UNIVERSITEIT VAN AMSTERDAM
Faculty of Science

Automated detection of unused CSS style rules by crawling web applications

Master Thesis

Kevin Adegeest

Supervisors:
Dr. Vadim Zaytsev

Version 1.0

Amsterdam, August 2015

Abstract

Cascading Style Sheets (CSS) is a style sheet language used for describing the look and formatting of a document written in a markup language. The most common application is the styling of websites. When websites grow so do their CSS files and with them the amount of unused selectors. These result in larger bandwidth usage, less responsive websites, and higher CPU load for users. Manually detecting the usage of selectors is a difficult and time intensive task. In many cases all possible states of a website must be manually visited to verify the existence of a selector. Still, limited research has been done on the automated detection of unused selectors.

We present an automated technique to crawl websites and detect unused selectors. Additionally we present a method to detect dynamic states cause by JavaScript. Results show that our crawler is capable of finding 88.5% of all web pages detected by Google and four other crawlers. Evaluations show that the style detector has an accuracy of 100% and is capable of detecting pseudo-classes and -elements. Tests done on five websites revealed that an average of 77% of all selectors is unused.

Contents

Chapter 1

Introduction

1.1 Relevance

Cascading Style Sheets is one of the fundamental standards for developing web applications [1]. It is a language for defining the presentation of HTML elements. This includes the colors, layout styles, but also the position and fonts to use. CSS separates the presentation from the applications structure, which increases maintainability in web applications. Unfortunately CSS code itself is not easy to maintain [13].

The structure of a web application is defined by its Document Object Model (DOM). The presentation, defined by CSS rules, is applied to the elements of the DOM tree. The DOM consists of HTML elements and can be altered with JavaScript.

In large web applications with dozens of DOM states one can easily miss the effect a change in the CSS rules has on the application. With the evolution of an application some rules may become obsolete. Due to difficulty of finding whether a specific rule is still being used, they often remain untouched. Over time this accumulates to a large file with a certain degree of unused or overridden rules. Another cause for unused rules is the use of CSS frameworks, such as Twitter Bootstrap. Earlier research has shown that up to 60% of rules remained unused [11]. This has certain consequences:

- All CSS code has to be downloaded before the browser can apply the styles to a website. The larger the size, the longer it will take for the rendering to start[14].
- Larger CSS code results in more data usage, which is undesirable and may be costly
- The web-browser is CPU intensive and rendering the layout consumes between 40 and 70% of total processing time [8]. When rendering a page, each rule is being applied to each element in the DOM tree. Thus, many unused rules require unnecessary checking.

By reducing the amount of unused rules one reduces the data usage, increases rendering speed of a website and thus responsiveness, and reduces battery drain.

1.2 Related research

Limited research has been done on the improvement and analysis of CSS. The main contributions to finding unused rules have been from the teams behind Crawljax [12] and CILLA [11]. To our knowledge no other research has used this particular approach. Other approaches used to detect unused rules are tree-rewriting [7], refactoring style sheets by detecting and removing clones [10], or using tree-logic to statistically refactor CSS [4].

The industry, however, has spawned tools for maintaining CSS code. One such type of tool are preprocessors such as SASS[1] and Less[2] that increase the maintainability of code with structuring, use of variables, and more. Another type is unused rule detectors, such as uncss[3], css-usage[4], Dust-Me Selectors[5]. Due to not being able to crawl a web application these three detectors cannot detect selectors on a single page. Though Dust-Me-Selectors can be given a sitemap with URLs to visit.

1.3 Structure of the thesis

Chapter 2 explains the goals and research questions of this thesis. The software choice and other information not directly related to the research, but useful nonetheless, are listed in chapter 3.

The remainder of the thesis is split into three parts.

