End User Involvement in Exploratory Test Automation for Web Applications

Version of December 12, 2011

Paolo Luigi Schipani

End User Involvement in Exploratory Test Automation for Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF ENGINEERING

in

COMPUTER ENGINEERING

by

Paolo Luigi Schipani
born in Rome, Italy

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology

Delft, the Netherlands
www.ewi.tudelft.nl

End User Involvement in Exploratory Test Automation for Web Applications

Author: Paolo Luigi Schipani
Student id: 4035801
Email: paololschipani@gmail.com

Abstract

The traditional way of developing websites as hypertexts, which can be navigated link by link, is progressively giving way to the AJAX approach, in which the entire hypertext can be contained in a single web page. The resulting page has the advantage of offering navigation by loading only specific parts of the page - only the changing content. Conventional web crawlers, applications which explore web pages in a systematic way, are not able to browse AJAX pages. In order to overcome this barrier, which prevents the execution of automated tasks such as web indexing or mechanized tests, the Software Engineering Group at TU Delft has developed Crawljax, a tool capable of crawling AJAX pages.

Crawljax already offers many possibilities. It provides default settings for simple page testing, but it can also be included in a Java project and be programmed to execute more complicated testing, or specific crawling in certain directions of the page. For example, Crawljax can include or exclude some buttons, check boxes, text areas and other elements of the page to help focus on a certain area to test. Through its various plugins it can benchmark websites, find invariants to use in regression tests, export a graphical representation of the states tree graph, and more. All of these possibilities are however restricted to Java programmers, willing to learn how to use a new tool to expand their limited crawling power. What Crawljax does not yet offer is a simple way, even for non-programmers, to create and execute specific test cases.

Here we present an extension on Crawljax, a way to simplify the process of running crawling sessions and integrity tests on webpages. We call this system CrawlMan, the Crawljax Manager. CrawlMan uses components of Crawljax and his plugins and libraries, connected to a Graphical User Interface, in order to provide automated, repeatable crawling and testing. The application allows a basic user to start crawling a

web page by simply inserting the selected URL, then shows a graphical representation of the result and uses it to guide the user in the refinement of the settings. The user can then crawl the same URL with more specific settings, inspect the new result and use the new suggestions to refine the settings, again and again. The obtained cycle, where the test results are used to improve the test itself, is the main project contribution. We evaluate our approach by means of analyzing the behavior of selected novice users during the execution of predefined tests.

Thesis Committee:

Chair:	Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft
External supervisor:	Dr. Ir. Ali Mesbah, Faculty ECE, University of British Columbia (UBC)
Committee Member:	Dr. Phil. Hans-Gerhard Gross, Faculty EEMCS, TU Delft
Committee Member:	Dr. Martin Pinzger, Faculty EEMCS, TU Delft

Preface

This master thesis is not just the result of a final project, but the conclusion of an exciting journey that brought me to discover an amazing country, torn out of the waters of the sea by the hands of men, and to meet many new friends. I have been honored by being accepted to study in the Delft University of Technology, and to live between such unweary people as the Dutch.

This thesis could simply not be possible without the work of Ali Mesbah and Arie van Deursen. The project in fact extends Crawljax, by adding a layer between user and program, to facilitate the setting and the interpretation of results. More than for their work, on which the project is based on, I would like to especially thank them for their time and personal constant support. There is not a moment that I was left alone in this journey, and I am very grateful for it.

These master studies would have not been possible if it was not for the support of my parents, which always encouraged me to make the best of my capabilities, even when it meant sending me away. Luckily, it is not distance that defines relationships with the people we truly love. I also want to thank my distant friends, patiently waiting for me to return and celebrate with them.

Last thanks go to the first users of CrawlMan, the group of friends that collaborated to the evaluation tests, giving me precious advices on how to make CrawlMan better, easier and more intuitive: Bassem Zarour, Christiaan Menkveld, Marcela Izaguirre, Marco Cova, Nicola Pambakian, Rogerio Canales Perez, Sandra Treviño Barbosa, Teun Janssen, Yani Mur. Thank you all for your unconditioned help.

Paolo Luigi Schipani
Delft, the Netherlands
December 12, 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The relentless evolution of the web is rapidly changing the way we learn to use computers and software applications. The simple vision of the Web as a collection of hypertexts, providing multimedia data and more generally information, has given way to Web 2.0 [3], a collection of web applications where the user can generate and share contents. On this direction, there soon will be no more need for a user to install software on his computer, as new techniques allow the porting of software applications into web applications, accessible with a browser from any part of the world.

One of the key technologies contributing to this change is **AJAX** [4] (Asynchronous JavaScript and XML). AJAX is a collective name for a set of techniques employed for allowing asynchronous data exchange between a web server and a browser client, suppressing the need to continuously reload the visited page, making web applications more similar to desktop applications. In the panorama of web application testing, the use of these technologies calls for new ways to automatically test and verify the behavior of web pages [5]. The evolution of the web calls upon an evolution of the methods for testing the web, switching from a mostly manual approach, where a user repeats testing scenarios by browsing pages and annotating the results, to an automated approach, where a testing framework allows a user to define tests and repeatedly execute them. Such a framework would automate the creation of tests, the verification of results and the refinement of these tests depending on their results. The framework would reduce costs and time for testing, while guiding the user in the definition of meaningful test cases. The purpose of this project is to provide this testing framework.

The project consists of a testing framework based on **Crawljax** [5, 6, 7, 8, 9, 10], a Java desktop tool for crawling AJAX pages developed at TU Delft by Ali Mesbah et al. Crawljax can detect changes in the DOM of a web page, thus composing a graph representation of crawled websites through their different states. The project, denominated **CrawlMan** (CRAWLjax MANager), is a web application, created to extend Crawljax by providing ease of use and graphical means for interaction. CrawlMan aims to provide a user-friendly way to modify the scope of the state space to examine, by creating settings for Crawljax in

response to user actions on the result of a crawling session. The project provides a simple interface for using Crawljax, eliminating the need for installation and the learning process of the application. When requesting the crawling of a website, the application returns a visual result, which is modifiable by the user. These modifications directly act on the crawling process, pushing it in a specific direction, where the interest of the user resides.

1.1 Project Motivation

Web applications are more and more popular every day, and AJAX techniques with them. The most renowned websites at the moment - Facebook [11], Gmail [12] (and other Google products), Wikipedia [13], Ebay [14] and more - make use of these technologies. A reliable way to test those applications is required, which cannot be tested using traditional methods. Crawljax offers such a way, filling a hole in the web application testing market, but it introduces in turn new challenges. Crawljax is a powerful tool, but it requires specific acquaintance with programming. Every desired setting, click or condition must be manually programmed in Java, and the result must be extracted in the same way. This aspect means that every new test to perform with Crawljax requires a new program to be written in Java. If the test needs to be changed, the program must be rewritten, then recompiled, and only then it can be executed again. Of course, the programmer needs to know the libraries and methods of Crawljax, and the right way to write settings and conditions, as the tests he writes must refer to elements and paths inside the applications he wants to try. The learning curve of Crawljax and the necessity to write a Java executable for every test constitute a barrier to its utilization, impossible to overcome for non-programmers. The overall purpose of this project is to take down this barrier, by extending Crawljax with an easy-to-use interface, turning it into a web application, allowing non-specialists end users to run Crawljax on their sites.

1.2 Research Questions

The goal of the project is offering a simple and reliable alternative for testing (AJAX) web applications. To that end we propose CrawlMan, a simple and intuitive framework for **Automated Exploratory Testing** (Section 7.1.3), effective and user friendly. Inexperienced users should be able to perform tests, and understand the result and modify the settings to accomplish their needs (Section 7.2). At the same time it must bring innovation in testing web applications, using Crawljax to perform tests in a dynamic, exploratory way. The success of this project will depend from the answers to the following questions:

- Given a specific web application behavior, is it possible for a common user to set CrawlMan to reproduce it and understand the result?
- Is CrawlMan effective in reaching interesting states and identifying potential faults?

- Is CrawlMan useful for automated exploration of a web application, and does the user benefit from the data it collects?

The answers will be provided during the evaluation (Chapter 6) of the project. A group of end users will be involved in the testing, to assess project reliability and intuitiveness.

1.3 Project Synopsis

The project presents itself as a web application, written in Java using Google Web Toolkit [15], a web application development framework based on AJAX. Through its various panels, it is possible to access all the possible settings of Crawljax, or simply start crawling and then refine the session using the suggestions presented by the application. The information inserted by the user into the application gets converted into settings, with no need to know Java programming or the structure of Crawljax. The result obtained, representing the possible states of the crawled website, is visualized as a tree graph, whose nodes can be selected for displaying their information. The original contribution of this project is the possibility to create new settings by directly acting on the graphical representation of the state-flow graph, and to use these settings immediately in a new crawling session. In this way the project realizes a flow of data between settings and result, each influencing the other. The crawling process is more easily directed in the desired way, while it is possible to rapidly build and execute routine tests for web pages, even the most complicated, as the ones built using AJAX techniques.

Chapter 2

Background

In the earlier times of the Internet era, websites were hypertexts, pages with fixed content connected together by links [3, 4]. Most pages contained just HTML, showed some text, pictures and a fancy background. The home page was the root of a tree of links and nodes, the pages, and every page would have its own address, composed by the main URL plus the name of the actual HTML file. Loading different content meant loading a different page.

What changed from then? We still navigate through web pages, we still use a browser. And the code our browser interprets is still the familiar HTML... with some additions. Some elements were introduced to the scene, new technologies to add dynamic content to the pages - Java applets and JavaScript, IFrames and other ways to modify content at run time. New content can now be loaded without reloading the page itself, and we are already used to it, we do not even notice it anymore. Our email warns us we received a new message. We press a button and a movie starts, while it gets downloaded frame by frame. Content loading became asynchronous, thus we do not need to reload the page anymore. A website is not a static collection of data anymore, it is a web application, which can perform operations and give the result to the user, can store data, can allow users to share data, and ultimately, in the limits of system resources and bandwith, it can perform everything a desktop application can do. Maybe one day we will not need to install any program anymore on our computer, as everything we need is reachable online. Google [16], which has lately become the leader in the field, already proposes a set of Office-like web applications, free for everyone online.

2.1 Crawlers

This project ultimately is an extension of a crawler [17], but what is a web crawler? It is an application, a tool for exploring the web. Web crawler is a general denomination for all kind of software capable of browsing the Internet automatically, given as input one or more addresses.

Crawlers can be useful for different reasons. One, which we are all indirectly familiar

with, is the automatic indexing of web pages. Web crawlers are in fact essential for the working of search engines, like Google [16], Yahoo! [18], Bing [19]. Web search engines need to continuously index new pages, as the World Wide Web is in a state of perpetual change. When a search is made in the engine, it looks for the query words in all its indexes, and then lists the corresponding URLs (Uniform Resource Locators) to the user. Crawlers can also be used for other tasks, such as retrieving specific infomation, making a copy of an entire website (web archiving), or automatic testing. A crawler can be given, for example, instructions to click on certain elements of the page it visits, and repeat that over and over, then create a report. Also, they are of academic interest, as means for information retrieval, the study of searching information in a set of documents.

How do crawlers explore websites, and how do they navigate from page to page? The input, as we said, is a list of addresses, URLs. When loading one address, the crawler inspects the content of the page, to find all the links to other pages. The crawler can then use these new found links to load other pages, inspect them and find more. If a page is not linked anywhere, it cannot be found: it is an invisible page, unless its URL is given as input to the crawler. This process is shown in Figure 2.1.

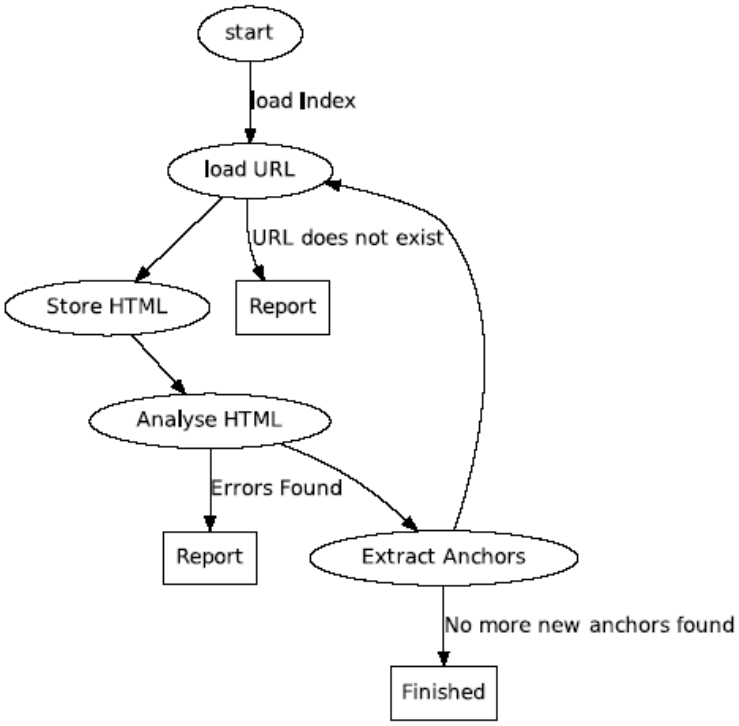

Figure 2.1: Operations of a crawler, as showed in [1].

This was the way of working of traditional crawlers, which relied on hyperlinks and the vision of the web as a collection of hypertexts. Then web applications came into the picture, using asynchronous techniques to load new content without changing their page

or address. While the pages of a hypertext represent its states, a web application can have different content through different states of the same page. Modern crawlers must be able to detect the changes in the page, to recognize that new content has been loaded, and consider the new content in the processing.

2.2 AJAX

AJAX [4], Asynchronous JAvascript and XML, is a collective name for a set of techniques used on the client side, responsible for rendering the page visualized in the browser. To understand it fully, we must comprehend the nature of web pages and the working of browsers. Web browsers, such as Internet Explorer [20], Firefox [21], Chrome [22], are desktop applications which can load a page defined by a certain address, the URL (Uniform Resource Locator). The server hosting the page provides it in a language comprehensible to the browser, the HTML language [23]. A page written in HTML language is a document forming a tree structure, where on top, forming the root, there is a 'html' element. Figure 2.2 shows the HTML code for a very simple web page.

```
<html>
  <head>
    <title>My page</title>
  </head>
  <body>This is my page.</body>
</html>
```

Figure 2.2: The code for a simple HTML page.

This hierarchical structure is commonly known as the HTML DOM - Document Object Model [24]. Through this structure and some client side language, such as JavaScript, it is possible to access the elements of a page, by their identifiers or positions. Then their content can be changed at runtime, for example as a response to the user clicking on a button. Furthermore, the style of the page, and other inherent properties, can be modified, new requests can be sent to the server, an entirely different content, seemingly a new page, can be showed without reloading the full page.

Here resides the problem. Traditional web crawlers rely on hyperlinks to figure out the structure of a website. With AJAX, and in general asynchronous techniques, there are no different URLs, and if there are they are not connected with the change of content [9]. This fact represents a problem for crawlers, and different solutions have been proposed, which we explore in Chapter 7.

2.3 Crawljax

Crawljax [5, 7, 9] is a project conceived by Ali Mesbah and the SERG group in TU Delft [25]. Crawljax is a web crawler, written in Java. It can crawl traditional websites, but more importantly it can crawl AJAX pages. Unlike other crawlers, it does not look for hyperlinks in the pages, but it examines their DOM looking for changes. This means that Crawljax does not try to find ways out of a website, but tries to find out all the possible states the website can assume, inside different pages or one unique page, according to the settings. This way Crawljax can explore all the possible content of a website, even the content appearing after specific sequences of user actions. The changes in the page are triggered by actions the crawler performs on the page, by firing events on the elements of the DOM. For example, Crawljax can be programmed to click all the buttons it finds, or all the buttons with some specific text.

Many different settings can be applied to Crawljax. It can be set to follow specific crawling conditions, for example to crawl, or not to crawl, a page if a certain element is visible, or it can be set to ignore elements of a page, so that some changes do not bring the crawler in a new state. Different DOMs are compared without the ignored elements, according to the selected *Oracle Comparator* (Section 2.4.11). There is also the possibility to check some condition, as the presence or absence of an element, or the validity of a JavaScript expression, on all the states of the web application. This condition, which has to always hold, is an *Invariant* condition. An important strength of Crawljax is its structure, which makes it easy to extend it by creating *plugins*. The *plugins* can be associated with Crawljax execution and collect data during the crawling to produce a result, push the crawling in a certain direction, perform operations of the states and many other tasks.

2.3.1 Crawljax Example

Crawljax is distributed as a Java library, to be imported in a Java project. The simplest example of utilization of Crawljax is inserting a configuration, and then running Crawljax through its class *CrawljaxController*.

```
CrawlSpecification spec
     = new CrawlSpecification("http://www.some_domain.com");
spec.clickDefaultElements();

CrawljaxConfiguration config = new CrawljaxConfiguration();
config.setCrawlSpecification(spec);

try {
     CrawljaxController crawljax = new CrawljaxController(config);
     crawljax.run();
} catch (CrawljaxException e) {
     e.printStackTrace();
     System.exit(1);
} catch (ConfigurationException e) {
     e.printStackTrace();
     System.exit(1);
}
```

Figure 2.3: A basic example of Crawljax's utilization.

As showed in Figure 2.3, the URL of the website to crawl is inserted through the *CrawlSpecification* class. We also apply the *Click Default* setting, which allows to click on every link and every button detected by Crawljax. In this example we do not collect the result, just crawl a website. As a user of Crawljax would run the class containing these instructions on his personal computer, he would see the default browser Mozilla Firefox open, load the specified URL, and automatically navigate through the website pages. To collect the result data for later utilization, one must extract the *State Flow Graph* produced by *CrawljaxController* or use a *plugin* to collect the data during execution. The second method, which is the advised one, offers more possibilities, because the browser and the current crawled state can be accessed through Crawljax - for example for capturing a screenshot of the current state. Figure 2.4 shows the way to add a plugin through *CrawljaxConfiguration*.

```
MyPlugin plugin = new MyPlugin();
config.addPlugin(plugin);
```

Figure 2.4: Adding an ad-hoc plugin to CrawljaxConfiguration.

There is a variety of settings which can be inserted through *CrawljaxSpecification*. It is possible to set Crawljax to click on all elements respecting a specific description, to crawl a state of the application only if a given condition is true, and much more (Section 2.4).

Clicking on all the detected images, for example, can be done as shown in Figure 2.5.

```
spec.click("img");
```

Figure 2.5: Setting Crawljax to click on all images.

For every new state Crawljax encounters, it clicks on all the HTML elements with tag 'img'. A more specific description can be given, inserting text, attributes and position of the HTML element inside the DOM representation of the state. A detailed explanation is offered in Section 2.4.

2.4 Crawljax Settings

This section describes all the settings currently offered by Crawljax. It is important to know what the possibilities of Crawljax are, because this project, being an extension of Crawljax, wants to able to do everything Crawljax can do, plus its own contribution.

2.4.1 Element Identification

There are various means for identifying an element of the DOM. Every element has a *tag* and a position inside the document, and it can have different attributes. The *tag* is the actual HTML element, for example a button in the HTML page is defined by the 'button' tag. The position is defined by an *xpath*, which is, as explained later in this section, a description of the path from the root of the DOM tree to the desired element. For example, the *xpath* of the button contained in the code of Figure 2.6 is '*/html[1]/body[1]/button[1]*'.

```
<html>
  <head>
    <title>My page</title>
  </head>
  <body>
    <button id="buttonId" type="submit">button text</button>
  </body>
</html>
```

Figure 2.6: A HTML page containing a button.

Every element of the DOM can be identified in these ways, which are implemented in Crawljax in three different manners, according to their utilization:

- Id - the 'id' attribute of a HTML element. It is used to identify input fields in input (Section 2.4.6) and *Set Values Before Click* settings (Section 2.4.7) .
- Tag, text, xpath and attributes - these four characteristics are used to identify DOM elements in the *Click* settings (Section 2.4.7) .
- Identification object - the user can define the characteristic of a DOM element he prefers between tag, id, name, xpath, text or partial text. It is used in *Expected* conditions and in the *Visible* crawl condition (Section 2.4.2).

Element characteristics

A HTML document is a hierarchy of elements, where the main element, or the root, is the 'html' element. An element is defined by one or two tags, depending if it needs or not a closure. The element 'br', for example, is never closed, so it just needs the start tag, as in Figure 2.7.

```
<br>
```

Figure 2.7: Newline HTML element.

A text box with no default text is defined by a single, closed tag, as in Figure 2.8.

```
<input type="text"/>
```

Figure 2.8: Empty text box HTML element.

A text box with some text, as all other tags which require content, is defined with a start and an end tag, as in Figure 2.9.

```
<input type="text">some text</input>
```

Figure 2.9: Text box HTML element with default text.

In this case we defined only one attribute, the input 'type', but we could have specified an *id*, a *name*, or other *attributes* as specified in the HTML standard [23]. It is not possible to use self made tags or attributes, because they will not be recognized by the standard HTML interpreters.

XPath

XPath [26] is a language, aimed at identifying elements inside an XML document, from where the name - XML Path. HTML, the language used in coding web pages, is a markup language, as the XML. Both languages present a hierarchical structure composed by tags, text and attributes, so that XPath represent a valid mean to identify elements in both languages.

If we examine again the HTML code example of Figure 2.6, we will notice that the same element, for example the button, can be addressed with different *xpaths*:

- */html[1]/body[1]/button[1]* - 'button' is child of the node 'body', which is child of 'html'.
- */html/body/button* - the previous expression emphasized that the button is the first 'button' child of the first 'body' child of the first 'html' tag. This expression selects all the button nodes at the defined level, but in this case the result is the same.
- */html//button* - selects all the buttons at any level after the 'html' tag.
- *//button[@text='button text']* - selects all the buttons with text 'button text', anywhere in the DOM.

These are examples of simple expressions one can build with the XPath language, but there are many other operators that can be used, and conditions that can be defined, to build all kinds of complicated *xpaths*.

2.4.2 Conditions

Many settings used by Crawljax employ some conditions, which must be verified during the crawling or influence its direction. There are two kinds of conditions:

- **Crawl conditions**, used by *Click When*, *Crawl*, *Invariant* and *Oracle Comparator* settings.
- **Expected conditions**, used by *Wait For* settings.

The settings will be explained in detail, but let us now examine the conditions.

Crawl conditions

Crawl conditions are conditions verified against the content of the page being crawled. For example, the JavaScript condition is an expression which gets executed in every page. If the expression returns true, the condition is considered true.

- JavaScript - true when the JavaScript expression is true.
- (Not) Regex - true when the regular expression does (not) match with some expression in the page.
- (Not) URL - true when the crawler is (not) visiting the defined URL.
- (Not) Visible - true when the specified element of the DOM is (not) visible.
- (Not) XPath - true when the element of the DOM specified by the *xpath* can (not) be reached.

Expected conditions

An expected condition is a condition that is not immediately true in the page, but it is expected to become true. In the condition we define an element, which is the expected element, through an *Identification* object.

- Element - true when the defined element gets loaded in the page.
- Visible - true when the defined element becomes visible in the page.

2.4.3 General Settings

These settings define general properties of Crawljax:

- Max Depth - maximum depth to reach during the crawling, defined as the number of events necessary to reach a certain state.
- Max States - maximum number of states to find.

