Automated Cross-Browser Compatibility Testing

Ali Mesbah*
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

Mukul R. Prasad
Trusted Systems Innovation Group
Fujitsu Laboratories of America
Sunnyvale, CA, USA
mukul.prasad@us.fujitsu.com

ABSTRACT

With the advent of *Web 2.0* applications and new browsers, the cross-browser compatibility issue is becoming increasingly important. Although the problem is widely recognized among web developers, no systematic approach to tackle it exists today. None of the current tools, which provide screenshots or emulation environments, specifies any notion of cross-browser compatibility, much less check it automatically. In this paper, we pose the problem of cross-browser compatibility testing of modern web applications as a 'functional consistency' check of web application behavior across different web browsers and present an automated solution for it. Our approach consists of (1) automatically analyzing the given web application under different browser environments and capturing the behavior as a finite-state machine; (2) formally comparing the generated models for equivalence on a pairwise-basis and exposing any observed discrepancies. We validate our approach on several open-source and industrial case studies to demonstrate its effectiveness and real-world relevance.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Dynamic analysis, web testing, cross-browser compatibility

1. INTRODUCTION

Web applications pervade all aspects of human activity today. The web browser continues to be the primary gateway channeling the end-user's interaction with the web application. It has been well known for some time that different web browsers render web content somewhat differently [18, 24, 25, 26]. However, the scope and impact of this problem has been rapidly growing due to two, fairly recent trends.

First, modern, rich-content web applications have a heavy client-side behavioral footprint, i.e., they are designed to execute significant elements of their behavior exclusively on the client-side, typically within the web browser. Further, technologies such as AJAX [12], Flash, and event-handling for dynamic HTML, which support this thick-client behavior, are the very aspects in which web browsers differ.

Second, recent years have seen an explosion in the number of available web browsers. There are nearly 100 different web browsers available today [31]. Coupled with the different kinds and versions of operating systems in which these operate, this yields several hundred different client-side environments in which a web application can be used, each with a slightly different client-side rendering of the application [7, 9]. In the context of the above problem, the key contributions of this paper are:

- We define the cross-browser compatibility problem and present an in-depth discussion of the issues surrounding possible solutions to it.
- We present a systematic, fully-automated solution for cross-browser compatibility testing that can expose a substantial fraction of the cross-browser issues in modern dynamic web applications.
- We validate our approach on several open-source as well as industrial case studies to demonstrate its efficacy and real-world relevance.

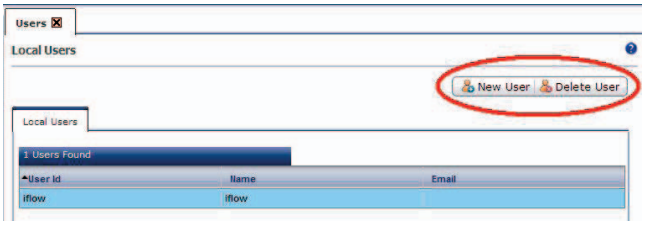
The rest of the paper is organized as follows. In the next section we discuss the genesis of the cross-browser compatibility problem and how it plays out in modern web applications. We present a formal characterization of the problem and put the solutions offered by this paper in that context. Section 3 reviews related work on cross-browser testing. In Section 4 we present our proposed approach for performing cross-browser compatibility testing, followed by a description of the salient implementation aspects of that approach, in Section 5. Section 6 presents an empirical evaluation of our approach on some open-source as well as industrial case studies. In Section 7 we discuss the strengths and weaknesses of our solution and lessons learnt from our experience. We conclude the paper in Section 8.

2. CROSS-BROWSER COMPATIBILITY

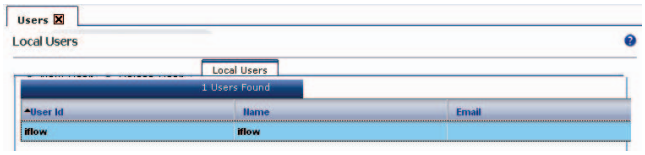
Problem Genesis. The *cross-browser compatibility (CBC)* problem is almost as old as the web browser itself. There are several reasons for its genesis and growth in recent years. The earliest instances of this problem had

*This author was a visiting researcher at Fujitsu Laboratories of America, when this work was done.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA



(a) Rendering in IE 8.0.



(b) Rendering in Firefox 3.5.

Figure 1: CBC issues in an industrial *Web 2.0* application: missing widgets in Firefox.

its roots in the parallel evolution of different browsers and the lack of standards for web content. Currently, standards do exist for basic web content (e.g., HTML 4.0) and for scripting languages such as JavaScript, JScript, and ActionScript (e.g., ECMA-262). However, in their bid to be inclusive and render websites that do not adhere to these standards (e.g., legacy websites) web browsers typically implement their own extensions to the web standards and in doing so differ in their behavior. The explosive growth in the number of browsers and client-side environments has only exacerbated this issue. In addition, modern, dynamic web applications execute an increasingly bigger fraction of their functionality in the client-tier i.e., within the end-user's web browser, thus further amplifying differences in observed behavior. Cross-browser compatibility is widely recognized as an important issue among web developers [26] but hardly ever addressed directly during the software development process. Typically web applications are developed with a single target client-side configuration in view and manually tested for a few more, as an after-thought.

Look-and-feel versus Functionality. The common perception is that the CBC problem is confined to purely 'look-and-feel' differences on individual screens. While that was indeed the case for the previous generation of web applications, the problem plays out in a more systemic, functional way in modern *Web 2.0* applications. Thus, we believe that the issue should be addressed as a generalized problem of checking functional consistency of web application behavior across web browsers, rather than purely a 'look-and-feel' issue. This view is also held by several web developers [10].

The following scenario illustrates how the problem plays out in modern web applications. The human end-user interacts with the web application, through the browser, by viewing dynamically served content and responding with user-events (e.g., `clicks`, `mouse-overs`, `drag-and-drop` actions) and user data (e.g., form data or user authentication data). The client-side machine in turn interacts with the server-side tiers of the web application by sending requests to it, based on the user input and the web application work-flow. The server side responds with new web content which is served by the client-machine to the user through the browser. Differences in the serving of a particular screen on the web browser can influence the set and nature of actionable elements available on that screen, which can influence what actions the user can and does take on the current screen. This can have a cascading effect by altering requests sent

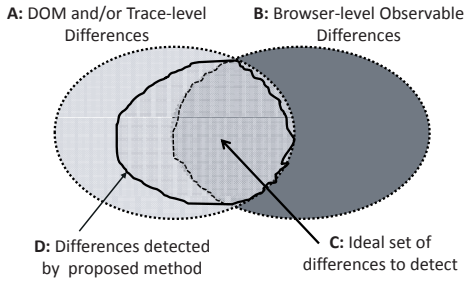


Figure 2: Browser incompatibility landscape.

