (*FP*) reported. Note that, as we discussed above, the issues reported here are the combined results of comparing the web pages rendered in Firefox (*i.e.,* our reference browser) with the pages rendered in Chrome and Internet Explorer.

As the results in Table III show, our technique was able to automatically discover and report a large number of cross-browser issues; overall, WEBDIFF reported 121 true issues of different types. In particular, WEBDIFF was able to identify the known issues in the Georgia Tech web site. These results provide a positive answer to RQ1: WEBDIFF can identify cross-browser issues in web applications.

As far as RQ2 is concerned, WEBDIFF reported 21 false positives for the nine subjects considered in the study, which corresponds to a 17% ratio of false positives. We believe this to be an encouraging result, for several reasons. First, the ratio is relatively low, as less than two in ten reports would be false positives. In other words, the developer time invested in investigating the reported issues would be well spent for the most part. Second, the ratio is actually lower than 17% for most subjects (six our of nine). Third, an investigation of the false positives showed that many of them are due to (1) minor differences in some elements containing text and background images that are mostly unnoticeable for the human eye and (2) presence of some variable elements that WEBDIFF failed to identify. We believe that careful engineering of the tool can help eliminate at least some of these false positives, as discussed in the next section.

One last point that is worth discussing is the cost of the approach. The analysis of each of the nine web pages considered took WEBDIFF less than five minutes to complete. Given that this time is clearly acceptable, especially for an analysis that can be run overnight and needs to be run only once per page, we did not perform a more detailed study of the cost of the technique.

C. Current Limitations

The current implementation of WEBDIFF has some limitations. In this section, we discuss these limitations and present possible ways to address them that we plan to investigate in future work.

Screen Capturing: Currently, WEBDIFF cannot capture screenshots of web pages that have elements whose scrolling behavior is different from that of the main page (*e.g.,* frames, fixed-position elements). The presence of these elements could cause either partial or erroneous visual information to be captured by WEBDIFF. Handling these elements is conceptually simple, but would require a more sophisticated engineering of the tool.

Embedded Objects: WEBDIFF has a limited ability to handle web sites that rely heavily on embedded objects, such as Adobe Flash elements. Since there is no readily available internal information for these elements, WEBDIFF currently considers them as a black box and ignores their internal details. To handle embedded objects, WEBDIFF would need to include a custom analysis component. The development of such a component, although possible, would also require an amount of development effort that we believe is not justified at this early stage of the research.

Detection of Variable Elements: To identify variable elements in a web page, WEBDIFF detects changes in web page elements when reloading the page twice in a small period of time. In this way, the tool may miss certain variable elements, which may in turn cause false positives, as it happened in our evaluation. A better detection of such variable elements would help reduce the number of false positives. For instance, the page could be reloaded several times and over a slightly longer period of time. Also, specific patterns could be used to identify variable elements (*e.g.,* in the case of external ads loaded in a page).

VII. RELATED WORK

The research work most closely related to ours is the technique for assessing web applications' compliance across browsers by Eaton and Memon [6]. Their technique requires developers to provide a set of positive and negative HTML pages. It then assigns a probability of being faulty to the HTML tags from these pages for a particular configuration space. Although useful, their technique requires developers to manually classify a large number of pages. Moreover the technique can reveal only issues related to HTML tags, and not problems related to other client-side components, such as style sheets or scripts.

More recently, Tamm released a tool that leverages both DOM and visual information to test the layout in one specific browser and find layout faults [7]. Although this tool uses the same information we use, it requires to manually change the web page being analyzed to hide and show elements while taking multiple screenshots. Our experience with a similar approach convinced us that it is too expensive and, at the same time, unable to detect many relevant issues.

Current industrial web authoring and testing tools have limited support for automated testing across browsers. Tools such as Microsoft Expression web [5] and Adobe BrowserLab [4], for instance, simply present side-by-side screenshots (or renderings) to the developers. While this helps automating part of the process, the developers still have to identify, understand, and fix layout issues manually.

There is related work in understanding web pages from a purely visual perspective. In particular, Cai and colleagues propose the VIPS algorithm [13], which segments a web page's screenshot into visual blocks to infer the hierarchy from the visual layout, rather than from the DOM. A limitation of their approach is that it assumes a fixed layout for a web page (*i.e.,* box header on top, main content in the middle, and footer at the bottom of the page) and clusters visible elements based on this assumption. While this might work for some class of applications, it is unlikely to work in general for web pages with complex layouts. Moreover, using the DOM allows us to report the specific HTML tag that is responsible for an issue, which can help developers understand and eliminate the issue.

Existing web testing tools, such as Selenium [14], help developers write or record test scripts or macros that can then be run on multiple browsers. All of these tools are limited by the fact that the test oracles need to be developed mostly manually. In fact, our technique could be combined with these tools and provide an automated test oracle that can compare executions across web browsers. More generally, our technique could be combined with any test-input generation technique for web applications (*e.g.,* [15]–[19]).

VIII. Conclusion

Cross-browser issues are a relevant and serious problem for web application developers. Existing techniques for detecting and fixing these issues are however still immature. In particular, many such techniques are limited by the fact of requiring a considerable amount of manual effort. In this paper, we presented an approach for automated identification of cross-browser issues that addresses the limitations of existing techniques. To do so, our technique compares web pages by leveraging both their structural (web page's DOM) and visual (web page's snapshots) characteristics. We implemented our technique in a tool, WebDiff, and used WebDiff to perform an evaluation on a set of real web application. Our results show that our technique is effective in identifying cross-browser issues and has a low false positive rate (below 20%).

There are several possible areas for future work. First, we will improve the implementation of WebDiff to eliminate some of the limitations described in Section VI-C, so as to further reduce the number of false positives generated by WebDiff. Second, and somehow related, we will investigate ways to improve histogram-based image differencing, which can also help the technique further reduce the false positives ratio. Third, we will perform additional experiments with the improved version of WebDiff and measure its performance.

A longer-term direction for future work is the extension of our approach to the case of versions of a web applications for desktop and mobile platforms. This is a challenging problem, as the application will necessarily look different in the two platforms, but it should provide the same, or at least similar, functionality.

Acknowledgments

This work was supported in part by the NSF awards CCF-0916605 and CCF-0725202 to Georgia Tech.

References

[1] W3Schools.com, "Browser statistics month by month," http://www.w3schools.com/browsers/browsers_stats.asp, May 2010.

[2] Cambridge Network, "Estate agents must update web browser compatibility ahead of microsoft announcement," http://www.cambridgenetwork.co.uk/news/article/default.aspx?objid=69332, March 2010.

[3] The Korea Times, "Korea sticking to aging browser," http://www.koreatimes.co.kr/www/news/biz/2010/02/123_61463.html, February 2010.

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

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

[6] 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.

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

[8] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10(1), pp. 100–107, 1998.

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

[10] G. Bradski and A. Kaehler, *Learning OpenCV*. O'Reilly Media, September 2008.

[11] 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.

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

[13] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "Vips: a vision-based page segmentation algorithm," Microsoft Research, Tech. Rep., November 2003.

[14] OpenQA, "Selenium web application testing system," http://seleniumhq.org/, May 2010.

[15] 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*, 6-10 2010, pp. 127 –136.

[16] 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*. New York, NY, USA: ACM, 2008, pp. 261–272.

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

[18] 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*, 2002, pp. 280–285.

[19] 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 the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 145–154.