- Max Runtime - maximum number of milliseconds to keep crawling.
- Click Default - click on all anchors and button elements of the DOM.
- Random Input - set random values inside input fields.
- Click Once - click only once on every element.
- Wait after event - milliseconds to wait after an event is fired.
- Wait after reload - milliseconds to wait after reloading an URL.
- Browser - the browser to use between Firefox, Internet Explorer, Chrome, a remote browser on a specified URL or the mock browser HTML Unit.

2.4.4 IFrame Settings

An IFrame is a HTML element used for displaying an external document inside a HTML page. The document is referenced inside the IFrame, avoiding duplication of content. Although at the moment Crawljax is not able to crawl IFrames, it presents two different IFrame settings. It is possible to disable the crawling of all IFrames, or it is possible to ignore specific IFrames. Every IFrame is defined by the 'id' attribute, passed as a string.

2.4.5 Thread Settings

Crawljax makes it possible to perform the crawling process in parallel on different browser instances and multiple threads. Through Crawljax thread configuration it is possible to set how many browser instances and threads to use. If the number of threads is not set, it is by default equal to the number of browsers, and vice versa. By default the number of browsers and threads to use is set to 1.

It is also possible to apply some more advanced settings:

- Browser booting - load the browsers in advance.
- Max Number of Creation Retries - maximum number of times Crawljax tries to open a browser.
- Sleep Time on Creation Failure - milliseconds to wait before trying to open a browser after a failure.

2.4.6 Input Settings

It is possible to add to the crawling one or more *input-value pairs*, in order to fill with a specific value the desired fields. The field must be of a HTML input type - text box, text

area, check box, radio button, select box, et cetera. The input can be identified by an 'id', which is a common HTML attribute. The possible values to insert are strings, representing the different options in the select box, except for check boxes and radio buttons, which take as input boolean values - true or 1 if checked, false or 0 otherwise. It is possible to set more than one value for the same field, so that all values will be inserted one after the other. If it is required to insert input in different fields before proceeding with an action, a *Set Values Before Click* setting (Section 2.4.7) should be used instead.

2.4.7 Click Settings

Crawljax is generally able to fire any kind of action on the elements of the DOM, but at the moment it is programmed to perform only click actions. There are five different kinds of *Click* settings in Crawljax. All of them define an element of the DOM to click, or not to click. This element is defined by an obligatory *tag*, and optional *text*, *xpath* and *attributes*. When only the tag is defined, all the elements sharing the tag will (not) be clicked. The possible *Click* settings are:

- Click - click an element of the DOM.
- Don't Click - do not click an element of the DOM.
- Set Values Before Click - set values in one or more input fields before clicking an element of the DOM. Input-value pairs are defined in the same way as input settings (Section 2.4.6).
- Click When - click an element of the DOM when all the defined conditions are true.
- Don't Click When - do not click an element of the DOM when all the defined conditions are true.

Conditions used in the *(Don't) Click When* settings are the same type of conditions used in the *Crawl* settings.

2.4.8 Crawl Settings

Crawl settings use crawl conditions to decide whether to crawl or not the newly found state. The settings need one condition, and zero or more preconditions. The preconditions are also *Crawl conditions*. Crawl and Invariant settings are defined in the same way.

2.4.9 Invariants

Invariants are settings that verify certain conditions on the entire visited website. If the condition is violated by some state during the crawling, Crawljax fires an Invariant violation

alert, which can be caught by specific plugins, as explained in Section 2.4.12. Crawl and Invariant settings are defined in the same way.

2.4.10 Wait For Settings

Crawljax can be set to wait on a certain (even partial) URL for one or more *Expected condition* to happen. An optional maximum waiting time can be specified in milliseconds.

2.4.11 Oracle Comparators

Oracle Comparators are a special type of settings, which allow to ignore specific parts of crawled pages. The analyzed DOMs are stripped off the desired attributes, and then compared. The simplified DOMs with equal results are grouped in a unique state. Zero or more *Crawl conditions* can be associated with a comparator. It is possible to use in Crawljax many type of comparators:

- Attribute - takes as input a list of HTML attributes to ignore.
- Date - ignores time and date regular expression patterns.
- Distance - applies Levenshtein Edit Distance and ignores differences above a given threshold.
- Plain Structure - strips the DOM elements of all attributes and content.
- Regex - ignores content matching the specified regular expressions.
- Script - removes all the 'script' elements from the DOM.
- Simple - eliminates whitespaces and linebreaks.
- Style - ignores style attributes.
- XPath - remove elements or attributes defined by XPath expressions.

Levenshtein Edit Distance

Levenshtein Edit Distance is a measure for comparing two strings. The number of changes needed to convert a string into another determines the distance, where the possible changes are insertion, removal or substitution. The specific implementation of Crawljax requires as input a threshold, a real number between 0 and 1. If 0, even completely different DOMs are considered the same; if 1, DOMs are considered the same only if they show no difference at all. This setting is a convenience for applying a general but effective comparison.

2.4.12 Plugins

Plugins are a mechanism to make Crawljax easily extendable. Crawljax can be executed with no plugins, but in that case there is no result, as plugins also collect the output. There are many kinds of plugins, each representing a point during Crawljax execution. When Crawljax finds itself at that point, it will fire the corresponding event to the attached plugins.

- Pre Crawling - called before loading the initial URL, returns the used browser instance.
- On New State - called when a new state is found.
- On Revisit State - called when the same state is revisited.
- On Invariant Violation - alerts of an invariant violation.
- On Browser Created - alerts when a new browser is started.
- On URL Load - called every time a URL is (re)loaded.
- On Fire Event Failed - warns if an event was fired unsuccessfully.
- Pre State Crawling - executed before crawling a state.
- Post Crawling - executed after the entire crawling is over.
- Proxy Server - associates to Crawljax a proxy and its settings.
- Guided Crawling - gives control to the plugin.
- Generates Output - makes it possible to set an output folder.

Figure 2.10 shows the working of plugins. The plugins are invoked at crucial moments of the crawling process - before loading the initial page, when loading a URL, when a new state is discovered or revisited, and at the end of the process. They can be used to collect data about the current state and the process itself, information that would not be saved by Crawljax. They can also be used to guide the crawler from inside, to intervene at specific moments.

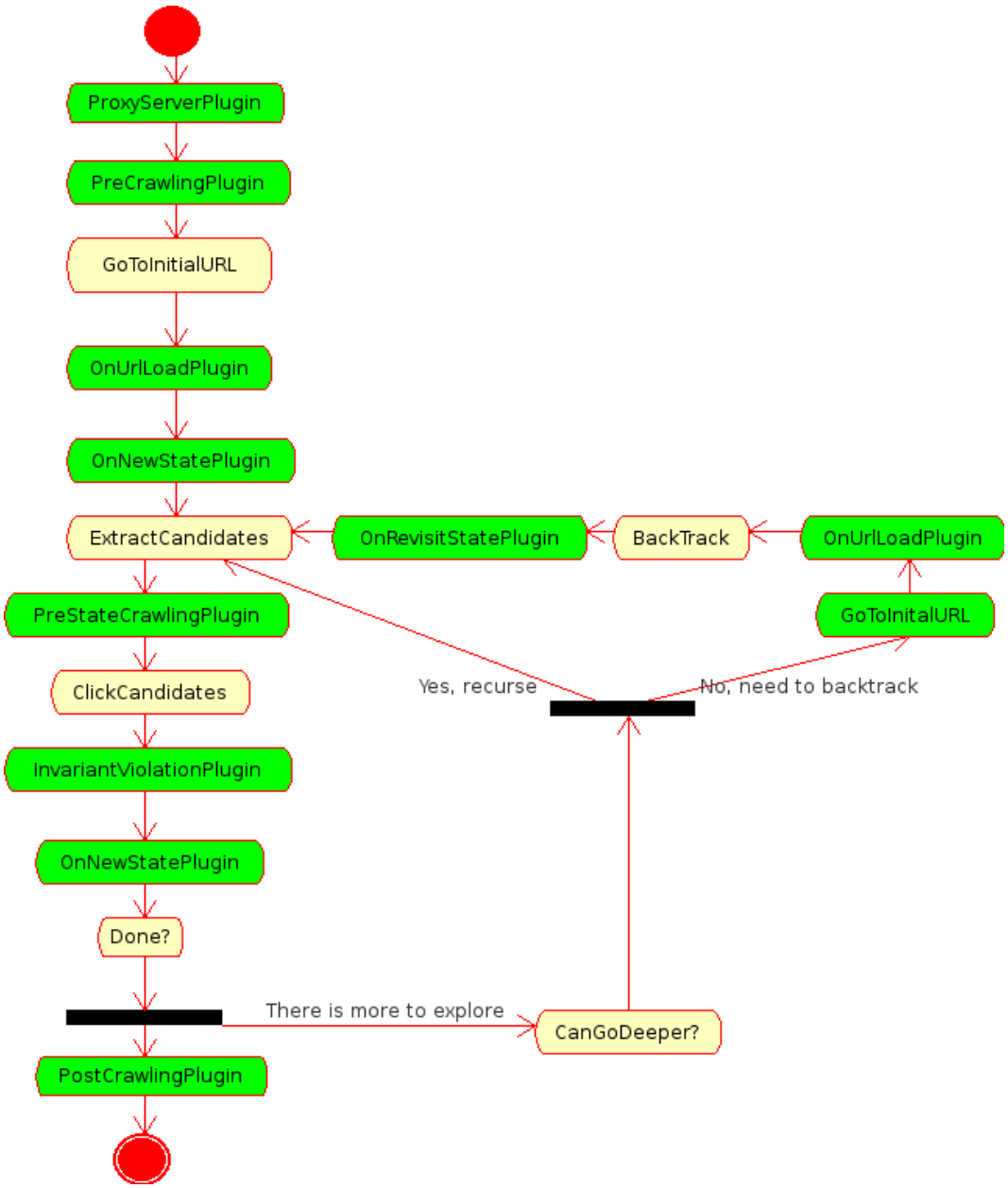

Figure 2.10: Crawljax plugins flow, as showed in [2].

Chapter 3

Requirements

The project is an extension of Crawljax, so a logical requirement is that it must allow access to all possibilities Crawljax has to offer, while adding its own twist. Section 2.3 described settings and features of Crawljax. The porting of Crawljax may not be a straightforward process, meaning that not all characteristics of the application could be directly portable into the project. This chapter describes what is necessary for making the project a satisfactory extension on Cralwjax, and what is needed to make the project a success, while the next chapter deals with the approach we followed.

3.1 Requirements

3.1.1 Crawljax Features

The project generally requires to cover all characteristics descibed in Section 2.3. The structure of Crawlax is defined through Java classes, organized in libraries which can be easily imported into a Java project. We decided to develop CrawlMan in Java, using the Eclipse IDE [27] and Google Web Toolkit [15]. As these environments are based on the principles of extensibility and compatibility, extending Crawljax seems not to pose specific threats to the programmer. Still we need to analyze what features of Crawljax make sense in a web environment.

In order to properly set up Crawljax, the project must allow element identification (Section 2.4.1), creation of the different types of conditions (Section 2.4.2), insertion of general (Section 2.4.3) and advanced settings. To keep the project extensible, we need to take advantage of the plugin structure (Section 2.4.12). Some of these requirements deserve a deeper investigation.

Thread Settings

Of all the settings, the thread settings impose closer attention, for various reasons. The first intuitive reason is that running Crawljax in multiple threads or browser instances means that the number of resources employed by a single user is multiplied, as well as the size of the result. Furthermore, in case different results are produced, all of them must be transmitted and showed to the user. The use of more resources can be justified only if it creates a valuable addition, as it does in the desktop environment for which it was ideated.

The second reason is that the thread settings allow crawling a website in parallel. This project represents a web extension of Crawljax. A user navigating CrawlMan, as explained in Section 2.4.3, is immediately able to crawl a website, by sending a request from his browser to the CrawlMan server. If the user wants to have two crawling sessions of the same website in parallel, what he can do is simply open CrawlMan in two different tabs on his browser and send two requests at the same time. A basic principle of a web server is in fact to satisfy multiple user requests in parallel.

A third reason is that thread settings allow the user to adjust the behavior of Crawljax when the crawling process is failing. One thing that the user expects when navigating a web application as CrawlMan is that the application does not crash while he is using it. The application must be reliable, and in case of a failure it must report an intelligible message. If the crawling process fails or is not smoothly performed, and the problem can be fixed by simply set some properties, the adjustment should be done automatically by the application. In our opinion the listed reasons are enough to exclude the thread settings from CrawlMan.

IFrame Settings

Crawljax offers settings to ignore eventual IFrames inside a website, but in fact Crawljax is not able to crawl IFrames at the moment. For the sake of future compatibility, IFrames settings are implemented in CrawlMan, but they are not presented to the user, as it would only generate confusion.

Invariant Settings

Invariants pose an indirect problem, as the result of Invariants is collected through *On Invariant Failure* plugins (Section 2.4.12). Using Invariant settings by itself does not present particular issues, but presenting the result of Invariant settings depends on the implementation of the plugin structure into CrawlMan, discussed in the next section.

Plugins

The Crawljax plugin organization makes it easy to extend Crawljax without touching its core libraries. There is no need of directly modifying Crawljax when a programmer uses

the plugins, as they can control the crawling, collect a result, report failures on *Invariants* and fired events, and more (Section 2.4.12). Thanks to the plugin structure, Crawljax can be extended, but the project can still benefit of every update to Crawljax, as it uses its libraries with no modifications, just additions. This is the reason why it is required that CrawlMan takes advange of the plugins.

An additional aspect of plugins is the fact that not only CrawlMan, but many extensions of Crawljax are in the form of plugins, for the same reason of mantaining compatibility. This fact hints that it should be possible to easily adapt existing plugins to the project, further extending CrawlMan. Implementing existing plugins is not a project requirement, but a significant possibility to consider.

3.1.2 Project Development

Some constraints apply to the project development. The project must be built using the Java programming language, as Crawljax libraries are written in Java. The project must be publishable on a web server, for online access. There are no constraints on the programming environment to employ, but Eclipse [27] and Google Web Toolkit [15] represent almost forced choices for their popularity, versatility and ease of use. An additional requirement, to improve portability and ease of compilation, is that the project must use Maven [28].

Maven

Maven is a tool which facilitates build and dependency management. It is executed with console commands, or it can be integrated in a development environment as Eclipse. The commands perform compilation, installation and generally build of projects. Directions for the tool are gathered in a Project Object Model (POM) file, which also describes the project structure, necessary libraries, authors and other information. When executing a build, Maven connects to a central Internet repository and downloads the specified libraries, automatically solving project dependencies. Thanks to Maven, a project can be transferred by copying its source files and ignoring the space consuming libraries, which will be downloaded directly on the computer where they are needed. Lastly Maven projects employ a common structure, separating source code from tests, resources and compiled units. The main project folder contains the POM file, the folder ‘src’ with the source code and the folder ‘target’ with the compiled units.

3.1.3 CrawlMan Features

The first requirement of the project is that the user must be able to insert a URL, send a crawling request and reliably receive a useful result in a finite time. The second requirement of the project is that CrawlMan must be easy to use and set, offering ready to use settings and suggestions to apply. The third but most important requirement is that the project must

innovate Crawljax, using its elements in an original way to help the user into obtaining the result he wants.

Settings and Result

CrawlMan must be able to apply all the required settings. Settings must be simple to create, modify and delete. The user needs to be able to understand the settings and easily check them, manually and automatically.

The result should be understandable and valuable to the user. The user is not able to see the guided browser performing the crawling, as in the case of a desktop application, so the recognized states need to be described exhaustively. The user needs to understand the behavior of the crawler, and the reasons behind it. In case the crawling fails, the application is required to show a message describing the error, in order for the user to correct the settings.

The user needs to be able to save his settings and result, to stop his crawling session and restart whenever he wants.

Settings Refinement

The main focus of this project is facilitating the process of settings refinement. It is not obvious when first looking at Crawljax what the meaning and purpose are of many of the settings it proposes. It is much easier to understand something when a good example is proposed. CrawlMan presents suggested settings using elements of the result, so that the user has a direct image of what is happening in the application. Then a report on the result is presented, telling the user how the settings did perform. New settings can be continuously added, progressively changing the result until the user obtains what he wants. CrawlMan's main requirement is allowing the exchange of data between settings and result, each influencing the other, in a way transparent to the user. Crawling can be performed again and again, without reloading or recompilation, speeding up the job of the user.

3.2 Scenarios

In this section we describe the possibilities of CrawlMan and what it has to offer to the user. Crawljax relies on the user to apply correct settings. It is in fact impossible for the application to know if the user wants to apply general settings, or if instead the settings are incomplete, because the way of crawling finally depends from the user's will. For example, the user could set the crawler to click on all buttons it finds, but he may be only interested in clicking on one specific button in a specific page. What CrawlMan can do is guiding the user into refining the settings, making the user conscious of what happens during the crawling process and using the elements of the result for suggestions. When inspecting the

result, the user checks if the element he wanted to click has been clicked, and selects the appropriate suggested setting for refinement.

3.2.1 Basic Scenario

The user must be able to insert a URL, send a crawling request and reliably receive a result. The basic requirement of the application is to work as an interface between the user and Crawljax. The user must be able to set Crawljax and examine its work, as if he had direct access to Crawljax.

3.2.2 Default Crawling

The application offers default settings to start crawling immediately. Using Crawljax's general settings (Section 2.4.3), Crawljax can click default elements such as buttons and anchors, and return a small result in a limited amount of time. After inspecting the result, the user can then refine the crawling, applying suggested settings or creating his own. As explained later in Section 6.3.1, Crawljax's *Click Default* setting functionality was substituted with ad hoc *Click* settings to comply with user expectations.

3.2.3 Advanced Crawling

The user can define all kinds of advanced settings, as *Click*, *Wait For*, *Crawl*, *Invariants* and *Oracle Comparators*. The user is guided in the definition of these settings by selecting suggestions presented using the result of the last crawling session. The suggestions cannot modify existing settings: it is up to the user to recognize the suggestions as more complete versions of the applied settings, and eventually substitute them.

3.2.4 Result Inspection

The result is presented to the user as a state tree graph. The root of the tree is the page corresponding to the initial address. The other nodes represent states of the visited application, found by comparing DOMs after firing the selected events.

The application must report failed invariants and fired events, and general application failure. The information related to every state needs to be present, as the URL, DOM source code and a recognizable image. The events leading to every state must be clearly reported.

3.2.5 Settings Suggestions

Clickables and other data found during the crawling are used to suggest settings around the application, especially in the result. As the user inspects the result, the events that lead to a

certain state are visible, so that the user can choose specific events to become permanent part of the settings. There is no assurance that the event will be repeated on the same element, as in the next session the crawler could find an element with the same characteristics. This is why the suggested setting automatically reports all the possible characteristics of the element.

Settings can be expanded with conditions. A simple suggested condition is for example a *URL* condition, if the URL of the selected state is different from the URL of the root. Conditions must be created independently from the settings, so that they can be used multiple times.

Chapter 4

Approach

This chapter defines conceptual challenges and solutions of the project. There are problems inherent to the development, solved by adopting certain programming strategies. Other problems, dependent on the final utilization of the project, rely for their solution on the main project contribution - using the result for setting refinement.

4.1 High Level Challenges

This section discusses challenges and issues presented by the project. Guidelines of CrawlMan are here resumed:

- All settings and result data must be accessible to the user in a clear way and at every moment.
- Usage and processing of the crawler must be transparent to the user.
- The application must accompany the user during result inspection and settings refinement, facilitating the user's task.

4.1.1 Main Project Challenge

CrawlMan must represent a valid, reliable test framework for web developers and testers. It is not sufficient for it to be an interface to Crawljax. It must allow dynamic setting, verification of the result and reliable execution. The user must be able to save his work and start back where he left. Most importantly, the application must provide valuable feedback to the user.

4.1.2 Interface Friendliness

Click settings can be considered easy to understand for the average user. The concept is that they allow to click on all the elements corresponding to a specific description. The same cannot be applied to the rest of the settings, so that the interface has to be clear and user-friendly. The challenge here is to make CrawlMan sufficiently easy to use and understand that a user who does not know Java, HTML, XPath and has never used a crawler before can successfully crawl and push the crawling in the desired direction. This is not a simple challenge, because it requires not only the expertise of a Java programmer but also the skills of a Web designer. The design of CrawlMan's interface must be focused on the user's experience, smoothening the user's interaction. Much of the knowledge needed for using Crawljax must be conveyed through graphical means inside CrawlMan - we do not want to force the user to read long manuals, but to start crawling immediately.

4.1.3 Project Modularity

It is important to keep the components of CrawlMan separated, to improve maintainability and expandability. This is necessary because the project is an extension on Crawljax, which is a growing project itself. By taking advantage of plugins instead of directly modifying its libraries we can easily substitute Crawljax libraries in the event of new version releases. The adoption of Maven [28] also helps in this matter, and also in enhancing modularity by enforcing a solid project structure.

While Maven helps in keeping source code, tests, resources and documents separated, Google Web Toolkit (GWT) [15] enforces a separation between server and client code. Client code in GWT is all the Java code which can be transformed in HTML and JavaScript, to be displayed on the browser. The requirement poses constraints on what can be programmed on the client side - only Java constructs compatible with GWT can be used, only GWT compatible libraries can be inherited, and data passed through client and server must be serializable. Crawljax and its libraries are not GWT compatible, so they cannot be directly inherited into the client side of CrawlMan. One problem is that it is not possible to send Crawljax results directly to the user, but that they must be converted to be compatible. The same holds for the settings, so the interface must represent a complete layer between the user and Crawljax.

4.2 Conceptual Solutions

Using the crawling result for guiding the user into refining the settings is the main concept of CrawlMan. The result of a crawling session is a graph of subsequent states assumed by the inspected website. CrawlMan does not report the raw collected data, but organizes them in an ordered graphical representation. The user visualizes a tree graph of nodes as the result, and he can select every node to show URL, DOM, events leading to the corresponding state, and other important data. By directly acting on the tree, the user can select states he

does (not) want to appear in the next crawling process, and generate according settings. As these settings reflect the will of the user, they help him understanding the general working of CrawlMan, and to perfect the settings. As they are naturally reflected in the result data, the user has an immediate feedback on the applied changes. The implementation of this mechanism resides in a set of suggestions distributed throughout the application and especially in the result, as explained in the next Chapter.

4.3 Conceptual Contributions

At the moment, there is no software tool or testing environment on the market comparable to CrawlMan. This project is in fact the first testing tool for AJAX pages, and websites in general, not to require installation, programming skills and crawling experience. More importantly, it is the only one making use of an AJAX crawler to automate web exploration and reconstruct the structure of a web site. As other tools (Section 7.2), CrawlMan offers automatic creation of test scripts, but the original contribution is in using the data collected during the test to improve the test itself. In this way it realizes an Automated Exploratory Testing process, which is not offered by any other web testing tool. The value of this contribution will be defined in the evaluation of Chapter 6, but the real success of this project will consist in its actual use in the web application testing world.

Chapter 5

Implementation

5.1 Interface Characteristics

The interface is formed by panels, divided in four categories:

- Main panel, to send the crawling request.
- Settings panels, for setting creation and editing.
- Result panel, for result inspection.
- Help panels, displaying specific information about the application.

Figure 5.1 shows the main panel - CrawlMan's homepage.

