CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODE

RÉSUMÉ

Puisque les entreprises tentent de minimiser leurs coûts, les applications web deviennent de plus en plus utiles en fournissant des applications multiplateformes qui peuvent être facilement portées aux appareils mobiles et environnements de bureau. Par contre, tester ces applications est beaucoup plus ardu que de tester des applications de bureau en raison du manque d'outils.

Dans ce mémoire, nous décrivons un outil de test d'applications web, Cornipickle, qui utilise une spécification expressive sur la mise en page. Le langage de spécification de Cornipickle fait des affirmations sur la mise en page des applications mais nous montrons qu'il permet aussi de tester des comportements. Grâce à lui, nous réussissons à détecter efficacement les bogues identifiés dans une étude de 35 applications webs. Nous montrons aussi la polyvalence de Cornipickle en l'utilisant dans une multitude d'applications et outils, notamment en l'associant à un robot d'indexation.

ABSTRACT

With businesses trying to minimize cost, web applications are becoming increasingly useful by providing cross-platform applications that can be easily ported to desktop and mobile. However, testing these applications is much more arduous than testing desktop applications due to the lack of tools.

In this thesis, we aim to describe a web application testing tool called Cornipickle which uses an expressive layout specification. The specification language for Cornipickle makes statements on applications' layouts but we show that it can also test behaviours. With it, we effectively achieve to detect the bugs found with our survey of 35 real-world web applications. We also show Cornipickle's versatility by using it in multiple applications and tools, notably pairing it with a crawler.

INTRODUCTION

The use of the web has seen significant changes since the inception of HTML in the early 1990s. Thanks to the use of databases, client- and server-side code, web sites have moved from being mere repositories of simple and static pages to become complex, interactive and stateful applications in their own right. Sites like Facebook or LinkedIn have become common fixtures of web users, and vendors like Microsoft now maintain a fully featured web-based version of their popular Office suite. Recent years have even seen the advent of full-fledged operating systems relying solely on the web stack of HTML/CSS/JavaScript, such as FirefoxOS[1], Chromium OS[2] and eyeOS[3]. The web is now a large and mature software ecosystem on a par with the complexity found in traditional desktop applications.

Due to the somewhat complex relationship between HTML, CSS and JavaScript, the layout of web applications tends to be harder to properly specify in contrast with traditional desktop applications. The same document can be shown in a variety of sizes, resolutions, browsers and even devices, making the presence of so-called layout "bugs" all the more prevalent. Such problems can range from relatively mundane quirks like overlapping or incorrectly aligned

[1] `https://www.mozilla.org/en/firefox/os/`

[2] `https://www.chromium.org/chromium-os`

[3] `http://www.eyeos.com`

elements, to more serious issues compromising the functionality of the user interface. The problem is exacerbated by the current lack of tools to test web applications with respect to their layout, and in particular the short supply of formally defined languages for expressing the desirable properties of a web application's content and display.

For example, currently, Selenium WebDriver lets the user write scripts in Java that test rendered web pages by accessing each element and its properties. However, the assertions, being in Java, do not provide clear and readable descriptions of their purpose. There are also visual techniques like WebSee (Mahajan and Halfond, 2015) that visually compare an input web page with the correct web page in order to identify layout problems like overlapping elements. This technique forces the generation of an oracle to be the "correct" rendering of the page and suffers from a lot of false negatives caused by different test contents or browser dimensions.

These problems are tackled in this thesis with the implementation of Cornipickle, a language and a tool to test web pages layouts at runtime. The language borrows from first-order logic, linear temporal logic and existing testing tools like Cucumber (Wynne and Hellesoy, 2012) to be used as an oracle for the tests. Using runtime monitoring ideas, the tool serializes web pages and tests whether it respects the properties defined prior. We achieved to set up a tool that is easy to use with its very expressive language, versatile (in the sense that it works in every browser and can express most bugs), and fast enough to execute at runtime on moderately big websites.

In order to cement Cornipickle as a general tool for testing web applications, it was used in other applications to test some types of bugs more specifically and in one other tool to ease its use through Selenium. First, a plugin for the very popular crawler Crawljax was developped to test behavioural bugs. In the same vein, bugs related to Responsive Web Design

(RWD) were detected with an application that uses Crawljax and a browser resizer. Finally, to facilitate the developpement of the latter, we wrapped Selenium's Driver class to allow testing with Cornipickle while using Selenium without additional work. This thesis is structured as follows:

Chapter 1 provides information on web applications, web browsers and bugs in websites. The last part of the chapter covers all the different types of bugs: layout bugs, non-layout bugs, RWD bugs and behavioural bugs.

The next chapter, Chapter 2, presents an overview of the work that has been done in the field of web application testing. We take a look at some testing tools, visual techniques, declarative specifications for Graphical User Interfaces (GUI), crawlers and RWD testing tools.

Chapter 3 describes our core testing tool, Cornipickle, that tests web application layouts using declarative constraints. We start by defining the language's grammar. Then, the tool's design goals and implementation will be described.

Next, Chapter 4 describes more specific tools and applications which use Cornipickle as the back end testing solution. We introduce a plugin for Crawljax, a stateful crawler, that allows the crawling of a website in a user-like way while testing Cornipickle assertions on every state. We also describe an application that tests RWD specifically. RWD is a design principle that focuses on displaying web pages correctly in any viewport dimension and without any loss of information. Since displaying a page correctly is harder with smaller space, the application horizontally shrinks the browser incrementally and tests the page on every resize. This is done on every page of a website with the help of Crawljax. Also, during developpement, a Selenium wrapper for Cornipickle was built due to some Crawljax limitations.

A conclusion summarizes the work, discusses the quality of our results and presents avenues for future works.

Most of the work in this thesis has already been published in the following research papers:

Sylvain Hallé, Nicolas Bergeron, Francis Guérin, Gabriel Le Breton, Oussama Beroual, "Declarative Layout Constraints for Testing Web Applications," In Journal of Logical and Algebraic Methods in Programming, Volume 85, Issue 5, Part 1, 2016, Pages 737-758, ISSN 2352-2208, https://doi.org/10.1016/j.jlamp.2016.04.001.

S. Halle, N. Bergeron, F. Guerin and G. Le Breton, "Testing Web Applications Through Layout Constraints," 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, 2015, pp. 1-8. doi: 10.1109/ICST.2015.7102635.

O. Beroual, F. Guérin and S. Hallé, "Searching for Behavioural Bugs with Stateful Test Oracles in Web Crawlers," 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST), Buenos Aires, 2017, pp. 7-13. doi: 10.1109/SBST.2017.8.

CHAPTER 1

BUGS IN WEB APPLICATIONS

The term “web application” was first introduced by the Java Servlet Specification 2.2 (Hunter, 1999). It refers to a single file containing every pages and resources that could be deployed accross web servers.

Today, web applications are websites that have functionalities similar to desktop applications, providing messaging or store services for example, in contrast to only displaying information from a file on a server. Technologies like Javascript, Ajax and jQuery are typically used in order to have fluid interactions with the server and a dynamic user interface.

We refer to web applications in this thesis because it is the most recent and complex product available in browsers. However, we include static and dynamic websites in this term.

Browsers are the means with which users display web pages. There are multiple different browsers on the market but some of the most widely used are Chrome, Firefox, Opera and Safari.

The component that is the most relevant to our subject is the rendering engine. The main flow of a typical rendering is described in Figure 1.1. The engine parses the HTML file into a DOM tree containing DOM nodes. DOM is an acronym for Document Object Model.

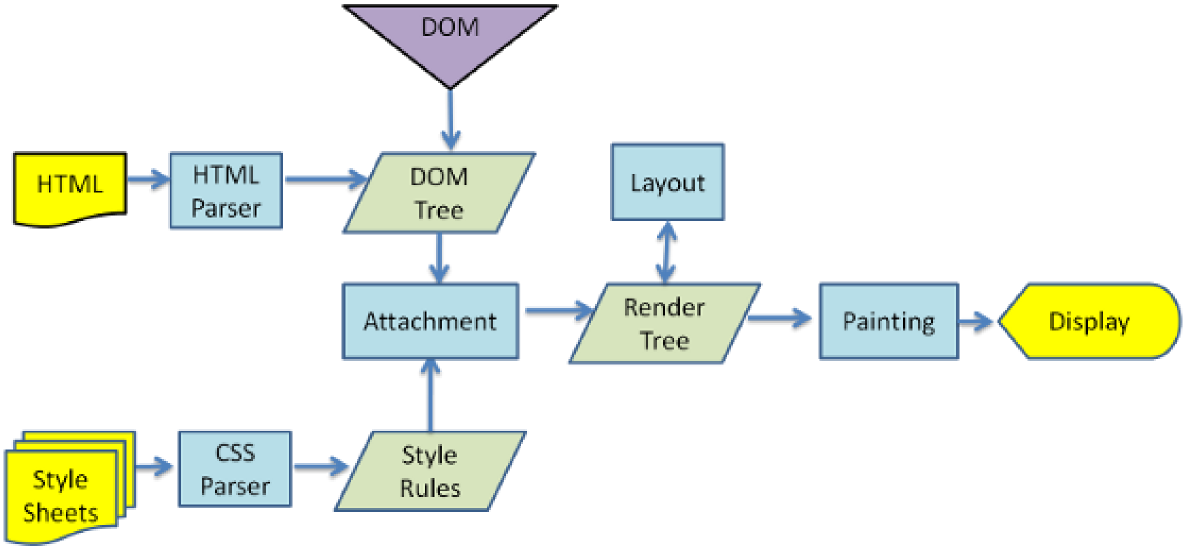


Figure 1.1: Main flow of Webkit (Garsiel and Irish, 2011)

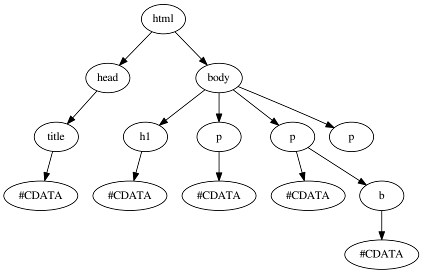
Essentially, it is a tree representing the HTML structure of the elements. Figure 1.2 shows the result of the transition from an HTML document to a DOM tree. To create the render tree with visual information (color, dimensions), the CSS files and the HTML file are parsed to obtain the CSS rules to be applied on the DOM tree. Then, a layout process goes through the render tree to position every node in the page with exact coordinates. Finally, the page is painted on the screen.

The complexity and coupling of HTML and CSS makes it also hard for the developer; building a bug-free web application is difficult today. As an experiment, we performed a survey of more than 35 web applications over a ten-month period in 2014–2015, and collected any bugs having an impact over the layout or contents of their user interface. The sites were surveyed in an informal way, by simply collecting data on bugs through the authors' daily use of the web. Table 1.1 gives the list of web sites and applications for which at least one layout bug has been found.

Every time such a bug was found, a bug report was created. This report contains a short

```
<html>
 <head>
  <title>My title</title>
 </head>
 <body>
  <h1>The first page</h1>
  <p style="color:red;width:400px">
   Hello world</p>
  <p style="font-size:14pt;width:200px;">
   Another <b>paragraph</b></p>
  <p style="width:400px;"></p>
 </body>
</html>
```

(a)



(b)

Figure 1.2: A simple DOM document. (a) HTML structure (b) Representation as a DOM tree; only element *names* are displayed: the remaining attributes and values are omitted for clarity.

- Academia.edu
- Acer
- Adagio Hotels
- Air Canada
- Air France
- AllMusic
- American Airlines
- Boingo
- Canadian Mathematical Society
- Customize.org
- Digital Ocean
- Dropbox
- EasyChair
- Elsevier
- Evous France
- Facebook
- IEEE
- Just for Laughs
- LinkedIn
- Liveshout
- Microsoft TechNet
- Monoprix
- Moodle
- Naymz
- NSERC
- OngerNeige
- ProQuest
- Rail Europe
- ResearchGate
- St-Hubert
- SpringerOpen
- TD Canada Trust
- Time Magazine
- Uniform Server
- YouTube

Table 1.1: Web sites and applications for which at least one layout-based bug has been found.

description of the bug (or the expected content of the page), a screenshot of the offending part of the page, as well as a snapshot of the page's contents.

In the following, we briefly present some of the bugs we discovered; many are manifestations of more than one type of layout problem, so that the classification given below is not a *partition* of all the encountered bugs. We shall highlight the fact that, unless stated otherwise, none of the bugs described here is a cross-browser problem —that is, the bug is present no matter what browser is being used to view the page, and is not caused by a divergent interpretation of the standards by two different browsers. Due to space limitations, we cannot provide screenshots or detailed information on all these bugs; however the complete bug database is publicly available online.[1]

1.1 LAYOUT BUGS

A *layout-based bug* is a defect in a web system that has visible effects on the content of the pages served to the user. This content can be anything observable by the client, including

[1] `https://bitbucket.org/sylvainhalle/cornipickle-bugs`

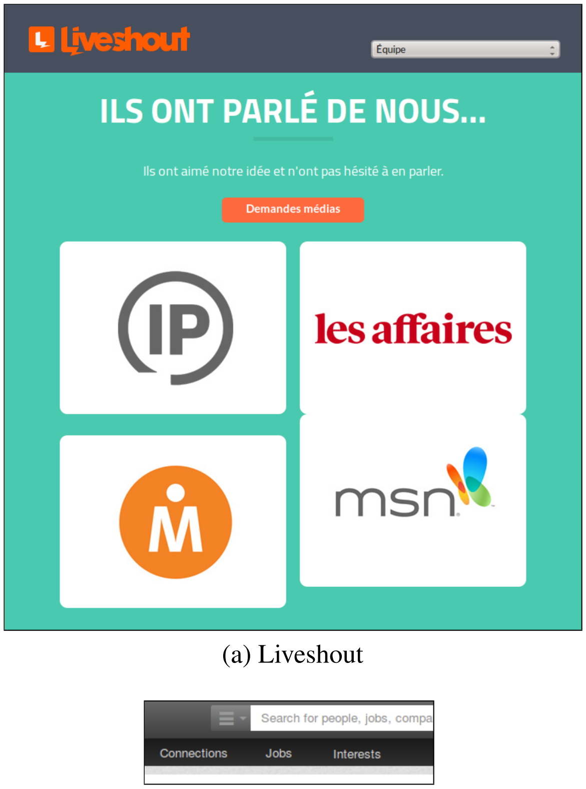
the structure of the page's DOM and the dimensions and style attributes of elements. Hence, unless it has observable effects on the client, we exclude from this definition anything that relates to the inner workings of the application: obviously any server-side state information (session variables, back-end database contents), but also any reference to client-side code, such as values of variables in local JavaScript code. In a nutshell, layout-based bugs are those that can be detected by client-side black-box testing.

This section shall first show that, although seemingly mundane, layout-based bugs are widely present in a large number of real-world applications and web sites. It will then demonstrate how, although based on page layout, bugs of this kind are not limited to simple, static presentation problems, and can in many cases reveal defects in the *behaviour* of the application.

A first category of bugs consists of *disruptions* in the layout or presentation of the page itself. By disruption, we mean any property of the page (position, size, number of elements) that contradicts the appearance or content expected by the designer for that page.

1.1.1 MISALIGNED ELEMENTS

A mundane but frequent layout problem is the presence of elements which should be visibly aligned but are not. Figure 1.3 shows an example of this bug from the web platforms LinkedIn and Liveshout. Sometimes, the misalignment is obvious, as in the case of Figure 1.3a, but sometimes, the misalignment is subtle, as an element may be off by a single pixel (Figure 1.3b).