- Part I: Crawling presents the process of reaching as many states of a web application as possible. The architecture is presented in chapter 4. Chapter 5 presents the basic algorithm for the crawler. The detection of links, forms, and dynamic states are explained in chapters 6, 7, and 8 respectively.
- Part II: Cascading style sheets discusses the detection of CSS rules on the website. Chapter 9 covers the basics difficulties of CSS. Chapter 10 discusses the retrieval of the style sheets and their rules. The bulk of this part is about the actual detection of selectors and dealing with the difficulties (chapter 11).
- Part III: Evaluation contains an evaluation of the created prototype by running it against a set of websites. We calculate the crawler's effectiveness and verify the style detector in chapter 12.

[1]SASS - http://sass-lang.com/
[2]Less - http://lesscss.org/
[3]uncss - https://github.com/giakki/uncss
[4]css-usage - https://addons.mozilla.org/en-US/firefox/addon/css-usage/
[5]Dust-Me Selectors - https://addons.mozilla.org/nl/firefox/addon/dust-me-selectors/

Chapter 2

Research questions

2.1 Detection

RQ1 How can unused CSS selectors be detected?

Before attempting to detect unused selectors it must first be clear how this could and should be done. The first question deals with obtaining the CSS selectors of an application. The second one deals with cross-referencing these selectors to the DOM elements for detection.

RQ1.1 How to retrieve the CSS rules of a website?

There are multiple ways to retrieve the CSS rules. One could extract the locations of external style sheets from a web page and parse them. Another way is to let the browser parse the CSS rules and then extract the resulting tree.

Comparisons will be done with regards to the speed and difficulty of retrieval process. The quality of the resulting tree (e.g. missing rules or properties, structure, and possibly size) will also be compared.

Answered in chapter 10.

RQ1.2 How to cross-reference the CSS rules to the DOM elements?

We are aware of two methods to do this:

1. Take the CSS rules as the base. The crawler tries to detect each rule on each page and as soon as it has found the rule once, it will remove the rule from the list.

2. The reverse situation, with the DOM as the base. It crawls the website, but retrieves the used CSS rules for every single element and then removes the rules.

The methods will be explored in chapter 11.

2.2 Crawler

RQ2 How can websites be crawled?

A crucial part of detecting unused rules by crawling is the crawling process itself. Due to its importance part of this thesis is dedicated to crawling. Part I discusses and answers the questions below.

RQ2.1 What elements cause state transitions?

The most important question is how to actually crawl a website. Before this can be fully answered we must find out how state transitions happen and what causes them. chapter 5 presents the elements known to cause state transitions.

RQ3 How to improve the detection by taking dynamic features into account?

Static websites (in which JavaScript does not influence the DOM) are getting less common while time passes. In order to accommodate for this transition solutions must be found for dealing with websites whose state changes while the URL remains unchanged. This part will be mostly theoretical due to time constraints.

RQ3.1 How can JavaScript influence the Document Object Model?

Currently JavaScript is used on 89.9% (July 2015) of all websites and it offers several ways to influence the state of a website[1]. How exactly is this achieved and how can it be detected? This is one of the main questions when attempting to include states reachable by evaluation JavaScript.

RQ3.2 How to deal with content hidden behind forms?

Some areas are hidden behind forms, how to ensure these areas are visited by the crawler?

2.3 Evaluation

RQ4 How to use detected unused selectors for their safe removal?

The fourth question deals with the actual removal of the detected unused selectors. The first sub-question deals with the choice of software. The other deals the application of the prototype.

RQ4.1 What software combination is best suited?
This is a matter of choosing between PhantomJS and Selenium, due to allowing JavaScript at run-time. The advantage of Selenium is that connecting to a different browser is less work. PhantomJS on the other hand is faster. It is crucial to find out which is better suited in terms of performance, scalability, possibly accuracy, and ease of use.

RQ4.2 What percentage of CSS rules are unused?
The objective is to create a prototype that can detect unused rules. Knowing what the exact results are is the most crucial question regarding this research.

RQ5 What is the effectiveness of our crawler?
RQ6 How accurate is the style detector?