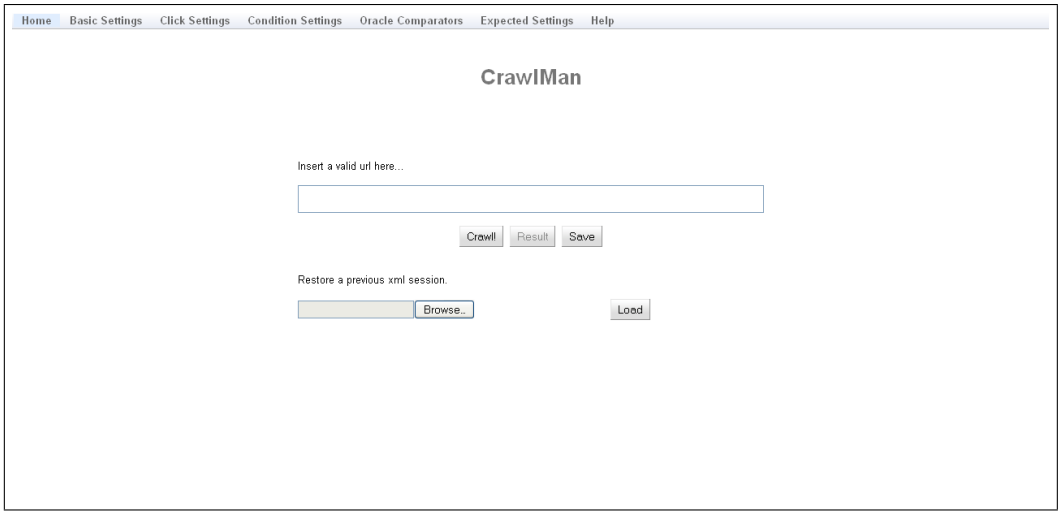

Figure 5.1: CrawlMan's homepage.

5.1.1 Immediate Crawling

The web application consents immediate crawling with default settings, so that a user can insert a URL and simply start crawling. The main panel basically consists of a text box to insert the URL, and a button to send the request. The default settings, modifiable in the 'Basic Settings' panel shown in Figure 5.2, include:

- Max States equal to 6.
- Max Runtime of 300 seconds, 30.000 milliseconds.
- HTMLUnit browser, which does not require a real browser.
- Click Default, for firing events on general clickables.
- Random Input, to fill the possible input fields.

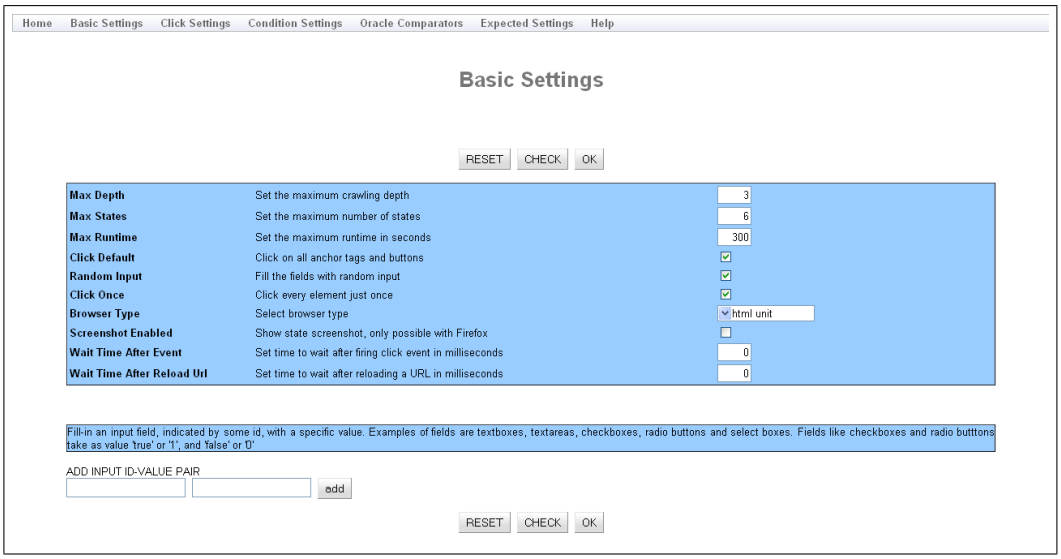

Figure 5.2: CrawlMan's 'Basic Settings' page.

The first three settings serve for producing a result in the shortest time possible. HTML Unit does not require the starting of a browser and content visualization, so that the crawling is substantially faster than using a real browser. The user is advised to mantain these initial settings, and then inspect the result to refine the crawling. When a request is sent, a clockdown counter starts, set on the *Max Runtime*. The duration of the process can be shorter, but not longer, so the user knows if something went wrong - connection loss, server or client problems.

The last two settings allow the crawler to find and click anchors and buttons, and to find and fill with random input all possible fields. An aspect of Crawljax is that it finds all the

fields and clickables it is programmed to find, so without the last two settings there would be no event producing new states, no navigation in the website. The last two settings can be then substituted by the user with *Click* and *Input* settings.

5.1.2 Setting Creation

Settings are organized in different panels, in order to avoid confusion to the user. In fact a basic user, not interested in advanced settings and deep testing, does not need the whole extension of settings Crawljax can offer, for example Invariants. The panels respect the organization depicted in Section 2.3.

All settings can be independently created. As conditions may be reused by more than one setting, their creation is as well independent from settings. At any moment the user can define and edit settings and conditions in the settings panel, after crawling he can also accept the suggested settings in the settings panels and principally in the result panel. The suggestions in the settings panels take the form of descriptions of HTML elements of the crawled pages, as shown in Figure 6.17. The suggestions in the result panel are instead ready to use settings, relevant to the selected node of the result tree. For example, the user may include or exclude the selected state from the next crawling session, which results in the creation of *Click* or *Don't Click* settings for clickables that generated the event bringing the crawler in that specific state. The user can then review the *Click* settings in the 'Click Settings' panel, shown in Figure 5.3.

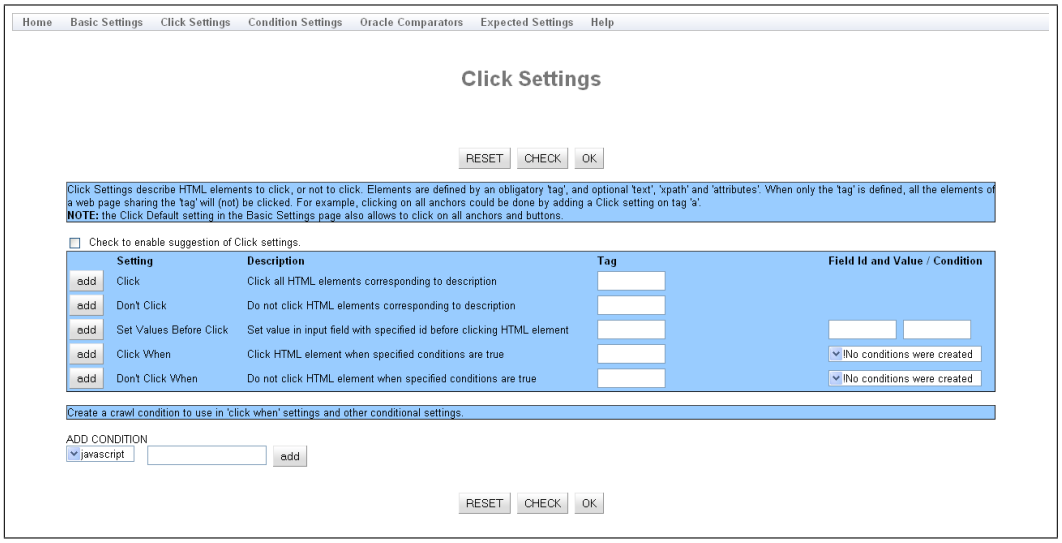

Figure 5.3: CrawlMan's 'Click Settings' page.

5.1.3 Result Editing

The result of the crawling process is a directed graph, whose nodes are the states reached by the crawler. We decided to render the graph as an organizational chart, a class offered by Google Web Toolkit [15] which offers some means of graphical interaction. The graph appears to the user as a tree of states, which can be selected to inspect their data (Chapter 6). By selecting a state, the following information is showed:

- URL address relative to the state.
- HTML DOM representation of the state.
- A screenshot of the state, in case CrawlMan was set to use Firefox in the 'Basic Settings' page.
- A set of **Incoming Edges**, describing click and input field events which brought into the state.
- A set of **Outgoing Edges**, describing click and input field events which brought outside of the state.
- A set of **Other Edges**, representing clickables which were not clicked or did not fire an event when clicked.

This information, collected during the crawling, gives the user a faithful representation of the session. It is also enough for creating and displaying the suggestions the next section describes. Some of the suggestions can avoid the entering of a certain state, so that the state is "eliminated" from the crawling. When the user accepts this kind of suggestions, the elimination is graphically showed in the displayed tree. In this way the creation of new settings is experienced as a form of result editing, more intuitive for the user.

5.1.4 Offering Crawling Suggestions

This section defines the actual features of CrawlMan representing the project contribution. The application mainly wants to suggest possible new settings to the user, or settings that require few adaptations to express the user's will. The features can be fundamentally divided into four groups:

- Suggestions of *Click* settings.
- Suggestions of conditions.
- Suggestions of *Oracle Comparators*.
- Application utilities.

The suggested settings benefit from the automatized processing of the result. They are settings that a user could create, but it would take time for the user to collect such specific information, such as the *attributes* and *xpath* of a HTML element. Suggested settings are shown to the user during the result inspection, when the user selects a node of the result tree. In the case of settings which can contain conditions, the innovation consists of suggesting a setting with conditions relative to the selected state, so that the user has a clear idea of the moment of the crawling in which that condition is true. As the condition yields true whenever the description fits, it is the user's task to verify the condition is met only in the desired cases. The proposed condition can be enforced by adding more than one condition to the setting.

By converting the result in GWT-compatible classes (Section 5.2.1), the suggestions can be processed on the client side, lightening the weight of the result, the need of bandwidth and the use of precious server resources.

Click Suggestions

During the inspection of the result, a user of CrawlMan can select a state of the application to see which elements where clicked to enter the state and from the state itself. The descriptions of these elements are presented together with buttons allowing to apply *Click* settings relative to the elements. In this way the user adds new settings by confirming the crawler actions, meaning he wants these actions to be repeated at the next crawling request. The presented click suggestions are:

- Suggestions on the reachability of a state, which can be included or excluded from next session with *(Don't) Click* settings.
- Suggestions of *(Don't) Click* settings, formed with the data of the elements clicked during the crawling session. The feature takes advantage of the *Click Default* setting to click on all the links and buttons, and the fact that Crawljax can click on elements respecting a general description, for example all the elements with tag 'img'.
- Suggestions of *Set Values Before Click* settings, using input data inserted during the session. The feature takes advantage of the *Random Input* setting of Crawljax to insert random data in all the possible fields.

Condition Suggestions

Many settings of Crawljax take as input conditions of different genres (Section 2.4.2). When a user selects a state from CrawlMan's result, the elements of the result are processed to generate a number of conditions relative to the selected state. The conditions can be immediately used to add *(Don't)Click When*, *Crawl*, *Invariant* and *Wait For* settings. They can also be optionally associated with *Oracle Comparators*. The suggested conditions are:

- (Not) URL conditions, when the URL of a state differs from the URL of the root in the state tree graph.
- (Not) Visible and Element conditions, using the data of the elements retrieved during the session, such as *id*, *name* and *text*.
- (Not) XPath conditions, composed with the *xpaths* of the retrieved elements.

The generated settings can be reviewed in their respective panels. Figure 5.4 shows the 'Condition Settings' page, where it is possible to modify *Crawl* and *Invariant* settings and to create new conditions.

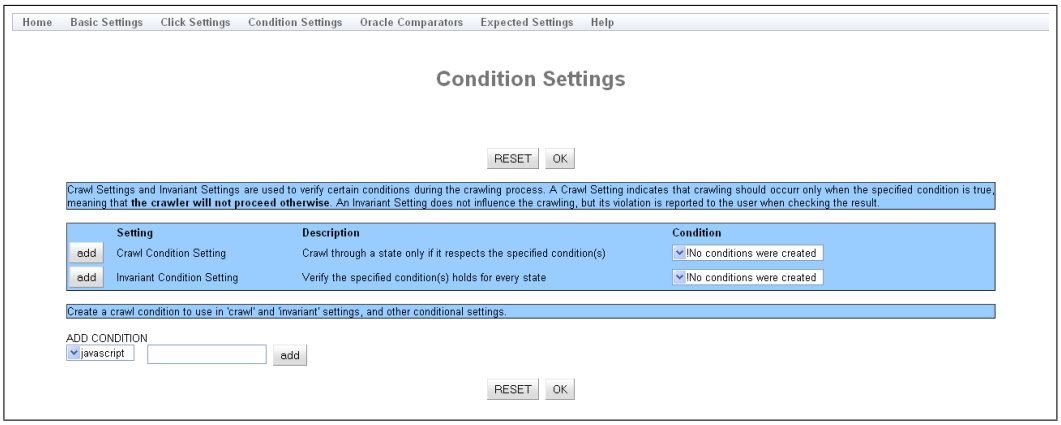

Figure 5.4: CrawlMan's 'Condition Settings' page.

Oracle Comparator Suggestions

Some of the data which is used to suggest conditions is also used for creating Oracle Comparator suggestions. In addition, an analysis of the selected state's DOM is performed to verify the presence of date patterns or 'script' elements. If these are detected, the corresponding Comparators are suggested. The possible suggestions are:

- Attribute Oracle Comparators, using attributes of the retrieved elements.
- Date Oracle Comparators, when the date regular expression pattern appears in the DOM.
- Script Oracle Comparators, when the 'script' tag appears in the DOM.
- XPath Oracle Comparators, using *xpaths* of the elements retrieved during the session. The user only has to select the *xpath* of an element to ignore.

A date pattern is detected when the state presents some text describing a date or a time, as in the case of a page hosting a digital clock. Figure 5.5 shows the 'Oracle Comparators' page, where the user can review the generated Comparators and create new ones.

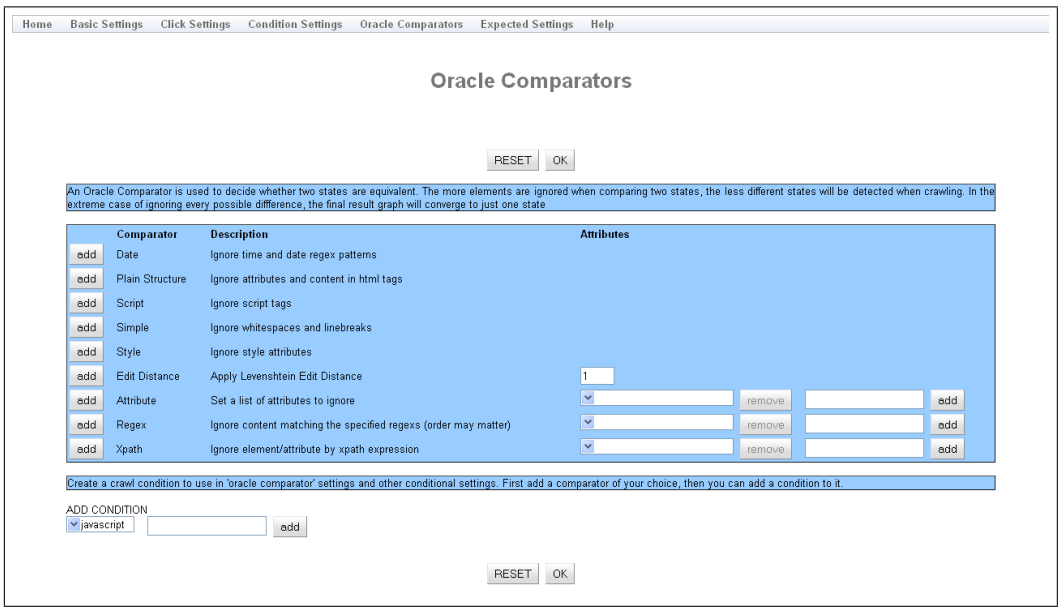

Figure 5.5: CrawlMan's 'Oracle Comparators' page.

Application Utilities

The following features are needed to simplify the usage of CrawlMan. They present to the user feedback on the settings and crawling result, and allow for session saving and reloading.

- Settings coherence common problems verification.
- Result report based on settings confrontation and collection of plugin data.
- Saving settings and crawling result for later (re)use.
- Loading settings and crawling result from a previous session.

The first feature warns the user when two or more *Click* settings reference the same element. The settings are compared on the base of tag, text, attributes and xpath. The analysis is performed between settings of the same type, for example all the *Set Values Before Click* settings, and settings of different type, for example *Click* and *Don't Click* settings. The warnings are displayed in red above the settings descriptions in the 'Click Settings' page.

The second feature performs a comparison between the defined *Click* settings and the clickables detected during the crawling session. It verifies that the elements defined by *Click*, *Set Values Before Click* and *Click When* settings appear in the crawling result, by comparing their tag, text, attributes and xpath with the crawling result clickables data. It also verifies that the elements defined by *Don't Click* and *Don't Click When* settings do not appear in the crawling result. The report is presented to the user in CrawlMan's 'Result' page.

The session saving and loading features are necessary for the user to interrupt and resume his work, and to share it with other users. We refer to Section 5.2.1 for a discussion of these features.

5.2 Technical Solutions

This section presents technical problems encountered during the development of CrawlMan and the solutions that were applied. Some of the issues are relative to the technologies involved, while some are dependent on the Crawljax implementation.

5.2.1 Google Web Toolkit

We decided to build CrawlMan as a web application, using Java and Google Web Toolkit (GWT) [15]. In order to present the result of Crawljax to users, the produced data had to be converted in a format transmittable through **Remote Procedure Call** (RPC) services, which is the way GWT implements asynchronous server requests and responses. All data carried by RPC services must be serializable, so that it can be converted into JavaScript by GWT. The nature of RPC services raised problems in three cases, involving data which cannot be serialized:

- Saving settings and result in XML format.
- Loading settings and result of a previous session from an XML file.
- Showing to the user screenshots captured during the crawling.

Saving and Loading User Sessions

A user session is composed of user created settings and the corresponding result. We needed a way to save the user session, so that the work of the user could be interrupted and resumed at any moment. Secondarily, the saved data had to be easily organizable, and we considered it a plus if the data could also be shared among users. At a first moment an attempt was made to create a database in the same machine hosting the server. Saving the user session in a database would have solved all the previous problems, but it would have added a great amount of complexity to the project:

- Tables and classes corresponding to the data had to be created.
- Access to a database would have required user log-in, so that only an authenticated user could save his session.
- Limitations on the amount of space conceded to every user had to be taken into account.
- An interface to let the user organize his data had to be added.
- Classes and functions to connect to the database had to be implemented.

A second solution, avoiding this increase in complexity, was found - letting the user save the session data in a file. The XML format, easy for conversion of serializable objects and simple for users to understand, was chosen. The solution is not devoid of problems itself. First of all, changes in the classes representing settings and result can make an old session incompatible. Once a structure for the user session is established, it should not be modified for future versions of the project. Furthermore, files are not conveyable through RPC services. To send an XML file from the server to the client and vice versa a conventional HTTP service must be used. This means that the data must be first sent to the server through an RPC service. When the server responds, a HTTP request is sent, so that the XML file can be transmitted along the HTTP response. The XML file is downloaded inside a hidden frame, avoiding the opening of a new window, which would not communicate an application feel and which is usually hindered by automatic popup blockers. The data is saved on the user session in the server between the calls, ensuring the data is deleted when the user finishes his work.

Screenshot Transmission

Crawljax is able to capture a screenshot for every new state in PNG format when crawling through an instance of Mozilla Firefox. As in the previous case, PNG files are not serializable, so they cannot be transmitted through RPC services. Screenshots are part of the result, so that they must be sent together with the rest of the data back to the user browser.

A way was found to convert an image into a serializable object, in the specific case a string in **Base64**, which can be saved into the result, sent through an RPC service and saved in an XML file. The string is then set as the URL field of an Image object, added into the Result page. Given the natural properties of Image HTML elements, the string is automatically converted into the corresponding PNG representation by the user browser.

5.2.2 Parallel Crawling

Although Crawljax is distributed as a Java library to be included in any Java project, it had never been used in a web application before. CrawlMan is the first project to run Crawljax

in a web server, and even if no specific problems were initially foreseen, one problem was encountered. Crawljax has been created as a desktop application, intended to be employed by one user at a time. There are settings (Section 2.4.5) which allow to crawl using different threads and different browser instances at the same time, providing parallel crawling. When using multiple threads, added plugins are shared through a static class. This aspect of Crawljax results to be a problem when the server is required to process different crawling requests, as we do not want the plugins to be shared through different threads. Every request must be serviced independently, because the data collected through plugins must produce independent result. To solve the issue, the static class, namely *CrawljaxPluginsUtil*, had to be changed to a class which can be instantiated. We refer to Section B.2 for the technical details of this operation.

These modifications can hinder the possibility of Crawljax to use multiple threads at the same time, but this is not important in the case of our project, where parallel crawling is performed by sending multiple requests to the server at the same time. This is the reason why the modifications were not included in the standard version of Crawljax, although they are necessary for CrawlMan to work. The decision caused the project to proceed using a special version of Crawljax. It can be possible to go back using Crawljax standard version in the future if the modification will be incorporated in Crawljax trunk.

5.2.3 Browser Loading Exception

As we explained, Crawljax was originally meant for desktop utilization. As such, reliability of the software was not the main goal, as an error would simply result in the interruption of the crawling process. The same cannot be said when using Crawljax in a web environment. An error during the crawling process must give control back to the primary interface, warning the user of the problem.

When the requested browser type cannot be loaded by Crawljax, an exception is generated. This can happen in various situations - the browser not being present in the system, the port for communication with the browser being locked, system resources being over exploited. In the case an exception is generated, the main Crawljax engine is not getting halted. The original code was modified to give back control to CrawlMan, by stopping the engine in case an exception is generated. The modifications are detailed in Section B.3.

5.2.4 Data Presentation

Crawling a website can generate a huge amount of data. Let us just consider the number of links in a single web page - during a simple test on Wikipedia [13], a page was found to contain more than two thousands different links. Imagine showing the result of a crawling session inside a single page. Just showing all the links in a page would take much of a client system resources. In order to limit the consumption of client resources, we decided to give the user the possibility to disable the generation of crawl conditions suggestions and the

capture of screenshots. Detected clickables which did not generate new states are showed ten at a time, not to clog the 'Result' page.

5.2.5 Click Default Setting

The *Click Default* setting is used to fire click actions on all anchors and buttons. When enabled, the setting has priority on other user-defined settings. When the user chooses a specific link to be clicked during the crawling, if *Click Default* is enabled, all the links preceding that one in the page will be clicked before it. As the user-defined *Click* settings are more important than default clickables, we decided to substitute the *Click Default* setting with the actual corresponding *Click* settings. Crawljax settings are executed in the order they are defined, so by adding the new settings after the user-defined ones, it is possible to make the crawler click on specific elements before turning to the default clickables. The solution was implemented to make Crawljax behavior more respectful of user expectations.

Chapter 6

Evaluation

We want CrawlMan to be a tool for testing web applications that should be easy to use for anyone. We investigate here the factors indicating if the purpose of this project has been reached. Specifically, it must be possible for the user to:

- Program CrawlMan to follow a desired behavior.
- Understand when and why the application is not following the desired behavior.
- Understand when and why the crawling process is possibly failing.
- Easily modify settings to obtain the desired result.