(a) Liveshout

(b) LinkedIn

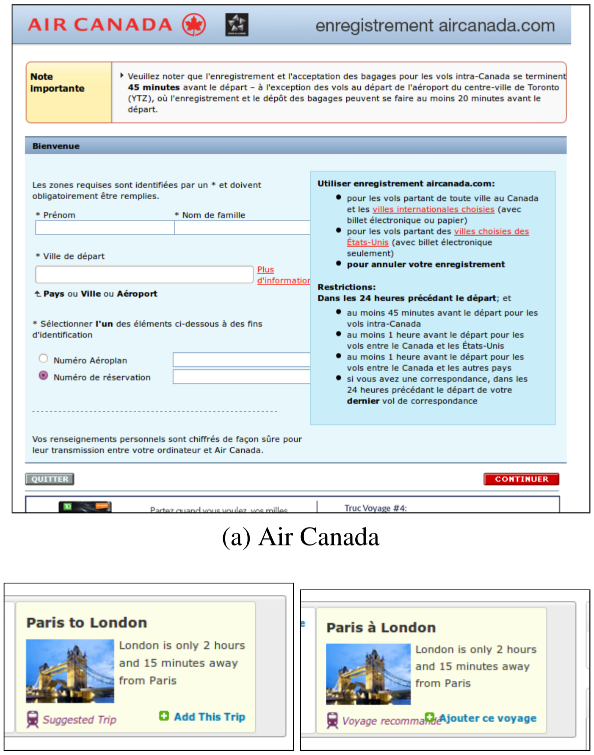
Figure 1.3: Examples of misaligned elements: a white box with an incorrect horizontal alignment (LiveShout); the “Interests” menu is one pixel lower than the others (LinkedIn).

1.1.2 OVERLAPPING ELEMENTS

A common layout problem is the overlapping of elements that should be disjoint in the layout of a page. In our empirical study, this bug was one of the most prevalent; problems of this nature have been found in sites as varied as Time Magazine, Air Canada, Moodle, and Rail Europe. Figure 1.4 shows examples of this problem. In some cases, the overlapping occurs through a disruption of the layout caused by the main document being resized below some minimum dimensions. Containers that were normally side by side are being forced on top of each other by the browser's layout engine. However, we also found that in many instances, this overlapping is caused by the fact that elements are absolutely positioned in a page with respect to their dimensions when the text they contain is in English. When displaying the web site in another language (such as French), it may occur that the corresponding text is longer than the English version, causing two elements that were disjoint to suddenly overlap. This has been observed, for example, in Figure 1.4 for the RailEurope web site.

1.1.3 ELEMENTS EXTENDING OUTSIDE THEIR CONTAINER

Another prevalent bug is the presence of elements that cause their parent container to extend beyond the dimensions it was designed for. In such a case, elements can be displayed past the boundaries of their parent container, causing unwanted overlapping with surrounding elements. This bug is the converse of the previous one: rather than extending their container outside its nominal dimensions, this time the elements that expand outside their container are simply clipped (i.e. trimmed off) from the display. Figure 1.5 shows an example of the latter type in the Moodle online teaching platform: elements from the table are clipped at the right; when analyzing the page's source code, it turns out that *four* more columns should actually be visible and are simply inaccessible, no matter how wide the parent window is resized. This



AIR CANADA enregistrement aircanada.com

Note importante

Veuillez noter que l'enregistrement et l'acceptation des bagages pour les vols intra-Canada se terminent **45 minutes** avant le départ – à l'exception des vols au départ de l'aéroport du centre-ville de Toronto (YTZ), où l'enregistrement et le dépôt des bagages peuvent se faire au moins 20 minutes avant le départ.

Bienvenue

Les zones requises sont identifiées par un * et doivent obligatoirement être remplies.

* Prénom * Nom de famille

* Ville de départ Plus d'informatior

Pays ou **Ville** ou **Aéroport**

* Sélectionner **l'un** des éléments ci-dessous à des fins d'identification

○ Numéro Aéroplan
● Numéro de réservation

Vos renseignements personnels sont chiffrés de façon sûre pour leur transmission entre votre ordinateur et Air Canada.

Utiliser enregistrement aircanada.com:

- pour les vols partant de toute ville au Canada et les villes internationales choisies (avec billet électronique ou papier)
- pour les vols partant des villes choisies des États-Unis (avec billet électronique seulement)
- **pour annuler votre enregistrement**

Restrictions:
Dans les 24 heures précédant le départ; et

- au moins 45 minutes avant le départ pour les vols intra-Canada
- au moins 1 heure avant le départ pour les vols entre le Canada et les États-Unis
- au moins 1 heure avant le départ pour les vols entre le Canada et les autres pays
- si vous avez une correspondance, dans les 24 heures précédant le départ de votre **dernier** vol de correspondance

(a) Air Canada

Paris to London

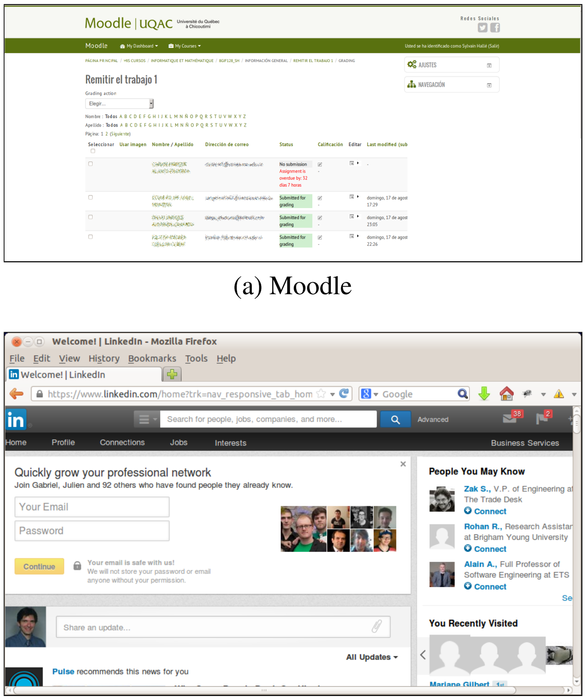
London is only 2 hours and 15 minutes away from Paris

Paris à London

London is only 2 hours and 15 minutes away from Paris

(b) RailEurope

Figure 1.4: Examples of overlapping portions of a page (a) Layout problem caused by insufficient document width; (b) caused by longer text in a French version of the site (RailEurope)



Quickly grow your professional network
Join Gabriel, Julien and 92 others who have found people they already know.

Your email is safe with us!
We will not store your password or email anyone without your permission.

Zak S., V.P. of Engineering at The Trade Desk

Rohan R., Research Assistant at Brigham Young University

Alain A., Full Professor of Software Engineering at ETS

Pulse recommends this news for you

(a) Moodle

(b) LinkedIn

Figure 1.5: Page contents being concealed from view, due to their extending beyond their parent container's dimensions.

makes the page more or less unusable.

A particular case is when the container is the whole browser's window. We have witnessed examples of web applications where, when the window is resized below a certain threshold, elements become inaccessible. This can be shown (or rather not) in Figure 1.5b: the top menu content disappears when the window is resized. The browser does provide a horizontal scrollbar, but this only scrolls the bottom part of the page —not the top menu. As a consequence, some buttons from the site's main menu can no longer be clicked.

We place in that category a bug we witnessed using Facebook's instant messaging window. When the parent window is resized below a specific width, the textbox allowing a user to enter a new message suddenly disappears from the interface, as is shown in Figure 1.6. All other elements of the window remain identical, yet it becomes impossible to type a message.

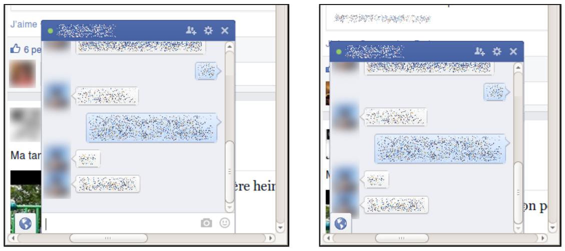
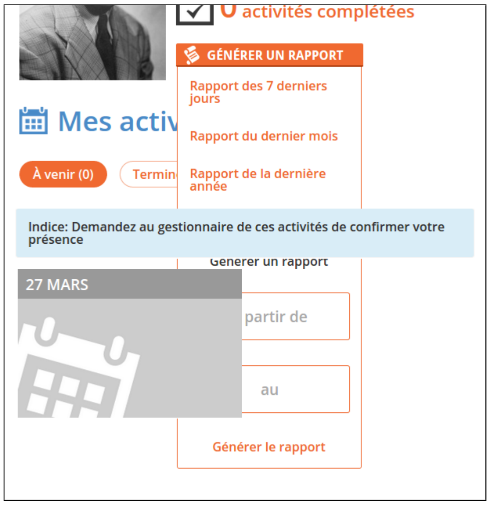


Figure 1.6: The "Facebook Bug". The textbox allowing a user to type an instant message (left) suddenly disappears (right).



Indice: Demandez au gestionnaire de ces activités de confirmer votre présence

Figure 1.7: An element incorrectly placed under another.

1.1.4 INCORRECTLY STACKED ELEMENTS

This problem happens when an element that should be rendered on top of another is actually placed under it, as is shown in Figure 1.7. In this example, clicking on the orange button at the top of the picture is supposed to bring up a drop-down menu; however, the content of this menu appears under the rest of the page's content, rather than on top of it, making some of its elements unusable.

As it turns out, it is hard to predict how the CSS standard decides on the stacking of elements in a page; this is a common source of errors. Most problems of this nature can be corrected by properly assigning the `z-index` property of the faulty element.

1.2 NON-LAYOUT BUGS

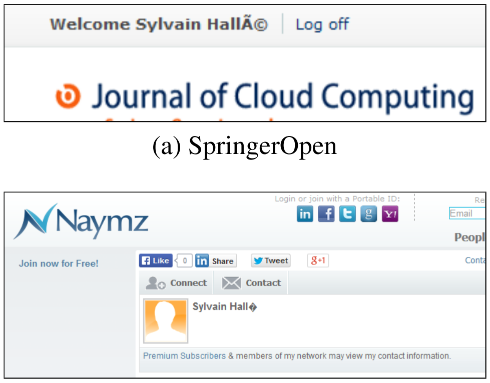
While not being layout problems *per se*, several other bugs can be detected by analyzing the contents and layout of a page. We shall see in the following pages that some of them even relate to the expected *behaviour* or functionality of the application, yet can still be detected through rules expressed on a single static snapshot of the application's viewport.

1.2.1 MOJIBAKE AND ENCODING PROBLEMS

Several sites and applications incorrectly handle string data outside 7-bit ASCII. Various character encodings, such as Cp1252 or UTF-8, can be used to represent so-called "accented" or "foreign" characters. However, the same character may be represented by a different binary value according to the encoding scheme being used; worse, some encodings, such as UTF-8, may use multiple bytes to represent a single character.

Problems arise when a system does not properly interpret the contents of a character string, believing it to be in some encoding while it actually uses antoher. This often results in the display of an incorrect, superfluous, or missing character; this phenomenon is called *mojibake*.[2] For example, the UTF-8 encoding of the character "é", when interpreted as a Cp1252 string, produces "Ã©", as shown in Figure 1.8. Conversely, the Cp1252 character "é" yields "�" when interpreted as UTF-8 (Figure 1.8b). While these characters may be legitimate data in their own right, their presence in an element's content most likely indicates an incorrect handling of string encoding by the application under test.

[2] Japanese term meaning "character transformation"

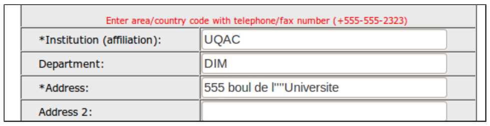

Welcome Sylvain HallÃ©
Log off
Journal of Cloud Computing

(a) SpringerOpen

Naymz
Login or join with a Portable ID:
Email
Peopl
Join now for Free!
Like
0
Share
Tweet
+1
Conta
Connect
Contact
Sylvain Hall�
Premium Subscribers & members of my network may view my contact information.

(b) Naymz

Figure 1.8: An example of Mojibake: (a) UTF-8 data displayed as Cp1252 (SpringerOpen); (b) Cp1252 data displayed as UTF-8.



Enter area/country code with telephone/fax number (+555-555-2323)

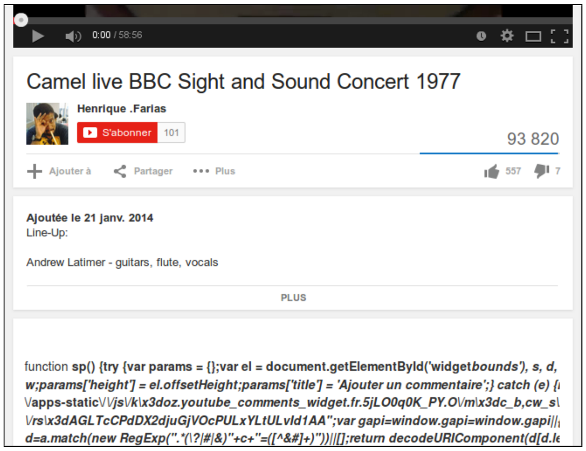
*Institution (affiliation):	UQAC
Department:	DIM
*Address:	555 boul de l""Universite
Address 2:	

Figure 1.9: Data fetched from a database is incorrectly escaped in IEEE PDF eXpress: note the presence of multiple apostrophes.

1.2.2 ESCAPING PROBLEMS

Escaping problems occur when strings with special characters fail to be properly encoded or decoded between two applications. The most frequent manifestation of this problem is when reading and writing character strings to a database. Some characters, such as the apostrophe, require to be doubled so as not to be confused with a string delimiter. Problems occur when a system fails to replace double apostrophes by single ones when displaying that data in a form. When saving the form contents back to the database, each apostrophe will again be doubled, resulting in *quadruple* apostrophes when reloading the page next time, and so on; this is shown in Figure 1.9. Hence looking for multiple apostrophes in a form can be used to detect incorrect escaping inside the source code.

A problem of a similar nature occurs when special characters from the page's source code



Camel live BBC Sight and Sound Concert 1977

Henrique .Farias

Ajoutée le 21 janv. 2014
Line-Up:

Andrew Latimer - guitars, flute, vocals

```
function sp() {try {var params = {};var el = document.getElementById('widgetbounds'), s, d,
w;params['height'] = el.offsetHeight;params['title'] = 'Ajouter un commentaire';} catch (e) {
\/apps-static\/\/js\/k\x3doz.youtube_comments_widget.fr.5jLO0q0K_PY.O\/m\x3dc_b,cw_s\
\/rs\x3dAGLTcCPdDX2djuGjVOcPULxYLtULvld1AA";var gapi=window.gapi=window.gapi||
d=a.match(new RegExp(".*(\?|#|&)"+c+"=([^&#]+)"))||[];return decodeURIComponent(d[d.le
```

Figure 1.10: JavaScript code is being displayed, rather than interpreted, on YouTube.

(such as HTML or JavaScript) are wrongly escaped. For example, a sequence like `<p>` may be transformed into `&lt;p&gt;`, resulting in the literal string `<p>` being displayed as text, rather than being interpreted as a paragraph delimiter. We have witnessed examples of this problem on EasyChair (which shows raw HTML code) and YouTube (which displays raw JavaScript code, as Figure 1.10 shows). Looking for escaping problems of this kind can prove especially important, as it can reveal a vulnerability that a malicious user could exploit.

1.3 RESPONSIVE WEB DESIGN BUGS

A few years ago, access to websites was conditioned by assumptions about the size of the device's screen. Desktop computers were designed to access websites with minimal window sizes, so if the size of the viewport was smaller than the supported size, the site would appear broken. Nowadays, an alternative approach is needed for proper site operation in a range of different viewport appliances and sizes.

StatCounter statistics dating from October 2017 show that the percentage (56.7%) of Internet users in the world via mobile devices and tablets is higher than the percentage (43.3%) of

Internet users via desktop computers (StatCounter, 2017). Thereby, site design must now take into account any device category depending on the size of the screen.

However, the rapid change of device properties could not be followed by web developers. They continued making assumptions from the client's request. The request for a resource through a browser was followed by a user agent string to identify the type of browser used. Reading the user agent string on the server side causes the release of two versions: a mobile version designed for a maximum width and a desktop version designed for minimum width.

Although the approach has many shortcomings, there was unfortunately no better technique. Among its defects, the fact that it does not fit with new devices entering the market, such as tablets that are somewhere in the middle of mobiles and laptops in size, brings the need for another special version for the site. In addition, other versions of the site must be developed in order to satisfy all user devices.