[1]http://w3techs.com/technologies/details/cp-javascript/all/all

Chapter 3

Overview

This chapter presents information that is not directly relevant to the research, but does answer certain research questions presented in chapter 2.

3.1 Software choice

Generally, crawlers come in two categories: static and dynamic. The former means that it downloads source code and extracts information using *Regular expressions* or libraries such as *BeautifulSoup4*. In these cases there is no Document Object Model, nor is any JavaScript executed. However, the dynamic version does have a DOM and does execute JavaScript due to partly running in a browser environment. The software interacts with the browser by giving tasks or requesting information.

For the detection of CSS we want to use the website just as actual users would. The reason being that simply parsing the source would not cover all the possible states a website can be in, due to JavaScript. So, to reach more states, and thus more possibilities to find selectors, the dynamic version is used.

Requiring access to a live browser limits the software choices. Due to our proficiency in JavaScript we decided to limit ourselves to libraries in JavaScript. The two main options are Selenium and PhantomJS with its many bridges.

One thing to note is that PhantomJS is both a browser usable through Selenium and a scripting environment. When referring to PhantomJS we mean the scripting environment, not the browser itself. The browser will be referred to as the PhantomJS browser.

3.1.1 Selenium

Selenium automates browsers. It is primarily used for (cross-browser) testing purposes. Its ability to interact with a browser, and a browser being the state of the art with regards to parsing JavaScript and CSS, makes it ideal. It is resource expensive in that it requires one to run a fully capable browser such as Chrome. Fortunately it is also capable of running the PhantomJS browser, which is headless and performant, and thus removes that disadvantage.

We decided not to go with Selenium because we were unsure whether it offered a simple way to detect the existence of selectors. In the final stage of this thesis we discovered Selenium does offer this functionality. At that point we figured it would be too late to investigate further and we had invested too much time in a PhantomJS prototype.

3.1.2 PhantomJS and CasperJS

CasperJS and PhantomJS were chosen due to our prior experience. PhantomJS is a headless webkit-based browser and scripting environment. It was chosen due to it being fully headless (it does not require a screen), which means that it outperforms other browsers and requires less resources. CasperJS is a library on top of PhantomJS that supplies much-used functionality, such as downloading and form filling.

One is able to drive PhantomJS by writing JavaScript code and running it with the `phantomjs` command in the terminal. The same goes for CasperJS scripts with `casperjs`.

Commands for CasperJS are written in JavaScript and it can be installed through NPM, Node's Package Manager. However, it is not a Node.js module in that it cannot be started with the `node` command and one cannot use the standard Node.js libraries. This limits the capabilities of the crawler and slows down development.

To circumvent the inability to use Node libraries certain bridges between Node and PhantomJS exist. These bridges are wrappers around the PhantomJS environment and can be ran from Node.js. Having such a bridge gives us access to a vast amount of libraries. Thus, a significant portion of time has been spent on creating different prototypes with different techniques and bridges. These will be discussed in short below.

3.1.3 Attempted prototypes

The initial crawler was written with plain CasperJS and PhantomJS. In all prototypes mentioned below an attempt was made to port the existing crawler. Due to differences in how the bridges function there is a major chance that the library did not succeed due to differences in architecture. A better approach would have been to truly start from scratch. Hence the reasons for discarding them mentioned below should not be taken as final.

Phantom[1] is the most used bridge and all communication between Node.js and PhantomJS is asynchronous through callbacks. Porting became difficult due to the original using many synchronous function for detecting the existence of rules. Similar to why Selenium was unsuitable.

Phantasma[2] Same as above but with promises instead of callbacks.

CasperJS socket communication An attempt to create a socket connecting between CasperJS (or PhantomJS' browser context) and a web server. Unfortunately this failed due to security restrictions.

PhantomJS web server PhantomJS has a simple built-in web server capable of receiving requests, gathering some data, returning the request. A promising solution, but ran into issues with concurrent crawling.