A challenge when testing web applications is reproducing real use case scenarios, ie., examples of utilization of a website as a final user would do. Is it possible for a CrawlMan user to set the application to execute a specific series of actions, as in a Script or a Recorder tool (Section 7.2)? CrawlMan can be used for simple tests such as clicking on every link of a web page, but this kind of tests can provide little information on the working of complex web applications. The user must be able to specify a behavior and verify it gets followed.

6.1 Pilot Tests

This section reports a set of preliminary experiments carried out to understand the challenges presented by using CrawlMan, and what aspects should be exercised during its evaluation. The tests were manually performed, as a normal user would employ CrawlMan. The experiments were carried out using Mozilla Firefox version 3.6.13.

6.1.1 Logging in to Facebook

Facebook [11] is one of the best known contemporary dynamic web applications [29, 30]. It is a social web application, connecting users through "friend subscription". Crawling Facebook and understanding the result represents a challenge by itself. The question arose if it was possible and easy to log-in into an application, using some predefined username and password. The test was performed on Facebook, being an iconic web application that many people know and use, with the intent of expanding the test to a use case for CrawlMan evaluation. The idea was discarded for the simple fact that Facebook does not allow creation of more than one personal profile, but the test resulted in a useful demonstration of the possibility of CrawlMan to log-in into a web application.

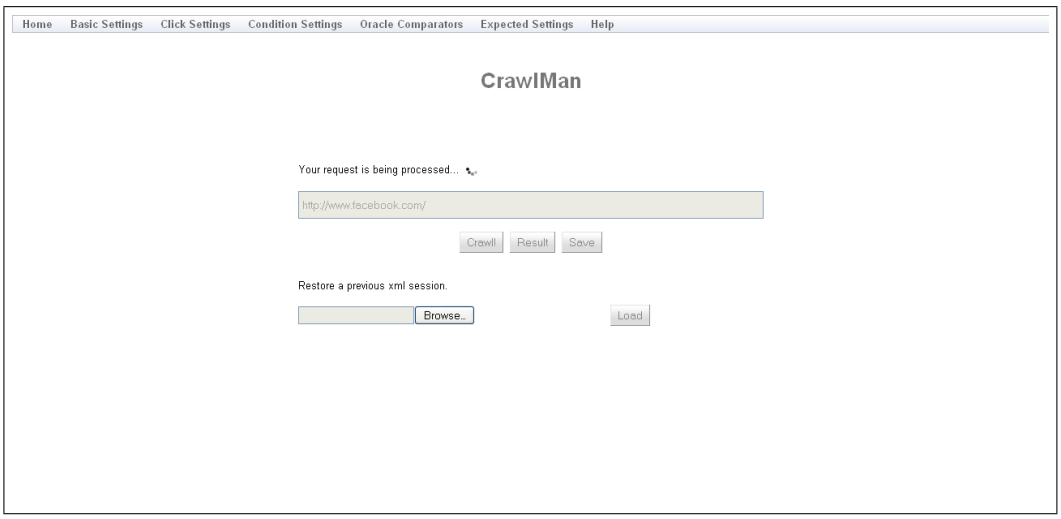

Figure 6.1: CrawlMan's homepage during the crawling process.

Figure 6.1 shows CrawlMan's homepage during the crawling process. After inserting Facebook's URL in the application and performing the crawling with default settings, as advised to final users, the result was inspected. Figure 6.2 shows the result for the crawling request. Using the presented suggestions it was easy to create a *Set Values Before Click* setting, to fill username and password fields and click on the log-in button. The *Click Default* and *Random Input* settings were then disabled, to make CrawlMan execute just the desired action. The crawling was performed again, expecting the crawler to log-in into Facebook. The test failed as the log-in button was not found.

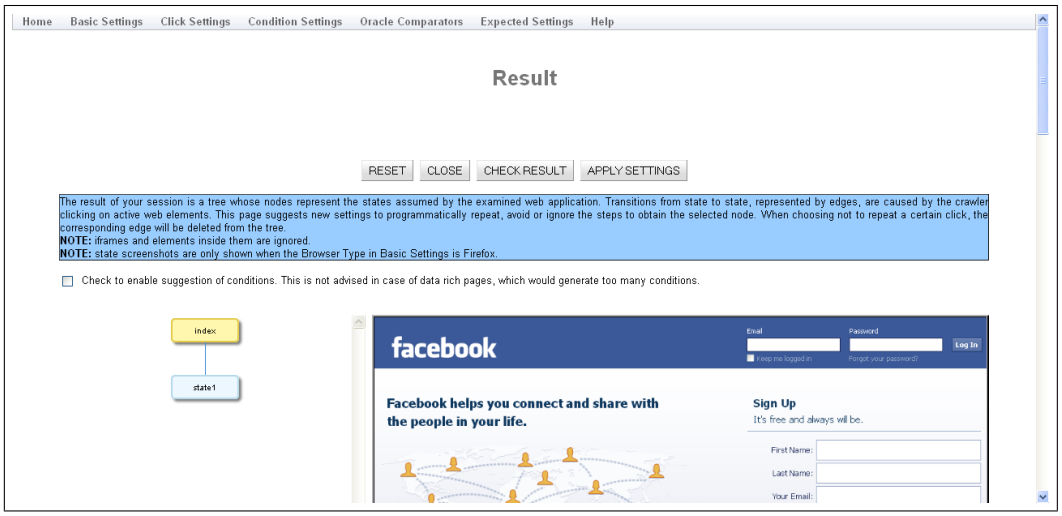

Figure 6.2: CrawlMan's 'Result' page for URL http://www.facebook.com.

What went wrong in these simple steps? The description of the log-in button in the *Set Values Before Click* setting did not correspond to the actual button inside Facebook's home-page. Upon inspection of the setting, we found that the definition of the button included a HTML attribute, 'id', which was dynamically generated at every loading of the page. After removing this attribute from the *Set Values Before Click* setting, the test was repeated and the log-in was successful. Figure 6.3 shows the code corresponding to the log-in button.

```
<input value="Log In" tabindex="4" type="submit"
  id="u174114_3" />
```

Figure 6.3: HTML code for Facebook's log-in button.

The test shows that it is up to the user to adjust CrawlMan settings in the most opportune way. The project cannot automatically point out too strict definitions, which cause test failure in dynamic environments as the one presented. CrawlMan can suggest settings to the user, but it is up to the user to adjust them to his needs. This test also demonstrates that dynamic application testing needs a certain degree of freedom in the test specification to actually work.

6.1.2 Sending Mail from Gmail

Gmail [12] is the Google free email service, with more than one hundred and fifty millions users. Sending one email from a test account to a defined address is a complicated task,

requiring first to sign in to the service, then load the email page, and finally set the fields with the right data to send the email. This test is intended to give us a good idea of the difficulties a user encounters when reproducing a complex behavior.

Log In

The first thing to do when reproducing a behavior is manually executing it. The address to load, from where is possible to sign in to Gmail, is `http://mail.google.com/mail`.

The page presents a number of informative and help links. We are interested in the 'Username' and 'Password' fields, and the 'Sign in' button. In order to log in, we must use a *Set Values Before Click* setting describing these three elements. Now that we familiarized ourselves with the the page, we proceed with some default crawling. Gmail's URL is inserted in CrawlMan's homepage to start crawling. The *Click Default* setting should automatically reveal buttons and fields, so that we can use the result suggestions instead of manually building the settings.

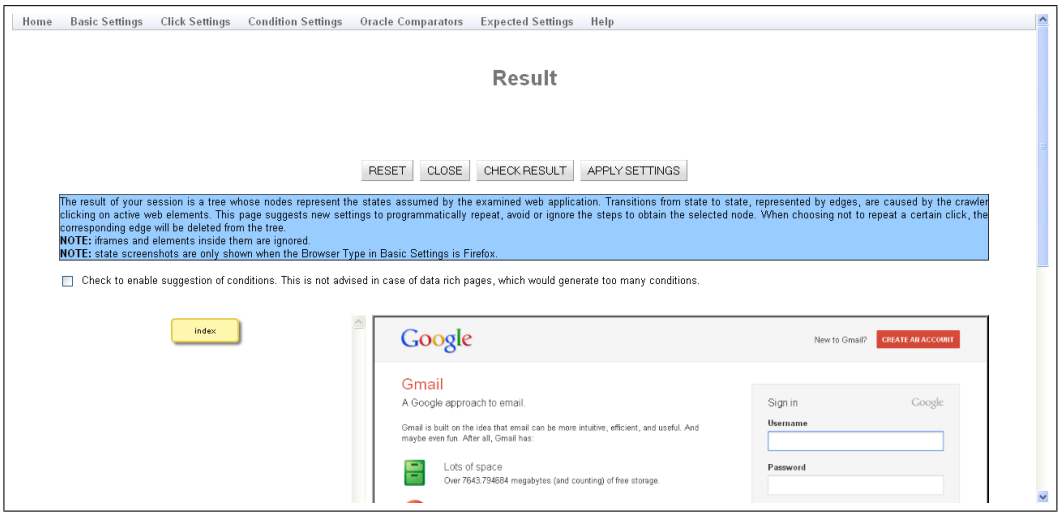

Figure 6.4: CrawlMan's 'Result' page for URL http://mail.google.com/mail.

Figure 6.4 shows the result for the crawling request. Unexpectedly, the result only shows the 'Username' field, identified as a text box with id 'Email', and the check box to click for the cookie persistence, which we are not interested in. The 'Sign in' button was clicked, but it was not possible to sign in given the incorrect credentials. At this point, we use CrawlMan's result suggestion to create a *Set Values Before Click* setting using the detected field button. We still have to manually add the password field, which means we have to find it inside the DOM representation, in the result page. The search is performed by looking for the word 'password' using the internal browser find functionality. The search rapidly highlights the presence of an input field of type password with id 'Passwd'. We proceed to modify the previously added setting inside CrawlMan's 'Click Settings' page. At this point

we visit the ‘Basic Settings’ page to disable the *Click Default* and *Random Input* options, not to mess with the more specific setting, then crawl again. The result shows the address of ‘state1’ corresponding to the address loaded after a successful log-in, so that we know the behavior was reproduced. Figure 6.5 shows CrawlMan’s ‘Result’ page when ‘state1’ is selected. The session is saved and the work can prosecute.

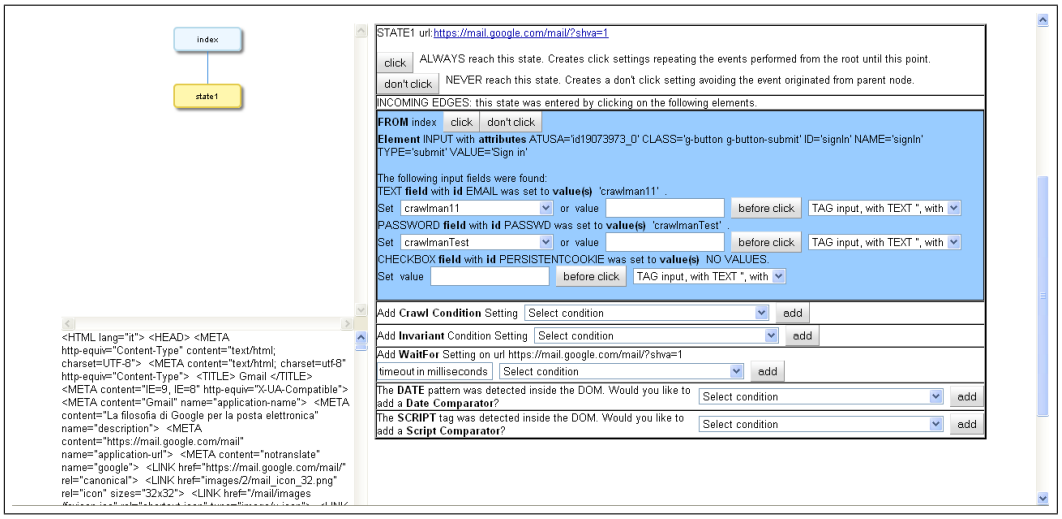

Figure 6.5: CrawlMan showing Gmail’s state corresponding to the ‘Inbox’ page.

Load Email Page

After signing in, Gmail’s ‘Inbox’ page is loaded. This is a page listing the recently received messages. The page contains many elements firing different actions related to the management of messages or other Gmail services. We are interested in the ‘Write’ button, the button allowing to access the email writing page. In order to make CrawlMan present the button in the result, we re-enable *Click Default*.

The resulting tree graph is composed this time by four nodes. The index is the log-in page. The second node, ‘state1’, is the page loaded after signing in. The other two states are reached by clicking other elements than the ‘Write’ button. Why was the button not clicked? Looking for the button inside the DOM should give us an answer.

After a fast search, it is clear that the DOM presents no button with text ‘Write’. The fact is suspicious, as the button we see must correspond to an element of the DOM. As the test was performed with HTMLUnit, which is a Browser Simulator and does not have a complete support of JavaScript, we decide to reexecute the test using Mozilla Firefox. This is easily changed into CrawlMan’s ‘Basic Settings’ page.

This time a much richer result is returned. The button we look for is again not listed between the detected clickables. The most likely reason why the button was again not detected is that what is showed as a button is instead a different element. We perform again

a search of the text ‘Write’ inside the DOM. We find this time a ‘div’ element with the corresponding text and attribute role equal to ‘button’, as shown in Figure 6.6.

```
<DIV class="J-Zh-I J-J5-Ji L3" role="button"
  style="-moz-user-select: none;"
  tabindex="0">
Write
</DIV>
```

Figure 6.6: HTML code for Gmail’s ‘Write’ button.

We create the corresponding *Click* setting, driving the crawler to click on all DIVs with text ‘Write’ and attribute role equal to ‘button’, disable the *Click Default* option again and crawl. This time we are able to enter the ‘Compose’ page, denoted by the address `https://mail.google.com/mail/?shva=1#compose`.

Send Email

We obtained a result tree with three states, as shown in Figure 6.7. The new ‘state2’ is now the state we wanted to reach. From ‘state2’ is possible to compose and send an email, given the right settings.

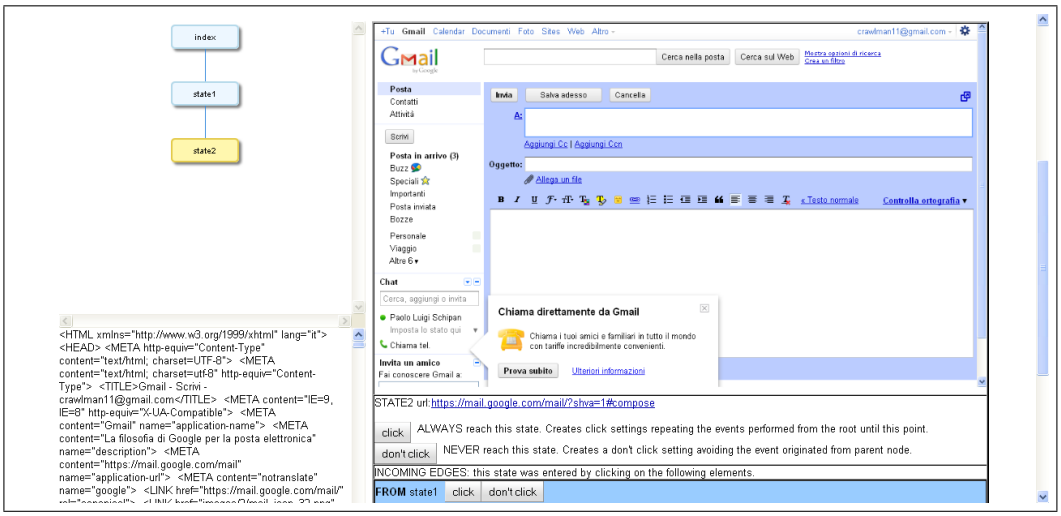

Figure 6.7: CrawlMan showing Gmail’s state corresponding to the ‘Compose’ page.’

Four elements of the page are of interest: the three text boxes to insert destination address, subject and body of the message, and the ‘Send’ button. The three input fields are

not shown in the result. This fact suggests us again that the elements we are looking for may not be defined as conventional text boxes. A search in the DOM reveals us that the three elements are indeed present in the page. The following HTML elements correspond respectively to the destination, subject and body field. They can be recognized by the *name* attribute, being 'to', 'subject' and 'body'. Their ids, necessary for input filling, are instead ':kz', ':kw' and ':k3', as shown in Figure 6.8. Using those ids we should be able to fill the inputs before clicking on the 'Send' button, if the ids are not dynamically generated.

```
<TEXTAREA aria-haspopup="true" class="dK nr" dir="ltr"
  id=":kz" name="to" spellcheck="false" tabindex="1"></TEXTAREA>

<INPUT class="ez nr" id=":kw" name="subject" spellcheck="true"
  tabindex="1">

<TEXTAREA class="Ak" id=":k3" name="body" spellcheck="true"
  style="height: 224.6px;" tabindex="1"></TEXTAREA>
```

Figure 6.8: HTML code for Gmail input fields in the 'Compose' page.

The 'Send' button, shown in Figure 6.9, is composed similarly to the previously discussed 'Write' button. The button definition and the input fields ids will be again used for building a *Set Values Before Click* setting.

```
<DIV class="J-Zh-I J-J5-Ji Bq L3" id=":ke" role="button"
  style="-moz-user-select: none;" tabindex="1"> <B>
  Invia</B> </DIV>
```

Figure 6.9: HTML code for Gmail 'Send' button.

After adding the setting, we proceed again to crawl. The result shows us that the input fields were not detected, and the message was not sent. It seems that the input fields cannot be detected for some reason. The description of the elements correspond to the most recent result, so that we know the elements ids are not generated dynamically. The analysis of the DOM helps us understand once again, revealing the inclusion of these elements in IFrames, which at present Crawljax is not capable to manage.

This test shows us that, in order to be able to understand and modify the crawling session, a user must be familiar with the basics of HTML and the possibilities of the appli-

cation. It is possible to define a complex behavior with CrawlMan, as long as the tested application is crawlable.

6.2 Evaluation Tests

We asked a group of testers with no crawling experience to use CrawlMan and complete the following tests. The tests want to verify usability and intuitiveness of the application, so the testers are offered no other information except the data provided by the tests and the application itself. The tests were performed in the arch of two months, from October to November 2011, as the results of the tests were used to improve the application and the tests themselves before proceeding.

Although various people contributed with their advices to the project, the tests we present were actually performed by four users. We present in Table 6.1 an overview of their background, and the tests they performed. The four users are master students at the TU Delft university, so that they share an engineering background, but only User 4 has programming experience. However, none of them has knowledge related to Crawljax or other crawlers.

	Age	Nationality	TU Delft MSc Study Course	Test 1	Test 2	Test 3	Test 4
User 1	25	Mexican	Sustainable Energy Technology, and Science Education and Communication	X		X	X
User 2	23	Dutch	Systems Engineering, Policy Analysis and Management	X	X	X	
User 3	24	Italian	Systems and Control	X	X		
User 4	24	Italian	Computer Engineering		X	X	X

Table 6.1: Users' background and performed tests.

The tests exercise the innovative features presented in the project (Section 5.1.4). Test 1 guides the user in refining the crawling with suggested *Click* settings. Test 2 introduces the user to *Set Values Before Click* and *Invariant* settings. Test 3 shows the concept and usage of *Oracle Comparators*. Finally, test 4 exercises *Crawl* settings and the suggestion of conditions. The tests do not exhaustively tests all the innovative features, which were manually tested separately, but the general comprehension and ability to use them by unexperienced users. Popular websites, ranked in the top one hundred most-visited sites ranking by Alexa [29] and Google Ad Planner [30], were chosen for the test, to ensure familiarity with the pages by the user and to demonstrate the capability of CrawlMan to handle real case problems.

6.2.1 Test 1: crawling Wikipedia

Wikipedia [13] is an interactive multimedia encyclopedia, written by its very same users, and the most popular encyclopedia on the Web [29, 30]. We have chosen to use Wikipedia

because of its popularity and the clear organization of its articles, divided into pages connected by links. In this test the user is required to find the shortest path between two unrelated Wikipedia pages, shown in Figure 6.10.

```
http://en.wikipedia.org/wiki/Zucchini
http://en.wikipedia.org/wiki/Julius_Caesar
```

Figure 6.10: Test links to Wikipedia's pages.

It is possible to navigate from the first page to the second page by using internal Wikipedia links. The user is first required to find the shortest path manually, then reproduce the behavior using CrawlMan. This test requires basic knowledge of CrawlMan and *Click* settings, given in the following. The test verifies if an unexperienced user is able to use CrawlMan to navigate between links.

The expected result of this test is the tree graph in Figure 6.11.

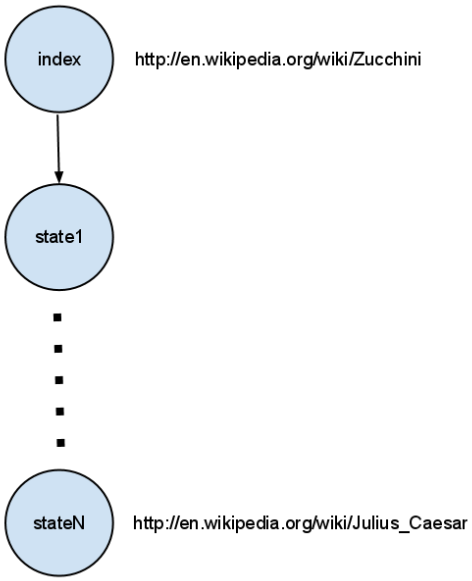

Figure 6.11: Test result tree. Links between nodes correspond to events generated by clicking on links between pages.

In case of a website with a hypertext structure, as Wikipedia, every node represents a page. One or more nodes separate the initial page from the final page. The edges connecting these nodes are links clicked to navigate from node to node.

Th test steps to be conducted by the user are:

1. Manually find the shortest path between the two pages.
2. Load CrawlMan's homepage.
3. Disable the *Random Input* setting, set *Max Depth* and *Max States* to the number of pages you had to navigate (including the initial one), select Firefox browser.
4. Insert the initial URL and proceed to crawl.
5. Inspect the result. The result consists of the node 'index', representing the initial page, and zero or more child nodes, representing the pages reached by navigating from the index page. What links were clicked? What should you have clicked instead?
6. Select the initial node. The application shows the elements of the page that were clicked (**outgoing edges**) and the elements which did not generate a new node (**other clickables**). These elements, which are the edges connecting nodes, are buttons and links, appearing respectively as elements with tag 'button' and 'a' (anchor). Buttons can also appear as elements with tag 'input' and attribute 'type=button' or 'type=submit'. From these edges, find the element you need to click to get to the second page.
7. Add the suggested setting by using the 'click' button. This will generate a *Click* setting to click on the defined element. Click on 'apply settings'.
8. Disable the *Click Default* setting. This setting tells the program to click on all anchors and buttons, which would lead to click on links we are not interested in just because they precede our link in the page.
9. Crawl again, then inspect the result. Did the application reach the desired state?
10. Load the 'Click Settings' page. This page shows the setting you applied from the result, telling the program to click on all anchors (elements with tag 'a') with a specific text. Create analogous settings to click on the other links you want to click. The program will programmatically search every state it navigates for corresponding elements, and click them all.
11. Confirm the settings, then crawl again. Inspect the result. Did the application reach the desired state?