On the other hand, the desktop site assumes that users have large screens (which is usually true) and therefore a large browser navigation window (browser viewport) for the simple reason that the screen is large, something that is not always true. Users may want to have more than one browser window displayed at the same time on the screen, which may invalidate assumptions about the available layout size for the site. Clearly, a better approach was needed. Developers have been following the emergence of CSS media queries that allow conditional style statements by media properties such as window size, see Listing 1.1 for an example. Adapting a site for a specific window size at runtime has become possible. The approach of having a single site respond to different multimedia properties (mainly the size of the window) at run time to enhance the user experience is referred as Responsive Web Design (RWD). The maintenance of several versions of a site is then no longer necessary.

Due to the somewhat complex relationship between HTML, CSS and Java Script and the

```
@media (max-width : 420px) {
    body {
      background-color : white ;
    }
}
@media (min-width : 421px) {
    body {
background-color : blue ;
    }
}
```

Listing 1.1: CSS code with conditions on the device's width

multiplication of the devices in the market nowadays, the layout of web applications tends to be harder to properly specify in contrast with traditional desktop applications. The same document can be shown in a variety of sizes, resolutions, browsers and even devices, making the presence of so-called layout "bugs" all the more prevalent. Such problems can range from relatively mundane quirks from elements overlapping to viewport protrusions.

Walsh et al. (2017) describe five types of bugs in RWD websites, which we shall now describe. Visual examples of these bugs are in Section 4.2.2.

1.3.1 ELEMENT COLLISION

When the viewport of a responsively designed page becomes narrower, one design strategy to account for the loss of width is to move horizontally-aligned page elements closer together. As the viewport contracts, however, elements may collide into one another, causing their contents to overlap. This can result in unintended effects such as overlaid text or images, or hidden content or functional elements, thus harming page usability.

1.3.2 ELEMENT PROTRUSION

When implementing a responsive design, one concern is ensuring that, as the viewport becomes narrower, HTML elements also adapt in size so that they are still large enough to contain their contents. When elements do not resize correctly, their contents may no longer “fit” and consequently protrude into surrounding parts of the page.

1.3.3 VIEWPORT PROTRUSION

As viewport space is squeezed, elements may not only start to overflow their containers, but also start to protrude out of the page’s root presentational HTML element (i.e., the body tag), thus appearing outside of the horizontally viewable portion of the page.

1.3.4 SMALL-RANGE LAYOUTS

Responsively designed web sites tend to use many CSS rules, which are activated and deactivated by different media queries. More than one media query in the CSS rules may evaluate to true at the same time for a given viewport width. For instance, two rules, one activated when the viewport is over 768 pixels wide, and another activated when the viewport is below 1024 pixels, will both be activated in the range 769–1023 pixels.

The logic that governs when a series of rules are “on” or “off” for a given sets of elements and viewport sizes can quickly become complex, to the point at which developers can frequently make mistakes that result in CSS rules being applied unintentionally at some viewport widths. A common fault of this type occurs when developers mix the use of the min-width and max-width qualifiers to define changes in layout. For instance, a developer may encode a media query “@media (max-width: 768px) for example”, and another as “@media (min-width:

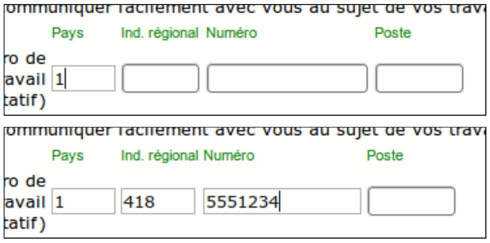
768px)". Since the viewport ranges defined by both of these expressions are inclusive, both will be activated at the 768 pixel viewport width. This clash of media queries can lead to strange layout effects, as two sets of rules will be activated when only one set was intended. These types of failures are difficult for developers to spot, since they occur only in small subranges of the entire range of viewport widths in which the page may be viewed.

1.3.5 WRAPPING ELEMENTS

If a containing element on a web page is not wide enough to contain its children, but is still "tall" enough, or has a flexible height, horizontally-aligned elements contained within it will "wrap" to form an additional, yet undesirable, row of elements and an unwanted presentational effect.

1.4 BEHAVIOURAL BUGS

Contrarily to a traditional web application, a Rich Internet Application (RIA) makes use of emerging modern web technologies such as AJAX (Garrett, 2005), Flash, and Silverlight. Therefore, novel testing problems are added to the existing problems in web testing field. An important characteristic of these applications is that they are inherently *stateful*: their code can store persistent data on the client (using WebStorage, CSS properties, JavaScript variables and objects) and on the server (using cookies, session storage and databases). Moreover, the state of the application is dispersed across a number of elements, and cannot simply be assimilated to the current page's URL (shown in the browser's address bar). Features such as asynchronous communication, delta updates, client side DOM manipulation, event handlers and timing (Arora and Sinha, 2012) make it possible to change the state of the application, without either requiring a full reload of a page, or changing the page's URL.



ommuniquer facilement avec vous au sujet de vos trav

Pays | Ind. régional | Numéro | Poste

ro de
avail 1
atif)

ommuniquer facilement avec vous au sujet de vos trav

Pays | Ind. régional | Numéro | Poste

ro de
avail 1 418 5551234
atif)

Figure 1.11: Moving elements: the position and border of the textboxes changes when writing text (NSERC).

A positive consequence of these features is that such an application can provide a richer (hence its name) user experience; without cookies and JavaScript, mundane operations such as shopping cart manipulations, user sessions (login/logout) and the like would not be possible. However, the presence of a state in the application also introduces the possibility for inconsistencies in the state displayed from one page to the next. These problems are called behavioural bugs, as they are the consequence of multiple pages interacting with each other and coming in a certain order.

1.4.1 MOVING ELEMENTS

Some elements have their position unintentionally changed upon interaction with a user. Figure 1.11 shows an example of such an issue on the web site for Canada's NSERC. Writing text in a previously empty textbox reduces its width by 4 pixels and shifts the remaining textboxes slightly to the left. This problem turned out to be more prevalent than we expected; variations include some elements' style (border or size) changing for no apparent reason in multiple other websites.

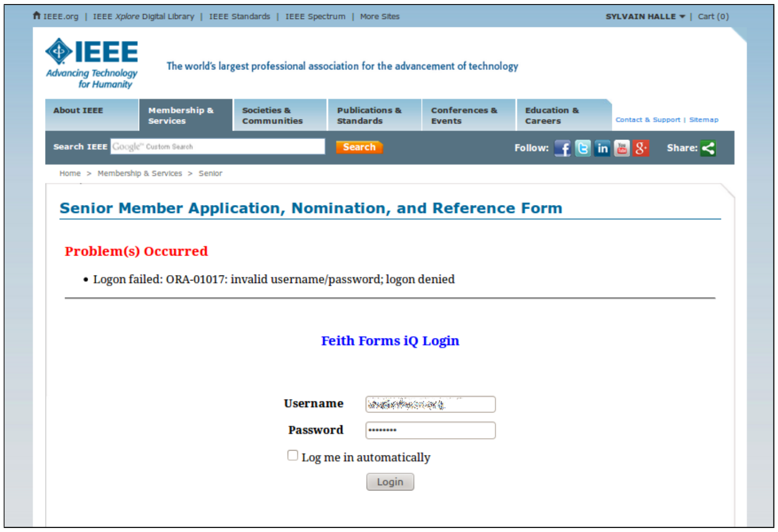
1.4.2 MALFUNCTIONING BUTTONS

Many applications show the user overlaid elements that behave as in-window “pop-ups”. However, in a number of cases, buttons in these windows are inoperative: clicking on them any number of times produces no effect whatsoever. This problem has been witnessed in sites as varied as American Airlines, Dropbox and BitBucket. In some cases the user is effectively stuck in the pop-up or the page that contains the button, and cannot move on without forcing a complete reloading of the document.

1.4.3 LOGIN STATUS CONFUSION

A couple of web sites in our survey exhibit inconsistencies in the status of a page, showing in the same window elements of a page normally reserved to logged in users (such as personalized menus) along with elements reserved to users that are logged out (such as a login form). We have witnessed such behaviour in IEEE’s web site for the Senior Member application form. After some timeout expires, the user is normally required to login again; however, the page presented to the user is the one shown in Figure 1.12. Notice how the user’s name is still present at the top-right of the page, even though it is supposedly logged out (as login credentials are required). Clicking on that name brings the same drop-down menu that is normally accessible only when the user is logged in. The same problem has been witnessed on Air Canada’s reservation site.

It shall be noted how an inherently *stateful* behaviour of the application (namely, the fact of being logged in or not) in this case reduces to the verification of a stateless property on the page’s contents.


IEEE.org | IEEE Xplore Digital Library | IEEE Standards | IEEE Spectrum | More Sites
SYLVAIN HALLE | Cart (0)
IEEE
Advancing Technology for Humanity
The world's largest professional association for the advancement of technology
About IEEE
Membership & Services
Societies & Communities
Publications & Standards
Conferences & Events
Education & Careers
Contact & Support | Sitemap
Search IEEE
Google Custom Search
Search
Follow:
Share:
Home > Membership & Services > Senior

Senior Member Application, Nomination, and Reference Form

Problem(s) Occurred

- Logon failed: ORA-01017: invalid username/password; logon denied

Feith Forms iQ Login

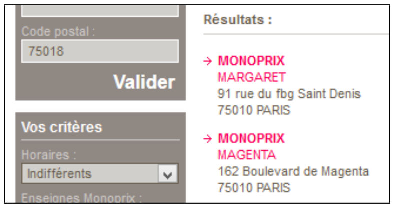
Username [redacted]

Password ••••••••

□ Log me in automatically

Login

Figure 1.12: Login/logout confusion in a web page: contents for a logged in user (top-right of the page) coexist with a login form reserved to users that are not already logged in.



75018

Indifférents

→ MONOPRIX
MARGARET
91 rue du fbg Saint Denis
75010 PARIS

→ MONOPRIX
MAGENTA
162 Boulevard de Magenta
75010 PARIS

Figure 1.13: Inconsistent search results: the user types a postal code in a search box (top left); the results that appear in the next page all have a different postal code (right).

1.4.4 INCONSISTENT SEARCH RESULTS

Another behavioural problem we witnessed is the inconsistency between a query made by a user, and the results of that query displayed by the application. Figure 1.13 shows an example of this problem on the web site for the Monoprix grocery stores. When searching for nearby stores, a user can type a postal code in a form field and then click on “Validate”. However, the search results displayed in the next page almost never show stores with the desired postal code. As with all bugs in the present category, this can only be observed by correlating multiple elements in two different states of the page: the text in the textbox, a click on a specific button, followed by the text in the list items that appear in the resulting page.

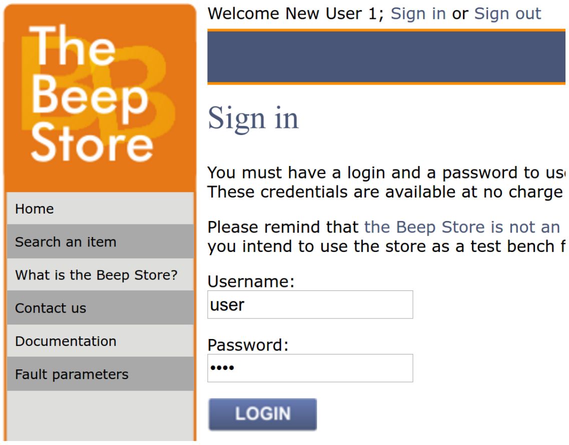
Our survey revealed a number of other oddities we do not describe here for lack of space: double vertical scrollbars, textboxes where text can by typed in but not deleted, buttons outside a window that is not scrollable, flickering tooltips, etc.

In addition to the bugs described above, we shall also include bugs taken from the *Beep Store* (Hallé and Villemaire, 2012), a demo RIA application developped for a research paper where it is possible to toggle behavioural bugs directly in the application. It is a stand-alone, client-server web application, implemented in PHP and JavaScript, that allows registered users to browse a fictional collection of books and music, and to manage a virtual shopping cart made of these elements. This application, whose features were extracted from an exhaustive study of real-world web applications, is an RIA in the pure sense of the term: user interactions are completely asynchronous, never require the reloading of the page, depend on past user actions, and consist of a single document whose various parts are shown or hidden depending on the current state of the application.

We detail here some of the bugs that will be used to test our application in Section 4.1.

1.4.5 MULTIPLE LOGIN

One of the bugs that can be toggled in the Beep Store lets the user land in the login page while being already logged in. This is detected by the fact that the “Sign in” link appears in the top action bar even after the user has successfully logged in, as is shown in Figure 1.14. Obviously, a well-built application would not provide a login button after a user is already logged in; note how this property is stateful, in the sense that the valid state of a page depends on the sequence of past actions made by the user (in this case, the fact that a successful login has occurred).



Welcome New User 1; Sign in or Sign out

Sign in

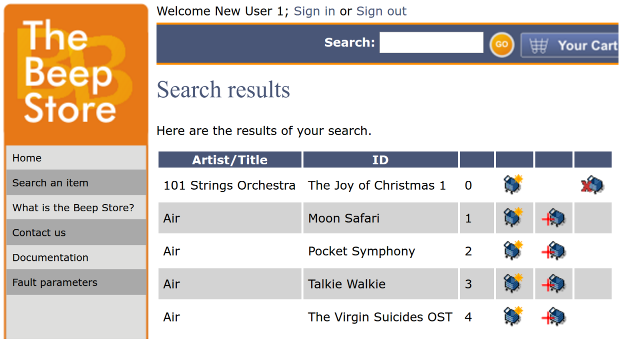
You must have a login and a password to us
These credentials are available at no charge

Please remind that the Beep Store is not an
you intend to use the store as a test bench f

Username:
user

Password:
••••

Figure 1.14: The multiple login bug. A user that is already logged in is presented the option to login again. Taken on the *Beep Store* website (Hallé and Villemaire, 2012)



Welcome New User 1; Sign in or Sign out

Search results

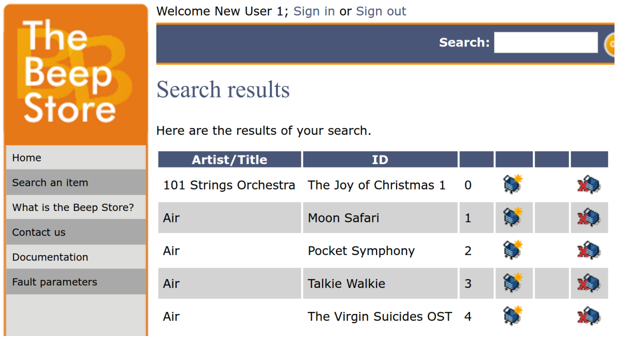
Here are the results of your search.

Artist/Title	ID				
101 Strings Orchestra	The Joy of Christmas 1	0	☆		✗
Air	Moon Safari	1	☆	+	
Air	Pocket Symphony	2	☆	+	
Air	Talkie Walkie	3	☆	+	
Air	The Virgin Suicides OST	4	☆	+	

Figure 1.15: The multiple carts bug. Buttons for creating a cart and adding items to the cart coexist on the same page. Taken on the *Beep Store* website (Hallé and Villemaire, 2012)

1.4.6 MULTIPLE CARTS

Another bug lets the user create multiple shopping carts even after creating a first one. Figure 1.15 shows an example of this bug: a cart has already been created, since the interface shows buttons allowing the user to add items to the cart; yet, buttons to create a new cart are also shown.



The Beep Store

Welcome New User 1; Sign in or Sign out

Search:

- Home
- Search an item
- What is the Beep Store?
- Contact us
- Documentation
- Fault parameters

Search results

Here are the results of your search.

Artist/Title	ID				
101 Strings Orchestra	The Joy of Christmas 1	0	🛒		❌
Air	Moon Safari	1	🛒		❌
Air	Pocket Symphony	2	🛒		❌
Air	Talkie Walkie	3	🛒		❌
Air	The Virgin Suicides OST	4	🛒		❌

Figure 1.16: The remove from nonexistent cart bug. Buttons for removing from the cart and creating a cart coexist on the same page. Taken on the *Beep Store* website (Hallé and Villemaire, 2012)

1.4.7 REMOVE FROM NONEXISTENT CART

This bug is the converse of the previous one: sometimes, the Beep Store lets the user remove an item from his cart before he even creates it. We can see in Figure 1.16 that the buttons for creating a cart are present along with the buttons to remove the items from it.

