Journal of Applied Intelligent System (e-ISSN : 2502-9401 | p-ISSN : 2503-0493)
Vol. 3 No. 1, August 2018, pp. 1 – 12
DOI:

An Automated Testing Framework for Cross-Browser Visual Incompatibility Detection

Zhen Xu[1], **James Miller***[2]
Department of Electrical & Computer Engineering, University of Alberta, Edmonton, Canada
E-mail : zxu3@ualberta.ca[1], jimm@ualberta.ca[2]*
**Corresponding author*

Received 8 December 2015; Revised 10 February 2016; Accepted 2 March 2016

Abstract - Due to the rapid evolution of web applications and computer techniques, visual incompatibility of web pages has become a problem across different browsers and platforms influencing the functionality of the web applications. At the present, researchers have made progress to address such issues; in addition, many commercial tools have emerged as well. However, drawbacks still exist in the existing work, where fully automate testing at the system level is still not achieved. In this paper, we attempt to propose a framework to detect the cross browser visual incompatibilities automatically. Highlights of the proposed framework include template based case organization, version based automation, and similarity embedded incompatibilities identification.

Keywords - Automated testing; Visual incompatibility; Visual similarity

1. INTRODUCTION

Web applications have become more and more popular nowadays. Compared with traditional applications with a client-server architecture, they are cross-platform, fully-functional, and easier to deploy (ready-to-use and no installation or configuration required). Developers of web applications, work hard on the goal of providing a universally identical user experience. However, due to the incompatibilities among browsers (and platforms), this goal is difficult to achieve. Visual differences of a web page rendered across browsers, in some cases, are expected or acceptable (such as fonts of text). However, in many other cases, they are incorrect and therefore may cause reading problems (such as missing or incorrectly presented content). The latter inconsistencies are considered as the cross-browser visual incompatibility (VI) in this paper.

The test of a web application to identify such VIs can be conducted manually, by reading and comparing each page through all the target browsers. This activity is highly manual and thus cost-intensive, time-consuming, and in many cases error-prone. On the one hand, a fully functional web application usually contains thousands of web pages, which makes it impossible to test all of them manually. On the other hand, many of the web pages are rendered from the same template, hence testing each of these pages is a repetitive and unnecessary task. In this paper, we propose an automated testing framework to solve these problems. The highlights of the framework include:

- template based case organization – extract different web pages rendered by the same source template as a single test case;
- version based automation – rerun the test case when changes of the source code are detected; and

- incompatibility identification with similarity estimation – provide both the list of visual. Incompatibilities and the quantitative similarity score for each pair of browsers.

The rest of the paper is organized as follows. Section 2 discusses related work, including the advantages and limitations of currently existing cross-browser testing tools. Section 3 describes the automated testing framework. It covers VI detection algorithm and the automation schemes. Section 4 illustrates the automated testing tool. Section 5 concludes the current progress and presents the future work.

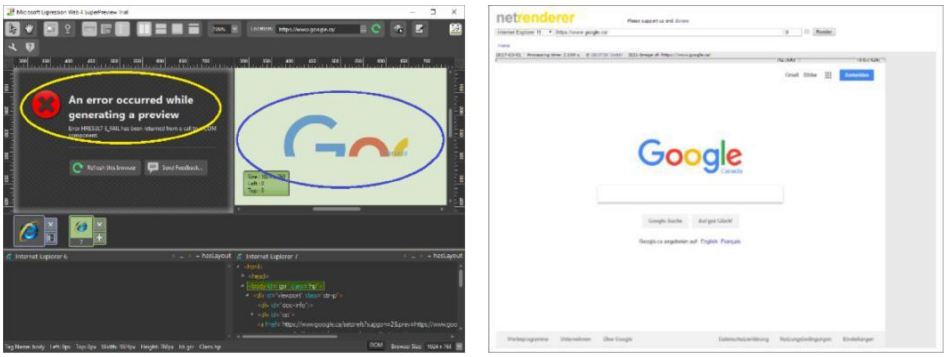
2. RELATED WORK

In this section, we will review existing research and tools regarding cross browser testing of web applications and web pages. There are many testing tools related to this area in the market. We will describe each in detail in this section by pointing out their pros and cons.