6.2.2 Test 2: crawling Ebay

Ebay [14] is a international auctioning web application. It offers to users the possibility to participate to auctions from all around the world, or create their own. The association with the popular online payment method PayPal [31] turned out to be a very successful formula, pushing both sites in the top forty most-visited on the Web [29, 30].

The purpose of this test is exercising *Click*, *Set Values Before Click* and *Invariant* settings. Starting from an Ebay page describing an object for auction, the user is required to insert a description to search in Ebay, and then load the 'Advanced Search' page. The defined *Invariant* must hold through the states.

The steps to conduct by hand on Ebay are the following:

1. Select an object on sale on Ebay. The URL of the selected page is the initial URL for the test.
2. Insert a search word in the text box at the top of the page, then press the 'Go' button.
3. Look inside the page for the anchor with text 'Advanced', then click it.

The test requires to repeat these same steps through CrawlMan. How many pages were visited? The visited pages are the states the application has to assume during the crawling process. The links between the states are the performed click actions. Figure 6.12 shows the expected result graph for this test.

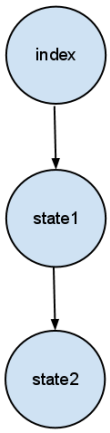

Figure 6.12: Test result expected tree for the Ebay case.

The corresponding CrawlMan steps are:

1. Load CrawlMan's homepage.
2. Go to the 'Basic Settings' page and change *Max States* to the number of states to visit.
3. Insert the initial URL and proceed to crawl.
4. Inspect the result. The result consists of the node 'index', representing the initial page, and zero or more child nodes, representing pages reached by navigating from the index page as shown in Figure 6.12. What links were clicked? What did you want to click instead?

5. Select the initial node. The application shows the elements of the page that were clicked (**outgoing edges**) and the elements which did not generate a new node (**other clickables**). These elements, which are the edges connecting nodes, are buttons and links, appearing respectively as elements with tag 'button' and 'a' (anchor). Buttons can also appear as elements with tag 'input' and attribute 'type=button' or 'type=submit'. From these edges, find the description of the 'Go' button.
6. The node data also show the input fields that were detected. These fields have been filled with random data input. Which of these fields corresponds to the text box near the 'Go' button? Looking into the DOM source code, showed at the bottom of the page, or directly at the original page could help with this operation.
7. Near each one of the detected input fields in Crawlman's result there is a button with text 'before click'. This button allows to create a *Set Values Before Click* setting using the already present value or one inserted by the user. Insert the value to search, select the description of the 'Go' button in the list, and then press the 'before click' button.
8. Click on 'apply settings' at the top of the page.
9. Go to the 'Basic Settings' page and disable the *Random Input* setting. The setting tells the program to insert random input in all the fields it finds.
10. Crawl again, then inspect the result. Did the application reach the desired state?
11. Find the anchor with text 'Advanced' between the elements of 'state1'.
12. Add the suggested setting by using the 'click' button. This will generate a *Click* setting to click on the defined element. Uncheck generated settings that are not needed, then click on 'apply settings'.

Applying an *Invariant* setting:

1. The first two visited web pages have various HTML elements in common. Can you identify them, looking at the actual pages?
2. From CrawlMan's result page is possible to create an *Invariant* setting. This setting can verify a condition on every new state during the crawling process. Create an *Invariant* to verify that the text box near the 'Go' button is present in every state.
3. Crawl again, then inspect the result. Press the 'check result' button to verify the result of *Invariant* settings. Was the *Invariant* successful on the first two states? What about the third state?
4. Can you explain the reason why the *Invariant* is still holding on 'state2'.

The *Invariant* setting is expected to fail on the last state, because apparently the search text box does not appear in it. The test actually succeeds because a different text box, but with the same 'id', appears inside the page. As the condition used by the *Invariant* checks for the presence of an element with the specified 'id', the condition is true also on 'state2'.

6.2.3 Test 3: crawling Mediafire

Mediafire [32] is a website that offers content storage and sharing. Users can subscribe to manage their content, but uploading is actually free for everyone. Mediafire appears in the top one hundred most-visited sites ranking by Alexa [29] and Google Ad Planner [30], and it is an AJAX web application. We are not using Mediafire for a test of the website features, but we are interested instead in the dynamic changes in the main page content.

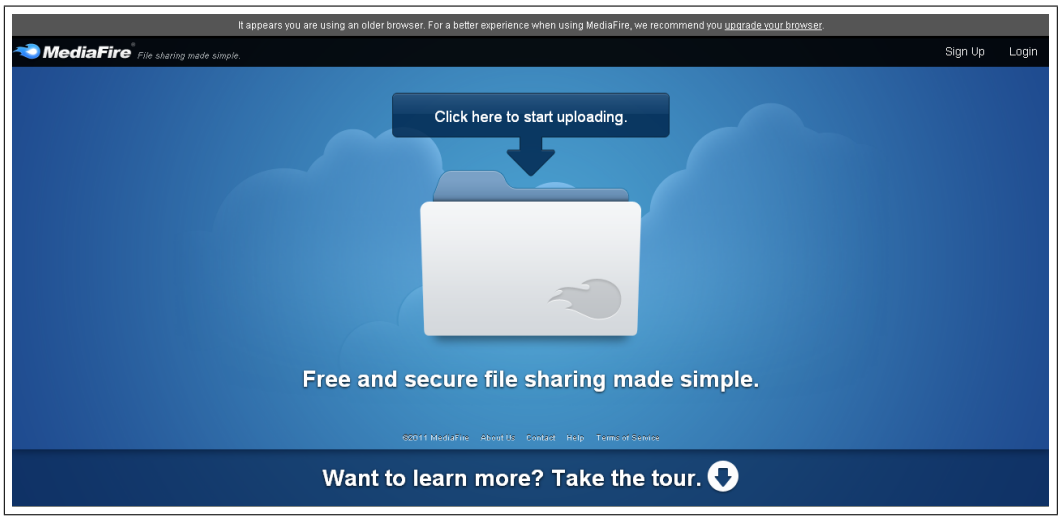

Figure 6.13: Mediafire homepage.

When loading Mediafire's homepage using Firefox 5.0 or other browsers not in their most recent version, a message appears on top of the page to advice the user to update their browser, as showed in Figure 6.13. When clicking on the message, a pop-up is showed with links to browser distributor websites. From the pop-up, the message can be dismissed, to return to the main page. After dismissing the message, the homepage appears unchanged expect for the absence of the message on top of the page. The browsers employed by Selenium, and thus employed by Crawljax and CrawlMan (Section 7.2.1), present this behavior, so that the test can be effectively performed. The behavior suggests that the initial and final states of the Mediafire web application can be considered equal if the top part of the homepage is ignored, which represents the perfect situation to verify the working of *Oracle Comparator* settings.

When applying an *Oracle Comparator* setting, we are telling the program to ignore parts of the detected states during their comparison. In order to recognize a new state Crawljax compares the current state's structure and content (the state's DOM) with the previous states. During the comparison, *Oracle Comparators* can tell Crawljax to ignore an element with a specific 'id', 'xpath', or a specific attribute of HTML elements, for example the 'type' or the 'name'. It is also possible to add more general *Oracle Comparator* settings, such as the *Simple Structure* comparator, which ignores all the content and attributes of detected

states, to compare only their structure, or the *Levenshtein Distance* comparator, which uses a threshold from 0 to 1 to decide on state equality. When using the 0 threshold, all states are considered the same, so that the result only consists of one state, the 'index'.

Initial steps of the test are:

1. Load CrawlMan's homepage.
2. Go to the 'Basic Settings' page and change *Max States* to three.
3. Insert the initial URL `http://www.mediafire.com` and proceed to crawl.
4. Inspect the result. The result consists of the node 'index', representing the initial page, and zero or more child nodes, representing pages reached by navigating from the index page. What links were clicked? What did you need to be clicked instead?
5. As the link we need to click is the first found in Mediafire's homepage, the behavior we needed to reproduce is actually the one obtained. Select 'state2', then click on the button to 'ALWAYS reach this state'.
6. Click on 'apply settings'.
7. Crawl, then inspect the result again. Were the settings respected?

We stated that the initial and final state of the application only differ for the presence of a message. This difference must reflect on the actual code of the page, which is presented by the result. After a line-by-line comparison of the DOMs of 'index' and 'state2', performed with a text comparator tool, it is clear that the sources of the states only differ in lines 36 and 411, respectively containing a 'body' and a 'script' HTML element.

```
<BODY class="home animating time-to-upgrade upload condensed windows ie7
  basic">

<BODY class="home animating upload condensed windows ie7 basic">
```

Figure 6.14: Line 36 of Mediafire's homepage DOM representation respectively for states 'index' and 'state2'.

```
<SCRIPT language="JavaScript" type="text/JavaScript">
  //<![CDATA[ kd='xnzuenj66k6; [...]

<SCRIPT language="JavaScript" type="text/JavaScript">
  //<![CDATA[ kd='k8qgaaawwap'; [...]
```

Figure 6.15: Line 411 of Mediafire's homepage DOM representation respectively for states 'index' and 'state2'.

As shown in Figure 6.14, the 'body' elements in the two states differ because of one parameter absent in the 'class' attribute in 'state2' - the self-speaking parameter 'time-to-upgrade'. The 'script' elements of Figure 6.15 differ instead because of the value of the attribute 'kd' inside the 'CDATA' element content. The remaining content was omitted because it did not present differences. We are not interested in the working or the meaning of these differences, but we know that they are related to the visual changes in the Mediafire homepage. The result page suggests the necessary settings to ignore these changes.

Applying *Oracle Comparator* settings:

1. Load CrawlMan's 'Result' page.
2. Click on the check box at the top of the page to enable the suggestion of conditions and *Attribute* and *XPath* comparators.
3. Select 'index', then reach the bottom of the page.
4. Add the correct *Oracle Comparator* setting to ignore the HTML attribute 'class'.
5. Add the correct *Oracle Comparator* setting to ignore the HTML element 'script'.
6. Click on 'apply settings'.
7. Go to the 'Basic Settings' page and disable the *Click Default* and *Random Input* settings, to ensure only the desired elements are exercised.
8. Crawl, then inspect the result again. What did change in the result? Where does the outgoing edge from 'state1' lead now?

After the addition of the correct *Oracle Comparators*, the changing parts of the page are ignored. The states 'index' and 'state2' are recognized as the same page, so that the click event performed on the pop-up in 'state1' leads back to 'index'. The final result graph is composed of only two nodes.

6.2.4 Test 4: crawling WordPress

WordPress [33] is a blogging platform offering free publishing space and second-level domains. WordPress is an interesting case of a web application which automatically creates new web applications. A user who subscribes to WordPress creates his own blog, supported by WordPress services but entirely customizable.

All blogs created through WordPress have a URL in the form of '*http://yourdomain.wordpress.com/*', where '*yourdomain*' can be substituted with the preferred expression, given the address does not already exist. This specific aspect makes WordPress an interesting website to crawl. The website homepage, in fact, contains a huge variety of links, but most of them lead outside of the original domain, '*wordpress.com*', to enter other pages with URLs of the form '*yourdomain.wordpress.com*'. Crawljax, and CrawlMan with it, is designed to automatically ignore the clickables **explicitly** bringing out of the original domain. The user is required to have CrawlMan find all the links present in the homepage which do not lead out of the original domain, and he is asked to do so by using only *Crawl* settings. Figure 6.16 shows the expected result at the end of the test.

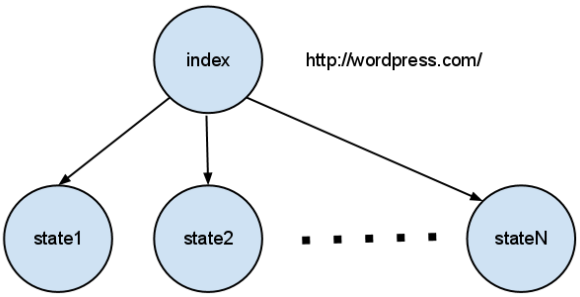

Figure 6.16: Wordpress test result tree.

Crawl settings tell the program to crawl or not the current state of the tested web site, depending on the outcome of the specified conditions. They are applied to every state detected during the crawling process, every time a crawling request is sent. There are various types of conditions, based on the current state's URL, or the visibility or position (XPath) of the elements in it, or more advanced as JavaScript and regular expressions to verify on the current state. All conditions can be also **Not** conditions, based on a negative expression. For example, by applying a *Crawl* setting with *URL* condition 'URL is 'something", we want the program to crawl the current state if its URL contains the word 'something' in it, as in '*http://www.mydomain.com/something.htm*'. If we use the condition 'URL is not 'something", we want the state to be crawled only if 'something' does not appear in its URL. If instead we apply a *Crawl* setting with a *(Not) Visible* condition, for example using the 'id' of a HTML element, we want the program to crawl the current state if the element with that 'id' is (not) visible inside the page. When adding a *Crawl* setting, only the states respecting the specified conditions will be crawled, which means the program will proceed

from these states to find new ones. If the ‘index’ state does not respect the condition(s), the result consists in no more than one state, the ‘index’ itself.

The corresponding CrawlMan steps are:

1. Load CrawlMan’s homepage.
2. Insert the initial URL `http://wordpress.com/` and proceed to crawl.
3. Inspect the result. The result consists of the node ‘index’, representing the initial page, and zero or more child nodes, representing pages reached by navigating from the index page. What links were clicked? Which nodes should be removed to comply with the expected result?
4. The WordPress ‘Sign Up’ page has been reached in more than one state, and more states have been found starting from this page. Even if it is the same page, the ‘Sign Up’ page is seen as different states because of randomly generated content, which can be ignored by applying *Oracle Comparator* settings, but this is not the point of the test. We are instead interested in adding a *Crawl* setting to proceed crawling when not in this page, so that the states corresponding to this page will not spawn more child nodes.
5. Click on the check box at the top of the page to enable the suggestion of conditions, then reach the bottom of the page to add the correct *Crawl* setting. Select a meaningful condition which will exclude the ‘Sign Up’ page from the crawling. The condition must be valid in the states we want to crawl and invalid in the states we do not want to crawl.
6. Click on ‘apply settings’ on the top of the page.
7. Crawl, then inspect the result again. What did change in the result?
8. Repeat the addition of *Crawl* settings for all the nodes that do not comply with the desired result. Augment *Max States* in the ‘Basic Settings’ page until all the conforming clickables in the ‘index’ page have been clicked. Crawl until the desired result is obtained.
9. Are there elements of the page which were not clicked? If so, can you explain why?

The ‘index’ state presents one link that was not clicked in the ‘Other Clickables’ section - a link with text ‘later’. The link is not clicked because it is invisible, and only displayed after clicking on the symmetric link ‘earlier’, which displays blog pages less recent than the ones already displayed in WordPress’s homepage. The results present states with URLs outside of the original domain, but this is due to the fact that the clicked elements refer to internal addresses - yet actually redirect the user to these final URLs.

The same result that we obtained, one root node connected to many leaf nodes, can be also obtained by specifying *Max Depth* as 1 in the ‘Basic Settings’ page of CrawlMan.

Different settings of CrawlMan can produce the same result, depending on the way they are used, to ensure the application is flexible enough to satisfy the user needs.

6.3 Interpretation

We discuss here the lessons learned and the modifications derived from the previous tests.

6.3.1 Test 1: crawling Wikipedia

The Wikipedia test was very informative and highlighted the necessity of some changes to the application. First of all, the test wants the user to navigate from link to link, using result suggestions. A small Wikipedia page contains around two hundred different links, but other pages involved in the test contain more than two thousand. The difficulty of showing the result of crawling such data-rich pages was immediately clear. An early version of the project simply showed all the detected clickables at once, in the same page. When visiting a Wikipedia page, the produced result blocked the browser while loading the page, as for every detected link various visualization elements had to be produced. Furthermore, the automatic generation of conditions was adding an even heavier burden. Conditions are in fact generated using *id*, *name*, *text* and *xpath* of the detected elements, which means that for every element at least four conditions are generated. It is straightforward that some modifications had to be applied to lighten up the workload of the application. The following measures were applied:

- Listing of clickables that did not generate new states were limited to ten at a time. The user can use internal arrows to navigate the result, so that all the clickables are shown but not all at once.
- Generation of conditions was disabled by default, but can be enabled through a check box in the 'Result' page.
- Capturing of screenshot was disabled by default, but can be enabled through a check box in the 'Basic Settings' page.

These measures successfully reduced the size of the result and the application workload on the user's machine, so that even data intensive results could be inspected.

The test demonstrated that an unexperienced user can use CrawlMan to explore a website, while also showing that the more severe difficulties of crawling are not derived from CrawlMan itself, but from the working of Crawljax. The users, having no familiarity with Crawljax, could not imagine the effect of their actions on the crawling process, and how new settings could influence the crawling and what they had to expect. The solution to this problem was to include a decision flow graph inside the CrawlMan tutorial (see Appendix A). This graph intends to show to a user the causes behind Crawljax behavior, so

that creation of settings and inspection of result are simplified. Furthermore, it was decided to add more explanations in the form of tooltips in the areas of the application requiring more user interaction.

Lastly, the Wikipedia test relies on the *Click Default* setting to automatically suggest links to the user. It was noticed that the setting was only useful in the initial crawling, while hindering the test during the successive steps. The test in fact requires the user to select the right link to click from the suggested clickables. However, after applying the right suggestion, the *Click Default* setting was forcing the program to click on all links encountered in the page before the desired element, resulting in the wrong clicks. The problem was caused by the fact that at every new found state Crawljax applies all the *Click* settings in order. The *Click Default* setting indicates to click on all anchors and buttons, and it always appears as the first *Click* setting, so that more specific settings are practically ignored. This aspect of Crawljax does not represent a problem in Crawljax itself, because the *Click Default* setting was not meant to be used in combination with specific *Click* settings. Nevertheless, it is not what a user expects when using it inside CrawlMan. The user would like the program to execute the specific settings, but also detect other clickables in order to receive new suggestions. The solution to this problem was to substitute Crawljax's *Click Default* setting with *Click* settings to position after the user-specified settings when *Click Default* is enabled.

6.3.2 Test 2: crawling Ebay

The Ebay test demonstrated that *Invariant* settings can be easily used by the common user through CrawlMan, if the user can understand the concept beside it. A practical example such as the proposed test is a perfect introduction to *Invariants* and other advanced settings. Therefore, it was decided to include the tests of this chapter in the CrawlMan website, along with the tutorial of Appendix A. The tests will guide the user in their first utilization of CrawlMan.

As the *Invariant* settings can succeed or fail independently from other settings, we decided to clearly distinguish their outcome inside the result report. The test highlighted that the user was confused by the previous form of the report, which did not inform about the success of *Invariants* in the presence of errors related to other settings.

The implementation of *Set Values Before Click* settings was intuitive and easy to use. A problem was detected related to the suggested clickables for these settings: when the user selects a state from the result containing many clickables and input fields, loading the page can be rather slow. We considered the possibility to add a check box to enable the suggestion of clickables for *Set Values Before Click* settings only when the user needs it, so that the standard visualization has a lower weight on the browser. We discarded the possibility because of the importance of the presence of all the clickables inside the suggestions, and the fact that is always possible to restrict the size of the result by employing more stringent definitions inside the settings.

Another confusing point for the user was the unclear separation between 'Incoming

Edges', 'Outgoing Edges' and 'Other Clickables' inside the result page. The solution was to emphasize the titles and borders of the three sections, so that the user could understand their elements embody different concepts. The 'Other Clickables' section was especially tricky: this section lists all the clickables detected in the selected state, but also clickables whose click failed. We decided to list failed clicks in the result report, and present this section just as a list of the detected clickables, to clearly present possible errors to the user.

The test encourages the user to explore the DOM of the selected state and the corresponding web page, in order to identify the elements he needs to perform the test steps. As CrawlMan wants to make the user familiar with HTML code, it was decided to reformulate the descriptions of HTML elements in the result page in a way that suggests to the user their original form. The best way to concisely describe HTML elements is using the HTML language, so it was decided to employ a kind of pseudo HTML in the descriptions, as shown in Figure 6.17. When the user looks inside the DOM and the original web page for a certain element, he now knows what to expect based on its description.

```
Element <A> with text BUTTON TEXT with attributes ID='buttonId'
  type='submit'
with xpath /HTML[1]/BODY[1]/BUTTON[1]

<A text='button text' id='buttonId' type='submit' />
```

Figure 6.17: Two different pseudo HTML representations of the button in Figure 2.6.

The direction of the content of list boxes was changed after this test. The content of list boxes was often exceeding the limits of the result page, causing the user not to understand the listed options and not to notice the presence of the list boxes sidebar, necessary for inspecting all the options. The solution for this problem was found outside Google Web Toolkit, applying a CSS style to list boxes and their options as shown in Figure 6.18.

```
.gwt-ListBox {
  direction: rtl;
  text-align: left;
}

.gwt-ListBox option {
  direction: ltr;
}
```

Figure 6.18: List boxes styles.

The styles indicate the direction of the content for list boxes as right to left, and for their options as left to right. This last addition was necessary as the resulting options text was messed up by the use of the first style.

6.3.3 Test 3: crawling Mediafire

The Mediafire test showed that *Oracle Comparator* settings can easily confuse the users. The concept of ignoring part of the state's DOM to merge different states in one was easy to understand for the test users. The difficulties appeared when the users were requested to apply the settings. The 'comparison' process was felt as separated from the 'ignoring' process, so that the users thought they had to add more settings than just an *Oracle Comparator*. It was suggested to use a different name for the settings, such as "Ignore Comparators", to make their function clearer. The suggestion was not followed as the terminology tails the original one used by Crawljax, to avoid confusion for possible users who want to access Crawljax's original documentation.

We must underline that the tests were done providing to the users as little information as possible, to verify the intuitiveness of the application. Due to the poor understanding of *Oracle Comparator* settings, more information was provided in the test itself and in CrawlMan's 'Result' page. We decided to visually separate the suggestion of comparators from the other settings, and to add some simple explanation lines.

Another point of confusion was the use of conditions together with *Comparators*. For other settings, when a list box with conditions is present it means adding a condition is mandatory. For *Oracle Comparator* settings the condition is optional, but this was not perceived by the user. The solution was to specify the optionality of the condition on the list box itself, so that the user understands it is not necessary to specify a condition - it depends of course on the test we want to perform.

6.3.4 Test 4: crawling WordPress

The concept of *Crawl* settings was the most difficult for users to understand. The test asks to select a condition which is false in the states we do not want to crawl, but true in all the other states. Once the users assimilated this concept, they easily proceeded in applying the correct settings. The first *Crawl* setting was hard to choose, but once the users inspected the result and verified its working, the rest of the test was easily performed. More information was added to the test to correctly guide the users.

The tests also shows how Crawljax selects the clickables to exercise. Many links inside WordPress's homepage clearly lead out of the original domain. Other links at first sight lead to internal pages, but instead redirect to pages outside of the original domain. The users were surprised to see these states inside the result, but easily realized the difference between the links by navigating through the actual pages.

Another good point of the test is that the users understood that the same result could be obtained using other settings - specifying to click on the desired elements or even in a simpler way by setting *Max Depth* to 1. The users suggested the methods based on the experience of the previous tests, so that the test also showed that the use of the application, once learned, is easy to remember.

6.4 Evaluation

This section uses the information collected during the previous tests to provide an engineering judgment on the CrawlMan project. We have often pointed out the need to make CrawlMan intuitive and easy to use, and to provide Automated Exploratory Testing. We investigate here the actual factors that would satisfy the two requirements.

6.4.1 Usability

Easy and intuitive are very vague terms. What we are trying to investigate is the usability of the project, which is actually defined by an ISO standard for human-targeted products [34]. The definition of usability is "*the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use*". More specifically, usability is composed by five factors [35]:

- Learnability: the degree on which the system is approachable by new users.
- Efficiency: ease and rapidity of completing tasks once the system has been learned.
- Memorability: ease of remembering how to use the system.
- Errors: user's tendency to errors, and capability to recover from them.
- Satisfaction: pleasantness of use of the product by the user.

We explored how CrawlMan can be used by unexperienced users, if good guidance is provided. With no guidance, testers demonstrated to understand the results of crawling, but not the refinement of settings. However, the simple tests proposed were enough to teach the testers group how to select and apply the correct result suggestions.

Once a user has learned to use CrawlMan, it is easy to remember it. Proceeding from test to test, users did not require an explanation of the features they already used. The memorability of the project is enhanced by the graphical representation of the result.

CrawlMan is approachable by new users, but they tend to make simple errors. The concepts behind the proposed settings were fundamental for the users to understand CrawlMan's usage during the tests. For this reason more information was added to the tests, and we decided to share more documentation on the application itself, such as the tests, the corresponding XML session files and the introductory Appendix presented at the end of this document.

Feedback from users showed that using CrawlMan is an interesting challenge, which makes testing somewhat pleasant. The test performed on Wikipedia (Section 6.3.1) particularly seemed to entertain the users, as the goal to reach was clear and thought-provoking.

The efficiency of CrawlMan is the less positive of the investigated factors. Even for an experienced user, it takes some time to refine a test to perfection, and to understand the result. This is due to the nature of the provided settings, which represent descriptions of HTML elements. Every time a setting matches an element of the crawled page, the element is taken into consideration, so that the crawling process can return more data than the user expects. Furthermore, because CrawlMan offers Automated Testing, it is the user expectation that the tests would be performed in a very brief time. The actual crawling process is not visible to the user, as it is conducted on the server side, so that the user does not take into consideration the actual time employed by the browser to load the pages under analysis. These aspects are inherent to testing a web application by using a crawler together with real browsers. However, CrawlMan gives to the users the possibility to directly define their settings, so that an experienced user can create the correct settings even before starting to crawl.

6.4.2 Effectiveness

To measure the ability of CrawlMan to provide Automated Testing we have to consider its effectiveness. The effectiveness of a system is part of its usability, in the sense of efficiency. We discuss here about effectiveness instead of efficiency to shift the focus on the comprehensibility of the results obtainable through the system, more than the ability of the users to obtain them.

The effectiveness of a product is also an ISO standard, defined as "*the accuracy and completeness of users' tasks while using a system*". A system must be able to complete a user's task to be effective. The question is, what tasks is a web testing framework required to complete? We can easily say that a user would like CrawlMan to detect an error, if it

exists in the subjected website. But CrawlMan represents an atypical case of web testing tool, placing the emphasis on the exploratory side. This is why we define the effectiveness factors for CrawlMan through the ability of CrawlMan's users to:

- Reach the desired application states.
- Exercise the elements of web pages.
- Detect eventual faults.
- Verify the holding of *Invariant* conditions.
- Learn information about the website under test.
- Improve the test using CrawlMan's suggestions.

We have seen that it is not possible to reproduce every desired behavior with CrawlMan. The project has some limitations, derived from Crawljax, such as the inability to crawl IFrames. It is possible that these limitations will be overcome in future Crawljax releases, but at the moment they hinder CrawlMan's ability to reach every possible application state and to exercise all the elements of a web page.

CrawlMan presents the eventual failing of *Click* and *Invariant* settings in a report generated in the result page. What other faults could be interesting to detect? The elasticity of *Invariant* settings combined with conditions offer many possibilities to the users willing to experiment.

CrawlMan is very informative about the website under test. The user learns about important elements of the visited states by inspecting the result, introducing him to the original HTML code. CrawlMan provides specific state data and an overview of the entire crawling process, to visually compare the generated states.

We employed unexperienced users to show that everyone can create a web test using CrawlMan's suggestions. We can easily say that the suggestions are a great simplification of the process of collecting the necessary data to create and perfect a test, and that users willing to learn using the application can easily make the best of CrawlMan's capabilities.

Chapter 7

Related Work

CrawlMan represents first of all a way to crawl and test AJAX pages, dynamic web applications and websites in general. The crawling capabilities of the project are derived from Crawljax, so that the project can ultimately be ascribed in the category of the *web crawling test tools*. Although a user can use CrawlMan for crawling a website, the focus of this project is not on the crawling, or offering an interface for making crawling easier. The project wants to be a testing framework for **Automated Exploratory Web Testing**, involving end users in the process. While Crawljax can be extended to be used in other common crawling environments, such as web indexing or information retrieval, these are different utilization scenarios that are not in the scope of the present project. This chapter does not want to compare Crawljax with other AJAX crawlers, it presents instead a panorama of the web testing techniques and tools, and analyzes them in comparison with CrawlMan.

7.1 Product Testing

We define here four general categories of product testing. The categories are often not clearly separated in practice, and they apply to all kinds of products, not just web applications and user interfaces. The description is not a detailed discussion of testing methods, but a general analysis to help framing the scope of CrawlMan and similar testing tools.

7.1.1 Manual Testing

The simplest way of testing a website is loading it in a browser and navigating through its pages. Manual Testing is a very common way of testing, especially employed in the final stage of a product evolution - acceptance testing. It is applied to every kind of product, not only web applications, as the best way to verify if a product corresponds to user expectations is trying it out. Manual testing is not synonym of sloppy, negligent testing. It is a way of testing the final product in the final utilization environment, testing also product integration if the product is part of a bigger system. Manual testing does not consist in randomly

exercising the functionalities of a product, but in rationalized tests that recreate scenarios of utilization to exercise vulnerable behavior.

Why do we mention manual testing? It is the form of testing which can actually involve end users. End users, in fact, usually do not have or need insight knowledge on the development process. What is required from end users is to understand what is the utility of the product and how to use it. In the specific case of web applications, the end user could require specialized knowledge related to the final utilization scenarios. For example, an end user of a banking web interface could be a bank employee, familiar with accounting, transactions and other financial operations. The technical, web knowledge which is required from every end user is just how to use a mouse and navigate a website, which can be considered common knowledge of every computer user.

7.1.2 Automated Testing

We indicate with Automated Testing the automation of tests that are usually manually performed. Test automation allows to repeat these tests in a mechanical way, making sure the tests do not change over time and the tested software maintains its characteristics even through modification. Automated tests are normally written and performed using some testing tool. Our interest is in *Web Interface Automated Testing*, specifically testing dynamic pages, and the existing tools presenting these capabilities. Porting of manual tests into an automated environment means reproducing with a tool the way a user would navigate a web application. As explained in Section 7.2, there are few ways to reproduce this behavior. Differences between automated web testing tools consist mostly in which kind of browser they exercise, how the tool interacts with the web page and the language used for scripting tests.

7.1.3 Exploratory Testing

Exploratory testing [36] is usually performed by an experienced tester. The tester could be a user aware of the technologies underlying the application. It is defined in [37] as "simultaneous learning, test design and test execution", which means that while the test is designed and performed, the tester gains knowledge to improve the test and design new ones. In a general view, all testing can be considered exploratory, but here the focus of the tester is not on executing a series of prebuilt tests over and over, but deciding every moment which is a valuable test to run, something which could lead to the discovery of undetected faults. The decision is based on many factors, such as the the product objective, interface, behavior, the known problems and weaknesses, recent modifications, and all other data relative to the development and testing of the project, but it ultimately relies on the tester's role, knowledge and skills.

Exploratory testing does not substitute systematic testing, it is employed to diversify it. It is itself a form of documented testing, except for the specific case of freestyle exploratory testing, where the tester only produces a report of newly detected bugs. Its strength is in

the speed of collecting feedback, which can then be used for creating new tests. The information collected during test execution helps refactoring the tests themselves, representing a flow of information between design and execution. As this aspect represents also the main strength of this project, CrawlMan can be defined a form of *Automated Exploratory Testing*. As defined in [38], Automated Exploratory Testing can be divided into passive or active. In the **passive testing**, the testing session is registered by some tool for later analysis, for example using a video capturing tool. Time and effort for reporting a bug are considerably reduced, as the conditions of its appearance are clearly documented. In the **active testing**, the test is designed during the execution. The tester is not required any programming knowledge, as this kind of testing follows a Keyword Driven Testing approach [39], in which the logical layer is divided by the test automation infrastructure layer by means of a translating interface. The tester inserts into the interface a set of keywords describing the elements to test, the action to perform and eventual input data. Those keywords are used by the infrastructure layer to generate and execute scripted tests. CrawlMan falls into this latter category, as an *Active Automated Exploratory Testing framework*, where the inserted keywords are formed by description of HTML elements, actions to perform such as click, crawl, ignore and other settings, and possible input data for input fields.

Does this mean CrawlMan cannot be used in a systematic approach? Absolutely not, because every session executed in CrawlMan can generate a script describing a test case, part of a larger test suite. A single test case can also be considered systematic if it is exercising all the elements of a certain kind, for example all anchors in a web page. Nevertheless, the nature of the definition of settings in Crawljax, capable of fitting many elements at the same time, leaves open the exploratory side of every test case, which will exercise newly introduced elements if their description corresponds to the already tested elements.

7.1.4 Systematic Testing

In Systematic Testing, opposed to Exploratory Testing, there is always a defined plan. Test cases are first designed, then executed, manually or in an automated way. Tests are collected in test suites, versioned for regression tests. Systematic Testing does not only consider the interface of a system, but it is performed at a granular, in-depth level. Functional and unit testing are forms of Systematic Testing, but the term generally includes all kinds of thorough inspection of a product. Systematic Testing ensures the absence of obvious bugs, the regression of the product to previous unstable situations and its overall functionality and respect for requirements. The test designer can be different from the test executioner, who need not have technical knowledge of the product.

7.2 Web Interface Testing Tools

Testing web applications client side is a specific brand of *Graphical User Interface (GUI) Testing*, as the user interface is a web interface, accessible through a browser. There are actually not so many techniques and tools which automatize the testing process. Following

the distinction made in [40] and better expressed in [41], we divide web testing tools in two categories:

- Browser Simulators, tools that can perform HTTP requests to a server, load and parse HTML content and execute actions on it as a browser would do.
- Browser Drivers, tools that drive browser instances and perform actions on the page through the browser.

Crawljax pertains to the second category, thanks to its use of the Selenium libraries (Section 7.2.1), and so does CrawlMan. The advantage of browser drivers is the power to execute tests reproducing the actual way a user would navigate the website. On the other side, browser simulators do not require installation of third party browsers and the utilization of a display environment, usually missing on dedicated machines and servers. Still, the use of browser simulators may be feasible just for executing simple smoke tests, as simulators do not support all the possibilities of JavaScript, as instead implemented by real browsers.

Another distinction must be made between testing tools, depending on the level of automation they support:

- Scripting tools, offering automation of test execution.
- Capture&Replay (or Recorder) tools, automating test creation and execution.
- Crawlers, allowing automated exploration.

Most of the AJAX testing tools, simulators and drivers, are of the Capture&Replay type [7]. Those tools record the actions of a user during the navigation of a web application, so that the session can be automatically repeated. The user's actions are converted in a script, which can be later manually modified. Capture&Replay tools make easy for an end user to write a test, and often to modify it, by employing a Keyword Driven Testing approach (Section 7.1.3). They can be used in a passive exploratory way, with the tool recording everything the tester does. The tool is generally a framework, allowing definition, modification and filing of tests.

The other option for interface testers is to create their own scripts. Scripting tools and languages automate the execution of the tests, but it is still left to the developer to gain the necessary knowledge and manually script the tests. The scripts call on browser simulators or browser drivers for test execution, usually part of the tool. Those tools are often extended to create Capture&Replay tools, which do not need scripting knowledge, enhancing end user involvement.

Until now we did not find a tool using a testing method similar to the one employed by CrawMan, capable of automatic exploration of a web application through a crawler to detect abnormalities and errors, and collecting data to improve the current testing session. There are similar tools based on traditional crawlers, developed for testing the Web 1.0 [7]. The

approach for testing AJAX interfaces until now has been through scripts, and adapting web applications for partial navigation by traditional crawlers. Static approaches for analyzing the server side code have been proposed, but these techniques cannot reproduce complex runtime behavior [7].

7.2.1 Tools Panorama

We present a description of the characteristics of some of the most used web testing tools, together with CrawlMan. We are interested here in test and scripting automation for AJAX, and tools that involve the end users by avoiding the need of programming knowledge, and we include scripting beyond AJAX tools which share some aspects with the project.

HTMLUnit

HTMLUnit [42] is a Browser Simulator in Java, modeling HTML documents and providing API to deal with them. It has a good JavaScript support, so that it can emulate Firefox or Internet Explorer. It is meant for use in another testing frameworks, and it presents no GUI to the user so that it can be used as a browser in headless environments. It supports XPath recognition, a feature shared by Selenium, which makes use of HTMLUnit.

HttpUnit

HttpUnit [43] is a Browser Simulator written in Java. As all simulators, it does not fully support JavaScript. It neither supports, and there is no plan to support, browser-specific JavaScript, meaning it does not emulate specific browsers. Its use can be easily integrated in testing frameworks such as JUnit. Its main difference from HTMLUnit is that it models the HTTP protocol instead of the returned document.

Imprimatur

Imprimatur [44] is a Browser Simulator tool, implementing the HTTP request and response as we have seen in HttpUnit. It does not offer support for JavaScript or other languages. The only thing Imprimatur does is validation of the response content using regular expressions. Test cases are written in XML and can be executed from the console or within Ant. Imprimatur is an open-source project written in Java.

MaxQ

MaxQ [45] is an open-source Browser Simulator tool with Capture&Replay capabilities. It works as a proxy between browser and server, recording requests and responses to create a script of the session. Scripts are written in XML or Jython, Java for Python, so that they are

easy to understand and modify, but can also work in a Java environment such as JUnit. The project itself is written in Java to be cross-platform, and can be run from the command line. It can be used with any browser which can be configured to use a HTTP proxy.

Sahi

Sahi [46] is an open-source tool written in Java and JavaScript. It works with any operating system and browser, being a proxy between client and server. Similarly to MaxQ [45], it can record a user session for later replay. It offers automatic proxy configuration for the browser, easy creation of checks on web elements and an intuitive interface. Sahi script is JavaScript based, and is injected directly in the visited page for interaction. Sahi is maintained by Tyto Software, offering also an advanced, paid version with customer support. It is a reliable choice, adopted by many companies such as Oracle, Accenture, and Deloitte.

Selenium

Selenium [47] is the name of a suite of tools pertaining to the category of Browser Drivers. Part of the suite are the Selenium Web Driver libraries, allowing to drive instances of Internet Explorer, Mozilla Firefox, Chrome, Safari, Opera, HTML Unit and browsers on remote servers. The remote servers can be created with Selenium Grid, also part of the suite. Web Driver libraries are cross-browser, cross-language and cross-system, as it is possible to use them with Java, C#, Ruby, Python, PHP and Perl on Windows, Linux, Solaris and OS X. Projects making use of the Web Driver libraries, for example Crawljax, usually take advantage of these possibilities, making them cross-browser and cross-system as well.

Selenium also offers a Firefox extension which acts as a Capture&Replay tool, the Selenium IDE. The environment allows recording, editing and debugging of tests written in Selenese, which is easy for non-programmers to learn, as it reflects the actions executed on the browser. The IDE and the other Selenium tools are open-source.

SlimDog

Slimdog [48] is an open-source Scripting tool, based on HttpUnit and written in Java. It uses its own set of commands, similar to Selenese (Section 7.2.1) and easy to understand for non-programmers. A test case is a text script, and multiple scripts can be part of a test suite by reuniting them in a folder. Tests are executed from the console or as an Ant task. Result reports are written to the console or to a file. Slimdog commands can be used in JUnit test cases.

Squish

Squish by FrogLogic [49] is a proprietary Eclipse based testing framework for GUI and Web interfaces. It is a Capture&Replay and Scripting tool, supporting Internet Explorer, Mozilla Firefox, Chrome, Safari, Opera on Windows, Linux, Unix and Mac OS X platforms. It offers its own scripting language and supports Python, JavaScript, Perl and Tcl. Like Selenium and HTMLUnit, it supports the use of XPath element definition. It can be considered a proprietary version of Selenium, with the advantage of offering customer support. It is currently used by many different companies such as Google, Intel, Siemens, Ericsson, Shell, Disney, Vodafone, Boeing and Mercedes-Benz, only to name a few.

VeriWeb

VeriWeb [50] is a tool very much like CrawlMan, crawling a web application to perform checks on basic correctness and functional and regression tests. It scans the DOM of a page in search for "active objects", elements of the page which usually have an associated action such as links, buttons, items in select boxes and form fields. When a list of possible actions is identified by the ChoiceFinder, a decision on the direction to follow is taken by the underlying VeriSoft implementation. VeriSoft [51] is a tool for systematic exploration of application space states. Unlike Crawljax, VeriSoft recognizes different states of a web application only if they correspond to different pages, which makes VeriWeb not fully AJAX compatible despite the ability to navigate through dynamic components.

The VeriWeb project has been terminated, so we could not perform an actual verification of its capabilities. VeriWeb uses a component called Web Navigator for driving existing browsers, but a list of supported browsers was not found as this element remained in a prototyping phase. Scenarios can be saved for later visualization and replay using WebVCR [52], but not for editing.

Watir

Watir [53], Web Application Testing in Ruby, is a set of Browser Driver libraries written in Ruby. By itself, Watir only supports Internet Explorer on Windows, but it becomes cross-browser and cross-platform by extending Selenium Web Driver libraries. Watir tests are easy to read and modify thanks to the Ruby language, so that Watir libraries are often extended by other tools using Ruby, such as Acceptance Testing frameworks like Cucumber. Watir is already cross-language, but there are also many tools similar to Watir, written in different languages as Java (Watij), .NET (WatiN), Perl (Win32-Watir), presenting the same capabilities. Watir is already used by major companies like HP, Oracle, SAP, Yahoo and Facebook.

WebUI

WebUI [54] is a proprietary Recorder tool by Telerik, for both desktop and web applications. It comes in two flavours, a stand-alone version or a plugin for Visual Studio developers. WebUI is only for Windows, but it can support various browsers: Internet Explorer, Mozilla Firefox, Chrome and Safari. Tests are scripted in C# or VB.NET, but using the Capture&Replay interface does not require programming knowledge.

Windmill

Windmill [55] is a Browser Driver, open-source testing environment with Capture&Replay capabilities, very similar to Selenium. Just like Selenium, it supports Internet Explorer, Mozilla Firefox, Chrome, Safari and Opera, but it is written in Python. Windmill tests can be written and generated not only in Python, but also in Ruby and JavaScript. Windmill also employs XPath for identifying DOM elements.

7.2.2 Comparative Analysis

The tools we have described fairly represent the present panorama of (AJAX) web testing tools. This section is intended to analyze the differences between the tools, and highlight the contribution brought in by CrawlMan. Table 7.1 summarizes the main aspects of the tools.

The implications of Table 7.1 are as follows. First of all, CrawlMan is currently the only AJAX testing tool employing crawling techniques. Web automated testing is often referred to as crawling, because a tool crawls through a website following a script listing the steps to execute on its pages, but we refer here to crawling as the action of a web crawler, a software program which can autonomously and systematically explore a web application, in our case Crawljax. The most similar tool is VeriWeb, but the project is actually unavailable, and there is no documentation of successful utilization of VeriWeb on an AJAX web application.

Some aspects of Browser Simulators must be clarified. As we said, they lack reliable support for JavaScript, which is a point in favor of Browser Drivers. HTMLUnit seems the most valid simulator on this point, being able to emulate Firefox and Internet Explorer as pointed in the table. Other simulators, like Imprimatur and MaxQ, do not even exercise the page and its elements, just the HTTP request and response. The suitability of these tools depends on the tests to be performed, for example they can easily execute file uploads, but drivers are rapidly integrating these possibilities. We can affirm that simulators are generally satisfactory for exercising a web server, but not the client interface itself, which often presents a complicated dynamic behavior in case of AJAX applications.

Regarding Browser Drivers, there are some of them, Watir and CrawlMan itself through Crawljax, which make use of Selenium Web Driver libraries. These share the same possibilities. Selenium and SlimDog are both indicated as Browser Drivers, considering their

Web Interface Testing Tools							
	AJAX	Simulator/ Driver	Script/ Record/ Crawl	Open Source	Cross Browser	Cross System	Scripting Language
CrawlMan	yes	driver	crawl	yes	IE FF CH HM	yes	XML
HTMLUnit	yes	simulator	script	yes	IE FF	yes	Java
HttpUnit	yes	simulator	script	yes	no	yes	Java
Imprimatur	yes	simulator	script	yes	no	yes	XML
MaxQ	yes	simulator	record	yes	no	yes	Jython XML
Sahi	yes	simulator	record	yes	yes	yes	JavaScript
Selenium	yes	driver	script	yes	IE FF CH HM	yes	C# Groovy Java Perl PHP Python Ruby Selenese
SlimDog	yes	driver	script	yes	HP	yes	SlimDog
Squish	yes	driver	record	no	IE FF CH SF OP	yes	JavaScript Perl Python Squish Tcl
VeriWeb	no	driver	crawl	no	yes	LNX SOL	Tcl DTD XML
Watir	yes	driver	script	yes	IE FF CH HM	LNX MAC WIN	Cucumber Ruby
WebUI	yes	driver	record	no	IE FF CH SF	WIN	C# VB.NET
Windmill	yes	driver	record	yes	IE FF CH SF OP	yes	JavaScript Python Ruby

LEGENDA			
Web Browsers		Operating Systems	
CH	Chrome	LNX	Linux
FF	Mozilla Firefox	MAC	Macintosh
HM	HTMLUnit	SOL	Solaris
HP	HttpUnit	WIN	Windows
IE	Internet Explorer		
OP	Opera		
SF	Safari		

Table 7.1: Web Interface Testing Tools overview.

internal utilization of simulators, HTMLUnit for Selenium and HttpUnit for SlimDog, as interchangeable with any possible real browser instance. At the moment this possibility is only available in Selenium and projects inheriting it.

The most interesting tools, and especially Browser Drivers, are the Capture&Replay tools which can act cross-system and cross-browser, helping the user create a basic test to modify and replicate. The category includes just two of the examined tools: Squish and Windmill. Some of the strong features of these tools are the clear depiction of test success or failure, ease of creating and modifying the tests, and intuitive definition of verification points. As those tools replicate a recorded session, it is relatively easy to verify if the session was correctly repeated. This is not the case of crawling tools, whose result can be different from time to time if a generic or random description of the elements to exercise is provided. It is important to emphasize these concepts in CrawlMan, to ensure valid testing capabilities.

Considering scripting languages, recording tools make it easy to modify the scripts by clearly showing to the user which action corresponds to every command. In addition Squish and Windmill both support scripting in JavaScript and Python. It is interesting how the scripts could be easily converted in other more user-friendly formats, for example using Cucumber [56], as Watir already does. In CrawlMan, it is possible for the user to save his session in XML and later reload it into the application. If the user really wants, it is possible for him to manually modify the script, but the use of the application and its dynamic suggestions is encouraged.

All tools use some means for identifying elements of the web pages, or widgets, by name, id, text or xpath. Claims on what is the best method vary, as the more stringent the definition of an element, the easier the test is going to fail when performed on a different version of a page containing that element. Especially the use of XPath directly connects the identification of an element with its position inside the DOM. Most tools, including CrawlMan, give the possibility to use XPath to refine the definition of elements. The possibility CrawlMan is adding is defining elements with a very general description, for example all tags of a certain type, which is not implemented by non-crawling tools. VeriWeb is partially implementing this feature by simply clicking every web element which is usually employed to fire an event.

Chapter 8

Conclusions

We present in this chapter an overview of the project's contributions. The possibilities of CrawlMan are discussed and what differentiates it from other web testing frameworks. Finally we describe possible future additions to improve the project.

8.1 Contributions

CrawlMan is a framework for Automated Exploratory Testing of Web2.0 websites. It is, at the moment we write, a framework unique in its genre, using an AJAX crawler to automatically explore web applications, and being itself an AJAX web application. This is the original contribution of CrawlMan.

CrawlMan extracts the state flow graph constructed by Crawljax to present it to the user, together with the collected information about nodes and edges, in an orderly way. Crawljax is the only existing crawler capable to process AJAX web applications because it can reconstruct the succession of different states. CrawlMan takes advantage of this aspect by using the resulting data to build an interactive model for the user to inspect. The model helps the user understand the result and the application suggests new settings to the user through the model itself. This aspect makes CrawlMan a framework for Exploratory Testing, because the execution of a test is used to improve the test itself. The tests are performed in an Automated way.

It may seem a secondary aspect, but being a web application CrawlMan does not need installation. The crawling process is performed on the server side, so that the resources of the user computer are untouched. The load on the client side of the application has been lowered to the minimum, by having the user enable the more computational intensive features only when he needs them. CrawlMan does not require installation, so that the tests do not depend on the user machine and are performed in an independent environment.

8.2 Research Questions Revisited

Here, we answer the questions raised by this project (Section 1.2), establishing if the goals of the project were successfully met.

First, it is possible for a common user to set CrawlMan to reproduce a specific web application behavior and understand the result. Tests were performed with chosen users with no experience of programming or HTML to demonstrate this aspect. The tests also showed that insight on the working of Crawljax can help the user in the process.

Second, the user tests also demonstrated that CrawlMan can be used to bring an application in the desired state. The report implemented in CrawlMan to analyze the result has proved useful to indicate potential faults, pointing the user in the right direction to correct his settings.

Finally, CrawlMan is useful for automated exploration of a web application. It cannot substitute other testing tools, but it can offer an interesting and diverse point of view of a website functionalities. CrawlMan represents a possibility that did not exist before, opening new testing panoramas for web developers as well as end users.

8.3 Reflection

The capabilities and ease of use of CrawlMan were demonstrated during the evaluation tests of Chapter 6. The tests also showed that the usefulness of CrawlMan depends on the understanding of the settings and the result by the user. We observed that knowledge about the working of Crawljax and the crawling process itself helps the user interpret the result, and appropriately modify the settings. As the information given during the tests was sufficient for unexperienced users to successfully use CrawlMan, we decided to include Crawljax-related information and the evaluation tests inside the CrawlMan website.

CrawlMan offers Automated Exploratory Testing of AJAX web applications, which means it offers a point of view different from other tools to web testers. It can be used like a ‘Capture & Replay’ tool, but only if the user is able to define very specific settings, in which he is helped by the result suggestions. The same test can be repeated on different browsers, and possibly on different operating systems taking advantage of the possibilities of Selenium libraries. Unfortunately, the capabilities of CrawlMan have limitations. It is more complicated to use than a ‘Capture & Replay’ tool, where everything a user is asked to do is perform a sequence of actions inside his browser, to have these actions repeated whenever they are needed. In CrawlMan, the user mostly gives a direction for the test to follow. Building a sequence of definite actions to repeat is done by executing various crawling attempts, which considerably lengthens the time needed to build an articulated test.

Even if the user can successfully define a complicated test involving a sequence of definite actions to perform on a website, the outcome of these actions cannot be verified in

a direct way as in 'Capture & Replay' tools. The settings of CrawlMan are description of HTML elements, so that every time the same description is encountered, the corresponding setting is applied. There is a one-to-multiple relationship between the settings of CrawlMan and the resulting data, instead of a one-by-one relationship as it happens for other tools. This leaves the interpretation of the result open to the user, who is required to know and understand the website under test.

Contrary to other tools, CrawlMan cannot execute a suite of tests, but every test must be executed independently. The functionality can be easily added, but it was decided to momentarily leave out such a feature because it would enormously augment the load on the hosting server. Furthermore, CrawlMan cannot be used to test file upload and download functionalities, as the file should be positioned on the server itself. Due to Crawljax limitations, it also cannot currently crawl IFrames.

8.4 Future work

We discussed in Section 8.3 how CrawlMan could be expanded with the possibility of executing a suite of tests. In this case it would be also useful to provide a printable report, maybe even incorporating the log produced by Crawljax. The addition of these features would require observation on the utilization of CrawlMan by real users, collecting important data as the server load to ascertain the features feasibility and benefit.

CrawlMan is hosted by the SPCI server of the EWI faculty of TU Delft. The server runs an Ubuntu operative system, so that the crawling process takes place only in an Ubuntu environment. Crawljax, through the Selenium libraries, offers the possibility to perform the crawling on a remote server which runs the actual browser instances. It would be interesting to have other servers available, running different OSs, to offer the user the possibility to perform the same test not only on different browsers, but also different environments.

Only limited stress tests were performed, as for extensive stress testing a pool of many more users should have been involved. Monitoring the user activity on the server could indicate other directions for expanding. Also, security tests should be performed to ensure the safety of the server. Google Web Toolkit already enforces correctness of requests and other simple security measures. Other necessary aspects, such as escaping HTML input and validation of inserted URL, *Max States* and *Max Time* were implemented. A form of avoiding an infinite loop where the user tries to use CrawlMan to crawl CrawlMan itself was also integrated in the project, but more specific security tests in different directions should be conducted.

We talked about the possibility of adding existing plugins offered by Crawljax to the project (Section 3.1.1). The plugins were not added as they were not needed for the aim of the project, still some functionalities they offer could be adapted to CrawlMan's use. For example, the possibility to create a JUnit test from a crawling session for the user to download is particularly interesting, but it was not considered at the moment because the project was especially intended for use by non-programmers.

Bibliography

[1] S. Lenselink, "Concurrent Multi-browser Crawling of Ajax-based Web Applications," MSc Thesis, SERG, TU Delft, 2010.

[2] *Crawljax: Writing Plugins*. Delft, NL: SERG, TU Delft. http://crawljax.com/documentation/writing-plugins/, July 2009.

[3] T. O'Reilly, "What is web 2.0: Design patterns and business models for the next generation of software," in *Communications and Strategies*, vol. 65, p. 17, ITS Communications and Strategies, 2007.

[4] J. J. Garrett, *AJAX: A new approach to web applications*. San Francisco, CA, USA: Adaptive Path. http://adaptivepath.com/publications/essays/archives/000385.php, February 2005.

[5] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling AJAX by Inferring User Interface State Changes," in *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)* (D. Schwabe, F. Curbera, and P. Dantzig, eds.), pp. 122–134, IEEE Computer Society, July 2008.

[6] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes," in *ACM Transactions on the Web (TWEB)*, ACM, 2011.

[7] A. Mesbah and A. van Deursen, "Invariant-Based Automatic Testing of AJAX User Interfaces," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers*, pp. 210–220, IEEE Computer Society, 2009.

[8] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-Based Automatic Testing of Modern Web Applications," in *IEEE Transactions on Software Engineering (TSE)*, vol. 37 issue 6, pp. 1–37, IEEE Computer Society, 2011.

[9] A. Mesbah and A. van Deursen, "An Architectural Style for AJAX," in *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)* (D. Paulish, I. Gorton, J. Tyree, and D. Soni, eds.), pp. 44–53, IEEE Computer Society, 2007.

[10] A. Mesbah and A. van Deursen, "A Component- and Push-based Architectural Style for Ajax Applications," in *Journal of Systems and Software*, vol. 81, no. 12, pp. 2194–2209, Elsevier, 2008.

[11] *Facebook*. Palo Alto, CA, USA: Facebook Inc. http://www.facebook.com/, February 2004.

[12] *Gmail: Email from Google*. Mountain View, CA, USA: Google Inc. http://mail.google.com/mail/, April 2004.

[13] *Wikipedia - The Free Encyclopedia*. San Francisco, CA, USA: Wikimedia Foundation Inc. http://www.wikipedia.com/, July 2004.

[14] *Ebay*. San Jose, CA, USA: Ebay Inc. http://www.ebay.com/, September 1995.

[15] *Google Web Toolkit - Google Code*. Mountain View, CA, USA: Google Inc. http://code.google.com/webtoolkit/, May 2006.

[16] *Google*. Mountain View, CA, USA: Google Inc. http://www.google.com/, September 1998.

[17] M. Kobayashi and K. Takeda, "Information retrieval on the web," in *ACM Computing Surveys (CSUR)*, vol. 32, pp. 144–173, ACM, 2000.

[18] *Yahoo*. Sunnyvale, CA, USA: Yahoo Inc. http://www.yahoo.com/, March 1995.

[19] *Bing*. Redmond, WA, USA: Microsoft Corporation. http://www.bing.com/, June 2009.

[20] *Internet Explorer*. Redmond, WA, USA: Microsoft Corporation. http://windows.microsoft.com/en-US/internet-explorer/products/ie/home, August 1995.

[21] *Mozilla Firefox*. Mountain View, CA, USA: Mozilla Corporation. http://www.mozilla.org/en-US/firefox/new/, November 2004.

[22] *Google Chrome*. Mountain View, CA, USA: Google Inc. http://www.google.com/chrome, September 2008.

[23] *HTML & CSS standards*. Cambridge, MA, USA: World Wide Web Consortium (W3C). http://www.w3.org/standards/webdesign/htmlcss, June 2011.

[24] *Document Object Model (DOM)*. Cambridge, MA, USA: World Wide Web Consortium (W3C). http://www.w3.org/DOM/, January 2009.

[25] *Software Engineering Research Group (SERG), TU Delft*. Delft, NL: http://swerl.tudelft.nl/bin/view/Main/WebHome, 2009.

[26] *XML Path Language (XPath)*. Cambridge, MA, USA: World Wide Web Consortium (W3C). http://www.w3.org/TR/xpath/, November 1999.

[27] *Eclipse - The Eclipse Foundation open source community website*. Ottawa, ON, CA: Eclipse Foundation Inc. http://www.eclipse.org/, January 2004.

[28] *Apache Maven Project*. Forest Hill, MD, USA: Apache Software Foundation. http://maven.apache.org/, June 1999.

[29] *Alexa Top 500 Global Sites*. San Francisco, CA, USA: Alexa - The Web Information Company. http://www.alexa.com/topsites/global, April 1996.

[30] *Top 1000 sites - DoubleClick Ad Planner*. Mountain View, CA, USA: Google Inc. http://www.google.com/adplanner/static/top1000/, June 2008.

[31] *PayPal*. San Jose, CA, USA: PayPal Inc. http://www.paypal.com/, December 1998.

[32] *Mediafire - Free File Sharing Made Simple*. Harris County, TX, USA: PayPal Inc. http://www.mediafire.com/, October 2006.

[33] *WordPress.com - Get a Free Blog Here*. San Francisco, CA, USA: WordPress Foundation. http://wordpress.com/, May 2003.

[34] ISO, *ISO/TR 16982:2002 - Ergonomics of human-system interaction – Usability methods supporting human-centred design*. ISO, Apr. 2005.

[35] J. Nielsen, *Usability Engineering*. Interactive Technologies, San Francisco, CA, USA: Morgan Kaufmann, September 1993.

[36] J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Boston, MA, USA: Addison-Wesley Professional, September 2009.

[37] J. Bach, "Exploratory testing explained," in *The Testing Practitioner*, (Den Bosch, NL), pp. 253–265, UTN Publishers, 2002.

[38] A. Zylberman and N. Shenar, *White Paper: Exploratory Automated Testing*. Norwalk, CT, USA: QualiTest Group. http://www.qualitestgroup.com/Automated_Exploratory_Testing.html, October 2003.

[39] H. Buwalda, *Key Success Factors for Keyword Driven Testing*. San Mateo, CA, USA: LogiGear Corporation. http://www.logigear.com/resource-center/software-testing-articles-by-logigear-staff/389–key-success-factors-for-keyword-driven-testing.html, November 2007.

[40] B. Pettichord, "Homebrew Test Automation - a One-Day Seminar," (Austin, TX, USA), http://www.pettichord.com/homebrew_automation.html, September 2004.

[41] G. Gheorghiu, "Agile testing: Web app testing with Python part 1: MaxQ," http://agiletesting.blogspot.com/2005/02/web-app-testing-with-python-part-1.html, February 2005.

[42] M. Bowler, "HTMLUnit." http://htmlunit.sourceforge.net/, May 2002.

[43] R. Gold, "HttpUnit." http://httpunit.sourceforge.net/, May 2000.

[44] T. Locke, "Imprimatur." http://imprimatur.wikispaces.com/, July 2006.

[45] J. Cooper, "MaxQ." http://maxq.tigris.org/, February 2003.

[46] N. Raman, "Sahi." http://sahi.co.in/w/, November 2005.

[47] G. Gheorghiu, "Tool Look: A Look at Selenium," in *Better Software* (H. Shanholtzer, J. McAllister, and L. Copeland, eds.), vol. 7, p. 38, Software Quality Engineering, October 2005.

[48] A. Mecky, "Slimdog." http://slimdog.jzonic.org/, November 2004.

[49] FrogLogic, "Squish - the cross-platform GUI test automation tool." http://www.froglogic.com/index.php, November 2003.

[50] J. Freire, M. Benedikt, and P. Godefroid, "Veriweb: Automatically testing dynamic web sites," in *Proceedings of the 11th International World Wide Web Conference (WWW2002)*, pp. 654–668, ACM, May 2002.

[51] P. Godefroid, "Model Checking for Programming Languages using Verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174–186, ACM, January 1997.

[52] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen, "Automating Web Navigation with the WebVCR," in *Proceedings of the 9th International World Wide Web Conference (WWW2000)*, pp. 503–517, ACM, May 2000.

[53] B. Pettichord, P. Rogers, and J. Kohl, "Watir: Web application testing in ruby." http://watir.com/, January 2005.

[54] Telerik, "Webui test studio." http://www.telerik.com/automated-testing-tools.aspx, February 2009.

[55] A. Christian, A. Goucher, and T. Riley, "Web application testing with Windmill," in *Beautiful Testing: Leading Professional Reveal How They Improve Software*, p. 352, O'Reilly Media, October 2009.

[56] M. Wynne and A. Hellesøy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Lewisville, TX, USA: The Pragmatic Bookshelf, January 2012.

Appendix A

Introduction to CrawlMan

This appendix contains a small introduction on the use of CrawlMan, and the necessary basic information on dynamic web pages and HTML elements. CrawlMan settings use this information, as the crawling and the construction of a tree graph is done by analyzing differences in the HTML code of the visited pages.

A.1 Dynamic Pages

During the first era of the Internet, later denominated Web 1.0 [3], the World Wide Web was a collection of hypertexts, pages connected through each other by links and offering static information. The structure of a traditional website can be represented with a tree graph, where the homepage, or index, is the root of the tree, the other pages represent the nodes of the tree and the links between them are the edges of the tree. An example is shown in Figure A.1.

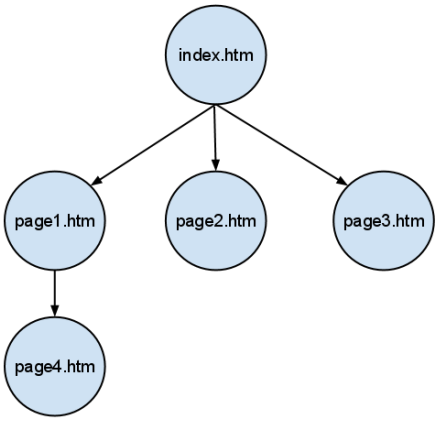

Figure A.1: Traditional website representation: every node corresponds to a different page.

Modern websites are instead composed by dynamic pages, changing their behavior according to the user actions, presenting content without the necessity of reloading. They more and more resemble desktop software, that is why we speak of web applications and Web 2.0 [3]. As new content does not correspond to different pages, we are forced to abandon the previous representation of websites. Web applications can still be imagined as tree graphs, where the root is the homepage, but the other nodes do not correspond to static pages, with different URLs. The nodes instead represent now states of the application, and the edges between them are the user actions which make the application change, as shown in Figure A.2.

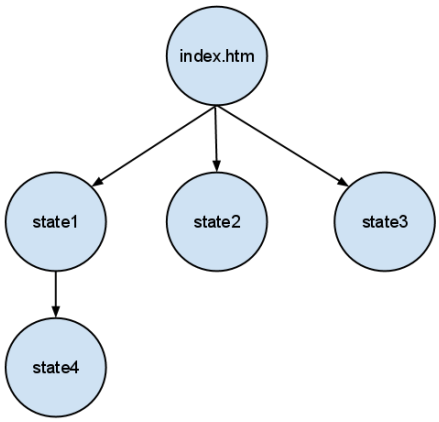

Figure A.2: Web application representation: the nodes are different states assumed by the application.

Contrary to other web testing tools, CrawlMan is able to present to the user a graph representation of the states of a web application, extracted by Crawljax [5, 7, 9]. This gives the user a clear idea of the working of the application, which is instead represented by other tools as a simple sequence of user actions, ignoring the fact that different actions bring the application in different states. The result of a crawling session is presented in CrawlMan as a tree graph, and information about the retrieved states is visualized when selecting a node.

A.2 HTML Fundamentals

A web page, as any person sees it when navigating through the Internet via a web browser, is a collection of data - text, active elements and multimedia content. Figure A.3 shows an example of web page.

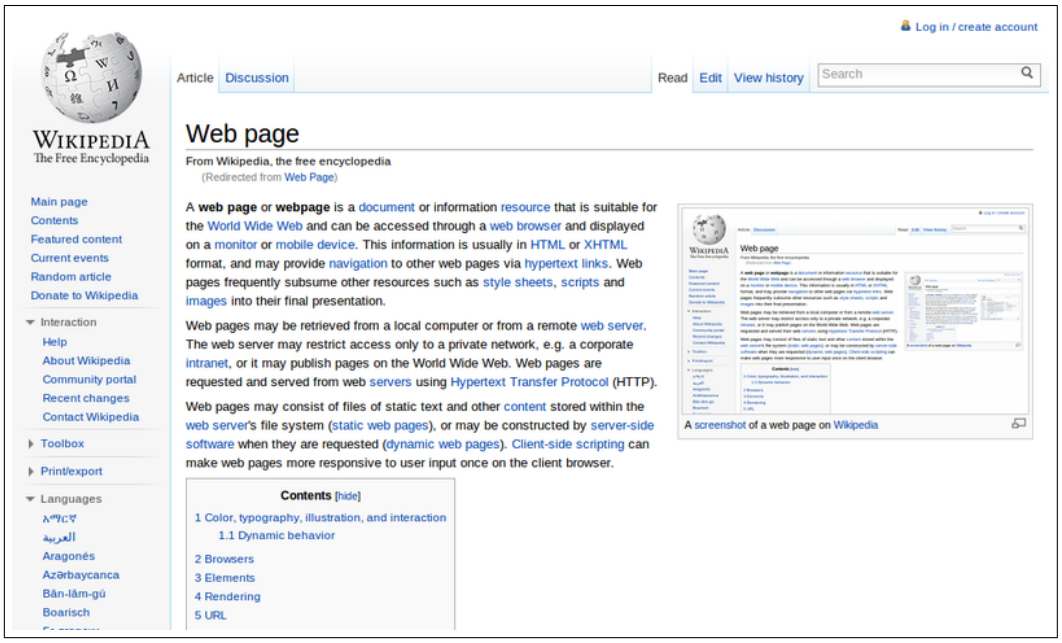

Figure A.3: Wikipedia web page describing web pages.

In reality, the page we see through a browser is the browser representation of the HTML code formatting of the data content, plus CSS styles and embedded functions. The representation follows a standard, so that a developer expects the code to be represented and behave the same way in every browser, but that is really up to the browser implementation. This means that elements we see on the page are actually HTML elements. For example the code for a button appears as in Figure A.4.

```
<button id="buttonId" type="submit">button text</button>
```

Figure A.4: HTML code for a button.

Every HTML element is formed by a tag (in this case 'button'), element attributes ('id' and 'type' in the example), and some content, which can be text or other HTML elements. The same HTML page is a HTML element, with tag 'html', and containing the 'head' and 'body' elements. The head of the document contains title, styles and functions, while the body contains all the other content we actually can see when visiting the page. Figure A.5 shows the code for a web page containing the previous button.

```
<html>
  <head>
    <title>My page</title>
  </head>
  <body>
    <button id="buttonId" type="submit">button text</button>
  </body>
</html>
```

Figure A.5: HTML code of a page containing a button.

Given the structure of HTML code, the page source can be itself represented as a tree, whose nodes are HTML elements. Figure A.6 is a tree representation of the previous code.

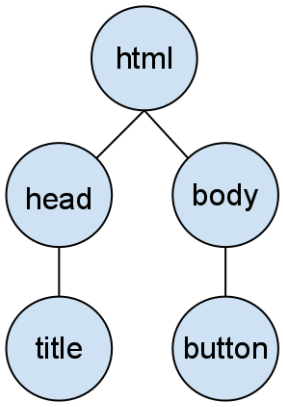

Figure A.6: Tree representation of HTML code.

HTML elements can be referenced by their tag, important attributes as id or name, or their position inside the HTML page. The standard language for defining elements by their position, and the one used by CrawlMan, is XPath. XPath takes advantage of the tree like structure for referencing elements. For example, the button in the previous code is referenced as shown in Figure A.7.

```
/html[1]/body[1]/button[1]
```

Figure A.7: XPath description of button.

The xpath of the button describes it as the first HTML element with tag ‘button’, inside the first element with tag ‘body’, inside the first element with tag ‘html’, although there cannot be more than one html and body elements inside the page. XPath also allows more general descriptions, like referencing all the buttons in a page, or at a certain level inside its tree representation, and it can even directly point to element attributes.

A.3 Using CrawlMan

Now that we know the basics of HTML pages and XPath, we are ready to use CrawlMan. Is it that simple? Of course, building complicated behavior will require more specific expertise, and some knowledge of JavaScript, a language used for embedding functions in web pages, could result useful, but CrawlMan is designed to guide you into your web exploration, so that an open, inquisitive mind should be all that you need for the moment.

First of all, CrawlMan comes with many different settings, but it is ready for use with default settings which are normally good for every website you would like to explore. For example, it is set to click on every button and link it finds, and to fill every input field with random data. In order to rapidly produce a result, a maximum size for the result and a maximum time to produce it are also set. The default browser to use is HTMLUnit, not a real browser but a browser simulator, which further reduces the crawling time. These settings can be modified in the ‘Basic Settings’ page, but it is advised not to modify them before receiving a first result.