These bugs are clearly behavioural because they are caused by prior actions by the user or actions that never happened. It should also be noted that, depending on the server's implementation, these bugs do not necessarily trigger error messages in the logs. We can easily imagine a case where an action is dismissed and does not create further problems but the client should have never been presented with the option.

CHAPTER 2

RELATED WORK

In this chapter, we will discuss other works that have been done in the field of web application testing. We will see some traditional testing tools with similar mechanisms as ours, as well as tools that use visual techniques and GUI specifications.

2.1 TESTING TOOLS

Analytics software (Google Analytics,[1] Piwik,[2] etc.) achieve some of the goals of a testing tool. They track HTTP requests and provide a dashboard to analyze basic data: country of origin, duration of a session, etc. However, as a rule, these tools do not handle Ajax, cannot be used as testing tools (i.e. do not evaluate conditions on the contents of the page) and do not report anything about the page's contents, the user's actions or the Ajax requests.

Closer to our goals are web testing software such as Capybara, Selenium WebDriver, Watij or Sahi. These tools provide various languages for describing the tests and writing assertions about the application. WebDriver scripts are written in Java; Capybara is an API[3] over the

[1] `http://www.google.com/analytics`

[2] `http://www.piwik.org`

[3] `http://makandracards.com/makandra/1422-capybara-the-missing-api`

```
WebDriver driver = new FirefoxDriver();
driver.get("http://...");
List<WebElement> items = driver.findElements(By.cssSelector("li"));
int left = -1;
for (WebElement item : items) {
  if (left == -1) {
    left = item.getLocation().getX();
    continue;
  }
  assert(left != item.getLocation().getX());
}
```

Listing 2.1: Code in Java that checks if all elements of a list are left-aligned

Ruby language; Watij uses WebSpec,[4] an API over Java; and Sahi uses SahiScript, an extension of JavaScript.[5] All these languages are *imperative* (i.e. procedural), and aimed at driving an application by performing actions. The testing part reduces to the insertion of assert-like statements throughout the script code.

As an example, Listing 2.1 is a snippet of Java code for Selenium WebDriver to check that all elements satisfying a CSS selector are left-aligned:

We can see how this code (which does not provide any explanation or counter-example when the assertion fails —this would require additional code) has a relatively low readability especially to those not familiar with the language. Barring syntactical differences, the other testing tools mentioned above produce expressions of a similar nature.

There is currently no tool that provides a *declarative* language, observes the contents of each page while an external agent (user, external driver) manipulates the application, and detects when the conditions for a given scenario are present and checks whether the conclusion occurs. This also means these tools do not work well with other software, such as automated crawlers (Crawljax (Mesbah et al., 2012), WebMate (Dallmeier et al., 2013) or WebMole (Hallé et al.,

[4] http://watij.com/webspec-api/

[5] http://sahi.co.in

2014)) or even other web testing software, that are responsible for driving the application or act as monitors that simply “look over the user’s shoulder” while the application is being used.

Moreover, all of the aforementioned tools (with the exception of Sahi) require some browser-specific plugin to operate, and as a consequence only support a handful of browsers —typically the “Big Five” (Firefox, Safari, IE, Opera and Chrome). Tools based on such solutions claim to work on “most browsers”. However, Table 2.1 reveals that the market share of browsers other than these five amounts to 20%, and rises to 63% when considering non-desktop devices (tablets, consoles and mobiles). Table 2.2 shows the market share of existing web testing tools, based on the market share of the browsers they support. Apart from Sahi, these testing tools do not reach more than three quarters of the market, and for some as few as 25% for non-desktop devices. Therefore the claim that “most users” are supported by them is relatively unsubstantiated.

On the other hand, ReDeCheck (Walsh et al., 2015) is an RWD testing tool. It is inspired from the alignment graph used in X-PERT, a concept that was proposed and developed by Choudhary et al. (2013b). ReDeCheck focuses specifically on layout bugs caused by responsive designs; it uses a Responsive Layout Graph (RLG), which takes into consideration of the alignment of the elements of the web page, visibility changes and other aspects of the page when the viewport width varies. As such, ReDeCheck can only verify a fixed set of predefined layout problems, and does not provide a general-purpose language for expressing assertions as Cornipickle does.

[6] HP QuickTest Professional

Browser	Share (all)	Share (non-desktop)
Chrome	36.44%	11.46%
IE	17.34%	0.14%
Firefox	14.86%	0.43%
Safari	7.44%	13.38%
Android	7.07%	24.3%
Opera	4.21%	11.11%
iPhone	4.85%	16.66%
UC Browser	2.65%	9.1%
Nokia	1.31%	4.5%
BlackBerry	0.54%	1.84%
NetFront	0.48%	1.63%
iPod Touch	0.27%	0.94%
IEMobile	0.51%	1.76%
360 Safe Browser	0.28%	0%
Yandex Browser	0.26%	0%
Maxthon	0.19%	0%
Silk	0.18%	0.62%
Sogou Explorer	0.18%	0%
Sony PS3	0.07%	0.23%
Unknown	0.13%	0.43%
Dolfin	0.08%	0.28%
Chromium	0.06%	0%
QQ Browser	0.1%	0.1%
Samsung	0.03%	0.09%
Puffin	0.06%	0.2%
Openwave	0.05%	0.17%
Pale Moon	0.02%	0%
Phantom	0.03%	0.07%
Netscape	0%	0%
Jasmine	0.02%	0.06%
RockMelt	0.01%	0%
TheWorld	0.02%	0%
Obigo	0.02%	0.07%
Tencent Traveler	0.02%	0%
SeaMonkey	0.01%	0%
Iron	0.02%	0%
CoRom+	0.04%	0%
SonyEricsson	0.02%	0.07%
Nintendo 3DS	0%	0.02%
Iceweasel	0.01%	0%
Mozilla	0.02%	0%
Other	0.13%	0.14%

Table 2.1: Market share for browsers as of February 2014, taken from `http://gs.statcounter.com`

Tool	IE	Firefox	Opera	Safari	Chrome	Share (all)	Share (non-desktop)
Capybara	Yes	Yes	No	Yes	Yes	76%	25%
Crawljax	Yes	Yes	No	Yes	Yes	76%	25%
HP QTP[6]	Yes	Yes	No	No	No	32%	1%
iMacros	No	Yes	No	No	No	15%	0%
QF-Test	Yes	Yes	No	No	No	32%	1%
Ranorex Studio	Yes	Yes	No	Yes	Yes	76%	25%
Sahi	Yes	Yes	Yes	Yes	Yes	80%	37%
TestComplete	Yes	Yes	No	No	Yes	69%	12%
Tosca Testsuite	Yes	Yes	No	No	No	32%	1%
Test Studio	Yes	Yes	No	Yes	No	40%	14%
Watij	Yes	Yes	No	Yes	No	40%	14%
WatIN	Yes	No	No	No	No	17%	0%
Watir	Yes	Yes	Yes	No	Yes	73%	23%
WebDriver	Yes	Yes	No	Yes	Yes	76%	25%

Table 2.2: Market share for browsers supported by various web testing tools

2.2 VISUAL TECHNIQUES

Some work has also been done on the use of image analysis techniques to identify layout problems (Tamm, 2009); in particular, WebSee (Mahajan and Halfond, 2015) is a tool implemented in Java that leverages several third party libraries to implement some specialized algorithms. It applies techniques from the field of computer vision to analyze the visual representation of web pages to automatically detect and localize presentation failures. WebSee identifies presentation failures, and then determines the elements in the HTML source of the page that could be responsible for the observed failures by comparing the visual representation of the rendered web page under test and its appearance oracle.

To this end, WebSee takes as input the HTML/CSS code of the page to analyze and an oracle (i.e. picture) of the expected rendering of the page. A set of differences between the rendered page and the reference picture is computed, and these differences are then clustered into groups that are likely to represent different underlying faults in the page. A second processing phase attempts to identify the HTML elements corresponding to the difference pixels, which are then ordered by some priority metric and output to the user. Applitools Eyes (Applitools, 2016) is a commercial tool following a similar principle; it uses pure image segmentation of web pages and a pixel-by-pixel visual comparison.

However, this approach is geared towards the detection of static, overlapping- or overflow-type bugs in a document, and currently does not support the checking of temporal patterns across multiple snapshots of the same page. Moreover, the approach generates lots of false positives if the rendered page contains text that is slightly different from the reference picture. This is the case when, for example, the browser's window has different dimensions, and causes some text to flow differently (but not necessarily incorrectly) with respect to the picture. To address this problem, the user has to manually define, for each oracle, so-called *dynamic regions* that should basically be ignored by the system when analyzing the page.

2.3 DECLARATIVE GUI SPECIFICATIONS

Declarative specifications of user interfaces has been the subject of research in the past. Early attempts include the MASTERMIND system, which uses a modelling language based on CORBA (Szekely et al., 1995); other approaches include the Auckland Layout Model (Lutteroth and Weber, 2008), Adobe's "Adam and Eve" (Parent et al., 2007) and property models (Järvi et al., 2008). We shall however discard these, and a lot of other GUI markup languages (QML, XAML), which are aimed at *generating* interfaces based on constraints, not to be used as assertions to be tested; moreover, they focus principally on solving linear constraints relative to the size and placement of elements inside a form.

To show why GUI-generating languages are not appropriate, let us consider the simple example of CSS. As was discussed earlier, CSS expresses what elements *should* look like, but not what they *should not* look like. Consider the following styling declarations, in the case where some element div2 is nested inside div1:

```
#div1 { ... width: 200px; }
#div2 { ... width: 300px; }
```

It is not possible to state that the contents of div1 should never be wider than its own declared width. If this is the case, the browser can either expand the element's box (disregarding its declaration), display elements outside its container (a problem we discussed extensively in Section 1.1), or cropping from that content anything that is too wide. The only solution is for the designer of the CSS document to make sure that the declared width of all elements that will be contained in div1 is smaller than 200 pixels. We will see later that, with our solution, a property expressed in our language could easily enforce at runtime that the width of div1 is always of 200 pixels.

One could argue that this can also be checked by statically analyzing the HTML document along with the CSS file (although no related work seems to address this question). There are cases, however, where this is not possible. If we delete the declaration for div2 in the example above, but that div2 contains an image that is 300 pixels wide, then there is no way to discover that the desired property is not fulfilled by simply looking at the CSS.

As a final example, consider the following code, which states that some menu's elements should be placed vertically and fixes their size:

```
#menu li {
  float: left;
  width: 200px;
  height: 50px;
}
```

Nothing allows the designer to specify that all elements should always be on the same line (i.e. have the same top position even if the value of top is defined). This will not be the case if the containing window is resized small enough for elements to float under. In such a case, one has to actually render the page inside a browser, with its actual dimensions, to discover the presence of the problem.

2.4 CRAWLERS

Crawlers are an integral part of web search engines and are dedicated to the collection and indexing of Web documents. They were originally developed for the purpose of archiving, retrieving information (find e-mail addresses or product pages for example). However, it was soon discovered that crawlers could be used for other purposes: in particular, a crawler can be seen as a state-space exploration tool, and as such, be used to perform automated testing.

A traditional crawl process starts from seed URLs. Web pages corresponding to these URLs are downloaded and the hyperlinks present on them are extracted and added to a set of URLs to visit, also called the crawl frontier. Since the number of URLs that populate the crawl frontier increases very rapidly, a criterion for prioritizing the downloading of certain pages is generally applied. In turn, the best ranked URLs within the crawl frontier are downloaded and new links are extracted. This operation is repeated until all accessible links are visited (Mirtaheri et al., 2013a; Olston and Najork, 2010).

Some basic features of traditional web applications are different in RIAs (Rich Internet Applications), which makes crawling RIAs more difficult than crawling traditional web applications. In RIA crawling, the crawl order respects the sequence of possible pages as if a user were using it. As we have seen, unlike in traditional web applications, a URL does not uniquely identify the state of the application, and hence cannot simply be requested from the server at any moment.

In an application with cart management features like the Beep Store, it would be possible for a traditional crawler to find behavioural bugs where there is none; the order of visit is crucial. For instance, in a scenario where the user can create a cart, delete a cart, add an item to a cart and edit a cart to change the quantity of an item, a bug that lets the user edit an item in his cart while not having a cart could be discovered. After creating a cart, adding an item to it and then deleting the cart, the traditional crawler's set of URLs to visit contains the URL to edit the item. Then, when trying to reach this URL, a bug occurs because the cart has been deleted. However, it can be a false positive because this sequence of actions is most likely not possible by a user. A traditional crawler is therefore not adequate to search for behavioural bugs.

In a typical RIA web crawler, the page associated with a seed URL is extracted and loaded into a virtual browser. A module is required to check if this is the first time the page constructed is encountered. An event extractor retrieves the JavaScript events from the page; one of these events is selected and executed on the page. The resulting page is then collected, and the process continues until all discovered actions have been exhausted (Choudhary et al., 2012). On the basis of this model, different exploration strategies (such as depth-first search, Greedy, Model-Based, and Component-Based) have been suggested (Mirtaheri et al., 2013b; Benjamin et al., 2011; Dincturk et al., 2012; Choudhary et al., 2013a; Dincturk, 2013).

Some tools have already been designed to test RIAs. For example, WebMate (Dallmeier et al., 2013) can extract the state tree of a web application. This tree is then compared with other browsers' state trees to find layout differences. However, it focuses on cross-browser compatibility and does not seem to support external user-defined tests.

WebMole is another automated crawler that generalizes existing approaches. It eliminates arbitrary backtracking by intercepting HTTP requests and make page jumps (Hallé et al., 2014). However, WebMole's goal is simply to extract the navigation graph of an application;

it does not allow a user to write test oracles to be evaluated while the application is being explored.

On the other hand, Crawljax (Mesbah et al., 2012) uses a depth-first strategy to crawling and produces a finite state machine of the application's behaviour. It is possible through its plugin architecture to test every state while they are being visited. However, the tests in question are left to be written by the user as pure Java code; this makes it hard, for reasons mentioned earlier, to easily write stateful test oracles.

2.5 MANUALLY TESTING RESPONSIVE WEB DESIGN

It is always a smart decision to test the design of a website on a variety of devices. But this classic method takes a lot of time, and because of the variety of devices available today, the developer may not have access to all of these devices. Apart from ReDeCheck seen in Section 2.1, the only tools that exist to ease this process display a page in a custom window of variable size using a web browser. However, most of these tools do not provide any information other than the rendering of the pages on different sizes which forces the task of evaluating the responsiveness on the developer. Not only identifying layout failures with a human eye is hard and time-consuming, but it is also subject to the developer's biases. We present here some of these tools:

With smart search and quick review features, Websiteresponsivetest (CSSChopper, 2017) supports all major browsers to provide the exact preview of the website on any specific device. It requires entering the URL of a web page to quickly assess its responsiveness. By providing accurate results in a matter of a few seconds, the tool saves a lot of time to the tester. User-friendliness and browser-compatibility are other added features which make it better than the other tools available.

Respondr (Respondr, 2014) allows checking responsiveness by simply entering the URL of a website. Additionally, the device for which the website or web page is tested can also be chosen from a list. Once the selection is done, a simple click on “G” is needed to receive a full analysis of website/web page for that specific device. The page can be easily previewed at an appropriate width.

Screenfly (Screenfly, 2017) is a multi-device compatibility testing tool which allows previewing web pages as they appear on different devices. The most popular ones include tablet and other smartphone devices like the Galaxy tab, Apple iPad, and Motorola Xoom. Additionally, it supports different screen sizes and resolutions. The site automatically detects if the entered URL has a mobile site and redirects the user to it. To switch between screen resolutions, all that is needed is to click on the icon of the device type or choose the device that is closest in screen resolution.

ViewPort Resizer (Wassermann, 2012) comes as a browser bookmarklet which can be used with any modern web browser. Dragging the bookmarklet on the Bookmarks bar is needed to get a toolbar at the top with buttons for different screen resolutions. As a user-friendly browser tool, ViewPort Resizer is completely configurable. It permits the selection of an initial range of screen resolution sizes and building a custom bookmarklet.

Responsinator (Pugsley and Hovey, 2013) helps site owners get an idea of how their site will work on the most popular devices. Just by typing the website URL, the site will quickly show on screens of various sizes.