The research papers in the literature, regarding this area, are limited. Mesbah and Prasad [1] propose an approach to automatically analyse web applications under various browsers and present the observed discrepancies on a pairwise basis. Choudhary et al. [2] investigate cross browser issues and propose an approach to automatically detect these issues based on differential testing. They implement their approach in the WebDiff tool with acceptable number of false positives. Later, they propose a more comprehensive tool, namely, CrossCheck, based on the WebDiff and the CrossT. The CrossCheck tool [2] combines the benefits of WebDiff and CrossT, and can provide both visual difference detection and functional difference detection. Subsequently, they present another tool called X-Pert [3]. In these models, they divided the detection of VIs into three aspects: structure XBI (cross-browser incompatibility) detection, text-content XBI detection, and visual-content XBI detection. The structure XBI detection employs the “alignment graph”, which records the hierarchical and geometrical information of each DOM element (e.g., element 1 is above element 2 and they are left and right aligned). This is a novel idea of XBI detection as it narrows down the numeric coordinates of elements into a limited number of relations based on the relative position. The text-content XBI detection compares the text of elements. The potential problem of comparing textual strings is that in a multi-language web page scenario (e.g., the English and the French version of Google’s home page), text is not the core content, and thus the pages are similar to users/developers while the comparison results suggests dissimilar, leading to false positive results. The visual-content XBI detection takes the screenshot images as the input – the images of the leaf DOM elements only, to be precise – and compares the colour histogram using ^{2} distance. The limitation of it is that leaf elements represent only part, and in many cases only a small part of the whole page; and using colour distribution to determine image similarity ignores the actual content, thus may also raise false positive results.

Figure 1 shows three tools that only support cross-IE incompatibility detections. The Expression Web SuperPreview supports only versions of IE 6 and 7 (IE 11 is in the list, but actually not supported). It provides functions such as side-by-side comparison, window size customization, and DOM inspection. The side-by-side comparison enables us to compare web pages with different browsers intuitively and conveniently on a single screen. The window size customization allows us to change the width and height of the view port to simulate different devices and screens. With the DOM inspection function, we can investigate the web pages in a responsive way. This tool returns error on many web pages (marked by the yellow circle in the figure); and when the page preview is acquired, it is limited in the current view port. Content outside of the view port will not be rendered (as shown in the blue circle). IETester can perform side-by-side comparison, but lacks in window size customization. With the extension of DebugBar, it can also perform DOM inspection. Although most versions of IE are claimed to be supported, many return errors. IE NetRenderer can only draw web pages with the target version of IE to generate screenshot image. Therefore, it provides no side-by-side comparison or DOM

Inspection. By investigating the tool, we also observe that it does not support window customization.



(a) Expression Web SuperPreview (b) IE NetRenderer

(c) IETester

Figure 1. The Three Tools for Cross-IE Incompatibility Detection

Browsershots, Browsera, BrowserBite, BrowserStack, and CrossBrowserTesting are five tools that support multi-browser and multi-platform detections. This meets the minimum request for VI detection. For the input, only Browsera can take multiple URLs as the input, while all the other tools allow only one URL per test. Thus, for web application developers to conduct full-site tests, most tasks will have to be performed manually. Also, Browsershots, BrowserStack and CrossBrowserTesting provide configurations to customize window size. As for the detection, Browsershots, BrowserBite and BrowserStack load a web page with all selected browsers and then simply take all the screenshot images as the results, without doing any VI detection. On the other hand, Browsera and CrossBrowserTesting provides both screenshot images and detection reports, as shown in Figure 2.

(a) Browsera (b) CrossBrowserTesting

Figure 2. Detection Reports of Browsera and CrossBrowserTesting

Browsera reports all the detection results into one page, where the web page drawn by selected browsers are all displayed in side-by-side sub-frames. The identified differences are

also listed. When the cursor is hovering above an identification, the corresponding element in each page will be highlighted. CrossBrowserTesting displays detection results in two pages, namely the summary view and the side-by-side view. After selecting a browser as the baseline, the summary view lists the number of differences between it and each of the remaining browsers. If the detection reveals that all browsers display the web page identically, then this summary is sufficient; however, if there exist VIs, the user can find details from the side-by-side view, where only the pages drawn by two selected browsers are displayed followed with a list of differences. When a difference is selected, the corresponding elements in the two pages are highlighted.

For web application developers, a good automated testing framework should be able to create test cases that cover all web pages automatically, run the tests automatically, and report test results automatically and accordingly. The aforementioned tools: a) all render web pages automatically, however only some of them can perform VI detection automatically; b) all of them take manual inputs, making it difficult to conduct full-site testing automation; c) only some of the tools automatically creates identification reports; and d) none of them support regression tests for web applications that are in development (BrowserBite supports schedule configurations that repeat the test daily or weekly, however the option is limited to every 15 minutes, and updating the schedule does not work correctly).

3. AUTOMATED TESTING FRAMEWORK

In our previous work [4], we developed a method to calculate the quantitative visual similarity of two web pages. The present paper extends this method by adding an extra step to identify different elements between the two pages, and uses the extended method as the core function of the automated testing framework to detect VIs.

3.1. Automated Page-Level Detection of VIs

The page-level detection employs the above extended method as the core function of the proposed testing framework. This method extracts block trees from the web page rendered by two different browsers, and uses the two block trees to detect VIs.

3.1.1. Block Tree Extraction from Web Page

The DOM tree contains all the information from a web page, but only the visible elements contribute to the visualization of the web page. Therefore, the first step of the block tree extraction is to remove invisible DOM elements. The next step is to merge semantically related elements into blocks. This is done by translating and applying the Gestalt laws of grouping as follows.

- The Gestalt law of simplicity shows people's tendency to recognize the simplest representation of objects. To interpret this law, we take each DOM element as the simplest representation of objects.
- The Gestalt law of closure indicates that people are inclined to construct complete shapes from incomplete ones. A DOM element is often overlapped by its child DOM elements, leaving the shape incomplete, but people are still able to recognize it as a complete rectangle. As such, to interpret this law, we treat all DOM elements as complete rectangular objects.
- The Gestalt law of proximity states that people have the tendency to group close objects and separate distant ones. Therefore, to translate this law, we merge elements into blocks based on this distance. In the web page scenario, we compare the distances between each pair of adjacent sibling DOM elements, those with smaller distances are "clustered" into a group, and those with larger distances are separated into different groups.
- The Gestalt law of similarity illustrates that people are prone to regard similar objects as a group. Here, similarity refers to the visual features related to background,

foreground, and size. If any of a list of sibling DOM elements is different from others in the above three aspects, we put it into a different group.

- The Gestalt law of continuity describes people's tendency to group aligned objects. In other words, if any DOM element is not aligned with its siblings, it is put into a different group.
- The Gestalt law of common fate reveals that people are inclined to put objects with the same motion into the same group. To translate this law, we focus on the scrolling behaviours when it comes to motion trends. Most DOM elements move accordingly when the user scrolls a web page, but some other elements may stay still, or move slower or faster. Such elements that do not move in the same way with others are place into a different group.
- The Gestalt law of symmetry tells us that people tend to perceive symmetric objects as a single group. Since this law is not common in web pages, we do not consider it in the present paper.
- The Gestalt law of past experience states that people are prone to rely on past experience when interpreting objects. Again, we do not consider this law in the present paper, because it is beyond the scope of web page analysis.

(a) Original Page

(b) Analyzed Page

```
[BODY]: left=0,top=0, ...
|- [FORM,DIV]: left=-1988,top=-1999, ...
| |- [HEADER,DIV,FOOTER]: left=0,top=0, ...
| | |- [DIV,DIV,DIV]: left=0,top=0, ...
| | | |- [DIV]: left=50,top=10, ...
| | | | |- [NAV,DIV]: left=218,top=10, ...
| | | | | |- [UL]: left=276,top=10, ...
| | | | | | |- [LI]: left=276,top=20, ...
| | | | | | | |- [UL]: left=276,top=22, ...
| | | | | | | | |- [A,A,A,A,A,A,A]: left=276,top=22, ...
| | | | | |- [INPUT,BUTTON]: left=802,top=13, ...
| | | |- [DIV]: left=50,top=61, ...
| | | | |- [A,NAV]: left=50,top=87, ...
| | | | | |- [UL]: left=416,top=115, ...
| | | | | | |- [A,A,A,A]: left=436,top=119, ...
| | | |- [NAV]: left=51,top=155, ...
| | | | |- [UL]: left=51,top=155, ...
| | | | | |- [LI]: left=52,top=155, ...
```

(c) Partial of the Block Tree

Figure 3. The Example of UAlberta's Home Page

Figure 3 shows an example of the block tree extracted from University of Alberta's home page. By applying the Gestalt laws of grouping, the semantically related DOM elements are grouped into blocks. In Figure 3b, semantically related elements are marked with the same background colours. For example, as shown in the yellow circle at the lower left part, the news items are marked with the same background colour. This is because they refer to the same topic.

As a comparison, the three boxes in the middle area (marked in the black circle) contain image, text and buttons respectively, indicating that they are semantically non-related, so they are marked with different colours. Figure 3c shows partial of the block tree, where each line denotes a single block. From this figure, we can find that a) the DOM hierarchy is well maintained in the block tree; b) the root block consists of the "BODY" element from the DOM tree; and c) some blocks contain only one DOM element while others merge a group of elements into one block. *3.1.2. VI Detection and Similarity Estimation*

The two block trees retrieved from two browsers of a web page are compared to detect VIs. During the comparison, a tree edit distance (TED) based mapping scheme, the extended subtree model [5], is employed. An overview of the model is given below:

- Subtree mapping. Regular TED mapping schemes only map tree nodes. However, in a web page scenario, content elements are stacked up so that lower elements are always covered by upper elements. Hence, when we see the content of a block in the web page, it is the content of a subtree that is rooted at the block. Consequently, the subtree mapping scheme is more accurate approach.
- One-time mapping. If two subtrees are mapped, then they will have common subtrees (if there are subtrees in them). However, to avoid duplications, we do not map these common subtrees again.
- One-to-many condition. Regular TED mapping scheme only maps each tree node once. In the extended subtree model, a subtree of one tree is allowed to be mapped to several subtrees.
- Subtree weight determination. A subtree mapping has a weight that is equal to the mean value of the weights of the two subtrees. The weight of a subtree is equal to the number of nodes that take this subtree as their largest subtree.

The mapping of two block trees reflects the visual compatibilities, i.e., are the two corresponding blocks similar or not. Therefore, the detection of VIs is to locate blocks that are not in the mapping results. In other words, blocks that are added, deleted or changed from one tree to the other tree contain VIs. The quantitative similarity of the two block trees, the extended sub tree (EST) value, is calculated by (1):

$$*(\, . \,) = \frac{\Sigma \quad \in \qquad \times(\qquad)}{\max(| \, |,| \, |)} \qquad (1)$$

where, and are the two trees; | | and | | are the sizes of the two trees, which equal to the numbers of nodes in and , respectively; is the mapping results; ( ) is the weight of the mapping ; is the coefficient to adjust the relation among mappings with different subtree sizes; and is a geometrical parameter to reflect the importance of the mapping

with respect to the position of block in and . = 1 when the node of and of in have the same depth, otherwise = 0, which is a constant in the range of (0,1).

3.2. Automated System-Level Testing for Detections

System-level testing is designed to evaluate all the web pages in a web application. The automated testing framework should be able to discover objective functions, trigger actions, and report outcomes without human intervention. To achieve this goal, three modules are proposed to construct the testing framework, namely the source parser, the schedule builder and the result reporter.

3.2.1. Source Parser

The core task of this module is to discover the objective functions. In the web application scenario, to detect VIs, the objective functions cover all the web pages because all of these pages must (ideally) be bug-free. Development of modern web applications relies on page templates. That is, utilizing one template dynamically generates similar web pages. Consider Google's search result page, when a user types in "online shopping", the search result page displays

dozens of online shopping related links; and when the user types in “health care”, the page displays another dozens of links, which are similar in the layout with the previous page except the details. This is because the search result page utilizes a template that shows different content according to the inputs. To test a web application, it is useless and impractical to test all possible web pages. Instead, we only need to test one case for each page template. Consequently, to conduct automated testing, the practical objective functions should be narrowed down to include only unique page templates.

Consider a typical Django project as an example, each component app of a Django project contains a source file named “urls.py”, where all the implemented URL entries are recorded and linked to the corresponding view methods. The view methods are further linked to the page templates that are used for displaying actual content. Therefore, in the Django project testing practice, the objective functions map to all these URL entries. To automatically test such projects, the source parser should be able to detect all possible URL entries. Meanwhile, some of these entries contain parameters, and thus, the source parser also needs to be able to detect these parameters and assign proper values to them. This may require accessing data models and querying databases.

3.2.2. Schedule Builder

As the name indicates, the schedule builder manages the schedule of testing automation. During the development of a web application, the source code keeps changing constantly, and it is necessary to repeat the tests through out the whole development period.

A straightforward solution to automatically repeat the tests is to set up a schedule based on the time. The second method we proposed to automatically run the tests is based on the source code changes. Not all the objective functions change all the time, so we should only re-test those that have changed and ignore those that have not changed. For instance, the developer may focus on one app of the web application today and another app tomorrow, so it is unnecessary to re-run tests on the second app. In this case, the schedule builder should monitor each template, and automatically triggers the actions to re-run the corresponding tests based on changes of the template’s source codes (for example, re-test after a defined number of updates to the source code).

3.2.3. Result Reporter

The automatic testing framework runs without human intervention; therefore, once the results are produced, it is possible for users to ignore their implications if the framework does not notify the developer. This is acceptable if a test case passes, but when the result fails, the result reporter module must notify the developer. Content of the notifications include a true/false assertion (i.e., indicating whether the template page is rendered identically in the target browsers), a quantitative value of the visual similarity (where 1.0 indicates identical and 0.0 indicates completely different), and a list of differences between the rendered pages.

Priorities must be added to the notifications automatically, and the result reporter must display the outputs accordingly. This is because automatic repetitions of the scheduled tests will generate significant numbers of results, and only those failed results (i.e., VIs that have been detected) require the developer’s attention. If the objective of a test is to confirm the template is rendered identically by all browsers, then all the results of “true”, “1.0”, or an empty list of differences are not important. In this case, the priority of these results should be set to lowest. As the opposite, the lower the similarity value (or the larger the difference list is), the higher the priority should be.

The difference list is easier to read if it is combined with the side-by-side display for locating VIs. Therefore, presentation of VIs must be done by rendering the web page in all the browsers simultaneously and highlighting the identified differences (for example, highlight them by changing the background colours or by outlining the borders of the related blocks).

Figure 4 shows the outline of the automated testing framework, where the green and yellow rectangles indicate the data and components of the framework, respectively; the green and yellow arrows denote the data and control flow, respectively; and the blue arrows refer to the notification flow.

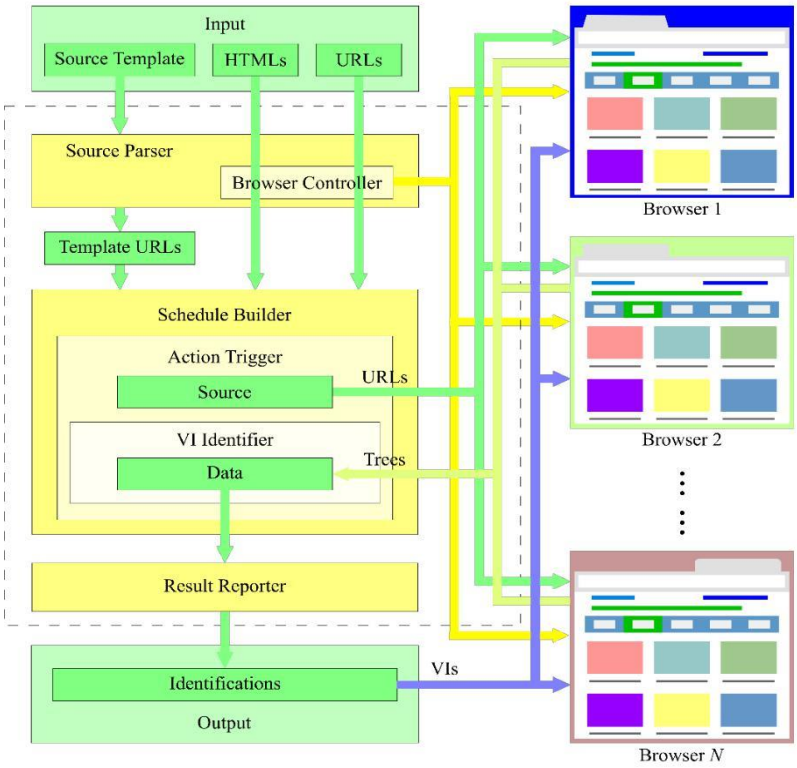


Figure 4. The Automated Testing Framework

- The framework accepts both templates of web applications and specific URLs/HTML code as the input. Although the templates require extra processing by the source parser, the specific URLs/HTML code can be directly input in the schedule builder.
- The browser controller registers and manages the supported browsers.
- The schedule builder manages the automation schedule, either by time, or by changes to the source code, or by both. According to the schedules, its subcomponent, the action trigger, conducts the testing process, where the sources are passed to the VI identifier for VI detection and similarity estimation.
- The result reporter collects all the test results, filters them by priorities, and notifies web application developers selectively. By updating the web pages in the browsers with the difference list, a side-by-side comparison provides fast location of VIs.

4. AUTOMATED TESTING TOOL

To conduction VI detection, the minimum request is that the automated testing tool must support multiple browsers and/or multiple platforms. The implementation, hence, is designed as a distributed system, where a central node communicates with and controls all leaf nodes. The leaf nodes run specific OSes and browsers and therefore consist of the testing farm, which renders the target web pages and collects the corresponding source data (i.e., the block trees). The central node deploys the automated testing tool as well as the target web application, and performs the testing automation. Figure 5 shows a sequence diagram of the tool's testing process.

4.1. Browser and Platform Registration

During the initialization of the automated testing tool, the supported browsers and platforms are to be configured. The core thread of this distributed system that is located in the

central node will send queries to all branches for browser detection. The active leaf nodes will respond to it with the configuration information, including the name and version of both its own operating system and installed browsers. Hardware configurations of the machine (either physical or virtual) could also be included if necessary, such as resolutions of mobile devices. Figure 6 shows the two initialization dialogs of the tool, where the browser registration illustrates examples of local browsers.

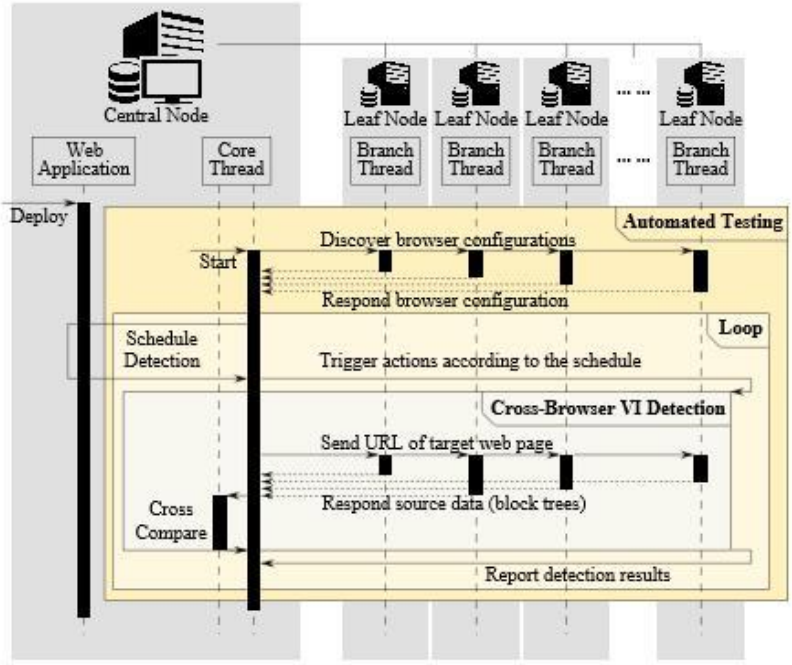
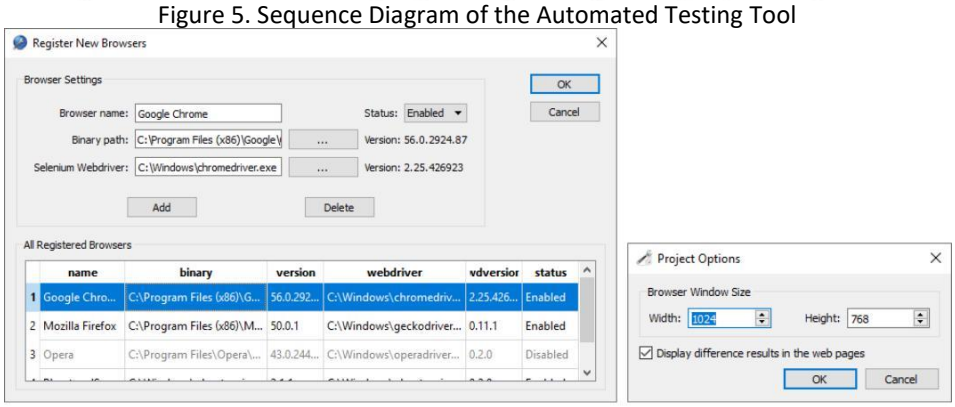


Figure 5. Sequence Diagram of the Automated Testing Tool



(a) Browser Registration (b) Project Options

Figure 6. Initialization Dialogs of the Automated Testing Tool

4.2. Template Based Test Case Organization

The automated testing tool analyses the source code that encode all the templates such as the full list of RESTful URLs, and generates test cases for each such template entry. As mentioned in the previous section, this task is done by the source parser. If necessary, the source parser will dig further information (for example dynamic content in the URLs, such as the user ID “MarcoXZh” in “https://github.com/MarcoXZh/”) from sub apps and the web application’s database. Note this step is project dependant – different web applications require different strategies for code analysis.