Phridge A very promising bridge as the evaluation function was synchronous, unlike in the bridges above. Unfortunately it did not run on Windows.

Additionally we tried different architectures based on PhantomJS as workers. In these attempts there was a Node.js process that spawned several workers which were in direct communication with PhantomJS processes. These workers received tasks from the main process, which then pass them onto the PhantomJS processes. There were some issues with parsing style sheets as each worker must be a standalone process, yet sheets must be exactly equal. In the end we decided against this option as there was too much unclarity regarding the synchronization of data. Though, if we were to start over this would probably be our method of choice.

In the end we chose a solution that somewhat resembles the worker one above. Instead of the crawler spawning workers, the crawler itself is a worker. A Node.js web server serves both the

[1] Phantom - https://github.com/sgentle/phantomjs-node
[2] Phantasma - https://github.com/petecoop/phantasma

Graphical User Interface and spawns the crawler process. The server also has end points that serve as database access points used by the crawler. A more thorough explanation follows in the dedicated chapters.

3.2 Browser context

In the following chapters the terms *browser context* and *browser environment* will be mentioned often. When developing with CasperJS and PhantomJS there are two contexts: the browser context and CasperJS/PhantomJS context. The former is within the browser, while the latter is in the scripting environment. Communication between both is only available through the `evaluate()` function. The function takes a function as an argument which is then executed within the browser context. Optionally values can be passed between both contexts by returning a value in the function (browser to scripting) and passing arguments (scripting to browser) to the evaluate function. Both contexts and their interaction are presented in Figure 3.1.

The evaluate function can only return data in native types, which are types that can be serialized to JSON.

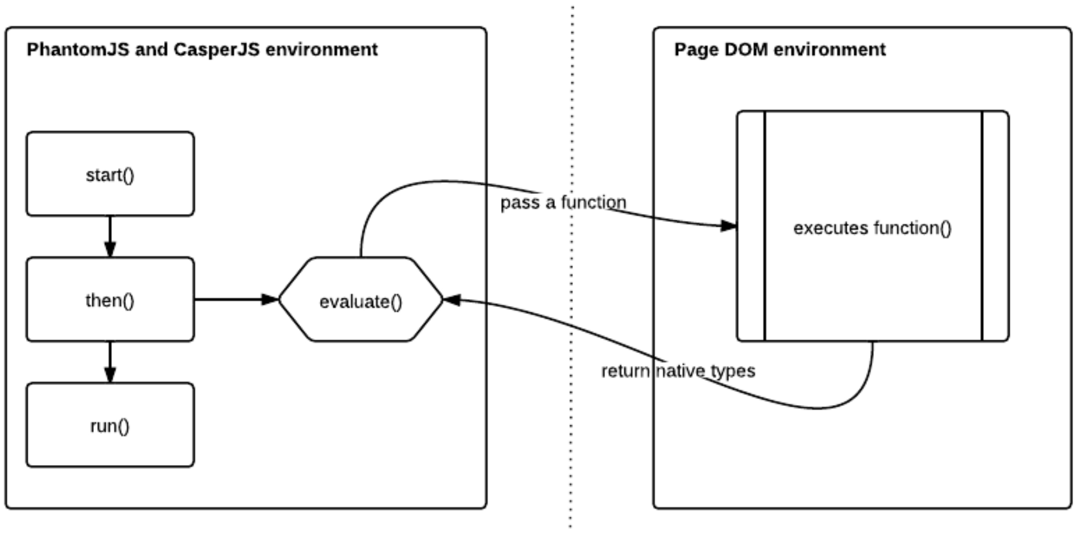

Figure 3.1: CasperJS and PhantomJS browser context and evaluate function

Part I
Crawling

Chapter 4

Architecture

4.1 Introduction

The prototype consists of two separate software packages: a web server and a crawler. The web server runs the crawler as a child process and communicates with the database. The crawler communicates with the web server and database through AJAX requests. The user communicates with the web server, which acts as the GUI as well. Both the web server and the crawler will be discussed in further details in their respective sections below. Figure 4.1 presents the basic architecture.

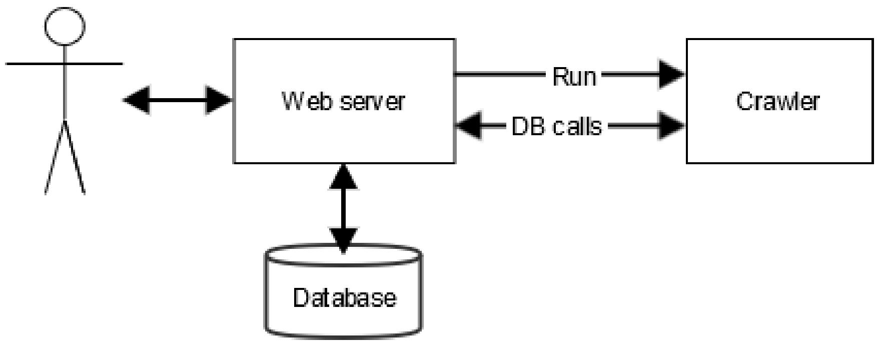

Figure 4.1: Basic representation of the architecture

4.2 Web server

The web server runs Node.js and uses Hapi[1] internally. It has two major interfaces:

Graphical User Interface A website serves as the GUI and is fully independent of the crawler. The data is retrieved from the database and visualizes the crawled data. It also serves as the control panel for running new crawl tasks through a web form, and lists all the crawled websites.

Storage interface It offers several endpoints (e.g. `/websites/{websiteId}/stylesheets`) for the crawler to request or store data. The crawler itself knows nothing about the internals of the web server or the database. Data is either stored in an sqlite3 database, due to it being lightweight and not requiring any configuration, or simply on the file system.

4.3 Crawler

The crawler itself is a CasperJS script. It sends data to the server by performing AJAX requests in the browser context. The web server receives the requests, stores or retrieves data from a database, and possibly sends data back to the crawler.

Internally the crawler consists of six major components:

1. **Crawler** is responsible for the processing and managing the queues of pending and visited URLs. It opens the pages and calls the different detectors.
2. **Link Detector** contains all code regarding the detection of URLs in pages. It scans the DOM for elements carrying URLs.
3. **Form Detector** detects forms on web pages, handles the storage of forms, and communicates with the GUI.
4. **State Machine** handles the exploration of dynamic states caused by JavaScript actions.
5. **CSS retriever** scans a web page for internal and external style sheets. It also downloads, parses, and stores the style sheets.
6. **Style detector** handles the detection of parsed CSS rules on web pages.

Figure 4.2 shows how the components relate to each other. The **main script** is the one that is started and defines the main flow of the program. It creates and configures the CasperJS object, loads the necessary modules, creates error callbacks, defines the global variables, and parses CLI options. The crawler is also started by this script, which then runs until all pages have been exhausted or the configured limit is reached.

The first four components will be discussed in the following chapters. The CSS retriever and style detector follow in Part II: Cascading style sheets.

[1]Hapi - http://hapijs.com/

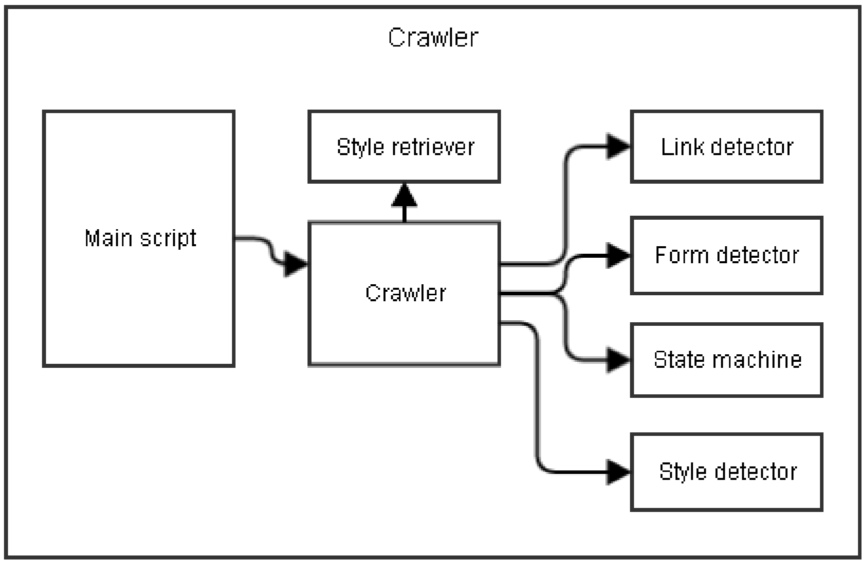

Figure 4.2: Components of the crawler

Chapter 5

Crawler

5.1 Algorithm

The crawling process itself is rather straightforward: dequeue a URL from the unvisited URLs, open the page, process the page, queue new URLs, and repeat.

Two queues are used: *pendingUrls* and *visitedUrls*, containing unvisited and visited URLs, respectively. The crawler compares all detected URLs to both queues to prevent duplicates. Algorithm 1: Basic crawling algorithm presents the basics of the crawler. The majority of the functionality is in the components that handle the detection of links, forms, and states.

Algorithm 1 Basic crawling algorithm

```
Require: baseUrl be set
Require: $pageLimit \geq 1$ OR $pageLimit = \infty$
 1: $visitedUrls \leftarrow \varnothing$
 2: $pendingUrls \leftarrow baseUrl$
 3: function CRAWL(page)
 4:     while $pendingUrls \neq \varnothing$ AND $visitedUrls < pageLimit$ do
 5:         $url \leftarrow pendingUrls.dequeue()$
 6:         $visitedUrls.enqueue(url)$
 7:         open the page
 8:         if Content-security-policy is active then
 9:             // xhr is forbidden when CSP is active
10:             // currently there is no way around this
11:             continue
12:         end if
13:         if isRedirection then
14:             follow, filter and store redirected URL
15:         end if
16:         DETECTLINKS()
17:         DETECTFORMS()
18:         EXPLORESTATES()
19:         DETECTSTYLES()
20:     end while
21: end function
```

5.2 Redirections

Occasionally a page is a redirection. The crawler follows those by default but branches off the default algorithm. Instead it marks the originating URL as visited and then runs the new URL through the filter. If the new URL is valid, and has not been visited before it simply processes it like any other page. If the URL is invalid or has been visited the page is skipped.

5.3 Content-security-policy

On some domains the *Content-security-policy* (CSP) is active. This is a new security measure that builds on the *Same-origin policy*, which was found to have been subverted. Initially all the code being served by the server was deemed safe by the browser. CSP prevents this by allowing the developers to create a whitelist of trusted sources. Only scripts (and other resources) that are explicitly whitelisted are allowed to be executed.

For each resource a special *directive* exists, such as **script-src** to white-list scripts. The directive that is of concern for the crawler is **connect-src**. This directive limits the origins to which you can connect through xhr and WebSockets[1]. The crawler uses xhr to send and retrieve data from the web server and thus the database. CSP blocks these requests, so communication is no longer possible. The crawler detects CSP and if one of the settings disables xhr, it skips the page.

This is far from desirable but we were unable to find a solution that did not require a complete rewrite of the architecture. This issue has been marked a limitation of the crawler.

[1]CSP - http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Chapter 6

Link detector

6.1 Algorithm

The link detection process is the most important process for the crawler. It is this function that makes sure the crawler can continue and reach as many pages as possible. If links are missed the crawler stops prematurely and the CSS detection will be less effective. The detection is not as straightforward as it may seem. It is not simply finding links and adding them to the queue. They must first be filtered. