X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications

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

Mukul R. Prasad
Fujitsu Laboratories of America, USA
mukul@us.fujitsu.com

Alessandro Orso
Georgia Institute of Technology, USA
orso@cc.gatech.edu

***Abstract*—Due to the increasing popularity of web applications, and the number of browsers and platforms on which such applications can be executed, cross-browser incompatibilities (XBIs) are becoming a serious concern for organizations that develop web-based software. Most of the techniques for XBI detection developed to date are either manual, and thus costly and error-prone, or partial and imprecise, and thus prone to generating both false positives and false negatives. To address these limitations of existing techniques, we developed X-PERT, a new automated, precise, and comprehensive approach for XBI detection. X-PERT combines several new and existing differencing techniques and is based on our findings from an extensive study of XBIs in real-world web applications. The key strength of our approach is that it handles each aspects of a web application using the differencing technique that is best suited to accurately detect XBIs related to that aspect. Our empirical evaluation shows that X-PERT is effective in detecting real-world XBIs, improves on the state of the art, and can provide useful support to developers for the diagnosis and (eventually) elimination of XBIs.**

I. Introduction

Cross-browser incompatibilities (XBIs) are discrepancies between a web application's appearance, behavior, or both, when the application is run on two different environments. An environment consists of a web browser together with the host operating system. Due to the increasing popularity of web applications, and the number of browsers and platforms on which such applications can be executed, XBIs are a serious concern for organizations that develop web-based software. For example, a search on the popular developer discussion forum stackoverflow.com, for posts tagged with "cross-browser" returned over 2500 posts over the past four years! Further, nearly 2000 of these have been active over the past year [1].

Because of the relevance of XBIs, a number of tools and techniques have been proposed to address them. In fact there are over 30 tools and services for cross-browser testing currently in the market [2]–[4]. Most of these tools are mainly manual and either provide tips and tricks for developers on how to avoid XBIs or render the same web application in multiple browsers at the same time and allow a human to check such renditions. Being human intensive, these techniques are less than ideal; they are costly and, especially, error-prone.

Researchers have therefore started to propose automated techniques for XBI detection (*e.g.,* [2], [5]–[8]). At a high level, these automated techniques work as follows. *First*, they render (and possibly crawl) the given web application in two different web browsers of interest and extract a possibly large set of features that characterize the application. This set may include behavioral features, such as finite state machine models that represent how the web application responds to various stimuli (*e.g.,* clicks, menu selections, text inputs). The set of features may also include visual characteristics of certain widgets or sets of widgets on a page, such as their size, their position, or properties of their visual rendition (*i.e.,* appearance). *Second*, the techniques compare the features collected across the two browsers and, if they differ, decide whether the difference is attributable to an XBI. Intuitively, these features are used as proxies for the human user's perception of the page and its behavior, so differences in features between two browsers are indications of possible XBIs. *Finally*, the techniques produce reports for the web-application developers, who can use the reports to understand the XBIs, identify their causes, and eliminate such causes.

The two most fundamental characteristics of XBI detection techniques are therefore (1) the choice of which features to collect and (2) the criteria used to decide whether a difference between two features is indeed the symptom of an XBI (*i.e.,* it can be perceived by a user as a difference in the web application's behavior or appearance). In existing techniques these choices are based primarily on intuition and experience and not on a systematic analysis of real-world XBIs.

Although such an approach is fine for an initial investigation, and in fact provided encouraging results in our initial evaluations (*e.g.,* [2], [5]), it must be improved for a more mature solution to the XBI detection problem. Case in point, the evaluation of our earlier approaches on a more extensive set of web applications generated a considerable number of false positives, false negatives, and duplicate reports for the same underlying errors, as we further discuss in Section VIII.

This paper presents X-PERT, a new comprehensive technique and tool for detection of XBIs that addresses the limitation of existing approaches. First, X-PERTś approach is derived from an extensive and systematic study of real-world XBIs in a large number of web applications from a variety of different domains. Besides showing that a large percentage of web applications indeed suffer from XBIs (over 20%), thus providing further evidence of the relevance of the problem, the study also allowed us to identify and categorize the most prominent feature differences that a human user would most likely perceive as actual XBIs.

Second, X-PERT is designed to be a comprehensive and accurate framework for detecting XBIs. It integrates differencing techniques proposed in previous work with a novel technique for detecting layout errors, by far the most common class of XBIs observed in our case study (over 50% of web-sites with XBIs contained layout XBIs). This allows X-PERT to detect the entire gamut of XBI errors and do so with very high precision.

Finally, by targeting the most appropriate differencing technique to the right class of XBIs, X-PERT usually reports only one XBI per actual error, unlike other techniques (*e.g.,* [5], [6]), which typically produce several duplicate error reports. For example, the movement of a single element on a page can have a domino effect on the positions of all elements below it. CROSSCHECK [6] might report all such elements as different XBIs, while our current approach would identify only the offending element. This improvement greatly simplifies the task of understanding XBIs for the developer.

This paper also presents a thorough empirical evaluation of X-PERT on a large set of real-world web applications. The results of our evaluation are promising: X-PERT was able to identify XBIs in the subjects with a fairly high precision (76%) and recall (95%) and with negligible duplication in the XBI reports. In fact, when compared with a state-of-the-art detector, X-PERT outperformed it in terms of precision, recall, and one-to-one correspondence between identified and actual XBIs.

The main contributions of this work are:

- A systematic study of a large number of real-world web applications that helps develop a deeper, realistic understanding of real-world XBIs and how to detect them.
- A comprehensive approach for XBI detection, called X-PERT, that integrates existing techniques with a novel approach to detecting layout XBIs in a single, unifying framework.
- An implementation of our X-PERT approach and a thorough empirical study whose results show that X-PERT is effective in detecting real-world XBIs, improves on the state of the art, and can support developers in understanding and (eventually) eliminating the causes of XBIs.
- A public release of our experimental infrastructure and artifacts (see http://www.cc.gatech.edu/~orso/software/x-pert/), which will allow other researchers and practitioners to benefit from them and build on this work.

II. MOTIVATING EXAMPLE

In this section, we introduce a simple web application that we use as a motivating example to illustrate different aspects of our approach. The application, referred to as **Conference** hereafter, is the web site for a generic conference. (In this paper, unless otherwise stated, we use the terms "web application" and "web site" interchangeably.)

Figure 1 provides an abstract view of this web site, as rendered in the Mozilla Firefox browser. The site consists of three interlinked dynamically generated pages that show the conference venue details, the key dates of the main conference activities, and the list of accepted papers. The buttons labeled **HOME**, **DATES**, and **PAPERS** can be used for navigating between different pages. Alternatively, the hyperlinks at the bottom of each page can also be used for navigation. The figure shows these inter-page transitions using two different kinds of edges, where dashed edges correspond to a button push, and solid edges correspond to a click on a link.

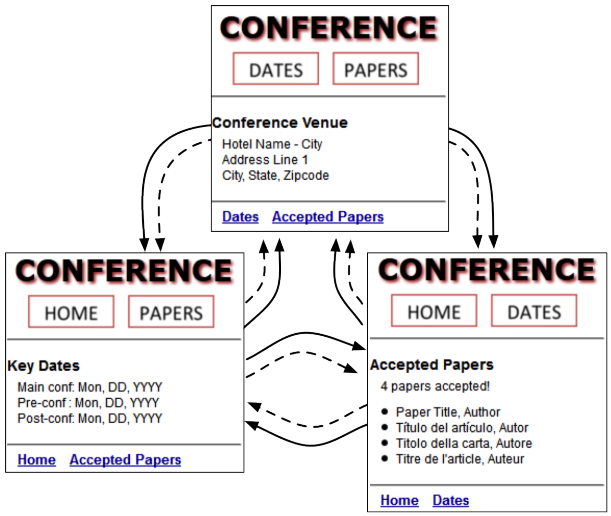
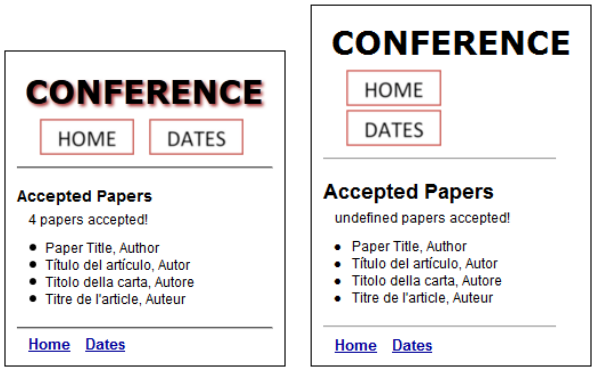


Fig. 1. State Graph for web application Conference in Mozilla Firefox.



(a) Mozilla Firefox (b) Internet Explorer

Fig. 2. One web page of Conference rendered in two browsers.

When rendered in Mozilla Firefox (FF) and Internet Explorer (IE), our example application manifests one behavioral and three visual XBIs (the latter shown in Figure 2). We discuss these XBIs and their causes individually.

XBI #1: Buttons HOME, DATES, and PAPERS do not produce any response when clicked in IE (*i.e.,* the dashed transitions in Figure 1 do not occur in IE), which prevents the users from accessing part of the application's functionality. The cause of this XBI is the following HTML code (shown for the DATES button only, as the other ones are analogous):

```
<img src="dates.png" onclick="navigate(event)" />
```

The application implements the buttons with `<img>` tags and associates the JavaScript event handler `navigate` to button-click events using the `onclick` attribute of such tags. Because IE does not support the `onclick` attribute for the `<img>` tag, the buttons are unresponsive in IE.

XBI #2: The buttons in the pages are arranged horizontally (left to right) in FF, but vertically in IE. The reason for this layout related XBI is that the application sets the total width of the button bar to 225 pixels. Due to differences in the default border and padding around button images in FF and IE, the second button does not fit in this width in IE and goes to the next line.

XBI #3: The number of accepted papers appears as `'undefined'` in IE. This issue is caused by the following JavaScript code:

```
var count = $("paperlist").childElementCount
                    + " papers accepted!";
$("papercount").innerHTML = count;
```

In the code, the list of papers is implemented as a list element (`<li>`) with id `''paperlist''`. The code uses property `childElementCount` to query the size of this list and adds it to the string that prints the number of papers. (We use `$("paperlist")` as a shorthand for the complete expression, which is `document.getElementById("paperlist")`.) Because the `childElementCount` property is not supported in IE, the query returns `'undefined'`, which results in the observed error.

XBI #4: The page title has a (red) shadow in FF and no shadow in IE. This last XBI is due to the following CSS property of the page title, which is an `<h1>` element:

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

Similar to the previous XBI, because the `text-shadow` property is not supported in IE, the shadow is absent in the IE rendering of the application pages.

III. Related Work

Previous work on cross-browser compatibility testing can be divided into the following four generations of techniques.

A. Generation 0: Developer Tool Support

A common resource used by web developers are browser-compatibility tables maintained by reference websites such as http://quirksmode.org and http://caniuse.com. Using these tables, a developer can manually lookup features used by their applications to know if the feature is supported by a certain set of browsers. However, these tables are generated manually and only have feature information for a limited set of browsers. Some web development tools such as Adobe Dreamweaver (http://www.adobe.com/products/dreamweaver.html) provide basic static analysis-based early detection of XBIs during development. However, these tools only check for certain types of XBIs related to specific DOM API calls or CSS features not supported by certain browsers. Modern XBIs often arise through a combination of browser, DOM, and JavaScript features, which is outside the scope of such tools.

B. Generation I: Tests on a Single Browser

Eaton and Memon [9] propose a technique to identify potentially problematic HTML tags on a page based on a given manual classification of good and faulty pages. Tamm [10] presents a tool to find layout related issues on a page, using visual and DOM information, that requires the user to manually alter the page (*i.e.,* add and remove elements) while taking screenshots. Both these techniques are human intensive and not well suited for modern web applications with a potentially large number of dynamically generated pages. Moreover, they test a web application within a single web browser, so they are not, strictly speaking, XBI-detection techniques.

C. Generation II: Multi-Platform Behavior & Test Emulation

This class of tools allows developers to emulate the behavior of their web application under different client platforms. Tools such as BrowserShots (http://browsershots.org) and Microsoft Expression Web SuperPreview (http://microsoft.com) provide previews of single pages, while tools such as CrossBrowserTesting.com and Adobe BrowserLab (https://browserlab.adobe.com/) let the user browse the complete web application in different emulated environments. While these tools do provide some emulation and visualization infrastructure, the core tasks of identifying the behaviors to explore and comparing them to extract XBI errors are still left to the user.

D. Generation III: Crawl-and-Compare Approaches

This class of techniques represents the most recent and most automated solutions for cross-browser testing. These techniques generally work in two steps. First, the *behavior capture* step automatically crawls and captures the behavior of the web application in two or more browsers; such captured behavior may include screen images and/or layout data of individual web pages, as well as models of user-actions and inter-page navigation. Then, the *behavior comparison* step automatically compares the captured behavior to identify XBIs.

WebDiff [5] uses a combination of DOM and visual comparison, based on computer-vision techniques, to detect XBIs on individual web pages. CrossT [2], conversely, uses automatic crawling and navigation comparison to focus on differences in the dynamic behavior caused by, for example, unresponsive widgets in a certain browser. CrossCheck [6] combines these two approaches and tries to achieve better precision through machine-learning based error detection. However, it still suffers from a high number of false positives and duplicate error reports. WebMate [7], a recent addition to this class of techniques, focuses mainly on improving the coverage and automation of the automated crawling, and its XBI detection features are still under development. QualityBots (http://code.google.com/p/qualitybots/) is an open source project that checks the appearance of a web application in different versions of the Google Chrome browser. The technique uses pixel-level image comparison and is not yet available for use with other families of web browsers. Browsera (http://www.browsera.com/), MogoTest (http://mogotest.com/), and Browserbite (http://app.browserbite.com/) are the very first industrial offerings in this category. They use a combination of limited automated crawling and layout comparison, albeit based on a set of hard-coded heuristics. In our (limited) experience with these tools, we found that these heuristics are not effective for arbitrary web sites. However, an objective evaluation of

the quality of these tools cannot be made at this time, since a description of their underlying technology is not available.

IV. Study of Real-World XBIs

As we discussed in the Introduction, the starting point of this work was a study of a large number of real-world XBIs. The goal was to provide a deeper understanding that could guide the re-targeting of existing XBI detection techniques and possibly the development of new ones.

In order to have an adequate sample of web applications for our study, we set the number of web sites to be studied to 100. Also, to avoid bias in the selection of the web sites, we selected them randomly using Yahoo!'s random URL service, available at http://random.yahoo.com/bin/ryl. For each web site selected, we followed the following process. First, we opened the web site using two different browsers: Mozilla Firefox and Internet Explorer. Second, one of the authors performed a manual examination of the site on the two browsers by studying both the visual rendering of the pages and their behavior when subjected to various stimuli. To limit the time requirements for the study, we selected a time limit of five minutes per site for the examination. This resulted in a total of over eight hours of manual examination, spread across several days. Finally, we analyzed the XBIs identified to categorize them based on their characteristics. We now discuss the finding of the study.

One striking result of our study is that the problem of XBI detection is quite relevant: among the 100 web sites examined, 23 manifested XBIs. This result is even more surprising if we consider that the examination involved only two browsers and a fairly limited observation time. More issues may appear if additional browsers and platforms, or a more extensive observation, were to be considered.

The study of the characteristics of the identified XBIs clearly showed three main types of XBIs: structure, content, and behavior. A finer grained analysis further allowed us to identify two subcategories for content XBIs: text and appearance. We describe these categories in detail below.

- *Structure XBIs:* These XBIs manifest themselves as errors in the structure, or layout, of individual web pages. For example, a structure XBI may consist of differences in the way some components of a page (*e.g.,* widgets) are arranged on that page. XBI #2 in the example of Section II is an instance of such an XBI.
- *Content XBIs:* These XBIs involve differences in the content of individual components of the web page. A typical example of this type of XBIs would be a textual element that either contains different text when rendered in two different browsers or is displayed with a different style in the two cases. We further classify these XBIs as *text-content* or *visual-content* XBIs. The former category involves differences in the text value of an element, whereas the latter category refers to differences in the visual aspects of a single element (*e.g.,* differences in the content of an image or in the style of some text). XBIs #3 and #4 (Section II) are instances of text-content and visual-content XBIs respectively.

TABLE I
Categorization of the real-world XBIs we found in our study.

Structure		13
Content	Text	5
	Visual	7
Behavior		2

- *Behavior XBIs:* These type of XBIs involve differences in the behavior of individual functional components of a page. An example of behavioral XBI would be a button that performs some action within one browser and a different action, or no action at all, in another browser. XBI #1 from Section II is a behavior XBI.

Table I shows, for each category of XBIs that we identified, the number of web sites in the study sample that exhibit that type of issue. Note that the sum of the values in the last column is higher than the total number of web sites with XBIs (23) because a single web site can contain multiple types of XBIs.

The above statistics, as well as a deeper analysis of each of the observed XBIs, provided the following key insights:

1) The three categories of XBIs are independent, that is, there is typically little or no correlation between the occurrence of XBIs in one category and another.
2) The three categories of XBIs are qualitatively quite distinct. Intuitively, while behavior of a widget refers to how it respond to a user action, structure denotes where and how it is arranged on the page, and content refers to its appearance.
3) Structure XBIs are by far the most common category, occurring in 57% (13/23) of the subjects that had XBIs. Further, we observed that we tended to recognize a structure XBI through a difference in the relative position of an element with respect to its immediate neighbors, rather than a difference of its absolute size or position. (We hypothesize that most users will do the same.)

The first two insights suggest that the three categories of XBIs could be independently detected, and techniques specialized to each category should be used. This insight also partly explains why use of image-comparison techniques for detecting structure and content XBIs had a high false positive rate in our previous work [5]. The third insight motivated us to develop a novel approach for detecting structure XBIs based on the concept of *relative-layout comparison*. This technique is presented in Section VI. It also explained why using the absolute size or position of elements to detect structure XBIs in our previous work [6] resulted in many false positives.

V. A Comprehensive Approach to XBI Detection

Our overall framework for XBI detection falls into the broad category of "crawl-and-compare" approaches described in Section III-D and draws heavily on the findings of our case study in Section IV. The behavior capture step is fairly similar to the one used in [6]. However, the behavior comparison step, unlike [6] or any other previous work, is organized as a set of four independent and orthogonal algorithms, each targeted to detect a specific category of XBIs: behavior, structure, visual-content, and text-content.

Algorithm 1: X-PERT: Overall algorithm

Input : url: URL of target web application
Br_1, Br_2: Two browsers
Output: $\mathcal{X}$: List of XBIs

1 **begin**
2 $\quad\mathcal{X} \leftarrow \emptyset$
3 $\quad(M_1, M_2) \leftarrow genCrawlModel(url, Br_1, Br_2)$
$\quad$// Compare State Graphs
4 $\quad(\mathcal{B}, PageMatchList) \leftarrow diffStateGraphs(M_1, M_2)$
5 $\quad addErrors(\mathcal{B}, \mathcal{X})$
6 $\quad$**foreach** $(S_i^1, S_i^2) \in PageMatchList$ **do**
$\quad\quad$// Compare matched web-page pair
7 $\quad\quad DomMatchList_i \leftarrow matchDOMs(S_i^1, S_i^2)$
8 $\quad\quad\mathcal{L}_i^R \leftarrow diffRelativeLayouts(S_i^1, S_i^2, DomMatchList_i)$
9 $\quad\quad\mathcal{C}_i^T \leftarrow diffTextContent(S_i^1, S_i^2, DomMatchList_i)$
10 $\quad\quad\mathcal{C}_i^V \leftarrow diffVisualContent(S_i^1, S_i^2, DomMatchList_i)$
11 $\quad\quad addErrors(\mathcal{L}_i^R, \mathcal{C}_i^V, \mathcal{C}_i^T, \mathcal{X})$;
12 $\quad$**end**
13 $\quad$**return** $\mathcal{X}$
14 **end**

Further, the algorithms for behavior, visual-content, and text-content XBI detection are adapted from [6] but orchestrated differently and more effectively in the current work. The algorithm for detecting structure XBIs, which usually constitute the bulk of XBIs (see Table I), is completely novel and a substantial improvement over previous work.

A. Terminology

Modern web applications are comprised of several static or dynamically generated web pages. Given a web page W and a web browser Br, $W(Br)$ is used to denote W as rendered in Br. Each web page is comprised of a number of web elements (*e.g.,* buttons, text elements) or containers of such elements (*e.g.,* tables). We use e to refer to an element of a web page. Further, each web page has a *DOM (Document Object Model)* representation, a layout, and a visual representation. We use D to refer to the DOM of a web page (or a portion thereof). The layout of a web page represents its visual structure. We model the layout as a set of potentially overlapping rectangles in a two dimensional plane and denote it as $\mathcal{L}$. Each rectangle represents an element of the page and is characterized by the coordinates (x_1, y_1) of its top-left corner and (x_2, y_2) of its bottom right corner. Thus, $\mathcal{L}(e) = ((x_1, y_1), (x_2, y_2))$ denotes the layout of element e. The visual representation of a web page is simply its two-dimensional image, as rendered within the web browser. Accordingly, the visual representation of an element is the image of the rectangle comprising the element.

B. Framework for XBI Detection

Algorithm 1 presents our overall approach for XBI detection. Its input is the URL of the opening page of the target web application, url, and the two browsers to be considered, Br_1 and Br_2. Its output is a list $\mathcal{X}$ of XBIs. The salient steps of this approach are explained in the following sections.

Crawling and Model capture: The first step is to crawl the web application, in an identical fashion, in each of the two browsers Br_1 and Br_2, and record the observed behavior as navigation models M_1 and M_2, respectively. The navigation model is comprised of a state graph representing the top-level structure of the navigation performed during the crawling, as well as the image, DOM, and layout information of each observed page. This is implemented by function $genCrawlModel()$ at line 3 and is similar to the model capture step in CROSSCHECK [6].

Behavior XBI Detection: The next step is to check the state graphs of navigation models M_1 and M_2 for equivalence. This is done using the algorithm for checking isomorphism of labelled transition graphs proposed in [2]. Function $diffStateGraphs()$ (line 4) performs this operation. This comparison produces a set of differences, $\mathcal{B}$, and a list $PageMatchList$ of corresponding web-page pairs S_i^1, S_i^2 between M_1 and M_2. The differences in $\mathcal{B}$ are attributable to missing and/or mismatched inter-page transitions. Since these transitions characterize the dynamic behavior of the web application, $\mathcal{B}$ represents the behavior XBIs as detected by Algorithm 1. The algorithm then iterates over the list of matched web-page pairs in $PageMatchList$ and compares them in various ways to detect other kinds of XBIs (lines $6-13$).

DOM Matching: To compare two matched pages S_i^1 and S_i^2, the algorithm computes a list $DomMatchList_i$ of corresponding DOM element pairs in S_i^1 and S_i^2. This is implemented by function $matchDOMs()$ (line 7) and done based on a *match index* metric for DOM element correspondence. This metric was first proposed in [5] and further developed in [6]. The match index uses a weighted combination of (1) the XPath (*i.e.,* path in the DOM—see http://www.w3.org/TR/xpath/), (2) DOM attributes, and (3) a hash of an element's descendants to compute a number between 0 and 1 that quantifies the similarity between two DOM elements. (See [6] for further details.) The computed $DomMatchList_i$ is used by several of the subsequent steps.

Structure XBI Detection: We introduce the notion of relative-layout comparison as the mechanism for detecting structure XBIs, which is one of the key contributions of this paper. Function $diffRelativeLayouts()$ (line 8 of Algorithm 1) compares pages S_i^1 and S_i^2 and extracts the set of relative-layout differences $\mathcal{L}_i^R$ that represent structure XBIs (also called relative-layout XBIs). The technique for detecting relative-layout XBIs is described in Section VI.

Text-content XBI Detection: These XBIs capture textual differences in page elements that contain text. To detect them, the text-value of an element is extracted from its DOM representation and compared with that of its corresponding element from $DomMatchList_i$. This operation is performed by $diffTextContent()$ (line 9) and is similar to the method for extracting the *LDTD* feature for machine learning in [6].

Visual-content XBI Detection: Visual-content XBIs represent differences in the visual appearance of *individual* page elements, such as differences in the styling of text or background of an element. To detect such errors, our approach takes the screen images of two corresponding elements and compares their color histograms using the χ^2 distance, similar to what we did in CROSSCHECK [6]. Unlike CROSSCHECK however, which compared *all* DOM elements and generated many false positives, our new approach applies visual comparison only to leaf DOM elements, where it is most effective at detecting

visual-content XBIs. Function $diffVisualContent()$ (line 10) implements this operation. The XBIs extracted in this step and in the previous one are then added to the XBI list $\mathcal{X}$ (line 11).

VI. Detecting Relative-Layout XBIs

Given a web page W and two different browsers Br_1 and Br_2, relative-layout XBIs represent discrepancies between the relative arrangements of elements on the layouts of $W(Br_1)$ and $W(Br_2)$. To accurately detect these issues, we propose a formalism for modeling the relevant aspects of a page layout, called an *Alignment Graph*. To detect relative-layout XBIs, our approach performs the following two steps: (1) extract alignment graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ from the layouts of $W(Br_1)$ and $W(Br_2)$, respectively; and (2) compare $\mathcal{A}_1$ and $\mathcal{A}_2$ for equivalence and extract differences as relative-layout XBIs.

In the following sections, we formally define the alignment graph and the algorithms for extraction and equivalence checking of alignment graphs.

A. The Alignment Graph

The alignment graph is used to represent two kinds of relationships between the elements (rectangles) of the layout of a web page, namely, *parent-child* relationships and *sibling*. We introduce the relevant definitions for these two relationships.

Definition 1 (Contains Relation): Given a set of elements D_s from a web page, a *contains relation*, $\prec: D_s \rightarrow D_s$, is defined between elements of D_s as follows. Given two elements $e_1, e_2 \in D_s$ with layout views $\mathcal{L}(e_1) = ((x_1^1, y_1^1), (x_2^1, y_2^1))$ and $\mathcal{L}(e_2) = ((x_1^2, y_1^2), (x_2^2, y_2^2))$ and XPaths $\mathcal{X}_1$ and $\mathcal{X}_2$, $e_1 \prec e_2$ if and only if

- $x_1^1 \leq x_1^2 \wedge y_1^1 \leq y_1^2 \wedge x_2^1 \geq x_2^2 \wedge y_2^1 \geq y_2^2$ and
- if $\mathcal{L}(e_1) = \mathcal{L}(e_2)$ then $\mathcal{X}_1$ is a prefix of $\mathcal{X}_2$

Thus, a contains relation exists between e_1 and e_2 if either (1) rectangle $\mathcal{L}(e_2)$ is strictly contained within rectangle $\mathcal{L}(e_1)$ of e_1 or (2) e_1 is an ancestor of e_2 in the DOM, in the case where $\mathcal{L}(e_1)$ and $\mathcal{L}(e_2)$ are identical.

Definition 2 (Parent node): Given a set of elements D_s from a web page, and two elements $e_1, e_2 \in D_s$, e_1 is a parent of e_2 if and only if $e_1 \prec e_2$, and there does not exist an element $e_3 \in D_s$ such that $e_1 \prec e_3 \wedge e_3 \prec e_2$.

Thus, the parent of an element e is basically the "smallest" element containing e. Note that Definition 2 allows for elements to have multiple parents. However, to simplify the implementation, we use a simple metric (*i.e.,* the area) to associate each element with at most one parent.

Definition 3 (Sibling nodes): Given a set of elements D_s from a web page. two elements $e_1, e_2 \in D_s$ are said to be siblings if and only if they have a common parent in D_s.

Parent-child and sibling relationships can be further qualified with attributes specifying the relative position of the elements with respect to each other. For example, a child could be horizontally left-, right-, or center-justified and vertically top-, bottom-, or center-justified within its parent. Similarly, an element e_1 could be above, below, or to the left or right of its sibling element e_2. Further, e_1 and e_2 could be aligned with respect to their top, bottom, left, or right edges. These attributes can be simply computed by comparing the x and y coordinates of the elements in question.

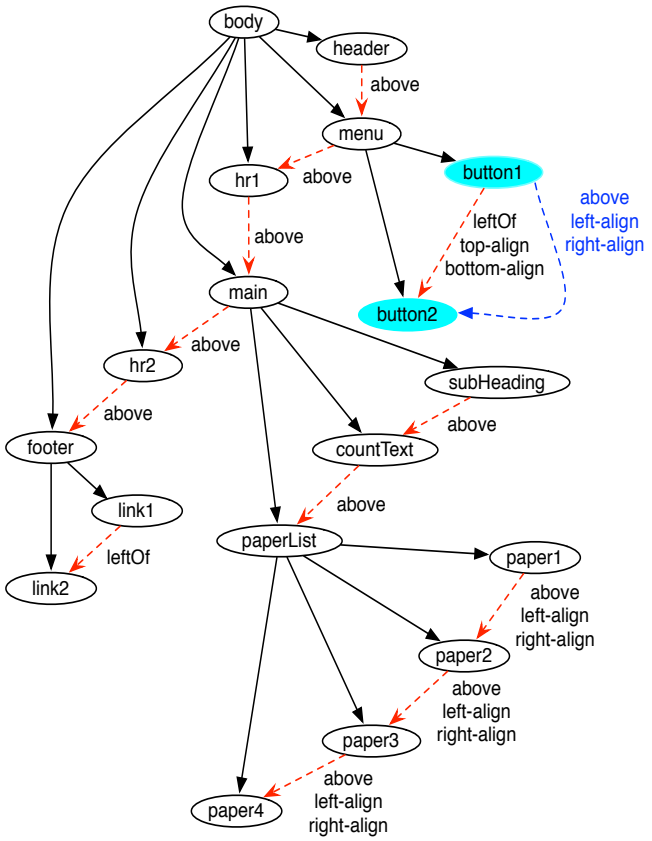


Fig. 3. Alignment Graph for the web pages in Figure 2.

Formally, an alignment graph $\mathcal{A}$ is a directed graph defined by the 5-tuple $(E, \mathcal{R}, \mathcal{T}, Q, \mathcal{F})$. Here, E is the set of vertices, one for each web page element. $\mathcal{R} \subseteq E \times E$ is a set of directed relationship edges, such that for elements $e_1, e_2 \in E$, there exists an edge (e_1, e_2) in $\mathcal{A}$ if and only if either e_1 is a parent of e_2, or e_1 and e_2 are siblings. (Although the sibling relation is symmetric, in practice only one of the edges (e_1, e_2) and (e_2, e_1) is sufficient to represent it, so we can arbitrarily choose one.) $\mathcal{T}$ is a set of the two types $\{parent, sibling\}$ for identifying an edge as a parent or a sibling edge. Q is a set of attributes (*e.g., left-align, center-align, above, leftOf*) used to positionally qualify the parent or sibling relationship. $\mathcal{F} : \mathcal{R} \mapsto \mathcal{T} \times 2^Q$ is a function that maps edges to their type and set of attributes.

Figure 3 shows the alignment graph for the web pages shown in Figure 2, where some sibling edges and edge attributes have been omitted to avoid cluttering. In the figure, parent edges are represented with black, solid lines, and sibling edges with red, dashed lines. Node labels indicate the element they represent. Nodes *button1* and *button2*, for instance, represent menu buttons HOME and DATES, respectively, and *header*, *footer*, and *main* represent the page header, footer, and the main content-bearing section (showing the accepted papers), respectively. The graph is identical for the web pages in Figures 2a (FF) and 2b (IE), except for the sibling edge

Algorithm 2: ExtractAlignmentGraph

Input : W: Web page to analyze, Br: Web browser
Output: $\mathcal{A}$: Alignment Graph

```
1  begin
2      D ← extractDOM(W, Br)
3      L ← extractLayout(W, Br)
4      D_f ← filterDOM(D)
5      foreach e ∈ D_f do addNode(e, A)
6
7      addParentEdges(A, L, D_f)
8      addSiblingEdges(A)
9      foreach (v1, v2) ∈ parentEdges(A) do
10         addParentChildAttributes(A, L)
11     end
12     foreach (v1, v2) ∈ siblingEdges(A) do
13         addSiblingAttributes(A, L)
14     end
15     return A
16 end
```

Algorithm 3: addParentEdges

Input : A: Alignment Graph being built, $\mathcal{L}$: Layout of web page, D_f: Filtered DOM of web page

```
1  begin
2      E ← getListOfElements(D_f)
3      sort(E, g) // Sort E using compare function g
4      while size(E) > 1 do
5          e ← removeLastElement(E)
6          for index ← size(E) to 1 do
7              p ← getElement(E, index)
8              if contains(p, e, L) then
9                  insertParentEdge(A, p, e, L)
10                 break
11             end
12         end
13     end
14 end
```

between the nodes *button1* and *button2*, which is represented as a dotted blue line for IE and as a red line for FF.