Figure 7 shows example pseudocode of the source parser for Django project analysis. This algorithm takes a Django project’s project name and root directory as inputs. It searches

the “manage.py” script for configurations of installed apps and databases (Line 6 to 8), and then parses the templates as follows:

```
1   ALGORITHM ParseSource_Django:
2       INPUT: project name: PN,
3              root directory PN: RD
4       OUTPUT: all template URLs: TS
5
6       config_script = get_config_script(RD.mangage.py)
7       apps = get_installed_apps(config_script)
8       database = get_database(config_script)
9       connect(database)
10      TS = [EMPTY_LIST]
11      FOR EACH app IN apps DO:
12          FOR EACH url, view IN urlpatterns(app.urls.py) DO:
13              IF contains_django_variables(url) THEN:
14                  variables = retrieve variables(url)
15                  model = retrieve_model(view)
16                  sql table = retrieve sql table(app.model)
17                  values = [EMPTY_LIST]
18                  FOR EACH var IN variables DO:
19                      value = query(sql_table, var)
20                      values.append(value)
21                  END FOR
22                  replace_all(url, variables, values)
23              END IF
24              TS.append(url)
25          END FOR
26      END FOR
27      close(database)
28      RETURN TS
29  END ALGORITHM
```

Figure 7. Extracting Templates from Django Projects

1. for each installed app, it checks its “urls.py” script to detect all supported URL patterns and the corresponding views;
2. for each URL pattern, if it contains variables, then the source parser needs to assign correct values by querying the view-model-table chain for them (Line 13 to 23);
3. after assigning values to the variables, or if the pattern is a regular URL and contains no variables, the URL pattern is added to the template list; and
4. the full template list includes all the URL patterns of all the installed apps. Due to the templates list consisting of URL patterns only extracted from the “urls.py” script, it will not store duplicated URL entries, and at the same time cover all the supported URLs of the web application.

4.3. Version Based Automation

Once the templates are extracted from the web application’s source code, the core thread sends signals to all the leaf nodes according to the predefined schedule for VI detection. This tool contains both time-driven schedules (i.e., triggers actions after a fixed time) and change-driven schedules (i.e., triggers actions after a fixed number of changes being made in the target source codes). Once the predefined time has expired or the predefined number of code changes has detected, a re-test is triggered. However, if the target source code remains unchanged (i.e., no changes of source code found by the diff process, or http response code of the target web page being 304), then the test will be skipped. Figure 8 shows the scheduler builder of the tool, which combines the functions of template extraction and schedule configuration. Note the three source entries at the right-side list view are raw HTML code – a regular URL without variables and a variable-included URL. The “`|$1|`”, “`|$2|`”, etc. are the variable names, and the corresponding values are stored however not displayed. Testing frequency of the schedule builder can be configured as either change-based or time-based or both. Collection of the target test cases’ screenshots can be customized, too.

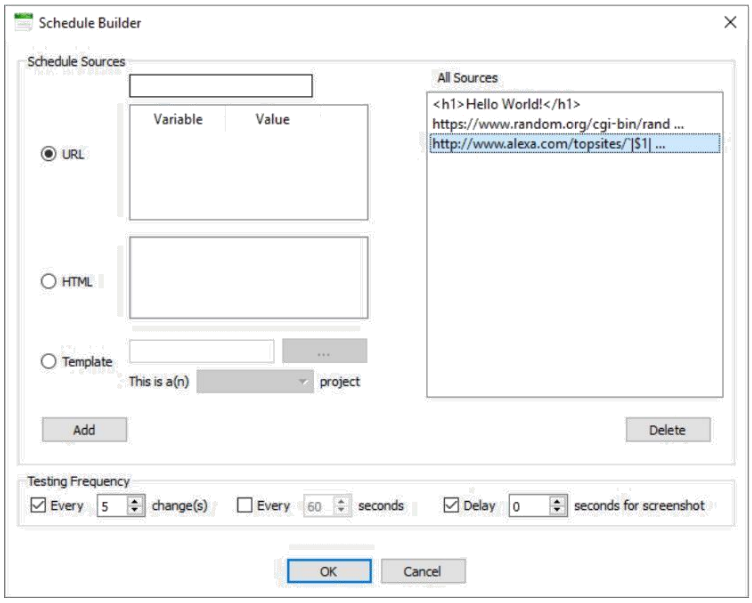


Figure 8. Schedule Builder of the Automated Testing Tool

4.4. Case Study

In this paper, we evaluate the efficiency of the automated testing tool though the case of University of Alberta's home page. We run the tool to compare the page in two popular browsers: Google Chrome version 57 and Mozilla Firefox version 52, and on two platforms: Windows 10 and CentOS 7. By comparing the results of CrossBrowserTesting and our VI detection, the following conclusions can be derived:

1. Both CrossBrowserTesting and our automated testing tool can locate VIs of web pages among different browsers; and at the same time, both can make the correct conclusion without raising false positives if two versions of a web page are identical.
2. Results of CrossBrowserTesting contains only VIs, lacking in intuitive conclusions to determine how similar the two versions of a web page are. Thus, if a test result contains ten small VIs and another test result contains one big VIs, it is difficult to figure out the priority for developers to start debugging. As the comparison, our tool calculates the EST similarity, which enables the priority judgement. Table 1 shows the EST values of the evaluation.
3. During the tree comparison, our EST model maps sub trees instead of nodes, thus it can avoid potential duplications of VI detection. As previously shown in Figure 2b, CrossBrowserTesting identified four VIs caused by the X coordinates of the elements. However, the second and the third VIs are child elements of the element in the first VI.

   Due to the mismatch of the parent element's X coordinate from the two versions of the page, its child elements consequently mismatch, too, Therefore, the second and the third VIs are actually a duplication of the first VI. Our EST model, by absorbing comparisons of child elements and mapping subtrees, prevents such hierarchical false positives from being detected.

Table 1. EST Similarity Values of the Cross-Comparisons

Browser1	Windows-Chrome	CentOS-Chrome	Windows-Chrome	Windows-Firefox
Browser 2	Windows-Firefox	CentOS-Firefox	CentOS-Chrome	CentOS-Firefox
EST Value	1.0000	1.0000	0.9603	0.9603

5. CONCLUSIONS

Diversity of present web browsers and platforms have brought cross browser issues to both web users and developers. To detect cross browser incompatibilities, many commercial

tools have been developed and relevant topics have gained attention among researchers as well. In this paper, we target the detection of VIs and attempt to propose a testing framework to detect these incompatibilities automatically. Three advantages exist in the automated testing framework. Firstly, the detection of VIs is based on source templates. By doing so, we narrow down the scale of testing. Second, automation is achieved by schedules based on both time and changes of the source code, which avoids human intervention and at the same time this further reduces the test ranges. Finally, the framework provides both a list of VIs (including a rendered presentation of these differences) and a quantitative similarity value as the result. This makes it possible to notify web application developers by priorities.

An automated testing tool is designed according to the framework. This tool allows the registration of browsers on both local and remote machines, and utilizes all these registered browsers to conduct VI detection. It extracts the templates depending on the type of the target web application. A Django example is employed showing that this tool can extract both plain URLs and URLs with variables, where the extraction of the latter is done by querying information from the web application's database. Version base automation of the tool is achieved by both time-driven and change-driven schedules. A case study is presented to illustrate the efficiency of the extended subtree model by comparing it with the CrossBrowserTesting. Conclusions reveal that the quantitative values indicate how similar the two browser versions of a web page are and serves as a reference to debug these VIs; and the subtree mapping scheme has eliminated duplications of the VI detection results.

ACKNOWLEDGMENT

The authors thank China Scholarship Council (CSC) for the financial support.

REFERENCES

[1] Ali Mesbah and Mukul R. Prasad. 2011. *Automated cross-browser compatibility testing*. Proceedings of the 33rd International Conference on Software Engineering. ACM, 561–570.

[2] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. 2012. *Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications*. 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, 171–180.

[3] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. 2013. *X-PERT: accurate identification of cross-browser issues in web applications*. Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 702–711.

[4] Zhen Xu, James Miller. *Estimating similarity of rich internet pages using visual information*. International Journal of Web Engineering and Technology 12, no. 2 (2017): 97-119.

[5] Ali Shahbazi and James Miller. 2014. *Extended subtree: a new similarity function for tree structured data*. Knowledge and Data Engineering, IEEE Transactions on 26, 4 (2014), 864–877.