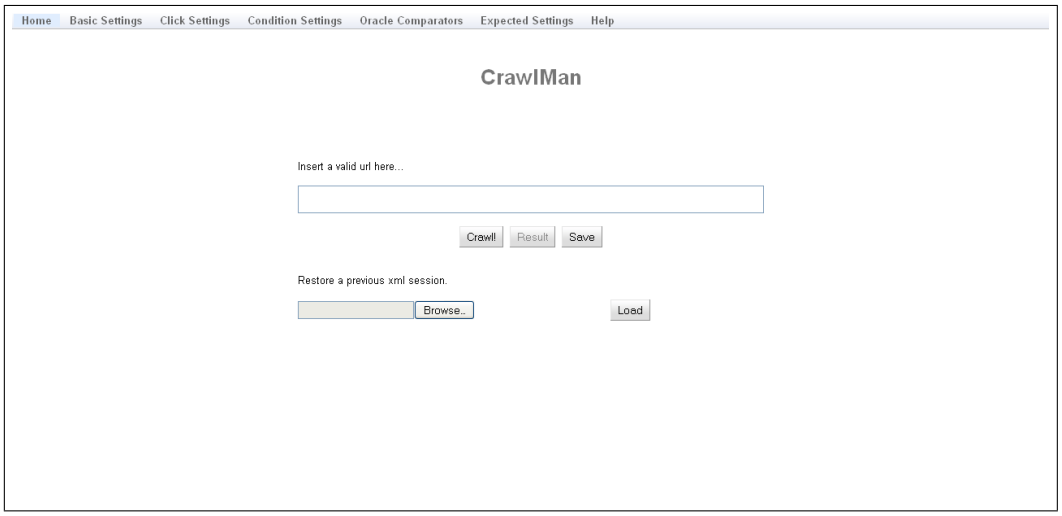

Figure A.8: CrawlMan’s homepage.

In order to start using CrawlMan, load its homepage, shown in Figure A.8, insert the desired URL, starting with ‘http’ or ‘https’, in the appropriate text box, then press the ‘crawl’ button. After a certain time, depending on the settings and the amount of data inside the

crawled pages, a result is returned. To visualize the result press the 'result' button, which should now be enabled. Figure A.9 shows the 'Result' panel when crawling Google's home-page. The result is presented as a tree, whose nodes can be clicked to visualize the relative information. A double click on a node makes the relative subgraph representation fold or unfold.

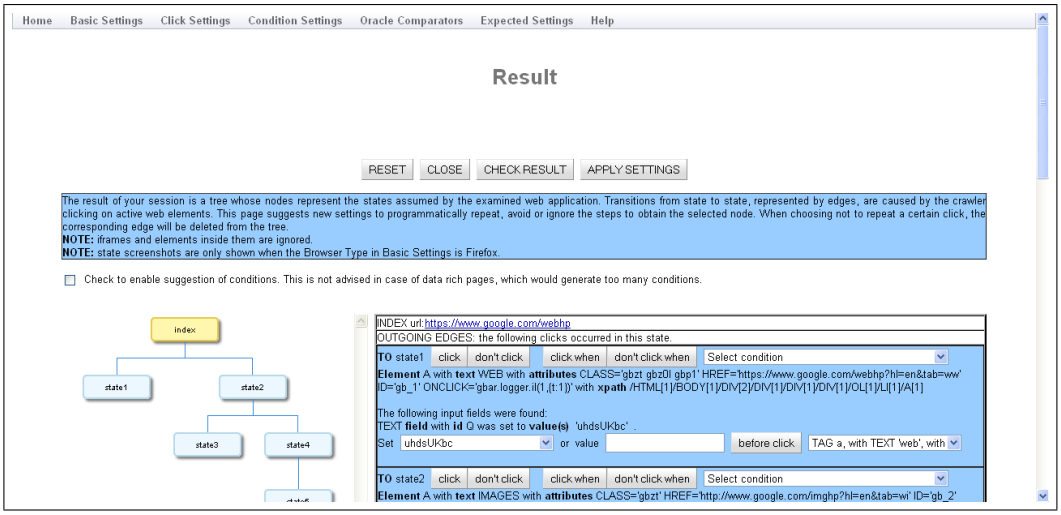

Figure A.9: CrawlMan 'Result' page for URL http://www.google.com/webhp.

As explained in Section A.1, edges between the nodes represent events which brought the application in the different states, as the clicking of a button or the changing of a variable. In the tree, the user can see only one edge connecting two nodes, but there actually can be more than one, and they can be directed from a child node to a parent node, if an event makes the application return to a previous state. The only way for a user to know the events connecting the states is inspecting node information. When a node is clicked, the following data is presented:

- URL address relative to the state.
- HTML DOM representation.
- A screenshot of the state, in case CrawlMan was set to use Firefox in the 'Basic Settings' page.
- A set of **Incoming Edges**, describing click and input field events which brought into the state.
- A set of **Outgoing Edges**, describing click and input field events which brought outside of the state.
- A set of **Other Edges**, representing clickables which were not clicked or did not fire an event when clicked.

As Crawljax can only perform click actions, the edges describe HTML elements that were clicked, with their tag, text, attributes and xpath. For every edge description, buttons appear to create corresponding *Click* and *Don't Click* settings to replace the default settings. Other buttons allow to create *Set Values Before Click* settings, using the retrieved information about input fields in the page. This setting type allows to enter some value in an input field before clicking on a certain element. For input fields like check boxes and radio boxes, the user can insert boolean values like 'true' or '1', and 'false' or '0'. Apart from simple click settings, CrawlMan uses the result to suggest:

- *Click When* settings, to click a HTML element when the specified condition is true.
- *Oracle Comparator* settings, to ignore specific parts of pages.
- *Crawl* settings, to crawl a page only is the specified condition is true.
- *Invariant* settings, to verify a condition on every state - the outcome is presented when clicking on the 'check result' button.
- *Wait For* settings, to wait for an element to be loaded or become visible on the specified URL.

Suggested conditions are relative to the selected state, but during the crawling are verified against all of the states. In fact, applied settings can only direct the crawling, and there is no assurance that the crawling session will be repeated the same over and over. Settings describe a desired behavior, for example a *Click* setting will cause the application to click on every element corresponding to the description, not just the element we took the description from. If we want the click to be executed on just one particular element, we can restrict the description with **xpath** and **conditions**.

When the behavior we want to reproduce is well described by the created settings, it is better to uncheck the *Click Default* and *Random Input* settings in the 'Basic Settings' page. The *Random Input* setting could in fact conflict with eventual other settings which fill input fields, and *Click Default* could interfere with other *Click* settings. This last setting in fact is comparable to adding two general click settings, one for clicking all buttons (all elements with tag 'button') and one for clicking all links (also called anchors, all elements with tag 'a'). The only reason why anchors and buttons appear in the result is the enabling of this default setting. The same is valid for all other possible clickables. For example, if we need the application to click on every image it finds and present all image HTML elements in the result, we will have to add a setting to click on every element with tag 'img'.

In case one of the selected setting in the result would cause a state not to be reached anymore during the next crawling, the node gets separated from the rest of the tree. As it is necessary for the crawler to reach a state before executing actions on it, CrawlMan also creates the necessary settings to reach the selected state when adding some of the suggested settings. The created settings can be confirmed in the result page, then modified in the relative pages. As the same settings could already be present, it is up to the user to select only the settings he really needs.

A.4 Crawling Process

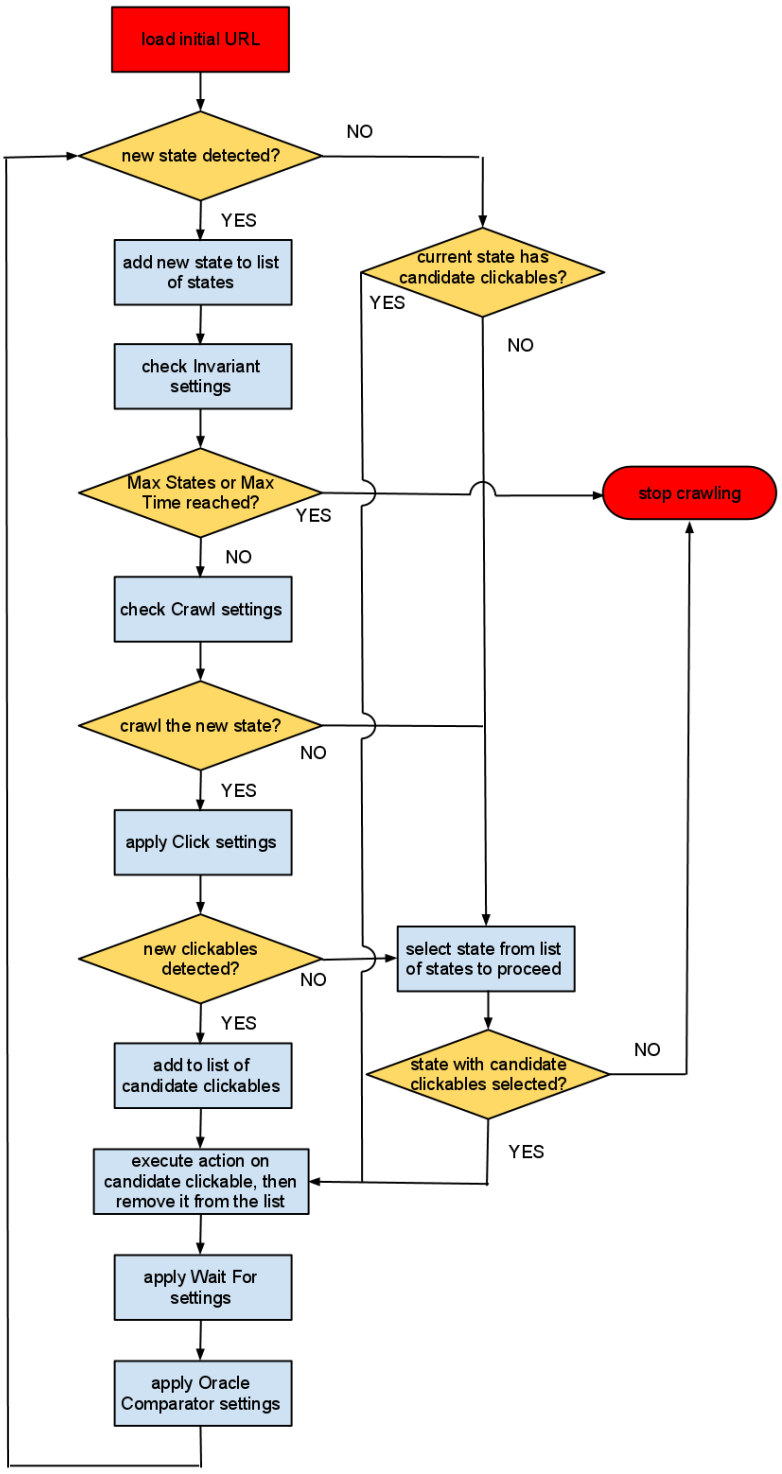

Figure A.10: Crawljax decisional flowchart.

When a crawling request is sent through CrawlMan, the request is converted to commands to a crawler, Crawljax, present on the server. Crawljax opens a browser instance to load the initial URL, then it starts crawling. First of all, the initial URL must correspond to a valid address, of a website which can be reached online. If a new state is detected, the state information is saved for more processing. *Invariant* settings are checked against the new state, then Crawljax decides to continue crawling or not on the base of the limiting settings, *Max States* and *Max Time*. Based on *Max Depth* and *Crawl* settings, Crawljax decides to proceed with the newly discovered state or a previous state which still has clickables which where not clicked. If the new state is processed, *Click* settings are applied to look for new clickables. The clickables found are saved and exercised one-by-one, until all clickables have been tried or a new state is found. The crawling process stops when the crawling constraints are reached or all the compliant clickables of all the suitable states are exercised.

Appendix B

Crawljax Modifications

B.1 Scope of the Document

This document is a technical report, related to the *CrawlMan*'s project. Here we list the modifications applied to the libraries of *Crawljax*. Crawljax is an open-source project, developed in Java. Crawljax's source code is available online for anyone to download, use, and modify, at the address `http://code.google.com/p/crawljax/`. The modifications we applied were necessary to adapt Crawljax to the use intended in the CrawlMan's project. The modifications can be divided in three groups:

- Adaptations for CrawlMan's necessities.
- Improvements on the code reliability.
- Addition of other functions.

B.2 Adaptations for CrawlMan's necessities

Google Web Toolkit, used for CrawlMan's development, requires the source code of the inherited libraries. The line of Crawljax's POM file shown in Figure B.1 had to be uncommented.

```
<resource> <directory>src/main/java</directory> </resource>
```

Figure B.1: POM setting for including the Java sources in the packaging of a Maven project.

One modification to Crawljax was really necessary because it hindered the possibility to use Crawljax in a web environment, as needed by CrawlMan. In the original version

of Crawljax it is not possible to use plugins to collect data at runtime when using two different instances of Crawljax in parallel (Section 5.2.2). The problem appears when the data collected during the crawling is used outside the plugins, as opposed to the traditional way of using plugins which is saving the data through a *GeneratesOutput* plugin during the execution of Crawljax (Section 2.4.12), at the end of the crawling process.

In CrawlMan, a user sends a crawling request from his browser to the server hosting CrawlMan. Every request in a web server is serviced by a different thread, which is instance of a web service. This thread adds a plugin to Crawljax in order to collect the result data, then calls Crawljax to execute the crawling. At the end of the crawling, the thread gets the result from the plugin and sends it back to the user.

```
private static final List<Plugin> PLUGINS = Lists.newArrayList();

/**
 * Set the Plugins, first removes all the currently loaded
 * plugins and add the plugins supplied.
 *
 * @param plugins
 *            the list of plugins,
 * if plugins is null no plugins are added.
 */
public static void loadPlugins(List<Plugin> plugins) {
  PLUGINS.clear();
  if (plugins == null || plugins.size() == 0) {
    LOGGER.warn("No plugins loaded because "
      + "CrawljaxConfiguration is empty");
    return;
  }
  CrawljaxPluginsUtil.PLUGINS.addAll(plugins);
  for (Plugin plugin : CrawljaxPluginsUtil.PLUGINS) {
    /**
     * Log the name of the plugin loaded
     */
    LOGGER.info(plugin.getClass().getName());
  }
}
```

Figure B.2: CrawljaxPluginsUtil's loadPlugins(List <Plugin> pluginList) method.

The plugins are added to Crawljax through a static class, called **CrawljaxPluginsU-**

til. The plugins are loaded by the method *loadPlugins(List <Plugin> pluginList)*, which first removes all the currently loaded plugins from the static list PLUGINS, as shown in Figure B.2.

When two or more request are sent to the server in parallel, two or more threads add plugins to Crawljax, so that every request deletes the plugins added by the previous request. When the threads try to extract the result from the added plugin, they find the result to be null, because it has already been deleted by the requests after them. Only the last request of all, given that no request has been sent after it, would receive a result. The problem also appears when calling two instances of Crawljax as two threads inside a class, then waiting for their resolution through a semaphore and verifying the presence of the result collected through a plugin.

Two solutions were explored:

- Removing the line '*PLUGINS.clear();*'. The plugins would be instead eliminated at the end of the crawling process.
- Making possible to instantiate the class *CrawljaxPluginsUtil*.

The clear downside of the first solution is that every program which uses Crawljax should be modified to accommodate the change. Another downside is that in presence of serious errors it is possible that the plugins would not be removed, so that the class would accumulate outdated data.

The second solution was applied. It has not been studied the influence of the changes on the usage of Crawljax's *Thread* settings, for which the *CrawljaxPluginsUtil* class was made static in the first place. This is due to the lack of an example showing the intended usage of the *Thread* settings together with plugins, and to the fact that the *Thread* settings are out of the scope of the CrawlMan's project. It can be however assured that Crawljax passes all the provided functional tests after the change.

Given the change to the class *CrawljaxPluginsUtil*, many other classes had to be changed in order to use an instance of this class instead of its static version. Overall, the change comported modifications of six core classes (CrawljaxPluginsUtil, Crawler, InitialCrawler, CrawljaxController, StateMachine, BrowserPool) and four test classes (PluginTest, OnFireEventFailedPluginTest, StateMachineTest, BrowserPoolTest).

B.2.1 CrawljaxPluginsUtil

The non instanceable constructor of this class was removed, and the static list of plugins was changed to a traditional list, as shown in Figure B.3.

```
private final List<Plugin> plugins = new ArrayList<Plugin>();
```

Figure B.3: List of plugins in class CrawljaxPluginsUtil.

The instructions using the list were changed accordingly.

B.2.2 CrawljaxController

Instructions were added in the class to create an instance of CrawljaxPluginsUtil in its constructor. The instance can be requested by other classes through the added method 'getPluginsUtil()', shown in Figure B.4.

```
private final CrawljaxPluginsUtil pluginsUtil;

/**
 * Return the pluginsUtil.
 */
public CrawljaxPluginsUtil getPluginsUtil() {
  return pluginsUtil;
}
```

Figure B.4: CrawljaxController's getPluginsUtil() method.

Given the modifications applied to the BrowserPool class, the constructor of CrawljaxController ends with the lines shown in Figure B.5.

```
public CrawljaxController(final CrawljaxConfiguration config)
  throws ConfigurationException {
  [...]
  pluginsUtil = new CrawljaxPluginsUtil();

  workQueue = init();

  browserPool = new BrowserPool(workQueue, configurationReader,
    pluginsUtil);
}
```

Figure B.5: CrawljaxController's modifications.

B.2.3 StateMachine

CrawljaxPluginsUtil was added as a parameter to the functions *update* and *checkInvariants*. As a consequence, the call to these functions has been modified in the relative classes - Crawler and StateMachineTest, as shown in Figure B.6.

```
public boolean update(final Eventable event,
  StateVertix newState, EmbeddedBrowser browser,
  CrawlSession session, CrawljaxPluginsUtil pluginsUtil)

private void checkInvariants(EmbeddedBrowser browser,
  CrawlSession session, CrawljaxPluginsUtil pluginsUtil)
```

Figure B.6: Modifications to functions calls.

B.2.4 BrowserPool

CrawljaxPluginsUtil was added as a parameter to the class constructor, as shown in Figure B.7.

```
private final CrawljaxPluginsUtil pluginsUtil;

public BrowserPool(CrawljaxConfigurationReader
  configurationReader, CrawljaxPluginsUtil pluginsUtil) {
  [...]
  this.pluginsUtil = pluginsUtil;
}
```

Figure B.7: Constructor for class BrowserPool.

B.2.5 Crawler, InitialCrawler, PluginTest, OnFireEventFailedPluginTest

Every usage of 'CrawljaxPluginsUtil' was substitued with 'controller.getPluginsUtil()'.

B.2.6 StateMachineTest

The use of the functions modified in the other classes was changed accordingly, by passing as parameter to the functions an instance of CrawljaxPluginsUtil, as shown in Figure B.8.

```
assertTrue(sm.update(c, state2, dummyBrowser,
  new CrawlSession(dummyPool), new CrawljaxPluginsUtil()));
```

Figure B.8: Modification to functions call.

The calls to CrawljaxPluginsUtil were substituted with the call to instances of the class, as shown in Figure B.9.

```
CrawljaxPluginsUtil pluginsUtil = new CrawljaxPluginsUtil();
pluginsUtil.loadPlugins(
  new CrawljaxConfigurationReader(cfg).getPlugins());
```

Figure B.9: Call to CrawljaxPluginsUtil's loadPlugins(List <Plugin> pluginList) method.

B.2.7 BrowserPoolTest

The class was changed to use the new BrowserPool constructor.

B.3 Improvements on the code reliability

During the testing, we encountered situations which originated irreversible errors during the execution of Crawljax. After these errors, the control was not given back to the main thread, CrawljaxController, so that the user was left expecting for a result that would never arrive. It was necessary to modify the code dealing with these errors to give the control back to the main thread. The modifications, applied to three classes, are explained in the following sections.

B.3.1 InitialCrawler

When a browser instance is not obtained, the InitialCrawler is not immediately terminated. The control is not given back to the CrawljaxController, which results in more errors and a faulty termination. The problem appears for example in the absence of a display environment when using a real browser for the crawling process. The solution was to return the process when the browser is null, as shown in Figure B.10.

```
public void run() {
  try {
    browser = controller.getBrowserPool().requestBrowser();
  } catch (InterruptedException e) {
    LOGGER.error("The request for a browser was interrupted.");
  }

  /**
   * Aborted session.
   */
  if (browser == null) {
    return;
  }

  goToInitialURL();

  [...]
}
```

Figure B.10: Terminating the InitialCrawler.

B.3.2 BrowserPool

A different constructor was added to the class BrowserPool, to include as parameter the CrawlerExecutor. The new constructor is used in the CrawljaxController in place of the old one, which is used only in the test classes BrowserPoolTest and StateMachineTest. The CrawlerExecutor is needed to stop the crawling process when the creation of the browser cannot be rescued. Figure B.11 shows the new constructor for the BrowserPool class.

```
private CrawlerExecutor executor;

public BrowserPool(CrawlerExecutor executor,
  CrawljaxConfigurationReader configurationReader,
  CrawljaxPluginsUtil pluginsUtil) {

  this(configurationReader, pluginsUtil);
  this.executor = executor;
}
```

Figure B.11: BrowserPool's new constructor.

The line to shutdown the executor was added in the *run* method, as shown in Figure B.12.

```
public void run() {
  [...]

  failedCreatedBrowserCount.incrementAndGet();
  LOGGER.error("Could not rescue browser creation!", e);
  executor.shutdownNow(true);

  [...]
}
```

Figure B.12: Terminating the CrawlerExecutor.

B.3.3 FormHandler

The function *handleFormElements* of this class is used to fill in the input elements of the web pages. When a *Set Values Before Click* setting of Crawljax specifies a valid input field together with an absent one, the method is executed on a null object, resulting in an uncaught exception. The for statement of the function was modified to the code in Figure B.13.

```
for (FormInput input : formInputs) {
  // Input is null when the specified input does not exist.
  if (input != null) {
    LOGGER.debug("Filling in: " + input);
    setInputElementValue(formInputValueHelper
      .getBelongingNode(input, dom), input);
  } else {
    LOGGER.warn("Input is null.");
  }
}
```

Figure B.13: Modifications to the class FormHandler.

B.4 Addition of other functions

The class StateVertix was modified for adding one function, *getUnprocessedCrawlActions()*. The function is built on the model of *getUnprocessedCandidateElements()*, and practically returns the same result, candidate elements, but taking into account the type of the candidate elements. In fact, because Crawljax uses at the moment only *click actions*, the type of the actions is omitted by the original function. The new function takes the type into account, in preparation for the future version of Crawljax which will also execute *hover actions*. This modification was not indispensable for the well functioning of Crawljax or CrawlMan. The function, shown in Figure B.14, is employed in CrawlMan's plugin to collect data about candidate crawl actions.

```
/**
 * Return a list of UnprocessedCrawlActions.
 *
 * @return a list of crawl actions which are unprocessed.
 */
public List<CandidateCrawlAction> getUnprocessedCrawlActions() {
  List<CandidateCrawlAction> list =
    new ArrayList<CandidateCrawlAction>();
  if (candidateActions == null) {
    return list;
  }
  for (CandidateCrawlAction candidateAction : candidateActions) {
    list.add(candidateAction);
  }
  return list;
}
```

Figure B.14: StateVertix's getUnprocessedCrawlActions() method.