ResponsivePX’s (Sharp, 2011) process involves entering the URL of the site and uses buttons to adjust the width and height of the viewport to find the exact breakpoint width in pixels. Extremely simple to use and enhanced functionality, this web design tool helps designers create usable and responsive sites.

Finally, FROONT (FROONT, 2012) makes responsive web design testing accessible to all kinds of designers without requiring any coding skills. Designs can be created in the browser with this intuitive drag and drop tool. By testing each URL specifically, it tests the designs on real devices right away.

CHAPTER 3

CORNIPICKLE: A WEB APPLICATION TESTING TOOL

One might be inclined to think that a static analysis of a page's HTML content and CSS declarations should be sufficient to detect all of the aforementioned problems. Yet, one must realize that CSS is not a language that can express prescriptive properties about the layout of a document. CSS declarations are a set of instructions processed by a layout engine, which can (and most often do) contradict or override each other. As we discussed in Section 2.3, giving CSS a prescriptive expressiveness would imply the possibility to specify how an element should *not* look, or that a particular style declaration may *not* be overridden by some other constructs that CSS does not provide. Aside from an absolute positioning of all elements in a page (thereby completely bypassing the layout engine), there is no formal guarantee that two elements with the same `left` property will always be vertically aligned. Moreover, one cannot use CSS to express constraints about the structure or content of a document (such as: the `#menu` element should not contain mojibake). Finally, some of the bugs described earlier involve the comparison of the content of a document at multiple moments in time, a thing that CSS obviously is not designed for.

3.1 A DECLARATIVE LAYOUT LANGUAGE

Therefore, in order to express prescriptive properties about the layout and contents of a page, a language on top of CSS is required. This language shall allow a user to elicit declarative properties on the *computed* styles of elements, and deal with events, regardless of whatever CSS declarations or client-side code that may have been declared. In this section, we shall now describe *Cornipickle*, a declarative language for expressing properties about a document's Document Object Model and CSS properties.

The language borrows features from a variety of other formalisms and tools. Its name is a nod to Cucumber (Wynne and Hellesoy, 2012), a general-purpose testing tool that uses regular expression matching to interpret structured-English constructs from an input file and convert them into executable test steps, yielding very human-readable test documents. Although Cornipickle is not a regex-based converter between text files and script commands, its grammar strives for the same kind of readability. Similarly, constructs from first-order and linear temporal logic, such as quantifiers and temporal operators, allow a user to specify complex relationships on various document elements at multiple moments in time, a feature that, as we saw in Section 2.1, is absent from many scripting languages.

The reader may wonder about our insistence on human readability. A similar behaviour could have easily been achieved using more formal (e.g. first-order logic) syntax or script-like constructs. This design choice stems from our past experience at using formal specification languages in a real-world testing context. It is fair to say that any solution involving some amount of formalism elicits from many practitioners a reaction ranging from pure disgust to sheer fright —and limits as much their will to use a tool for real. Moreover, a human-readable specification can also be understood and interacted upon by non-technical people, such as graphic designers, much more easily than logic or code.

To improve readability, the grammar allows various words to be inserted within the various constructs; these words have no effect on the parsing and interpretation of the expression.

3.1.1 GRAMMAR

In Cornipickle, properties are expressed as assertions on the content and attributes of some *snapshot* of a page taken at a given moment. The precise way such snapshots are taken from a given web application will be detailed in the next section. In the following, we first describe the language's grammar, summarized in Table 3.1, by illustrating its various constructs.

Selecting Elements

The main unit in Cornipickle are page *elements*, which are selected in order to express properties about them. These properties may apply to all selected elements, or at least one. Therefore, element selection is done through classical first-order quantification, using the English-like syntax `For each $x in S` or `There exists $x in S`. In these expressions, `S` denotes either a CSS selector,[1] or another set previously defined by the user. CSS selectors are expressed using jQuery's syntax `$(...)`. A special selector called `CDATA` can be used to refer to the text content of leaf nodes in a DOM tree (i.e. the actual clear-text snippets that compose the page).

If `$x` is a variable quantified using the mechanism described above, one may access to the DOM or CSS attributes of this element using the syntax `$x's property`. For example, the width of the element would be written as `$x's width`. Element attributes (which are either character strings or numbers) can then be compared using connectives such as

[1] A CSS selector is a path expression that defines what elements of a document are the subject of a given set of rules. These expressions are defined by a regular grammar, as stipulated in the CSS standard.

⟨*S*⟩ ::= ⟨*predicate*⟩ | ⟨*def-set*⟩ | ⟨*statement*⟩ | ⟨*context*⟩

Cornipickle statements

⟨*statement*⟩ ::= ⟨*foreach*⟩ | ⟨*exists*⟩ | ⟨*binary-stmt*⟩ | ⟨*negation*⟩
| ⟨*temporal-stmt*⟩ | ⟨*userdef-stmt*⟩ | ⟨*let*⟩ | ⟨*when*⟩

⟨*binary-stmt*⟩ ::= ⟨*equality*⟩ | ⟨*gt*⟩ | ⟨*lt*⟩ | ⟨*regex-match*⟩ | ⟨*and*⟩ | ⟨*or*⟩ | ⟨*implies*⟩

⟨*temporal-stmt*⟩ ::= ⟨*globally*⟩ | ⟨*eventually*⟩ | ⟨*never*⟩ | ⟨*next*⟩ | ⟨*next-time*⟩

⟨*context*⟩ := The following rules apply when (⟨*statement*⟩) ;

First-order logic

⟨*foreach*⟩ ::= For each ⟨*var-name*⟩ in ⟨*set-name*⟩ (⟨*statement*⟩)

⟨*exists*⟩ ::= There exists ⟨*var-name*⟩ in ⟨*set-name*⟩ such that (⟨*statement*⟩)

⟨*when*⟩ ::= When ⟨*var-name*⟩ is now ⟨*var-name*⟩ (⟨*statement*⟩)

⟨*let*⟩ ::= Let ⟨*var-name*⟩ be ⟨*property-or-const*⟩ (⟨*statement*⟩)

⟨*and*⟩ ::= (⟨*statement*⟩) And (⟨*statement*⟩)

⟨*or*⟩ ::= (⟨*statement*⟩) Or (⟨*statement*⟩)

⟨*implies*⟩ ::= If (⟨*statement*⟩) Then (⟨*statement*⟩)

⟨*negation*⟩ ::= Not (⟨*statement*⟩)

Temporal statements

⟨*globally*⟩ ::= Always (⟨*statement*⟩)

⟨*never*⟩ ::= Never (⟨*statement*⟩)

⟨*next*⟩ ::= Next (⟨*statement*⟩)

⟨*eventually*⟩ ::= Eventually (⟨*statement*⟩)

⟨*next-time*⟩ ::= The next time (⟨*statement*⟩) Then (⟨*statement*⟩)

Table 3.1: The BNF grammar for Cornipickle (Part I)

`is greater than` or `equals`; regular expression matching is made through the `match` predicate, and substring inclusion is asserted with `contains`. Basic assertions about elements can also be combined using the classical Boolean connectives `and`, `or`, `if...then`, `not`.

Events and Temporal Operators

In Cornipickle, user-triggered events such as key presses and mouse clicks are represented as attributes on the element that is the event's target. For example, an element that has been clicked by the user will momentarily possess an `event` attribute with value `click`. Hence asserting that some element `$x` has been clicked can be written as `$x's event is "click"`.

The inclusion of events in the language naturally leads to the notion of *sequences* of document snapshots. Consequently, Cornipickle provides operators borrowed from Linear Temporal Logic (LTL) to express assertions about the evolution of a document's content over time. The `Always` φ construct allows one to assert that whatever φ expresses must be true in every snapshot of the document. Similarly, `Eventually` φ says that φ will be true in some future document snapshot, and `Next` φ asserts it is true in the next snapshot.

A special construct called `The next time` φ `then` ψ asserts that whatever ψ expresses must hold, but only once φ evaluates to true. For example, one can use this construct to express that something must be observed after an element has been clicked. This is but a slight rewriting of LTL's "until" operator.

One particular purpose of temporal operators is to compare the state of the *same* element across multiple snapshots. This can be done in Cornipickle with the construct `When $x is now $y` φ. If `$x` refers to the state of an element captured in some previous snapshot, then `$y` will contain the state of the same element in the current snapshot.

Operators

⟨*equality*⟩ ::= ⟨*property-or-const*⟩ equals ⟨*property-or-const*⟩
| ⟨*property-or-const*⟩ is ⟨*property-or-const*⟩

⟨*gt*⟩ ::= ⟨*property-or-const*⟩ is greater than ⟨*property-or-const*⟩

⟨*lt*⟩ ::= ⟨*property-or-const*⟩ is less than ⟨*property-or-const*⟩

⟨*regex-match*⟩ ::= ⟨*property-or-const*⟩ matches ⟨*property-or-const*⟩

⟨*constant*⟩ ::= ⟨*number*⟩ | ⟨*string*⟩

⟨*property-or-const*⟩ ::= ⟨*elem-property*⟩ | ⟨*constant*⟩

⟨*number*⟩ ::= ˆ\d+

⟨*string*⟩ ::= ˆ"[ˆ"]*"

Auxiliary constructs in statements

⟨*el-or-not*⟩ ::= element | ε

⟨*set-name*⟩ ::= ⟨*css-selector*⟩ | ⟨*userdef-set*⟩ | ⟨*regex-capture*⟩

⟨*userdef-set*⟩ ::= ⟨*string*⟩

⟨*var-name*⟩ ::= ˆ\$[\w\d]+

CSS selector

⟨*css-selector*⟩ ::= $(⟨*multi-css-sel*⟩)

⟨*multi-css-sel*⟩ ::= ⟨*css-sel-contents*⟩ , ⟨*multi-css-sel*⟩ | ⟨*css-sel-contents*⟩

⟨*css-sel-contents*⟩ ::= ⟨*css-sel-part*⟩ > ⟨*css-sel-contents*⟩ | ⟨*css-sel-part*⟩ ⟨*css_sel-contents*⟩ | ⟨*css_sel-part*⟩

⟨*css-sel-part*⟩ ::= ^[\w\u0023\.*\-\$]+

CSS attributes

⟨*css-attribute*⟩ ::= nodeValue | value | height | width | top | left | right | bottom | color | id | text | background | border | event | cornipickleid | display | size | accesskey | checked | disabled | min

Table 3.2: The BNF grammar for Cornipickle (Part II)

User-defined set in extension

⟨*def-set*⟩ ::= A ⟨*def-set-name*⟩ is any of ⟨*def-set-elements*⟩

⟨*def-set-name*⟩ ::= ˆ.*?(?=is)

⟨*def-set-elements*⟩ ::= ⟨*def-set-element*⟩ , ⟨*def-set-elements*⟩ | ⟨*def-set-element*⟩

⟨*def-set-element*⟩ ::= ⟨*constant*⟩

User-defined predicate

⟨*predicate*⟩ ::= We say that ⟨*pred-pattern*⟩ when (⟨*statement*⟩)

⟨*pred-pattern*⟩ ::= ˆ.*?(?=when)

User-defined statements

⟨*userdef-stmt*⟩ ::= empty

Table 3.3: BNF grammar extensions for Cornipickle

All these constructs can be freely mixed. For example, the following property asserts that every list item eventually moves down the page at some moment:

```
For each $x in $(li) (
  Eventually (
    When $x is now $y (
      $y's top is greater than $x's top
))).
```

Extending the Grammar

An important feature of Cornipickle, which contributes to the readability of its specifications, is the possibility for users to extend the basic vocabulary with their own definitions. These new definitions are read by the interpreter, and can then be used freely like any basic language element.

Predicates can be defined with the construct `We say that...when`. The text between `that` and when is interpreted as a literal string that can contain variables. The text after when describes how that expression should be evaluated in terms of the existing vocabulary. For example, one can define the expression “left-aligned” as follows:

```
We say that $x and $y are left-aligned when (
  $x’s left equals $y’s left ).
```

The phrase $x and $y are left-aligned can then be reused (possibly with different variable names) in later assertions. Users can also define sets —either sets of character strings, numbers, or sets of elements from a page, by enumerating them using the A...is any of construct:

```
     A Mojibake is any of "Ã©", "Ã’", "Ã¨".
```

Note that the name of the set need not be a single word; the parser interprets as a name anything between A and is any of.

3.1.2 LAYOUT BUGS REVISITED

Equipped with such a grammar, it is possible to give examples of properties corresponding to some of the bugs we discovered in our survey. For example, with Mojibake the set defined as above, the presence of character encoding issues in a page can be detected with:

```
We say that $x doesn’t contain $y when (
  Not ($x’s text matches $y’s value )).

For each $text in $(CDATA) (
  For each $mojibake in "Mojibake" (
    $text doesn’t contain $mojibake
)).
```

Note how we first add to the grammar the construct `doesn’t contain`, only to improve the readability of the following statement. Similarly, specifying that a specific class of elements should never move can be written as:

```
We say that $x is immobile when (
  Always (
    When $x is now $y (
      ($x’s left equals $y’s left)
      And
      ($x’s top equals $y’s top )
))).

For each $item in $(li) ( $item is immobile ).
```

The intuitiveness of specifications can be further highlighted in this (slightly contrived) last example, which states that at least one list item has the value of some other list item the previous time the user clicked on a button called “Go”:

```
We say that I click on Go when (
  There exists $b in $(button) such that (
   ($b’s text is "Go")
   And
   ($b’s event is "mouseup")
)).

Always (
  If (I click on Go) Then (
    There exists $x in $(.value) such that (
      The next time (I click on Go)
      Then (
        There exists $y in $(.value) such that (
          $x’s text equals $y’s text
))))).
```

As with the Java example shown in Section 2.1, the readability of this specification should be contrasted with the procedural code one would need to write to detect the same issue, which is objectively longer, and arguably much less clear in its intent. For example, Listing 3.1 shows how to detect that bug in jQuery.

```
$(document).mouseUp(function(event) {
  var e = arguments.callee;
  if ($(event.target).text() === "Go") {
    var current_values = [];
    $(".value").each(
    current_values.push($(this).text());
    );
    if (e.lastValues !== undefined) {
      var found = false;
      for (var v in current_values) {
        if ($.inArray(v, e.lastValues)) {
          found = true;
          break;
        }
      }
    }
    if (!found)
      console.log("Error");
    e.lastValues = current_values;
  }
};);
```

Listing 3.1: Javascript code with JQuery detecting that at least one list item has the value of some other list item the previous time the user clicked on a button called "Go"

We make no formal claim regarding the completeness of the language or its expressiveness. However, anecdotal evidence reveals that all the layout bugs in our survey can be expressed by an appropriate expression, suggesting it is well suited for the task at hand.

3.1.3 FORMAL SEMANTICS

We shall now proceed to describe the formal semantics of Cornipickle. The first step involves formalizing the structure, content and style properties of a displayed page.

We first define a set A of *attribute names*. This set includes all attributes from the W3C Document Object Model Level 2 (Hors et al., 2000) and all Cascading Stylesheet (CSS) properties that can be associated with an element. A *DOM node* is a function $\nu : A \to V$, which associates to every attribute name a value taken from some set V. We use the special value “?” to indicate that an attribute is undefined for a given node. We single out a subset $E \subset V$ that denotes element *names*, standing for the actual HTML tag name that represents the element (e.g. `a`, `h1`, `img`, etc.).

We shall denote by N the set of all DOM nodes. The set T of DOM *documents* comprises all trees whose nodes are DOM nodes. In line with the convention adopted by most web browsers, text elements can only appear as leaves and are given the special element name `#TEXT`. We already saw Figure 1.2 that represents one such document. If we let ν refer to the second paragraph of the document’s `body`, we have for example that $\nu(\text{elementName}) =$ “p”, $\nu(\text{style.color}) =$ “red”, and so on. We extend ν to values by defining $\nu(v) = v$ for all $v \in V$.

We let $\kappa : T \times N \rightarrow 2^N$ be the function that, given a document $t \in T$ and a node $\nu \in N$, yields the set $\kappa(t, \nu)$ of all children of ν in T. Similarly, if we let C be the set of all CSS selectors, function $\chi : T \times S \rightarrow 2^N$ will return the set of nodes in t matching some CSS selector $c \in C$. User-triggered events are taken into account by assuming that some elements carry an attribute with the special name “event”, whose value describes the event this element is related to. For example, a user clicking on a button would have that button carry the “event” attribute “click”. In this way, it is possible for a snapshot of a document to relay information about dynamic events occurring in the application.

The semantics of Cornipickle is defined on *traces* of DOM documents; a trace is a finite sequence of elements of T, which we will denote $\bar{t} = t_0, t_1, \ldots, t_k$. Since all expressions involving constructs defined with `We say that` can easily be converted back into expressions that only use constructs from the basic language, it suffices to define the semantics for that basic language. A trace $\bar{t}$ will be said to satisfy some Cornipickle expression φ, noted $\bar{t} \models \varphi$, when its evaluation returns the value true ($\top$). the notation $\bar{t}^i$ indicates the suffix of $\bar{t}$ starting at its i-th event.

$$
\begin{array}{rcl}
\bar{t} \models \nu\texttt{'s}\ a\ \texttt{equals}\ \nu'\texttt{'s}\ a' & \Leftrightarrow & \nu(a) = \nu'(a') \\
\bar{t} \models \nu\texttt{'s}\ a\ \texttt{equals}\ v & \Leftrightarrow & \nu(a) = v \\
\bar{t} \models \texttt{Not}\,\varphi & \Leftrightarrow & \bar{t} \not\models \varphi \\
\bar{t} \models \varphi\ \texttt{And}\ \psi & \Leftrightarrow & \bar{t} \models \varphi \text{ and } \bar{t} \models \psi \\
\bar{t} \models \varphi\ \texttt{Or}\ \psi & \Leftrightarrow & \bar{t} \models \varphi \text{ or } \bar{t} \models \psi \\
\bar{t} \models \texttt{If}\ \varphi\ \texttt{Then}\ \psi & \Leftrightarrow & \bar{t} \not\models \varphi \text{ or } \bar{t} \models \psi \\
\bar{t} \models \texttt{There exists}\ \xi\ \texttt{in}\ \texttt{\$(}c\texttt{)}\ \texttt{such that}\ \varphi & \Leftrightarrow & \bar{t} \models \varphi[\xi/\nu] \text{ for some } \nu \in \chi(\bar{t}_0, c) \\
\bar{t} \models \texttt{For each}\ \xi\ \texttt{in}\ \texttt{\$(}c\texttt{)}\ \varphi & \Leftrightarrow & \bar{t} \models \varphi[\xi/\nu] \text{ for all } \nu \in \chi(\bar{t}_0, c) \\
\bar{t} \models \texttt{Always}\,\varphi & \Leftrightarrow & \bar{t} \models \varphi \text{ and } \bar{t}^1 \models \texttt{Always}\,\varphi \\
\bar{t} \models \texttt{Eventually}\,\varphi & \Leftrightarrow & \bar{t} \models \varphi \text{ or } \bar{t}^1 \models \texttt{Eventually}\,\varphi \\
\bar{t} \models \texttt{Next}\,\varphi & \Leftrightarrow & \bar{t}^1 \models \varphi \\
\bar{t} \models \varphi\ \texttt{Until}\ \psi & \Leftrightarrow & \text{There exists } i \geq 0 \text{ such that} \\
 & & \bar{t}^i \models \psi \text{ and } \bar{t}^j \models \varphi \text{ for } j < i \\
\text{When } \xi \text{ is now } \xi'\ \ \varphi & \Leftrightarrow & \text{There exists } \nu' \in t_0 \text{ such that} \\
 & & \nu(id) = \nu'(id) \text{ and } \bar{t} \models \varphi[\xi/\nu']
\end{array}
$$

Table 3.4: The formal semantics of Cornipickle; $a, a' \in A$ are DOM attribute names, $v \in V$ is an attribute value, $c \in C$ is a CSS selector, ξ and ξ' are variable names, $\nu, \nu' \in N$ are DOM nodes, and φ and ψ are Cornipickle expressions.

The complete semantics is defined recursively in Table 3.4. One can see that in formal terms, the expressiveness of Cornipickle's language amounts to a first-order extension of Linear Temporal Logic where events are tree structures of name-value pairs, similar to that used by the BeepBeep runtime monitor (Hallé and Villemaire, 2012); however, BeepBeep lacks the possibility of creating user-defined grammatical constructs. Moreover, the language has been extended with constructs which, although they do not increase expressiveness, improve the readability of specifications, such as `The next time`.[2]

The case of construct `When $x is now $y` warrants some explanation, however. This construct is used to refer to the same element of a document at two different moments in time. Due to the dynamic nature of web applications, it is not sufficient to simply use `For each $x in $(s)` followed by `For each $y in $(s)`, with the same CSS selector `s`. Elements in a page may be arbitrarily moved to any part of a document, and therefore fetching elements with the same selector provides no guarantee they will range over the same domain twice. Cornipickle solves this problem by giving to each element a unique attribute, called cornipickleid (shortened to id in the table). This ID never changes, whatever manipulations the application may apply on an element. The construct `When $x is now $y` evaluates variable `$y` with the element having the same `cornipickleid` as that given to the valuation of variable `$x`, allowing to compare the attributes of the same element in two distinct snapshots of the page.

Since Cornipickle always deals with finite traces, any expression whose truth value remains unresolved upon reaching the end of the trace is assumed to return a special "undefined" result, denoted by "?". We shall see next how Cornipickle can produce intuitive counter-examples in the case a property is violated; this is a feature all aforementioned tools currently lack.

[2] Formally, $\bar{t} \models$ `The next time` φ `then` ψ if and only if $\bar{t} \models$ `Not` φ `Until (`φ `And` ψ`)`.

3.2 A TOOL FOR VERIFYING LAYOUT-BASED SPECIFICATIONS

We now describe the implementation of an interpreter for the automated evaluation of Cornipickle specifications on web applications. This implementation is made of roughly 7,000 lines of Java and JavaScript code and is available under the GNU General Public License.[3] A video of the tool in action on simple examples can also be viewed online.[4]

3.2.1 DESIGN GOALS

Apart from the core functionality to be implemented, the development of the tool was motivated by a number of important design goals.

No Browser-Specific Plugins

First, the solution must work on as many combinations of browsers and operating systems as possible. This explicitly excludes browser-specific (or browser-limited) plugins, such as Chrome plugins, Firefox plugins, or the use of tools like Selenium WebDriver. For the same reason, the solution must not rely on the presence of JavaScript frameworks (jQuery, Prototype, etc.) and be stand-alone. This entails that our tool can operate in unusual browser/OS combinations, such as BoatBrowser on an Android phone, or Qupzilla under Haiku OS.

Client-Side Information Gathering

Secondly, the evaluation of specifications must be done based on information gathered on the client; this discards the prospect of performing a static evaluation of HTML and CSS on the

[3] `https://github.com/liflab/cornipickle`

[4] `http://youtu.be/90YitGRRx2w`

server side. This is mandatory for many reasons. One must consider the fact that the CSS standard is not interpreted in the same way by all browsers. For example, CSS stipulates that the width of an element does not include the padding, yet some versions of Internet Explorer include the padding in the width and render the same element with different dimensions. To a large extent, verifying constraints by looking at the HTML+CSS alone would imply emulating each browser's rendering engine, complete with its specific "quirks", to reach a faithful verdict.

In addition to the aforementioned issues, all the applications we surveyed contain client-side code that may change the layout of a page after the layout engine has processed the static declarations found in the initial HTML document and CSS files that are processed at load time. Such code, programmed to be executed when the page is being loaded, completely overrides whatever style declarations the original CSS files may initially define. Therefore, in all cases, it does not suffice to analyze the set of HTML and CSS files defined by the application, as any of this content can be modified on-the-fly through interactions with the user once the page is loaded.

No Client-Side Interpretation

Thirdly, the interpretation of Cornipickle specifications should not be done on the client side. This is done so as not to impose an undue computing burden in the browser, and allows the use of another language than JavaScript for the implementation of this functionality. More importantly, it allows behavioural properties involving more than one page snapshot to be handled by the tool. Using strictly client-side code, a problem arises when a complete page reload occurs, as this resets the state of any JavaScript object associated to that page. Since behavioural specifications require saving information from the past, some means of preserving that information in the client, across page reloads, would need to be devised. HTML5 does provide storage facilities, but using them would limit browser support (for example, only to Opera 11.5, Safari 4 and IE 9 onward[5]).

However, the storage facilities were used to keep the users' sessions. Indeed, after evaluating a page, the server serializes itself and returns the verdict along with the serialized object to the client. When the client has a new page to evaluate, he can then send the serialization to the server in order to continue where it left off after the last request. This method allows the server to be on an external machine and receive requests from multiple clients with different traces and properties.

Runtime Interpretation

Finally, it should be possible for a user to add, delete or modify the specifications being evaluated by the tool. This presents a challenge because of the special `We say that` construct, which allows one to add new grammatical constructs into the basic language. This is different from usual function or predicate definitions available in most languages, where the syntax of function calls is fixed and only new function *identifiers* are added at parse time. This necessitated the development of a BNF parser, called Bullwinkle[6] that could accept new parsing rules at runtime, as opposed to most other parsers that require a compilation step every time the grammar changes.

3.2.2 ARCHITECTURE AND TYPICAL USAGE SCENARIO

The combination of all these requirements more or less imposes an architecture for our tool, where server-side code takes care of gathering and evaluating specifications, while client-side

[5] `http://www.html5rocks.com/en/features/storage`

[6] `https://github.com/sylvainhalle/Bullwinkle`

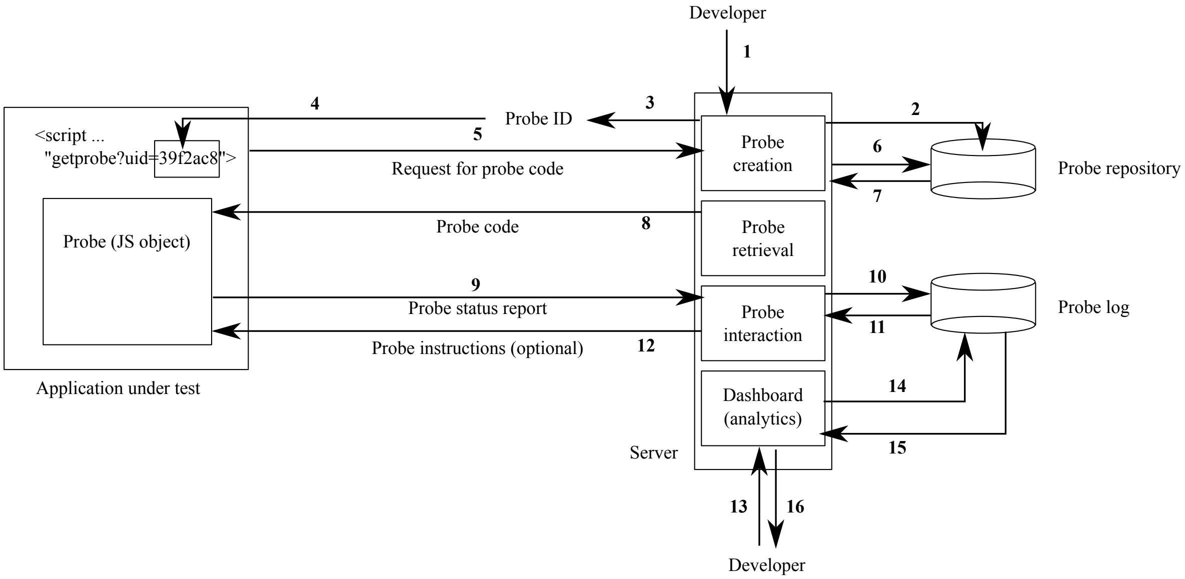
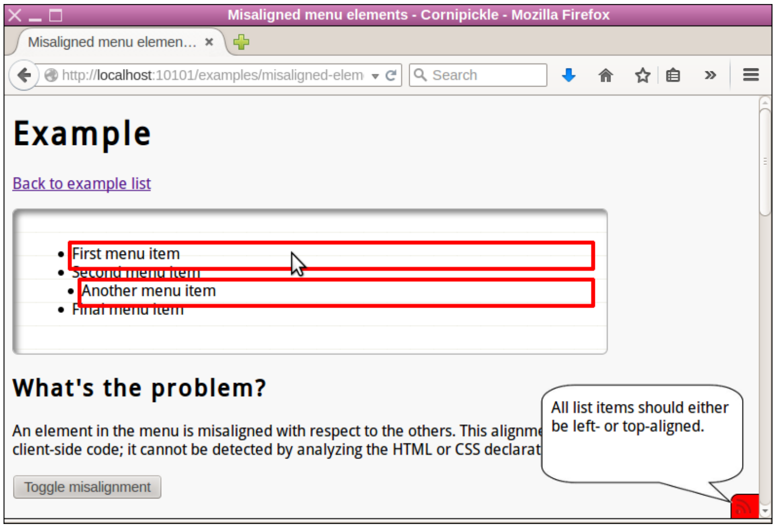


Figure 3.1: The schematics of Cornipickle

code only acts as a *probe*, querying relevant information about the current state of the page and relaying it to the server for further processing. Such client-server interaction has an advantage, though: client-side code can be relatively lightweight, and stateless (that is, be reset every time the page reloads), as any stateful processing can be done by the server.

The schematics of Cornipickle are illustrated in Figure 3.1. A developer first writes a set of declarative statements (1), which are stored in Cornipickle's memory (2). This set of statements is given a unique ID, which can be referred to in JavaScript code to be inserted into the application so that the probe can be loaded in every page (3–4); this addition is generic and differs only in the ID string. When a page from the application is to be loaded (5), Cornipickle dynamically creates the JavaScript probe corresponding to the set of assertions to evaluate and sends it back to the client (6–8).

This probe is designed to report a snapshot of the relevant DOM and CSS data upon every user-triggered event. When such an event occurs, the probe collects whatever information is relevant on the contents of the page and relays that information to the Cornipickle server


Misaligned menu elements - Cornipickle - Mozilla Firefox
Misaligned menu elemen...
http://localhost:10101/examples/misaligned-elem
Search
Back to example list
First menu item
Second menu item
Another menu item
Final menu item
Toggle misalignment

Example

What's the problem?

An element in the menu is misaligned with respect to the others. This alignme
client-side code; it cannot be detected by analyzing the HTML or CSS declarat

All list items should either be left- or top-aligned.

Figure 3.2: A screenshot of Cornipickle in action.

(9), which saves it into a log (10–11). Optionally, information on the current status of the assertions being evaluated (true/false) can be relayed back to the probe (12). An analytics dashboard can retrieve the saved log and be consulted by the developer, to query the state of all properties input at the beginning of the process (13–16).

Figure 3.2 shows an example of Cornipickle in action. In this case, the injected probe inserted a small icon in the lower right-hand side of the window, which turns red whenever a property is violated. The probe sends an Ajax request with the page's content and the serialization of the last interpreter state. The interpreter uses the latter to restore itself and sends back the verdict with the updated serialization of the interpreter. This way, the Cornipickle server can reside on a different server than the application under test, and still have punctual two-way communication with its probe.

Since the amount of data that can be relayed in this way is limited, Cornipickle takes care of sending a probe that fetches only the information required for the evaluation of the specifications provided by the user. Hence the probe is given instructions on what elements of the page are of interest, and what DOM and CSS attributes are needed on these elements. This is done by fetching the set of all attribute names occurring in an expression, and the set of all CSS selectors used in quantifiers.

The probe traverses the DOM structure in a depth-first manner, and produces one output node for each DOM node visited. By default the output node is blank: it only acts as an empty placeholder, to preserve the parent-child relationship between output nodes. Only if the current node's location matches one of the CSS selectors will attributes and values be added to the node, and only for those attributes present in the expression to evaluate. Further reductions can be achieved by trimming all subtrees that only contain blank nodes. Hence the DOM structure produced by the probe can be seen as a "hollowed out" version of the original document, containing only nodes and attributes that matter for the evaluation of a property.

Incidentally, it shall be noted that this filtering is by itself relatively coarse. Consider for example the following expression:

```
For each $x in $(p) (
  If $x's height equals 400 Then (
    For each $y in $(h1) (
      $x's width is greater than $y's width.
))).
```

This expression fetches every *p* and *h1* elements and verifies that every *p* with a height of 400 pixels are larger than the *h1* elements.

Cornipickle will be instructed to fetch the width and height of all paragraphs and headings; yet, one can see that only paragraphs of height 400 are actually necessary for deciding on the truth value of the property. Moreover, information about headings only matters if one such paragraph exists in the document, otherwise the property is vacuously true. Therefore, filtering conditions could be refined; a compromise should be reached between the bandwidth savings of such filtering, and the computing power it requires on the client-side to evaluate the conditions.

CHAPTER 4

TOOLS AND APPLICATIONS USING CORNIPICKLE

Aiming to prove Cornipickle as a generic testing tools for web applications, we used it in multiple tools and applications for more specific work.

4.1 USING CORNIPICKLE WITH A CRAWLER TO VERIFY BEHAVIOURAL BUGS

Considering many issues with current solutions on testing behavioural bugs, we propose in this section an architecture that combines an RIA web crawler (in this case, Crawljax) with the high-level language interpreter for web test oracles Cornipickle. Crawljax is responsible for exploring a web application by taking its state into account, while we employ the operators borrowed from linear temporal logic provided by Cornipickle to express assertions about the evolution of a document's content over time. This architecture was coded in an open source plugin for Crawljax [1].

[1] http://github.com/liflab/crawljax-cornipickle-plugin

4.1.1 MONITOR STATUS SERIALIZATION AND THE INTERACTION WITH CRAWLJAX

Crawljax is a tool for automatically exploring the dynamic state of modern web applications. Via programmatic interfaces, it has the capacity to interact with the client side code of the application. We use it for exploring the behavior of the web application to be tested. To detect clickables, Crawljax analyzes a web page and systematically uses them to explore the dynamic behavior of the application (Mesbah et al., 2012; Tanida et al., 2013). The detected changes in the dynamic DOM tree are committed as new states of the behavior. Many options are available with Crawljax to configure the crawling behavior. We can, for example, specify the links or the widgets to click on in the course of the crawling.

In one variation, Crawljax performs a depth-first search, stores the history of event executions and only executes an event if it has not been executed before, regardless of the application state. Another more aggressive mode forces Crawljax to execute all events and in-effect makes Crawljax perform a standard depth-first crawling strategy (Taheri, 2015).

Crawljax interacts with Cornipickle through its plugin architecture. Figure 4.1 demonstrates the workflow of the combined system to detect the behavioural bugs of the application under test. Every time a state is created or visited (1), Crawljax serializes the page and sends it to the interpreter for evaluation (2), the same way the probe sends the page to the Cornipickle server in the traditional architecture. After the page has been evaluated by Cornipickle, the verdict is returned (3) and our plugin outputs the result (4).

It is important to remember that each state is visited by Crawljax in the same sequence a user would. Even when it goes back to an earlier state, it starts at the beginning of the crawl and takes the same path until the desired state is reached.

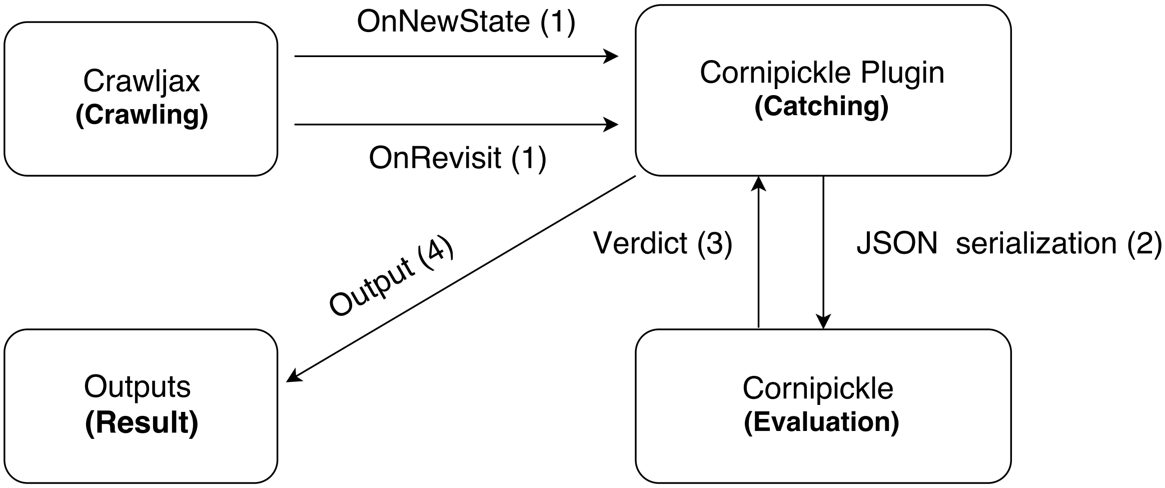


Figure 4.1: Interaction and serialization workflow (Crawljax-Cornipickle)

4.1.2 EXPERIMENTS AND RESULTS

In this section, we explain the behavioural bugs described in Section 1.4, and show how they can be caught by Crawljax by evaluating Cornipickle assertions during its exploration of an application.

Multiple Login The bug where you can login while already being logged in can be caught using these statements:

```
We say that we are signed in when (
  There exists $p in $(#action-band) such that (
    $p's text matches "^Welcome.*"
)).

We say that we are in the login page when (
  There exists $div in $(#sign-in) such that (
    Not ( $div's display is "none" )
)).

Always (
  If ( we are signed in ) Then (
    Not ( we are in the login page )
)).
```

We say that x when (y) constructs are used to define phrases that can be used later like macros. For example, when you use *x* in another statement, it evaluates to true if *y* is true. The two *We say that* explain how you define being signed in and being in the login page in the Beep Store.

There exists x in y such that (z) is used to assign to the variable *x* an element of the set *y* where *z* is true. We can see that the set *y* in the second predicate is composed of all the elements with the id "sign-in" and it makes sure that *z* is true with at least one of them. In this case, the set is composed of only one element but it could be composed of more.

The *x matches y* construct checks if *x* matches the regular expression *y* and the *x is y* construct checks if *x* is equal to *y*. Finally, the *Always (x)* statement makes sure *x* is true in every snapshot. In a nutshell, it should never happen that the action band says "Welcome" while the *div* with the id "sign-in" is displayed. It could have been simpler in another application to check the URL for the login page but, as said before, the Beep Store only has a single page with certain blocks being displayed.

Listing 4.1 shows how one could catch the same bug purely with Crawljax and its plugin architecture. The readability is much poorer and, with more complex properties, we can see how the code could get complex and lengthy.

Multiple Carts The multiple carts bug is the bug that occurs if it is possible to create a second cart after creating a first one. It can be caught using this property:

```
We say that we are signed in when (
  There exists $p in $(#action-band) such that (
private enum Verdict {TRUE, FALSE, INCONCLUSIVE};
private Verdict m_verdict;

@Override
public void onNewState(CrawlerContext context, StateVertex newState) {
  if(m_verdict == Verdict.INCONCLUSIVE) {
    EmbeddedBrowser browser = context.getBrowser();

    Identification identificationActionBand =
        new Identification(Identification.How.id, "action-band");
    boolean signedIn = false;

    Identification identificationSignInDiv =
        new Identification(Identification.How.id, "sign-in");
    boolean currentlyInLoginPage = false;

    if(browser.elementExists(identificationActionBand)) {
      WebElement actionBand = browser.getWebElement(identificationActionBand);
      if(Pattern.matches("^Welcome.*", actionBand.getText())) {
        signedIn = true;
      }
    }

    if(browser.elementExists(identificationSignInDiv)) {
      WebElement signInDiv = browser.getWebElement(identificationSignInDiv);
      if(signInDiv.isDisplayed()) {
        currentlyInLoginPage = true;
      }
    }

    if(signedIn) {
      if(currentlyInLoginPage) {
        m_verdict = Verdict.FALSE;
      }
    }
  }
  output(context, newState);
}
```

Listing 4.1: The code needed in Crawljax to catch the multiple logins bug without Cornipickle

```
    $p’s text matches "^Welcome.*"
)).

We say that we create a cart when (
  There exists $button in $(.button-create-cart)
    such that (
    $button’s event is "click"
)).

The next time ( we are signed in ) Then (
  The next time ( we create a cart ) Then (
    Always (
      Not ( we create a cart )
))).
```

The next time x Then (y) temporal statements evaluate to true if *y* evaluates to true but only after *x* does. So, after we sign in and after we click on the “create cart” button, we should never be able to click on “create cart” again.

Remove from Nonexistent Cart This bug allows the user to remove an item from a cart that has not been created. The Cornipickle property that detects it looks like this:

```
We say that we are signed in when (
  There exists $p in $(#action-band) such that (
    $p’s text matches "^Welcome.*"
)).

We say that we create a cart when (
  There exists $button in $(.button-create-cart)
    such that (
    $button’s event is "click"
)).

We say that we remove from cart when (
  There exists $but in $(.button-remove-from-cart)
    such that (
    $but’s event is "click"
)).
```

```
The next time ( we are signed in ) Then (
  Not ( we remove from cart ) Until (
    we create a cart
)).
```

(x) Until (y) means that *x* needs to be true until *y* evaluates to true. So, after we sign in, we should not click on the "remove from cart" button until we click on the "create cart" button.

A crude empirical evaluation reveals that evaluating a state with these Cornipickle properties takes between 36 to 74 milliseconds per page with an Intel Core i5-3470 CPU. The reader should keep in mind that, although the properties were pretty simple, the Beep Store is a big application to serialize because even the non-displayed blocks have to be included.

4.2 USING CORNIPICKLE WITH A CRAWLER TO VERIFY RESPONSIVE WEB DESIGN BUGS

Other than the ReDeCheck tool dicussed in Section 2.1, the existing solutions to test RWD are mostly visual and essentially consist of displaying the page in different resolutions, letting the developer notice bugs by himself. We developped an application that uses Cornipickle to find these bugs. It is available on Github [2].

We used as a core the plugin developed in the previous section to which we added a browser resizer. During development, we realized that we would also need a way to use Selenium's Web Driver with Cornipickle automatically.

[2] https://github.com/Faterou/ResponsiveTests

4.2.1 COMPONENTS

In order to find RWD bugs, we also created a plugin that resizes the browser from a given width to another width. Because having a vertical scrollbar is not considered a problem in responsive design, only resizing horizontally is the correct approach in discovering RWD bugs. Since we explicitely want to find bugs related to RWD, the plugin slowly reduces the browser's width; these bugs show up on lower widths where the space available becomes increasingly scarce in reference to wider widths. It is possible to provide to the plugin the upper bound, the lower bound and the amount of pixels for each decrement. The plugin also highlights the bugs it finds and takes a screenshot of the page.

Since Crawljax does not give direct access to the Web Driver it uses, we had to find a workaround. This workaround gives up the standard use of a plugin in Crawljax (in Crawljax's web application for example) but it lets us provide our own driver and keep a reference to it.

Then, we can create a wrapper for Selenium's Web Driver and provide it to Crawljax. The wrapper modifies the `click` function to add a Cornipickle evaluation. We end up with a tool that lets you use Cornipickle without changing Selenium's workflow. We called the tool CorniSel and it can be found on Github [3].

4.2.2 EXPERIMENTS AND RESULTS

Website behaviours are unique to every website. For that reason, the detection of behavioural bugs needs specific properties. On the other hand, Responsive Web Design is a general approach to web design, similar to design patterns in traditional languages. Failures in the implementation of this design should be detectable with general properties. Since a website

[3] `https://github.com/liflab/CorniSel`

has specific constraints, general rules cannot always apply. Thus, the properties described in this section are only warnings, and a detection should not mean it is a bug in every case.

As usual, our tool takes as input properties written in the Cornipickle language. This section will have all the properties used for our tests.

Scrollbar bug

One of the first indications of a poorly responsive website is the presence of an horizontal scrollbar. To detect this bug, a simple Cornipickle property can be defined:

```
We say that there is an horizontal scrollbar
  when (
  the page's width is less than
    the page's scroll-width
).

Always (
  Not ( there is an horizontal scrollbar )
).
```

In this property, catching that there is an horizontal scrollbar can be achieved by comparing the width of the viewport with the scroll-width. The horizontal scrollbar should never be seen so it was surrounded with the *Always... Not...* construct.

Element Collision

The next property detects the bug where elements overlap with each other:

```
We say that $x x-intersects $y when (
  (($y's right - 1) is greater than $x's left)
  And
  (($x’s right - 1) is greater than $y’s left)
).

We say that $x y-intersects $y when (
  (($y’s bottom - 1) is greater than $x’s top)
  And
  (($x’s bottom - 1) is greater than $y’s top)
).

We say that $x is visible when (
  Not ( $x’s display is "none" )
).

We say that $x and $y are the same when (
  $x’s cornipickleid equals $y’s cornipickleid
).

We say that $x and $y are not the same when (
  Not ($x and $y are the same)
).

We say that $x and $y overlap when (
  (($x is visible) And ($y is visible))
  And
  (
    ($x x-intersects $y)
    And
    ($x y-intersects $y)
)).

We say that $x and $y do not overlap when (
  Not ($x and $y overlap)
).

Always (
  For each $x in $(body *) (
    For each $y in $($x > *) (
      For each $z in $($x > *) (
        If ( ($y and $z are not the same) And
          ($y and $z do not overlap) ) Then (
          Next (
            When $y is now $a (
              When $z is now $b (
                $a and $b don’t overlap
)))))))).
```

This property starts with some language definitions to simplify the core of the property at the end. It describes horizontal and vertical intersections, a visible element, two elements that are the same and overlaps.

The first definition uses “right - 1” because elements that intersect should intersect by at least 2 pixels. It overcomes a problem where we receive dimensions and coordinates in integers (pixels) but the browser can work with floating-point numbers in cases of elements having dimensions in ratios. When one of these numbers is rounded, a 1 pixel difference can occur between what is displayed and what is serialized. It is true that we may miss bugs that are legitimately 1 pixel off but it is important to not punish good practices like using percent dimensions.

The second definition identifies if an element is visible by checking if the *display* property is “none” because these elements do not cause any layout change. Also, this value is affected consciously by the developer so their position on the page is correct.

The third definition describes two elements that are the same using the “cornipickleid” property. This property is a unique value given to every element on the page during the serialization phase. Since it is unique, it can be used to identify if two references point to the same element.

The last construct defines two elements that overlap. If they are both visible and they intersect vertically and horizontally, they are considered in a collision.

Finally, the three *For each* constructs gather all the elements and their direct children. It allows to test pairs of siblings for their overlap property. Note that it does not check whether an element overlaps with a cousin because that cousin would need to violate the Element Protrusion property described later. The property could be done by testing every element with every other element but it is costly in performance. The two *When x is now y* constructs gather the pair in the next screenshot in order to compare it with itself in the screenshot before. Overall, the property says that if two siblings do not overlap at one point in time, these two siblings should not overlap either in the next point in time.

We could not find an example of a visible overlapping bug because most overlaps are caused by an element protruding outside of its parent and this bug is caught by the next property. However, we caught non-visible element collisions, so it is safe to claim that we can also catch visible ones.

Element Protrusion

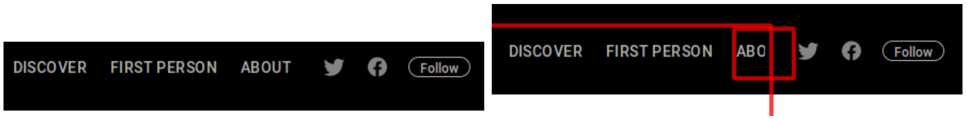
This property tackles the problem of elements that overflow their container. It can be written in the Cornipickle language in this fashion:

```
We say that $child is horizontally
  inside $parent when (
  ($child’s left is greater than
    ($parent’s left - 2))
  And
  ($child’s right is less than
    ($parent’s right + 2))
).

We say that $child is vertically
  inside $parent when (
  ($child’s top is greater than
    ($parent’s top - 2))
  And
  ($child’s bottom is less than
    ($parent’s bottom + 2))
).

We say that $x is visible when (
  Not ( $x’s display is "none" )
).
```

```
We say that $child is fully
  inside $parent when (
  If ( ($child is visible) And
    ($parent is visible) ) Then
  (
    ($child is horizontally inside $parent)
    And
    ($child is vertically inside $parent)
)).

Always (
  For each $x in $(*) (
    For each $y in $($x > *) (
      If ($y is fully inside $x) Then (
        Next (
          When $x is now $a (
            When $y is now $b (
              $b is fully inside $a
))))))).
```

The first definitions describe how childs can be horizontally and vertically inside their parents. For the same reason than the Element Collision property, we use “parent’s left - 2” to allow a protrusion of 1 pixel outside of the parent’s container. The property at the end has two *For each* constructs that return a pair composed of any element in the page and any of its direct children. Then, if the latter is fully inside the former in an initial screenshot, the same pair should be fully inside in the next one.