from the client-tier to the server tiers, the response received and subsequent content served on the client, thus influencing the overall evolution of the web application on the client side. Figure 1 shows screen-shots taken from a real enterprise (proprietary) *Web 2.0* application, illustrating this scenario. While the application renders correctly in Internet Explorer, the Firefox rendering of it is missing the `New User` and `Delete User` widgets, marked with a red eclipse in IE. Thus, the human user would be unable to exercise these features from this screen, thereby completely removing a very important set of behaviors, namely the application administrator being able to add and remove users, from the evolution of the web application under Firefox.[1]

Problem Definition. Our working definition of the CBC problem, for the purposes of this paper, is based on the Venn diagram of Figure 2. The set B depicts those cross-browser differences that can be observed by the human user on the web browser, while the set A represents those cross-browser differences that are reflected as differences in the client-side state of one or more screens (e.g., differences in the DOM representation or CSS properties of various DOM elements) or differences in the set of possible traces, i.e., alternating sequences of screens and user actions, on the client-side. Further, there is the set of differences $B - A$ which can be visually observed by the user but is not reflected as a difference in the client-side state or in the evolution of that state. These differences are typically, though not necessarily, stylistic and stem from different browsers' rendering of *identical* web content (e.g., CSS files). Although such differences are important in their own right, they are not the focus of this paper. There is also the set $A - B$ of differences, which exist in the DOM representation but do not result in any user-observable differences in behavior. We argue that these differences are not relevant since the final metric of incompatibility is that which can be observed by the end user. The set $C = A \cap B$, which comprises the differences reflected in both the user-observable behavior as well as the underlying client-side state, represents the most interesting target for our technique. However, as explained below in Section 4, typically the set of differences detected by our technique is a set D, where $C \subseteq D \subseteq A$.

3. RELATED WORK

Currently there are several tools and services, both open-source and commercial, that claim to address the problem of cross browser compatibility checking. They fall under two broad categories, namely:

Tools capturing screenshots. These web services can capture and provide screen-shots of how a particular

[1] We verified that this was an issue of DOM-level differences, not simply one of screen-resolution or the size of the browser window.

URL, specified by the user, would be rendered, under various client-side platform combinations of specific browsers, operating system and screen resolutions. Typical offerings include IE NetRenderer [15] and ieCapture [14], which focus on Internet Explorer as well as others like BrowserShots [7], BrowserCam [3], Browser Photo [6], and Litmus [20] which include a broader set of browser-OS combinations.

Tools providing emulation environments. These tools and web services offer a slightly more advanced functionality than the previous category in that they allow the user to interact with a particular web application as if it were being served on a particular client-platform (browser-OS combination). Representatives in this category include Adobe BrowserLab [5] and CrossBrowserTesting.com, which span a wide variety of client-side platforms, as well as more specific ones like BrowserCamp [4] targeted for Mac OS X browsers, Xenocode Browser Sandbox [32] for windows-based environments, IETester [16] which is specific to IE and iPhoney [17] which is an emulator for the iPhone platform.

The offerings in both of the above categories suffer, in varying degrees, from the following fundamental limitations:

1. **Compatibility Checking:** None of the above tools specifies any notion of cross-browser behavioral compatibility, much less check it automatically. It is left to the human user to define their own notion of compatibility and validate it on observed behavior, usually manually. This makes the whole process very ad hoc and hence the quality of the check itself, very suspect.
2. **Behavioral Coverage:** Modern dynamic web applications display rich, stateful behavior which cannot be adequately covered with a few screenshots or manual examination of a few traces. The above solutions do not provide any automatic method of systematically and comprehensively exploring web application behavior. Such an exploration is typically needed to expose bugs embedded deep in the web application work-flow.
3. **Scalability:** Since the process of actually browsing the web application under different environments and comparing observed behavior is completely manual, it is difficult to apply the tools to several browser-OS platforms in a consistent, repeatable manner.

The *Acid Tests* [2] developed under the *Web Standards Project* present a complementary approach towards achieving cross-browser compatibility. The Acid Tests are a battery of 100 tests that check a given browser for enforcement of various W3C and ECMA standards. However, the fact that almost all common browsers fail some, and sometimes most of these tests[2] is another indication that cross-browser compatibility issues will continue to exist in the foreseeable future. A related effort in this category is the *SputnikTests* [30] project which provides a conformance test suite to check JavaScript engines for ECMAScript 3 compliance. These approaches are complementary to ours since they check web browsers (or their component engines) for standards' compliance rather checking the real-time behavior of a specific web application of interest.

Another related body of work is that of GUI testing [21, 33]. Indeed, web applications can be seen as a special class of general GUI-based applications. However, the specific architecture, execution environment (the web browser), implementation technologies (JavaScript, Ajax) and API standards (W3C HTML, CSS, DOM) of web applications allow us to employ a simple finite state machine model and develop very efficient equivalence checking algorithms around it, whereas [21] use GUI forests, event-flow graphs, and integration trees.

Eaton and Memon [11] presented one of the first techniques for automated identification of cross-browser issues on specific web pages. Their technique identifies potentially problematic HTML tags on a page using a user-supplied, manually generated classification of good and faulty pages. Concurrent to our work, Choudhary et al. [8] have proposed the WebDiff tool, which analyzes the DOM as well as screen-shots of pairs of screens to automatically locate cross-browser issues. In both these papers the focus is on identifying cross-browser differences in individual screens. In contrast, our approach also identifies more systemic cross-browser issues that manifest in the overall trace-level behavior of the web application. Thus, these techniques are somewhat complementary to our work.

4. OUR APPROACH

Our overall approach consists of a two-step process. The first step is to automatically crawl the given web application under different browser environments and capture and store the observed behavior, under each browser, as a finite-state machine *navigation model*. The crawling is done in an identical fashion under each browser to simulate exactly the same set of user-interaction sequences with the web application, under each environment. The second step consists of formally comparing the generated models for equivalence on a pairwise-basis and exposing any observed discrepancies. The crawling technology (i.e., model generation process), the generated model and the equivalence check are discussed in more detail in the following sections.

4.1 State Space Exploration

Our approach for automatically exploring the web application's state space is based on the Crawljax [22, 23] work. 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 or capabilities of Crawljax the interested reader is referred to [22, 23]. We have extended Crawljax in several ways, to apply it to the CBC problem. This is discussed in Section 5.

4.2 The Navigation Model

The navigation model produced by crawling the web application is a finite-state machine. The states represent the screens observed by the end-user, on the web browser, and the transitions represent user-actions (e.g. a button click) that cause the web application to transition from one screen to another. Each transition is labeled with the user action that caused it. Each state (screen) is represented by its underlying programmatic representation as viewed by the web browser. For the purpose of meaningfully and efficiently comparing multiple navigation models we found it useful to view and analyze the navigational model hierarchically. The top level is a graph representation of the finite-state machine with the states (screens) represented merely as unnamed vertices. We refer to this as the *state graph*. At the second level is the full programmatic representation of

[2] IE8 scores 20/100 on the Acid3 Test [1].

each screen/state which we refer to as the *screen model* of the state. These are formally defined below. Conceptually, the state graph captures the set of traces, i.e., alternating sequences of user-actions and screen transitions, without referring to the details of each screen, whereas the screen model of each screen captures precisely this detail, but without any knowledge of transitions leading up to or out of the screen.

Definition 1 (State Graph) *A State Graph G is a labeled, directed graph, with a special designed start vertex. It is denoted by a $5-tuple$, $G(V, E, o, \Sigma, \mathcal{L})$, where V is the set of vertices, E the set of (directed) edges, o the special designated start vertex, Σ an alphabet of labels and $\mathcal{L} : E \rightarrow \Sigma$ is a labelling function that assigns a label from Σ to each edge. Further, G has the following specific characteristics:*

1. *Each label from Σ can appear on at most one outgoing edge from a given vertex,* i.e.*, no vertex can have two (or more) outgoing edges with the same label, although the same label can appear on multiple edges in the overall graph.*
2. *G can have multi-edges,* i.e.*, $\exists e_1, e_2 \in E : s(e_1) = s(e_2)$ and $d(e_1) = d(e_2)$.*
3. *Each node in V is reachable from the root r, i.e., G is composed of a single connected component.*
4. *G can be cyclic.*

The special node, o typically denotes the start or `index` screen (state) of the web application. For an edge $e = (u, v)$ we refer to its source vertex u by $s(e)$ and destination vertex v by $d(e)$. We use $Out(v)$ and $In(v)$ to denote, respectively, the set of outgoing and incoming edges of vertex v.

Definition 2 (Screen Model) *A Screen Model T is a rooted, directed, labeled tree. It is denoted by a 5-tuple, $T(Q, D, r, \Lambda, \delta)$, where Q is the set of vertices, D is the set of directed edges, $r \in Q$ is the root vertex, Λ is a finite set of labels and $\delta : Q \rightarrow \Lambda$ is a labelling function that assigns a label from Λ to each vertex in Q.*

In our current implementation, the screen model is essentially an abstracted version of the DOM tree of a given screen, displayed on the web browser. Note that this model could easily be generalized to include (and compare) other aspects of the client-side state such as JavaScript variable values or CSS properties. Q, D and r have obvious meanings in terms of the DOM tree. The label of a particular node is a combination of the HTML tag-name of the DOM node as well as a set of name-value pairs representing the attributes of the DOM node. We use a single label as a convenient abstraction for these multiple pieces of data, in order to formalize the model and the equivalence check performed on it (Section 4.3). Details of the actual implementation are discussed in Section 5.

4.3 Equivalence Checking

The equivalence check of the navigation models mirrors our hierarchical view of them. We first extract the state graph models from the respective navigation models and compare them at a trace-level. This provides a set of trace-level differences between the two navigation models, i.e. a set of traces that exist in one model and not in the other, and vice versa. It also pairs up each screen in the first model with its most likely counter-part in the second model. Next, candidate matching pairs of screens, produced by the first step, are compared in terms of their underlying DOM representations. This unearths detailed screen-level differences

Algorithm 4.1: ModelEquivCheck(M_1, M_2)

```
G1 ← StateGraph(M1)
G2 ← StateGraph(M2)
if (TraceEquivCheck(G1, G2) = false)
  then OutTraceDiff(G1, G2)
V1 ← Vertices(G1)
for each v1 ∈ V1
do  if (v1.match ≠ null)
      then  v2 ← v1.match
            T1 ← GetScreen(M1, v1)
            T2 ← GetScreen(M2, v2)
            if (ScrnEquivCheck(T1, T2)
                = false)
              then
                OutScrDiff(T1, T2)
```

whose effects may be confined to the individual screen or may play into trace-level differences. Our proposed method attempts to prune out some of the provably benign differences (discussed in Section 5). Note that providing a detailed diagnosis for the cause of each observed difference is beyond the scope of this work at its current state.

Algorithm 4.1 is the overall algorithm for matching a pair of navigation models M_1 and M_2, generated through the crawling. The function StateGraph returns the underlying state graph, which is an input to the TraceEquivCheck (Algorithm 4.2). TraceEquivCheck checks and annotates the two graphs. It returns false if $G_1 \not\equiv G_2$. OutTraceDiff extracts trace-level differences from the annotated graphs. V_1 is the set of vertices from G_1. The function GetScreen extracts and returns the detailed screen representation of a vertex v of a state graph from its corresponding navigation model. This is used for the equivalence check, ScrnEquivCheck, done by the screen-matching algorithm, explained later. The function OutScrDiff extracts and presents these screen-level differences to the user.

Trace Equivalence Check. The notion of trace equivalence on state graphs and the precise algorithm to perform such a comparison are discussed in the following.

Definition 3 (Trace Equivalence) *Given state graphs $G_1(V_1, E_1, o_1, \Sigma, \mathcal{L}_1)$ and $G_2(V_2, E_2, o_2, \Sigma, \mathcal{L}_2)$, G_1 and G_2 are said to be trace-equivalent, denoted as $G_1 \equiv G_2$, if and only there exists a bijective mapping function $\mathcal{M} : V_1 \rightarrow V_2$ such that the following are true:*

1. *$\forall u, v \in V_1, (u, v) \in E_1 \Leftrightarrow (\mathcal{M}(u), \mathcal{M}(v)) \in V_2$*
2. *$\forall e_1(u_1, v_1) \in E_1, e_2(u_2, v_2) \in E_2$ such that $\mathcal{M}(u_1) = u_2$ and $\mathcal{M}(v_1) = v_2 \Rightarrow \mathcal{L}_1(e_1) = \mathcal{L}_2(e_1)$*
3. *$\mathcal{M}(o_1) = o_2$*

Algorithm 4.2 implements the trace-level equivalence check on the state graphs G_1 and G_2 as an isomorphism check. The function Out(v) returns the set of outgoing edges of vertex v, Label(e) returns the label of edge e and the function Lookup(l, *edgeSet*) returns an edge having the label l from the set of edges *edgeSet* or *null* if none exists. Dest(e) returns the destination vertex of edge e. It is assumed that the *match* field of each edge and the *visited* field of each vertex is initialized to *false* and the *match* field of each vertex in is initialized to *null* (in G_1 and G_2). The above algorithm is a simple variant of depth-first search and linear-time in the sizes of G_1, G_2 i.e., $O(|V_1| + |V_2| + |E_1| + |E_2|)$.

It is important to note here that we have chosen to use the same alphabet of labels Σ for both G_1 and G_2, in order to develop and present the theoretical under-pinnings of our approach. However, in practice, edge labels representing otherwise identical transitions, also have cross-browser differences. Our tool implementation, described in Section 5,

Algorithm 4.2: TRACEEQUIVCHECK(G_1, G_2)

```
procedure MATCH(u1, u2)
 u1.visited ← true
 u2.visited ← true
 u1.match ← u2
 u2.match ← u1
 for each e1 ∈ OUT(u1)
   do { e2 ← LOOKUP(LABEL(e1), OUT(u2))
        if (e2 ≠ null)
          then { v1 ← DEST(e1)
                 v2 ← DEST(e2)
                 if ((v1.visited = false)&
                 (v2.visited = false))
                   then { e1.match ← true
                          e2.match ← true
                          edgeCt += 2
                          MATCH(v1, v2)
                   else if ((v1.match = v2)&
                     (v1.visited = true)&
                     (v2.visited = true))
                       then { e1.match ← true
                              e2.match ← true
                              edgeCt += 2
main
 global edgeCt
 edgeCt ← 0
 o1 ← STARTVERTEX(G1)
 o2 ← STARTVERTEX(G2)
 MATCH(o1, o2)
 if (edgeCt = |E1| + |E2|)
   then return (true)   comment: G1 ≡ G2
   else return (false)  comment: G1 ≢ G2
```

uses a set of abstractions and transformations to reconcile these differences and establish edge-label equivalence.

Theorem 1 *Algorithm 4.2 returns $G_1 \equiv G_2$, if and only if they are trace equivalent according to Definition 3.*

We have not included the formal proof of the above theorem (and others below) in the paper due to space constraints. However, the interested reader can find them at this link. [3]

In the case where Algorithm 4.2 finds that $G_1 \not\equiv G_2$, consider the sub-graphs implied by the partial match produced by the algorithm, i.e. the sub-graph implied by all nodes and edges that were matched to a counter-part in the other graph. Specifically, consider subgraph $G_1'(V_1', E_1', o_1, \Sigma, \mathcal{L}_1')$ of G_1 , where $E_1' = \{e \in E_1 : e.match = true\}$, $V_1' = \{v \in V_1 : v.match \neq null\}$, $\mathcal{L}_1' : E_1' \rightarrow \Sigma$ and $\mathcal{L}_1'(e) = \mathcal{L}_1(e) \forall e \in E_1'$. The implied sub-graph G_2' of $G_2(V_2', E_2', o_2, \Sigma, \mathcal{L}_2')$ can be similarly defined. As the following theorem states, subgraphs G_1' and G_2' are indeed valid state graphs themselves and trace-equivalent as per Definition 3.

Theorem 2 *When Algorithm 4.2 certifies $G_1 \not\equiv G_2$ the subgraphs G_1' and G_2' induced by the partial match computed by it, are valid state graphs and trace-equivalent as per the criteria of Definition 3.*

Further, it can be shown that when Algorithm 4.2 returns $G_1 \not\equiv G_2$, it not only produces a trace equivalent partial match but actually a *maximal* partial match, i.e. there do not exist a pair of edges e_1 and e_2, where $e_1 \in E_1$ but $e_1 \notin E_1'$ and $e_2 \in E_2$ but $e_2 \notin E_2'$ which can be added to G_1' and G_2' respectively, along with their source and sink nodes such that the resulting graphs would also be trace-equivalent.

Theorem 3 *If $G_1 \not\equiv G_2$, the trace-equivalent partial matches G_1' and G_2' computed by Algorithm 4.2 are maximal matches.*

[3] URL for proofs of the above theorems: `http://www.flacp.fujitsulabs.com/~mprasad/2b5b385a6d/proofs.pdf`

Note, that although the algorithm computes *maximal* matches, it is easy to show by a simple example that the match need not be the *maximum* match possible. It is possible to have a variant of Algorithm 4.2 that back-tracks on matching decisions made in order to compute the absolute maximum match. In our experience, the maximal match computed by the linear-time algorithm above is sufficient for most practical purposes.

Screen Equivalence Check. Since the screen model is represented as a rooted, directed, labeled tree (Definition 2), it is natural to compare two given screen models T_1 and T_2 based on the isomorphism of the respective trees. Thus, screen models $T_1(Q_1, D_1, r_1, \Lambda, \delta_1)$ and $T_2(Q_2, D_2, r_2, \Lambda, \delta_2)$ are said to be equivalent, denoted $T_1 \equiv T_2$, if and only if there exists a bijective mapping $\mathcal{N} : Q_1 \rightarrow Q_2$ such that:

1. $\mathcal{N}(r_1) = r_2$
2. $\forall q \in Q_1, \delta_1(q) = \delta_2(\mathcal{N}(q))$
3. $\forall u, v \in Q_1, (u, v) \in D_1 \Leftrightarrow (\mathcal{N}(u), \mathcal{N}(v)) \in D_2$

Since screen models are rooted, labeled trees, screen matching (the function call SCRNEQUIVCHECK(T_1, T_2) in Algorithm 4.1) can be performed in linear time by a simple variant of the tree isomorphism algorithm originally due to Aho, Hopcroft and Ullman [13]. However, our implementation is built upon a more contemporary solution to this problem that respects the special syntax and structure of DOM trees. Further details of this check are presented in Section 5.

5. TOOL IMPLEMENTATION

We have implemented our CBC testing approach in a tool, called CROSST, which comprises two components. For the state exploration component, we have used and extended on the open source AJAX crawler CRAWLJAX [23].[4] The input consists of the URL of the web application under examination, a list of supported browsers (IE, Firefox, and Chrome), and a list of user interface elements for inclusion and exclusion during the crawling execution. Using the given settings, the web application is crawled in each of the browsers and the inferred navigation model is serialized and saved into the file system. The implementation of the navigation model itself is based on the JGraphT[5] library. The nodes are state abstractions in terms of the browser's DOM tree and the edges are comprised of the DOM elements and event types that cause state transitions. In the second component, the saved navigation models are used to automatically conduct a pair-wise comparison of the web application's behavior in the given browsers, which itself has two parts, namely, trace-level and screen-level comparison.

Trace-level Comparison. For detecting trace-level differences, we have implemented Algorithm 4.2 in Java, which produces as output a list of edges and nodes that are mismatched in either of the two compared graphs. In order to compare the labels of two edges, as defined in Definition 3, our method retrieves the DOM element and the corresponding event type from each edge and tries to reconcile the two labels. Two edges are considered a match if the edit distance between the two based on a combination of event types, tag names, attributes and their values, as well as the XPath positions of the two elements on the corresponding DOM trees, is lower than a similarity threshold. The edge comparison method is fully reconfigurable: for instance, if persistent ID attributes are used for elements in a given web application, the method can be easily configured to use the IDs instead of the combination sketched above.

[4] `http://crawljax.com`

[5] `http://jgrapht.sourceforge.net`

Table 1: Examples of browser DOM differences.

Description	FF	IE	CH
uppercase, lowercase	`style="display:`	`style="DISPLAY:`	`style="display:`
semicolon, whitespace	`style="display: none;"`	`style="DISPLAY: none"`	`style="display: none;"`
colgroup	`<COL width="5%"/>`	`<COLGROUP><COL width="5%"/>`	`<COL width="5%"/>`
middle, center	`valign="middle"`	`valign="center"`	`valign="middle"`
attribute value order	`style="visibility:inherit; width:256px;"`	`style="WIDTH:256px; VISIBILITY:inherit;"`	`style="visibility:inherit; width:256px;"`
attribute addition	`<INPUT name="reason" value=""/>`	`<INPUT name="reason"/>`	`<INPUT name="reason"/>`
element addition	`<body>`	`<body><iframe id="yui_hist_iframe"`	`<body>`

Screen-level Comparison. Each node on the state graph represents an abstraction of the DOM tree instance of a particular screen of the web application, as viewed in the browser. As discussed in Section 2, not all screen-level differences should be flagged as candidate CBC issues. For instance, Table 1 shows some of the internal DOM level differences (e.g., elements, attributes, their values and order of appearance) in Firefox (FF), Internet Explorer (IE), and Chrome (CH), that typically do not cause any observable functional differences in the way a web application is presented to the end-user (the set $A - B$ in Figure 2). Based on our case studies, we have started to document such differences between browsers and develop suitable abstractions that allows our screen-matching algorithm to ignore them and present the user with only the true CBC issues.

Our implementation for detecting DOM-level mismatches is built as an extension of the `differencing` engine in XMLUnit[6]. To minimize the number of false positives (i.e., irrelevant differences), we have written a `DifferenceListener` to ignore case sensitivity, node-level white space, attribute order, node text values, attribute value order, and whitespace between the values. In addition, our tool takes as input a list of patterns that can be excluded, such as the ones shown in Table 1. A custom `ElementQualifier` is used to recursively compare nodes and their children to allow for node permutations in the graphs.

Additionally, as a realization of the OutScrDiff in Algorithm 4.1, we have implemented a visualization plugin for our tool, which while crawling, makes a snapshot of each screen change in the browser at hand. Later, when a mismatch is found between two screens, the corresponding snapshots are used to create a visualization report of the differences. More importantly, the report also contains the transition paths that lead to the mismatched screens, along with the two corresponding DOM trees having the differences highlighted (see Figure 5).

6. EMPIRICAL EVALUATION

To assess the efficacy and utility of our approach and the corresponding implemented tool, we have conducted a number of case studies following guidelines from [19]. Our evaluation addresses the following research questions:

RQ1 What is the effectiveness of our approach in revealing observable trace-level and screen-level differences in different browsers?

RQ2 How effective is our DOM-level comparison in minimizing the number of false positives?

RQ3 What is the tool's performance and automation level?

6.1 Subject Systems

The Organizer. Our first experimental case is an open source Ajax web application, called the Organizer, developed by Zammetti in his book 'Practical Ajax projects with Java technology' [34] and available for download.[7] It is a Java EE personal information manager using WebWork, HSQLDB, Spring JDBC, and the Prototype Ajax library.

[6] `http://xmlunit.sourceforge.net`

Table 2: Experimental data.

Exp. Subject	Browser Pair	Detected States	Detected Transitions	Crawled Paths	Avg. DOM Nodes	Crawling (min)	Comparison (sec)	Manual Effort (min)
Organizer	FF	13	53	51	112	4	7	2
	CH	14	60	54	112			
Ind-v1 *	FF	100	107	89	650	11	38	3
	CH	100	107	89	666			
Ind-v2 *	FF	200	200	178	230	28	45	3
	IE	200	199	178	245			
Tmms	IE	31	48	23	56	5	7	2
	CH	24	34	18	56			
Cnn *	FF	100	99	65	246	12	32	2
	IE	100	99	68	394			

IND V1 and V2. Our second experimental subject consists of two different versions (Ind-v1 and Ind-v2) of a large industrial enterprise *Web 2.0* application. Ind is a business process manager composed of approximately 214095 LOC in 248 client-side JavaScript files, 90742 LOC in 353 JSP files, and 58701 LOC in 43 Java files. YUI[8] is the Ajax library used in Ind. A complete user interface redesign forms the main difference between the two versions.

TM Management System. Tmms is a live, form-based, enterprise web application for submission, approval, distribution and archiving of technical memoranda. It is a very typical example of the many, relatively small but important, legacy web applications that are common-place in enterprise settings and usually very poorly compliant with modern web browsers. It is written in JavaScript and PHP.

Public Domain Sites. We have experimented with a number of public domain sites, taken from the list of Mozilla's bug reports on browser incompatibility issues.[9] Here we discuss the CNN 2008 US presidential elections page (referred to as Cnn).[10]

6.2 Experimental Setup

For each experimental subject, we configured our tool by providing the URL, elements to be included (e.g., `crawler.click("div")` and excluded (e.g., `crawler.dontClick("div").withAttribute("class", "logout")`) in the crawling session, as well as the browser

[7] `http://www.apress.com`

[8] `http://developer.yahoo.com/yui/`

[9] `http://tinyurl.com/ykjc3hl`

[10] Retrieved 10 Mar 2010: `http://edition.cnn.com/ELECTION/2008/results/president/`

Table 3: Case study results.

Exp. Subject	Trace-level Mismatches	Trace-level False Pos.	Screen-level Mismatches (% of pair of screens)	Screen-level False Pos. (%)	Avg. DOM-level Mismatches (per trace-level matched screen pair)	Method
Organizer	7	0	23	33	8	CrossT
	-	-	100	84	102	XDiff
Ind-v1	0	0	43	18	32	CrossT
	-	-	100	65	436	XDiff
Ind-v2	7	2	40	37	14	CrossT
	-	-	100	74	529	XDiff
Tmms	14	0	50	13	36	CrossT
	-	-	100	50	93	XDiff
Cnn	16	0	41	12	42	CrossT
	-	-	100	55	737	XDiff

to be used. The browser specifications used in our case studies are as follows: Firefox version 3.5, Chrome version 4.1, and Internet Explorer version 8.0. To constrain the state space, for this experiment, so that we can manually assess the correctness of the output, we set the crawl depth to 3 for all the subjects.

As shown in Table 2, for each subject, we measure the number of automatically detected states, transitions, and crawled paths. To give an indication of the size of the screens, we calculate the average number of DOM nodes per screen. For three of the subjects (marked with a *) we specifically set a maximum number of detected states (e.g., 100 or 200) to constrain the experimental data. To address RQ3, we report the time taken for the crawling and state exploration phase (in minutes), for the graph equivalence checking (in seconds), and manual effort required to configure and use CrossT (in minutes).

In the second phase of the experiment, the saved navigation models are checked for equivalence at a trace and screen level. We measure the number of trace-level and screen-level mismatches (pair of nodes) reported by CrossT, as well as the number of DOM-level differences per pair of screens, output as potential matches by the trace-level comparison.

To form a baseline for comparison, we manually inspect each subject in the given browsers, to document the *observable* trace-level and screen-level differences. In addition, we use the default settings of XMLUnit's `diff` method (further referenced as XDiff) as a baseline to evaluate the performance of our approach for detecting cross-browser DOM-level differences.

To measure the effectiveness, we analyze the observed false positives and where possible false negatives. A false positive is a reported mismatch that is not observable in the browsers and as such not relevant for cross-browser testing, i.e., the set $D - C$ in Figure 2. A false negative is an observable mismatch that is undetected (set $C - D$).

6.3 Results

Table 3 shows the results of our case studies. The *trace-level mismatches* column presents the sum of unmatched edges in both state graphs, i.e., edges in one state graph that have no counterpart in the other. The next column displays the number of false positives for the reported unmatched edges. The *screen-level mismatches* column shows the percentage of the screen pairs from the two models that were reported to be potentially matched at the trace-level but have difference at the screen-level, i.e., DOM-level. The next column reports the percentage of false positives. For such mismatched screen pairs, the table also shows the average number of DOM-level differences that are detected by our tool and XDiff. Note that the number is representing all the differences (e.g., at the node, attribute, children level) that need to be resolved, before the two DOM trees (and hence the screen pair) could be seen as a match.

Trace-level Mismatches. The trace-level comparison produced no false negatives and only 2 false positives (on Ind-v2). The false positives are due to the fact that the edge labels could not be matched deterministically: no persistent ID attributes were present and the combination of the attribute names/values and XPath expressions did not suffice. The 5 mismatches correctly reported for Ind-v2 are caused by a screen-level failure, which is discussed below under **Screen-level Mismatches**.

As an example of correctly detected trace-level mismatches, Figure 3 depicts the state graphs produced for the Organizer in Chrome and Firefox. As it can be seen, there are 7 state transitions (shown in bold with label 'IMGid:logoff') that lead to state 7 (shown in red) that are present in the Chrome version but are missing from the graph in Firefox. All these 7 transitions are caused by a `click` on the `logoff` tab in the application. The reason for this difference, is that the `logoff` JavaScript function associated with the `onclick` attribute of the corresponding DOM element is not executed in Firefox. Thus, after clicking on the element, there is no state change and hence no transition detected.

The Cnn case is interesting as well, since the missing traces are caused by a bug in the Firefox instance,[11] which results in many of the main content panes never finishing loading. The trace-level mismatches are mainly due to the transitions originating from content loaded into the pane, which are present in the IE graph but are missing in the Firefox one. The mismatches in the Tmms web application are due to the functionality being totally broken in Chrome, since many of the clickable elements do not cause any state changes. In fact, after filling in the form, clicking on the submit button merely clears all the filled input fields instead of submitting the form.

Screen-level Mismatches. Screen-level mismatches are typically more difficult to assess than trace-level differences. The reason is that a screen-level mismatch could be caused by a series of DOM-level differences, and since the number of DOM-level differences could be large, despite our abstraction mechanisms, more effort is required to check whether a reported mismatch is a relevant observable one.

The first observation is that our tool outperforms XDiff, which represents the current industrial practice, in constraining the number of DOM-level differences to be examined (reductions of up to 37 times in Ind-v2) and reducing the number of screen-level differences and false positives (reductions of up to 4.5 times in Cnn).

Figure 5 shows one instance of the detected DOM-level mismatches in Ind-v1, taken from our output visualization plugin. In this case, the `class` attribute which is on the anchor tag in Firefox, has been moved into a newly added tag element, namely a `LABEL` in Chrome. This detected DOM-level difference is observable in the two browsers as depicted in Figure 6, i.e., the menu icons are missing in Chrome. This type of behavior is the root cause of many DOM-level differ-

[11] `https://bugzilla.mozilla.org/show_bug.cgi?id=463639`

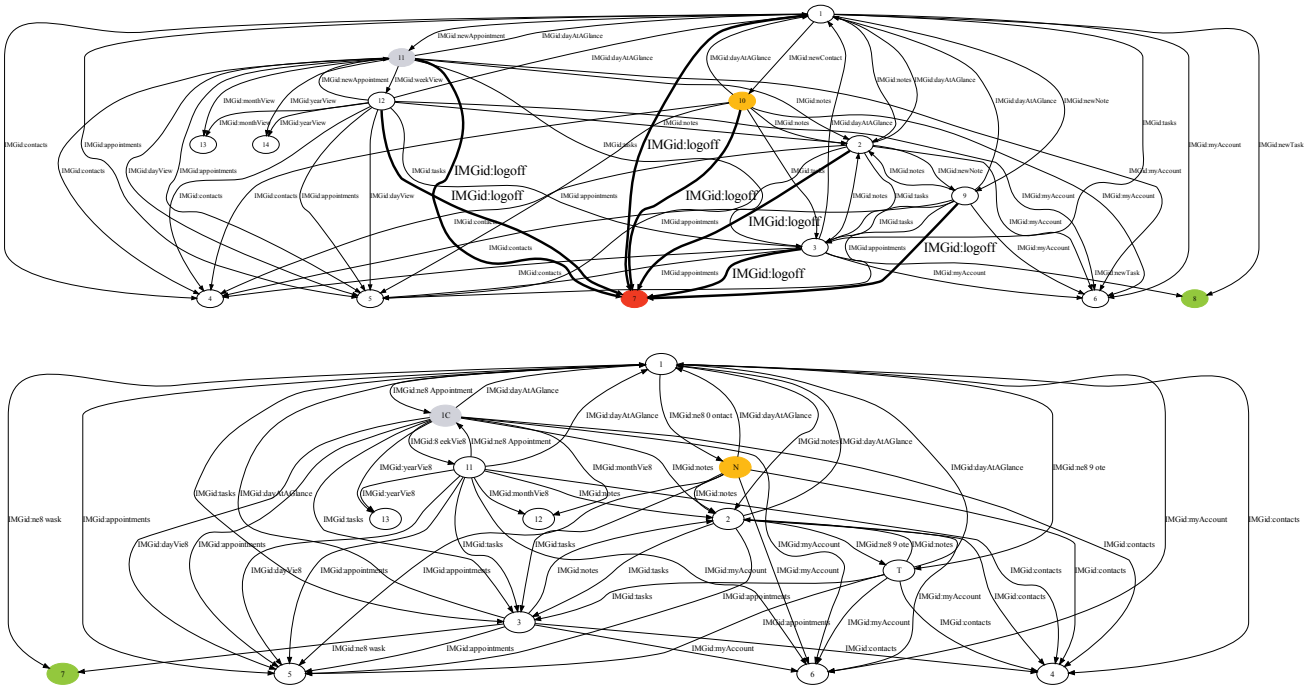


Figure 3: Generated Organizer graphs in Chrome (top) and Firefox (bottom) with the detected differences.

```
% The Organizer's DOM in Chrome %
<FORM class="cssmain" id="contactcreateshow" ...>
 <TABLE class="cssmain">
 <INPUT id="contactcreateshow_createddt" .../>
% The Organizer's DOM in Firefox %
<FORM class="cssmain" id="contactcreateshow" ...>
 <INPUT id="contactcreateshow_createddt" .../>
 <TABLE class="cssmain">
```

Figure 4: Example of the Organizer's DOM differences in Chrome (top) and Firefox (bottom).

ences we witness in IND-v1, propagated to all the instances of the screen-level mismatches.

As far as IND-v2 is concerned, an interesting instance of screen-level mismatch is manifested in the way some of the IFrames' (used as content panes) internal content simply disappears after a series of clicks in Firefox. This is caused by the way the style attributes are dynamically added and removed through JAVASCRIPT to the corresponding parent element (`DIV`) on the DOM tree.

A typical example of false positives, and the difficulty of resolving them, is shown in Figure 4, which presents DOM-level differences for the same `FORM` element in Chrome and Firefox. Here, the `INPUT` element resides outside the `TABLE` in Firefox, and is pulled in the `TABLE` in Chrome. Although in this case, there is no clear observable difference, it is very difficult to discard such differences automatically, without adversely affecting the false negative rate.

7. DISCUSSION

Scope. Cross-browser compatibility spans a wide range of issues from purely visual 'look-and-feel' differences to lack of functionality and availability in different browsers. Lack of functionality is believed to be a much more severe problem since it can easily result in loss of revenue. Ideally, each web application should behave exactly in the same manner regardless of the rendering browser. Our premise in this work is that cross-browser differences, that influence the functionality and usability of a web-application, do exist in practice. Further, many of these differences manifest themselves as differences in the client-side state of the application under different browsers. These differences can be captured in an automated way by using one browser's behavior as an oracle to test another browser's output (and vice versa) to detect mismatched transitions and screens. The results of our case studies validate these claims.

Note that the present implementation only uses the DOM portion of the client-state for analysis but could easily be extended to extract differences based on CSS properties or even JAVASCRIPT variable values. Our current approach does not target CBC issues caused by differences that result from different browsers' different rendering of *identical* web content, i.e. issues that never result in any differences in the programmatic client-side state or traces thereof. This would be an interesting area for future work.

Automation Level. Manual crawling or the record-and-replay approach embodied by tools such as Selenium[12] are other alternatives to the model capture phase of our approach. However, we believe that CRAWLJAX represents a more automated, scalable and robust solution to behavior exploration and integrates very well with the subsequent equivalence checking.

As far as the manual effort is concerned, the main task of the tester consists of (1) providing the URL of the deployed web application (2) setting the crawling specifications [22, 23] and (3) choosing the desired web browser types for compatibility testing, which usually takes a few minutes. The state exploration and comparison steps are fully automatic.

[12] `http://seleniumhq.org`

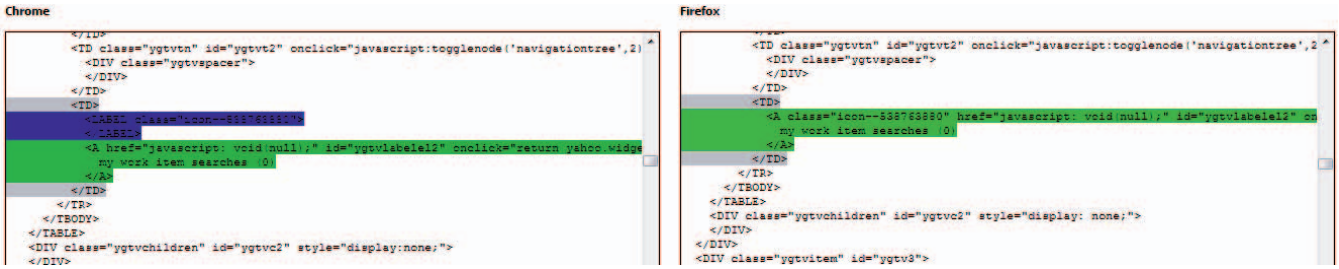


Figure 5: Detected Ind-v1 DOM-level differences in Chrome (left) and Firefox (right).

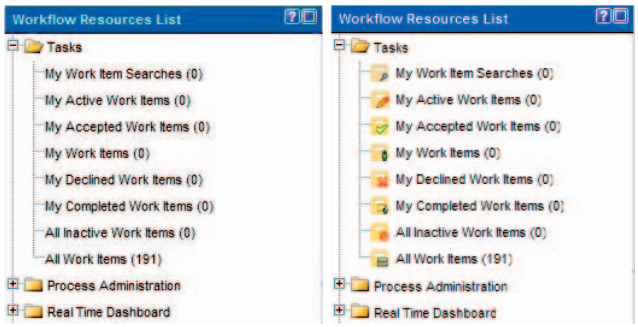


Figure 6: Observable differences of Ind-v1's Menu in Chrome (left) and Firefox (right).

Note that it is possible for the tester to tune the screen-level comparison metrics manually if desired. This optional step includes for instance specifying which DOM-level differences can be ignored to reduce false positives. The time needed for the actual crawling and generation is fully dependent on the required state space and the crawling settings. In our experiments, the longest model generation case was for IND-V2, which took around 28 minutes (for two browsers) to explore and generate 400 states. The graph comparison phase is quite fast, e.g., it took less than 45 seconds for IND-V2. It requires no manual effort if the default settings are used. In case custom DOM elements or attributes need to be ignored by the comparison method, they obviously have to be provided, which in our case studies was not needed. In short, the actual model generation and comparison requires very limited manual effort. Although we do provide some degree of visualization of the detected mismatches to make it easier, assessing the correctness of the output is of course still a manual task.

Completeness. While our proposed technique can expose several interesting and important cross-browser issues in a given web application it cannot expose all of them. This is limited on the one hand by the scope of the algorithm itself (discussed above) and on the other hand by the fraction of application behaviors covered during the crawling. Obviously, the number of cross-browser issues exposed co-relates to the number of behaviors covered by the crawling. For the purposes of this investigation we used simple resource bounds to limit the crawling. This served us well since typically the same cross-browser issues appear repeatedly on different states and under different application behaviors. Thus, a complete or comprehensive crawl of the application may not even be necessary to expose key cross-browser issues. In any case, optimizing, guiding or bounding the crawling to specifically target cross-browser issues are interesting avenues of future research, in their own right, but are beyond the scope of this paper.

Correctness. The most conclusive outcome from the evaluation is that within the automatically explored state space, our technique is capable of detecting trace-level mismatches with a high level of certainty. As per Theorem 1, Algorithm 4.2 guarantees a zero false positive and false negative error rate, at the trace-level, provided edge-labels in the state graphs have been matched (See our formal proof [3]). Our label-matching algorithm (Section 5) is quite successful in achieving this in practice (no false negatives and merely 2 false positives). Detected screen-level mismatches are also reasonably accurate, with the false positive rate ranging from 12% for CNN (lowest) to 37% for IND-V2 (highest).

As far as screen-level false negatives are concerned, for the ORGANIZER and TMMS we can claim that there were no false negatives. For the other subjects, because of the large state space we are not able to present any concrete numbers, i.e., it is difficult to claim that we have detected all the possible CBC issues, since that requires manually examining and comparing a couple of hundred states in different browsers.

Catalog of Browser DOM Differences. Ideally, we would have a comprehensive list of all DOM-level differences that are merely bound to different browser rendering and do not have any observable side-effects. The more complete this list, the more accurate our approach will be in terms of reducing the number of false positives. Unfortunately no off-the-shelf catalog exists today. This is a learning process, which requires examining the web applications at hand and documenting the noticeable differences. Our catalog is indeed growing with every new case study we conduct.

Non-deterministic and Dynamic Behavior. Comparing tree-structured data such as HTML, is known to be a difficult problem in web-based regression testing [28, 29], even within the same browser, because of the dynamic and non-deterministic behavior in modern web applications [27]. Some of the false positives in our results are due to precisely this reason. In this work, we have presented the extra challenges involved in dealing with the different ways browsers handle and render DOM changes, from the initial HTML page through subsequent DOM updates via JAVASCRIPT. Recent solutions proposed, for instance in [27], could help resolve some of the false positives we currently report.

Browser Sniffing. Browser sniffing is a technique used to determine the web browser of the visitor in order to serve content in a browser-specific manner. An interesting pattern we have witnessed in our case studies is when a JAVASCRIPT library treats one browser as the default and uses `if-else` statements for other *specific* browsers to account for CBC issues. The problem starts when a user is employing a browser not tested for and the default content is generated by the library. The CBC problem depicted in Figures 5 and 6 is

an example of this type of failure. Our technique is capable of correctly spotting all instances of such differences.

Threats to Validity. Some of the issues influencing the external validity of our results have been addressed in the above discussion. Although the number of web applications chosen in our evaluation is limited, we believe they are representative of modern web applications present in open-source and industrial environments. As far as the internal validity is concerned, since we use external libraries that act as wrappers around native browsers, our evaluation results could be affected by implementation differences. We have tried to mitigate this threat by extensively testing the browsers and their crawling behavior, for the model generation part.

8. CONCLUDING REMARKS

In this paper we have proposed an automated method for cross-browser compatibility testing of modern web applications. The results of our evaluation, on several open-source and industrial web applications, clearly point to the ubiquity of the problem. They also demonstrate the efficacy of our approach in automatically exposing relevant cross-browser issues at a trace-level as well as at a screen-level.

Our current approach can be enhanced in several ways. It can immediately benefit from a larger catalog of known DOM-level, cross-browser differences that are merely bound to different browser rendering and have no observable side effects. This would directly reduce the screen-level false positive rate and improve the precision of our approach. Another useful direction would be some automated method of detecting observable cross-browser differences, not reflected at DOM level. In concert with our tool, such a method could provide a very potent cross-browser compatibility tester for the end-user.

9. REFERENCES

[1] Acid 3: Wikipedia. Retrieved 4 Mar 2010. http://en.wikipedia.org/wiki/Acid3.

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

[3] BrowserCam. http://www.browsercam.com.

[4] BrowserCamp. http://www.browsrcamp.com.

[5] Adobe BrowserLab. https://browserlab.adobe.com.

[6] Browser Photo. http://www.netmechanic.com/products/browser-index.shtml.

[7] BrowserShots. http://browsershots.org.

[8] S. R. Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Proc. of the 26th IEEE Int. Conf. on Softw. Maintenance (ICSM'10)*, pages 1–10, 2010.

[9] CrossBrowserTesting. http://www.crosbrowsertesting.com.

[10] J. Dolson. What is "Cross-browser compatibility?". http://www.joedolson.com/articles/2008/03/what-is-cross-browser-compatibility/.

[11] C. Eaton and A. M. Memon. An empirical approach to testing web applications across diverse client platform configurations. *Int. Journal on Web Engineering and Technology (IJWET)*, 3(3):227–253, 2007.

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

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language and Computation.* Addision-Wesley, 1979.

[14] ieCapture. http://www.iecapture.com.

[15] IE NetRenderer. http://ipinfo.info/netrenderer/.

[16] IETester. http://www.my-debugbar.com/wiki/IETester/HomePage.

[17] iPhoney. http://sourceforge.net/projects/iphonesimulator/.

[18] E. Kiciman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proc. 21st Symp. on Operating Sys. Principles (SOSP'07)*, pages 17–30, 2007.

[19] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Softw.*, 12(4):52–62, 1995.

[20] Litmus. http://litmusapp.com.

[21] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proc. of The 10th Working Conf. on Reverse Engineering*, Nov. 2003.

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

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

[24] R. Ramler, E. Weippl, M. Winterer, W. Schwinger, and J. Altmann. A quality-driven approach to web testing. In *Proceedings of 2nd Conf. on Web Engineering (ICWE'02)*, 2002.

[25] F. Ricca and P. Tonella. Web testing: a roadmap for the empirical research. In *Proc. Int. Symp. on Web Site Evolution (WSE '05)*, pages 63–70, 2005.

[26] J. Rode1 and M. A. P.-Q. Mary Beth Rosson. The Challenges of Web Engineering and Requirements for Better Tool Support. Technical Report TR-05-01, Computer Science Department, Virginia Tech., 2002.

[27] D. Roest, A. Mesbah, and A. van Deursen. Regression testing Ajax applications: Coping with dynamism. In *Proc. 3rd Int. Conference on Softw. Testing, Verification and Validation (ICST'10)*, pages 128–136. IEEE Computer Society, 2010.

[28] E. Soechting, K. Dobolyi, and W. Weimer. Syntactic regression testing for tree-structured output. In *Proc. Int. Symp. on Web Systems Evolution (WSE'09)*. IEEE Computer Society, 2009.

[29] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proc. 20th Int. Conference on Aut. Softw. Eng. (ASE'05)*, pages 253–262. ACM, 2005.

[30] SputnikTests. http://code.google.com/p/sputniktests/.

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

[32] Xenocode Browser Sandbox. http://spoon.net/browsers/.

[33] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.

[34] F. W. Zammetti. *Practical Ajax projects with Java technology.* Apress, 2006.