B. Extracting the Alignment Graph

Algorithm 2 describes our approach for extracting the Alignment Graph $\mathcal{A}$ of a target web page W with respect to a web browser Br. The algorithm first extracts the DOM D of $W(Br)$ ($extractDOM()$, line 2) and the layout $\mathcal{L}$ of $W(Br)$ ($extractLayout()$, line 3). Function $filterDOM()$ then reduces D to D_f by pruning away DOM elements that have no bearing on the visible layout of the page (*e.g.,* <a>). Line 5 adds one vertex to $\mathcal{A}$ for each element in D_f. Layout $\mathcal{L}$ is then analyzed to deduce parent-child relationships between elements in D_f and insert parent edges between the corresponding vertices in $\mathcal{A}$. This is implemented by function $addParentEdges()$ (line 6) and, similarly, for sibling edges by function $addSiblingEdges()$ (line 7). The layout of each parent-child element pair is further analyzed to infer alignment attributes qualifying this relationship, which are then added to the relevant edge in $\mathcal{A}$ (lines $8-10$). This is similarly done for sibling edges through function $addSiblingAttributes()$ (lines $11-13$).

Algorithm 3 computes the parent-child relationships among the nodes in D_f and inserts edges representing them into $\mathcal{A}$. First, the algorithm inserts the elements of D_f into a list E (function $getListOfElements()$, line 2). Then, list E is sorted using a compare function g (line 3) that satisfies the following property:

Property 1: For a pair of elements $e_1, e_2 \in D_f$, if $e_1 \prec e_2$ then $g(e_1, e_2) = -1$.

Finally, the algorithm iteratively removes the last element e from the sorted list E (line 5). It then scans E from left to right, while comparing e with each element, until it finds an element p such that $p \prec e$ (function $contains()$, line 8). From Property 1 of the sorted list E, p can be inferred to be the parent of e, so the algorithm adds a parent edge (p, e) to $\mathcal{A}$ (function $insertParentEdge()$, line 9).

It is fairly straightforward to prove that, given a compare function g satisfying Property 1, Algorithm 3 finds precisely one parent element, consistent with Definition 2, for each element in set D_f that has a parent, and adds a parent edge to $\mathcal{A}$ accordingly. Note that there are many possible compare functions g that satisfy Property 1. In our current implementation, we use a function that orders elements based on their geometric area and XPath.

C. Comparing Alignment Graphs

After extracting alignment graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ for $W(Br_1)$ and $W(Br_2)$, respectively, our technique checks the two graphs for equivalence; any difference found constitutes a relative-layout XBI. To do so, the technique uses the DOM matching approach we discussed in Section V-B, which can determine corresponding elements in $W(Br_1)$ and $W(Br_2)$, and consequently, corresponding vertices in $\mathcal{A}_1$ and $\mathcal{A}_2$. Given a node $e \in \mathcal{A}_1$ (resp., $\mathcal{A}_2$), let $m(e)$ denote its corresponding node in $\mathcal{A}_2$ (resp., $\mathcal{A}_1$) as computed by our matching approach. Next, our technique iterates over each edge $r = (e_1, e_2)$ in $\mathcal{A}_1$ and checks that the corresponding edge $r' = (m(e_1), m(e_2))$ exists in $\mathcal{A}_2$. It further checks that these edges have identical labels, that is, $\mathcal{F}_1(r) \equiv \mathcal{F}_2(r')$. This check ensures that r and r' are of the same type and have an identical set of attributes. Any discrepancies in the edge correspondence is recorded as an error. The process is repeated in a similar way for $\mathcal{A}_2$. Each XBI is detected and reported in the form of differences in the "neighborhood" of a given element and its counterpart in the two alignment graphs. The neighborhood refers to the parent and siblings of the element, and to the edges between them. For example, in Figure 3, the comparison would yield a single XBI on *button1* caused by attribute differences between the $button1 \rightarrow button2$ sibling edges in FF and IE. In FF, the edge indicates that (1) *button1* is to the left of *button2* and (2) the top and bottom edges of *button1* and *button2* are aligned. For IE, conversely, the dashed blue sibling edge indicates that *button1* is above *button2*, and the left and right edges of the two buttons are aligned. This is indeed the only layout XBI between the web pages in Figures 2a and 2b.

VII. Implementation

We implemented our approach in a prototype tool called X-PERT (*Cross-Platform Error ReporTer*), which is implemented in Java and consists of three modules: *Model collector*, *Model comparator*, and *Report generator* (Figure 4). Module *Model*

collector, accepts a web application and extracts its navigation model from multiple browsers using an existing web crawler, CRAWLJAX [11]. CRAWLJAX acts as a driver and, by triggering actions on web page elements, is able to explore a finite state space of the web application and save the model as a state-graph representation. For each state (*i.e.,* page), *Model collector* also extracts an image of the entire page and geometrical information about the page elements by querying the DOM API.

Module *Model comparator* (MC) performs the checks needed to identify the different classes of XBIs defined in Section IV. First, the *Behavior checker* detects behavior XBIs by checking the state-graphs for the two browsers. Then, it passes the equivalent states from the two graphs to the *DOM matcher*, which matches corresponding DOM elements in these states. These matched elements are then checked for structural and content XBIs by the *Layout checker* and *Content checker*. The *Layout checker* implements the new relative-layout detection algorithm described in Section VI. Each element on the page is represented as a layout node, and its edge relationships are inferred using the geometric information captured earlier. To find differences in the neighborhood of matched nodes in two alignment graphs, X-PERT checks the nodes' incoming and outgoing edges, along with the corresponding edge attributes. Any discrepancy observed is attributed to the elements being compared. The *Content checker* compares both the textual and visual content of leaf DOM elements on the page. X-PERT performs textual comparison using string operations defined in the Apache Commons Lang library (http://commons.apache.org/proper/commons-lang/). To compare visual content, X-PERT uses the implementation of the χ^2 metric in the OpenCV computer vision toolkit [12].

Finally, the *Report generator* module generates an XBI report in HTML format, meant for the web developer, using the Apache Velocity library (http://velocity.apache.org/). These reports first present the behavioral XBIs, overlaid on a graph, to depict missing transitions or states, if any. A list of XBIs is presented along with the pages where they appear. The user can select a particular page to see an instance of the XBI. These instances are identified using the XPaths and screen coordinates of the elements involved and also highlighted on two side-by-side screenshots of the affected page.

VIII. EVALUATION

To assess the effectiveness of our technique for detecting XBIs, we used X-PERT to conduct a thorough empirical evaluation on a suite of live web applications. In our evaluation, we investigated the following research questions:

- RQ1: Can X-PERT find XBIs in real web applications? If so, was the new relative-layout XBI detection algorithm effective in detecting the targeted issues?
- RQ2: How does X-PERT's ability to identify XBIs compare to that of a state-of-the-art technique?

In the rest of this section, we present the subject programs we used for our evaluation, our experimental protocol, our results, and a discussion of these results.

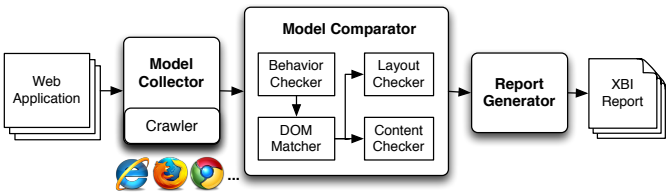


Fig. 4. High-level architecture of X-PERT.

A. Subject Programs

Table II shows the fourteen subjects we used in our evaluation. Along with the name, URL, and type of each subject, the table reports the following information: number of states explored by X-PERT, number of transitions between these states, and minimum, maximum, and average number of DOM nodes analyzed per web page. (This latter information provides an indication of the complexity of the individual pages.)

The first six subjects (*i.e.,* Organizer, GrantaBooks, DesignTrust, DivineLife, SaiBaba, and Breakaway) had been previously used in the evaluation of CROSSCHECK. In addition, Organizer was also used for evaluating CROSST [2]. Conference is our motivating example from Section II and was developed to show different classes of XBIs in a web application. The following three subjects—Fisherman, Valleyforge, and UniMelb—were obtained from the study of real world XBIs presented in Section IV. The main criteria for picking these subjects was the presence of known XBIs found in the study. All of the subjects mentioned so far had known XBIs, some of which were detected by previous techniques. To further generalize our evaluation, we selected four additional subjects using an online random URL service—http://www.uroulette.com/. (We used this alternative service because the Yahoo! service we used in the study was discontinued.) These additional subjects are Konqueror, a web-based file manager, UBC, a student organization site, BMVBS, a mobile web application for the German ministry, and StarWars, a fan site.

B. Protocol

For our experiments, we set up X-PERT on a 64-bit Windows 7 machine with 4GB memory. X-PERT was configured to run two web browsers: the latest stable versions of Internet Explorer (v9.0.9) and Mozilla Firefox (v14.0.1). Our choice of these two browsers was due to their use in previous studies. In fact, the chosen browsers do not have any bearing on the technique and can be replaced with any browser of choice. Two subjects, Organizer and Conference, were hosted on a local Apache web server, whereas the remaining subjects were used live from their actual web sites. Note that we do not report any data on the performance of the tool because the whole analysis, including crawling, terminated in less than an hour.

To investigate RQ2, as a tool representative of the state of the art we selected CROSSCHECK [6]. Unlike X-PERT, CROSSCHECK does not combine XBIs across different web pages, thereby having the same XBIs possibly reported multiple times. Therefore, to perform a fair comparison, we implemented such a grouping on top of CROSSCHECK. We

TABLE II
DETAILS OF THE SUBJECTS USED IN OUR EMPIRICAL EVALUATION.

Name	*URL*	*Type*	*States*	*Transitions*	DOM Nodes (per page)		
					max	*min*	*average*
Organizer	http://localhost/organizer	Productivity	13	99	10001	27482	13051
GrantaBooks	http://grantabooks.com	Publisher	9	8	15625	37800	25852
DesignTrust	http://designtrust.org	Business	10	20	7772	26437	18694
DivineLife	http://sivanandaonline.org	Spiritual	10	9	9082	140611	49886
SaiBaba	http://shrisaibabasansthan.org	Religious	13	20	524	42606	12162
Breakaway	http://breakaway-adventures.com	Sport	19	18	8191	45148	13059
Conference	http://localhost/conference	Information	3	12	878	817	853
Fisherman	http://fishermanslodge.co.uk	Restaurant	15	17	39146	15720	21336
Valleyforge	http://valleyforgeinn.net	Lodge	4	12	5416	4733	5046
UniMelb	http://www.economics.unimelb.edu.au/ACT/	University	9	8	15142	12131	13792
Konqueror	http://www.konqueror.org	Software	5	4	17586	15468	16187
UBC	http://www.ubcsororities.com	Club	7	7	20610	7834	12094
BMVBS	http://m.bmvbs.de	Ministry	5	20	19490	12544	15695
StarWars	http://www.starwarsholidayspecial.com	Movie	10	9	28452	19719	22626

call this improved version CROSSCHECK+ in the rest of the paper. The reports generated by these tools were manually inspected to find true and false positives. In addition, we manually analyzed the web pages analyzed by the tools to count all issues potentially detectable by a human user, which we use as an upper bound for the number of issues that a tool can detect. We use this number to calculate the recall of the results produced by the two tools.

C. Results

To answer RQ1, Table III(a) presents a detailed view of X-PERT's results when run on the 14 subjects considered. The table shows, for each subject, the true and false positives reported by X-PERT for each of the four types of XBI we identified, along with an aggregate total. As the results show, X-PERT reported 98 true XBIs and 31 false positives (76% precision). The detected issues included all four types of XBIs, with a prevalence of structure XBIs (60), followed by behavior (33) and content (5) XBIs. Based on these results, we can answer RQ1 and conclude that, for the subjects considered, X-PERT was indeed effective in finding XBIs. We can also observe that the new relative-layout XBI detection algorithm was able catch most of the issues in our subjects.

Table III(b) summarizes and compares the results of X-PERT and CROSSCHECK+, which allows us to answer RQ2. The table shows, for each subject, its name, the number of XBIs found by manual analysis (XBI), and the results of the two tools in terms of true positives (TP), false positives (FP), recall (Recall), and duplicate reports (Dup) produced. As the table shows, X-PERT outperformed CROSSCHECK+ in terms of both precision and recall for all of the subjects considered, and often by a considerable margin. For subject DesignTrust, for instance, X-PERT produced 3 false positives, as compared to 122 false positives produced by CROSSCHECK+. On average, the precision and recall of X-PERT's results were 76% and 95%, respectively, against 18% and 83% for CROSSCHECK+. Our results also show that the X-PERT reported a negligible number of duplicate XBIs—only 1 versus the 52 duplicate XBIs CROSSCHECK+ reported. We can therefore answer RQ2 and conclude that, for the cases considered, X-PERT does improve over the state of the art.

IX. DISCUSSION

As our empirical results show, X-PERT provided better results than a state-of-the-art tool. We attribute this improvement, in large part, to our novel relative-layout detection technique. From our study of real world XBIs, presented in Section IV, it was clear that layout XBIs are the most common class of XBIs. In previous approaches, such as WEBDIFF or CROSSCHECK, these XBIs were detected indirectly, by measuring side-effects of layout perturbations, such as changes in the visual appearance or in the absolute size or position of elements. However, as demonstrated by our results, detecting side effects is unreliable and may result in a significant reduction in precision. In addition, a single XBI can have multiple side effects, which when detected by previous techniques would result in duplicate error reports.

One solution for eliminating duplication, used in previous techniques, is to cluster related XBIs. However, clustering can be imperfect, thereby including unrelated issues in one cluster or separating related issues across multiple clusters. Moreover, developers still need to manually sift through the errors in a cluster to find the underlying cause of the XBI and related side effects. To alleviate this problem, X-PERT focuses each differencing technique (*i.e.,* visual comparison, text comparison, and layout differencing) where it can be most effective at detecting XBIs. By focusing the techniques on very specific problems, each XBI can be detected in terms of its principal cause, rather its side effects, which can be used to provide a better explanation of the XBI to the developers. In addition, we observed that such focused orchestration can detect more errors, which explains the improvement in the recall of the overall approach.

Another potential advantage of X-PERT is that it separates the individual techniques into different components, unlike previous approaches. Although we did not demonstrate this aspect in our study, intuitively this separation could allow developers to tune each of these components based on the kind of web application under test. For instance, developers could selectively use the behavioral detector, if such issues are more common in their web applications, or could turn it off to focus on other kinds of XBIs.

TABLE III
EMPIRICAL EVALUATION RESULTS.

NAME	BEHAV.		STRUCT.		CONTENT				TOTAL	
					TEXT		IMAGE			
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
Organizer	1	0	9	0	0	0	0	0	10	0
GrantaBooks	16	0	11	0	0	0	0	0	27	0
DesignTrust	2	0	5	3	0	0	0	0	7	3
DivineLife	7	0	3	6	1	0	0	0	11	6
SaiBaba	2	0	2	9	0	0	0	0	4	9
Breakaway	0	0	10	2	0	0	0	0	10	2
Conference	2	0	3	0	1	0	1	0	7	0
Fisherman	1	0	3	1	0	1	1	0	5	2
Valleyforge	0	0	2	2	0	0	1	0	3	2
UniMelb	2	0	0	0	0	0	0	1	2	1
Konqueror	0	0	0	0	0	0	0	6	0	6
UBC	0	0	0	0	0	0	0	0	0	0
BMVBS	0	0	0	0	0	0	0	0	0	0
StarWars	0	0	12	0	0	0	0	0	12	0
TOTAL	33	0	60	23	2	1	3	7	98	31

(a) X-PERT's detailed results.

NAME	XBI	X-PERT				CROSSCHECK+			
		TP	FP	Recall	Dup	TP	FP	Recall	Dup
Organizer	10	10	0	100%	0	8	2	80%	13
GrantaBooks	27	27	0	100%	0	27	1	100%	0
DesignTrust	7	7	3	100%	0	6	122	86%	3
DivineLife	11	11	6	100%	0	10	24	91%	3
SaiBaba	5	4	9	80%	0	4	53	80%	10
Breakaway	13	10	2	77%	1	7	49	54%	12
Conference	7	7	0	100%	0	7	0	100%	0
Fisherman	5	5	2	100%	0	4	5	80%	8
Valleyforge	3	3	2	100%	0	1	1	33%	0
UniMelb	2	2	1	100%	0	2	27	100%	0
Konqueror	0	0	6	100%	0	0	11	100%	0
UBC	0	0	0	100%	0	0	1	100%	0
BMVBS	1	0	0	0%	0	0	2	0%	0
StarWars	12	12	0	100%	0	10	91	83%	3
TOTAL	103	98	31	95%	1	86	389	83%	52

(b) X-PERT's results compared to those of a state-of-the-art technique.

X. THREATS TO VALIDITY

As with most empirical studies, there are some threats to the validity of our results. In terms of external validity, in particular, our results might not generalize to other web applications and XBIs. To minimize this threat, in our study, we used a mix of randomly selected real-world web applications and applications used in previous studies. The specific browsers used in the evaluation should not have affected the results, as our technique does not rely on browser specific logic and operates on DOM representations, which are generally available. Thus, we expect the technique to perform similarly on other browsers.

Threats to construct validity might be due to implementation errors in X-PERT and in the underlying infrastructure—especially with respect to the integration with the browser to extract DOM data. We mitigated this threat through extensive manual inspection of our results.

XI. CONCLUSION AND FUTURE WORK

Because of the richness and diversity of today's web platforms, XBIs are a prominent issue for web application developers. To address this issue, in this paper, we presented an automated approach that can accurately detect XBIs in web applications. The definition of our approach was guided by our findings in a systematic study of real-world XBIs in a large number of web applications. To target different types of XBIs, our approach integrates three existing techniques with a novel relative-layout XBI detection algorithm. Most importantly, the approach applies different techniques to different aspects of a web application, which allows it to maximize the overall effectiveness of the approach. We implemented our approach in a tool, called X-PERT, and used the tool to perform an extensive empirical evaluation on a set of real world applications. Our results show that X-PERT can identify XBIs accurately (76% precision) and effectively (95% recall), and that it can outperform a state-of-the-art technique.

One possible direction for future work is to investigate techniques that can automatically eliminate the XBIs identified by our approach through browser-specific automated web page repairs. Another possible direction is the identification of cross-platform incompatibilities for those (increasingly common) applications that are developed in different variants (*e.g.,* desktop, web, and mobile variants) and should behave consistently across variants.

ACKNOWLEDGMENTS

This work was supported in part by NSF awards CCF-1161821, CNS-1117167, and CCF-0964647 to Georgia Tech, and by funding from IBM Research and Microsoft Research.

REFERENCES

[1] Stackoverflow, http://data.stackexchange.com/stackoverflow/query/77488/posts-for-cross-browser-issues, August 2012.

[2] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceeding of the 33rd International Conference on Software Engineering (ICSE)*. ACM, May 2011, pp. 561–570.

[3] C. Chapman, "Review of cross-browser testing tools." http://www.smashingmagazine.com/2011/08/07/a-dozen-cross-browser-testing-tools/, August 2011.

[4] I. Safairis, "15 useful tools for cross browser compatibility test." http://wptidbits.com/webs/15-useful-tools-for-cross-browser-compatibility-test/, March 2011.

[5] S. Roy Choudhary, H. Versee, and A. Orso, "WebDiff: Automated identification of cross-browser issues in web applications," in *Proceeding of the 2010 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, September 2010, pp. 1–10.

[6] S. Roy Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE, April 2012, pp. 171–180.

[7] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "Webmate: a tool for testing web 2.0 applications," in *Proceedings of the Workshop on JavaScript Tools (JSTools)*. ACM, June 2012, pp. 11–15.

[8] Browserbite, "Cross browser testing with computer vision," http://app.browserbite.com/.

[9] C. Eaton and A. M. Memon, "An empirical approach to evaluating web application compliance across diverse client platform configurations," *International Journal of Web Engineering and Technology*, vol. 3, no. 3, pp. 227–253, January 2007.

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

[11] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, vol. 6, no. 1, pp. 3:1–3:30, March 2012.

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