This property was able to find a bug on the website `https://www.thelily.com/`. It can be seen in Figure 4.2 where the *div* with the menu buttons ends up outside of the menu bar and out of sight.



(a) All the buttons are correctly in the menu bar.

(b) The highlighted “About” button is protruding outside of the menu bar, its parent.

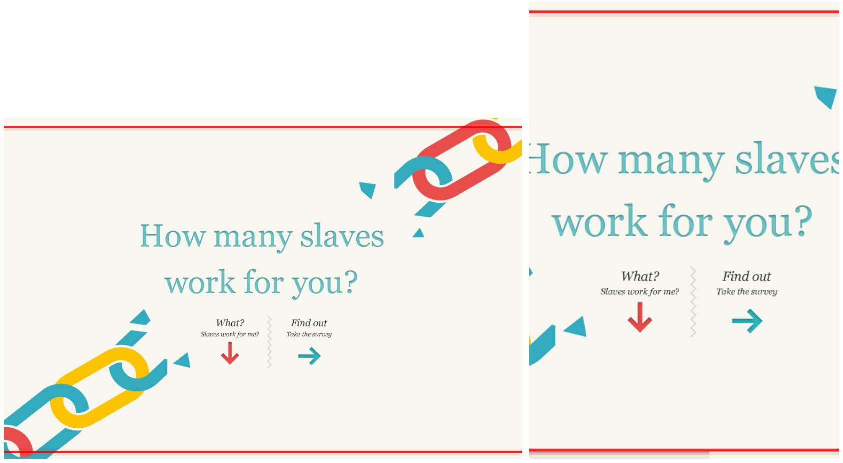
Figure 4.2: The Element Protrusion bug on the website `https://www.thelily.com/`.

Viewport Protrusion

In a Viewport Protrusion bug, elements are found outside of the viewport. This Cornipickle property can detect that behaviour:

```
We say that $x is visible when (
  Not ($x’s display is "none")
).

We say that $x is fully inside the viewport
  when (
  If ($x is visible) Then
  (
    (($x’s left + 2) is greater than 0)
    And
    ($x’s right is less than
      (the page’s width + 2))
)).

Always (
  For each $x in $(*) (
    If ( $x is fully inside the viewport )
      Then (
      Next (
        When $x is now $y (
          $y is fully inside the viewport
))))).
```

An element that is inside the viewport needs to be visible; if it is hidden, it does not affect anything. Then, once more, it checks “left + 2” and “width + 2” because of the 1 pixel



(a) The *div* is highlighted but there is no observable bug.

(b) At 440px, we see the same *div* with an observable bug where the text overflows outside the viewport.

Figure 4.3: The Viewport Protrusion bug on the website `https://www.slaveryfootprint.org/`.

rounding problem. If the element's left property is greater than 0 and its right is less than the width of the page in a first screenshot, it needs to keep these properties in the next.

On the website `https://www.slaveryfootprint.org`, a Viewport Protrusion was found in a large width. Figure 4.3 shows how non-observable bugs can create problems at lower widths. Here, the middle *div* is found to be a bit outside of the viewport but it does not show any observable issue. However, it's at 440px that we can clearly see the content overflowing out of the viewport.

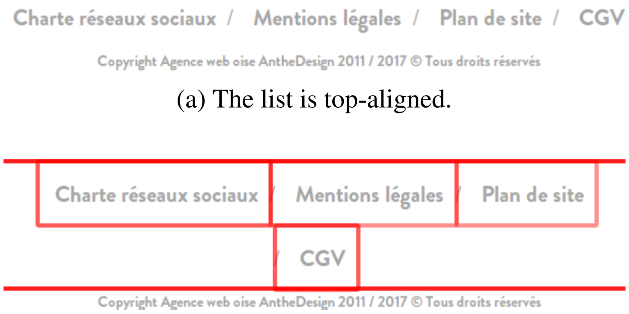
Wrapping Elements

Wrapped elements are elements that are pushed on an additional line after they were aligned with other elements on a single line. We limited our implementation to elements that are inside a list.

```
We say that $x and $y are top-aligned when (
  $x's top equals $y's top
).

We say that $x and $y are left-aligned when (
  $x's left equals $y's left
).

We say that the list $x is aligned when (
  For each $y in $($x > li) (
    For each $z in $($x > li) (
      ($y and $z are top-aligned)
        Or
      ($y and $z are left-aligned)
))).

Always (
  For each $x in $(ul) (
    If (the list $x is aligned) Then (
      Next (
        When $x is now $y (
          the list $y is aligned
))))).
```

We say that two elements are top-aligned and left-aligned when their top and left values, respectively, are equal. A predicate can then be constructed to define lists that are aligned when they are both top and left aligned. Finally, all the lists are taken in a first screenshot in order to compare their elements' alignement. They then need to still be aligned in the next screenshot.

An example of a wrapped element can be seen in Figure 4.4. It could be argued that this is not a bug, however, at lower widths, the list is top-aligned again. This shows that having this list top-aligned is the desired layout.


Charte réseaux sociaux / Mentions légales / Plan de site / CGV
Copyright Agence web oise AntheDesign 2011 / 2017 © Tous droits réservés
Charte réseaux sociaux / Mentions légales / Plan de site
/ CGV
Copyright Agence web oise AntheDesign 2011 / 2017 © Tous droits réservés

(a) The list is top-aligned.

(b) At a lower width, the "CGV" element gets pushed on an additional line. The list was red highlighted by our tool.

Figure 4.4: The Wrapping Element bug on the website `https://www.anthedesign.fr/`.

Small-Range Layouts

The Small-Range Layouts bug was not implemented for tool limitations reasons; properties in Cornipickle can only make assertions on pages, not on the verdicts of other properties. However, this failure is more of a statistic on the result of the other properties. A Small-Range Layout is basically a web page where a small interval of widths shows no failure for the other properties.

CONCLUSION

Looking at the state of web applications and their growing usage, it was clear that more refined testing tools were needed. Web applications have the advantage of being cross-platform which cuts development costs. Frameworks like Electron (used, for example, by Atom, Discord and Slack) can even produce a desktop application from a web application. Likewise, Apache Cordova, among others, deploys a mobile application from a web application. Hence, testing the web application at the core of these other applications becomes important to limit bug propagation.

Initially, we wanted to design a testing tool which had a very expressive specification language, did not need browser-specific plugins, had most of the workload on server rather than on client and used runtime interpretation. These goals were met but some only partially.

We achieved to develop a specification language that is very expressive compared to regular code. With inspiration from first-order logic, the generality of the grammar provides the means to express lots of bugs, static or not. We showed how we can catch visual and behavioural bugs with the language.

To work as an oracle for this specification language, we designed Cornipickle as a server-side tool that extract the pages' content on the client and returns whether they are correct according to the specification. We wanted the most versatile client-side code in order to not restrict any browser/OS combination. It meant we had to use standard Javascript without plugins or partially supported features. We mostly achieved that goal. Along the way, we had to use HTML5's Session Storage to hold the serialized state of the interpreter. Because the server can only hold one instance of the interpreter (with one state history and specification), the task had to be given to the client and cookies can not hold enough data. For now, we think it is supported sufficiently but it could be improved.

We then applied Cornipickle to more specific work. To help test behavioural bugs, we paired it with a crawler that traverses the state graph of a website in a user-like manner. By evaluating on every state, we can test the website quickly and automatically. It was successful in finding the stateful bugs introduced in previous works.

Similarly, with a Cornipickle and Crawljax application, we achieved to crawl websites and test for RWD bugs. The RWD bugs we decided to test were the types of bugs tested by the tool ReDeCheck. We showed that we can catch these types of bugs except one. The Small-Range Layout bug is more of a meta-bug where it can be caught by analyzing the verdicts of other properties. Our results were partially good given that we get many false positives, and we can only find the first bug occurence of any type. False positives are however a problem in ReDeCheck too. We still were able to show that we can express these RWD properties and even more thanks to the expressiveness of the language.

We effectively proved Cornipickle as a multipurpose testing tool for web applications. It can test layouts as well as behaviours and RWD. Future works could improve bug finding by enabling it to find multiple bugs of one property and improve browser compatibility by removing HTML5's client-side storage. The next step for Cornipickle could be to fix the bugs that it finds at runtime.

BIBLIOGRAPHY

Applitools. 2016. Applitools Visual Test Automation. `http://www.applitools.com` Accessed on April 25, 2016.

Arora, A. and M. Sinha. 2012. “Web Application Testing: A Review on Techniques, Tools and State of Art”, *International Journal of Scientific I& Engineering Research*, Volume 3(2), Pages 1–6.

Benjamin, K., G. Von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I. V. Onut. 2011. “A Strategy for Efficient Crawling of Rich Internet Applications”. In *11th International Conference on Web Engineering ser.ICWE’11*, Pages 74–89. Heidelberg: Springer-Verlag.

CSSChopper. 2017. Responsive Web Design Testing Tool - Online. `http://www.websiteresponsivetest.com/` Accessed on September 18, 2017.

Choudhary, S., E. Dincturk, S. Mirtaheri, G.-V. Jourdan, G. Bochmann, and I. Onut. 2013a. “Building Rich Internet Applications Models: Example of a Better Strategy”. In *Web Engineering, ser. Lecture Notes in Computer Science, F. Daniel, P. Dolog, and Q. Li*. Volume 7977, Pages 291–305. Springer.

Choudhary, S., M. E. Dincturk, S. M. Mirtaheri, A. Moosavi, G. von Bochmann, G.-V.

Jourdan, and I.-V. Onut. 2012. "Crawling Rich Internet Applications: the State of the Art". In *CASCON*, Pages 146–160. IBM Corporation.

Choudhary, S. R., M. R. Prasad, and A. Orso. 2013b. "X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications". In Notkin, D., B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, Pages 702–711. IEEE / ACM.

Dallmeier, V., M. Burger, T. Orth, and A. Zeller. 2013. "WebMate: Generating Test Cases for Web 2.0". In Winkler, D., S. Biffl, and J. Bergsmann, editors, *SWQD*. Volume 133, Series *Lecture Notes in Business Information Processing*, Pages 55–69. Springer.

Dincturk, M. E. 2013. "Model-Based Crawling - An Approach to Design Efficient Crawling Strategies for Rich Internet Applications". Phd Thesis, University of Ottawa.

Dincturk, M. E., S. Choudhary, G. von Bochmann, G.-V. Jourdan, and I.-V. Onut. 2012. "A Statistical Approach for Efficient Crawling of Rich Internet Applications", *ICWE*, Pages 362–369.

FROONT. 2012. FROONT. http://froont.com/ Accessed on September 18, 2017.

Garrett, J. 2005. Ajax: A new approach to web applications - adaptive path. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/ Accessed on October 24, 2017.

Garsiel, T. and P. Irish. 2011. https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/ Accessed on October 16, 2017.

Hallé, S., G. Le Breton, F. Maronnaud, A. B. Massé, and S. Gaboury. 2014. "Exhaustive Exploration of Ajax Web Applications with Selective Jumping". In *ICST*, Pages 243–252. IEEE Computer Society.

Hallé, S. and R. Villemaire. 2012. "Constraint-based invocation of stateful web services: The Beep Store (case study)". In Lago, P., G. A. Lewis, A. Metzger, and V. Tosic, editors, *4th International ICSE Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2012, June 4, 2012, Zurich, Switzerland*, Pages 61–62. IEEE.

Hallé, S. and R. Villemaire. 2012. "Runtime Enforcement of Web Service Message Contracts with Data", *IEEE T. Services Computing*, Volume 5, Issue 2, Pages 192–206.

Hors, A. L., P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. 2000. Document Object Model Level 2 Core. http://www.w3.org/TR/DOM-Level-2-Core.

Hunter, J. 1999. What's new in Java Servlet API 2.2? `https://www.javaworld.com/article/2076518/java-web-development/what-s-new-in-java-servlet-api-2-2-.html` Accessed on October 16, 2017.

Järvi, J., M. Marcus, S. Parent, J. Freeman, and J. N. Smith. 2008. "Property models: from incidental algorithms to reusable components". In Smaragdakis, Y. and J. G. Siek, editors, *GPCE*, Pages 89–98. ACM.

Lutteroth, C. and G. Weber. 2008. "Modular Specification of GUI Layout Using Constraints". In *ASWEC*, Pages 300–309. IEEE Computer Society.

Mahajan, S. and W. G. J. Halfond. 2015. "WebSee: A Tool for Debugging HTML Presentation Failures". In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, Pages 1–8. IEEE.

Mesbah, A., A. van Deursen, and S. Lenselink. 2012. "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes", *ACM Transactions on the Web*, Volume 6, Issue 1, Page 3.

Mirtaheri, S. M., M. Dincturk, S. Hooshmand, G. V. Bochmann, and G. Jourdan. 2013a. “A Brief History of Web Crawlers”. In *CASCON ’13 Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, Pages 40–54. IBM Corp. Riverton, NJ, USA ©2013.

Mirtaheri, S. M., D. Zou, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut. 2013b. “Dist-RIA Crawler: A Distributed Crawler for Rich Internet Applications”. In *8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, Pages 105–112. IEEE Computer Society Washington.

Olston, C. and M. Najork. 2010. “Web Crawling”, *Foundations and Trends in Information Retrieval*, Volume 4, Pages 175–246.

Parent, S., M. Marcus, and F. Brereton. 2007. ASL Overview. Tech. Report, Adobe Systems. `http://stlab.adobe.com/group__asl__overview.html`.

Pugsley, T. and A. Hovey. 2013. Responsinator. `https://www.responsinator.com/` Accessed on September 18, 2017.

Respondr. 2014. Respondr. `http://respondr.io/` Accessed on September 18, 2017.

Screenfly. 2017. Screenfly. `http://quirktools.com/screenfly/` Accessed on September 18, 2017.

Sharp, R. 2011. Responsivepx. `http://responsivepx.com/` Accessed on September 18, 2017.

StatCounter. 2017. `http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet` Accessed on October 23, 2017.

Szekely, P. A., P. N. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. 1995. “Declarative interface models for user interface construction tools: the MASTERMIND approach”. In Bass, L. J. and C. Unger, editors, *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*. Volume 45, Series *IFIP Conference Proceedings*, Pages 120–150. Chapman & Hall.

Taheri, S. M. M. 2015. “Distributed Crawling of Rich Internet Applications”. Phd Thesis, University of Ottawa.

Tamm, M. 2009. Fighting Layout Bugs. Google Test and Automation Conference (GTAC 2009), Zurich, Switzerland, October 21–22, 2009. `https://www.youtube.com/watch?v=WY3C6FHqSqQ`.

Tanida, H., M. R. Prasad, S. P. Rajan, and M. Fujita. 2013. “Automated System Testing of Dynamic Web Applications”. Volume 303, Pages 181–196. Springer Berlin Heidelberg.

Walsh, T. A., G. M. Kapfhammer, and P. McMinn. 2017. “Automated Layout Failure Detection for Responsive Web Pages Without an Explicit Oracle”. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Series “ISSTA 2017”, Pages 192–202, New York, NY, USA. ACM.

Walsh, T. A., P. McMinn, and G. M. Kapfhammer. 2015. “Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages (N)”. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Series “ASE ’15”, Pages 709–714, Washington, DC, USA. IEEE Computer Society.

Wassermann, M. 2012. Viewport Resizer. `http://lab.maltewassermann.com/viewport-resizer/` Accessed on September 18, 2017.

Wynne, M. and A. Hellesoy. 2012. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf.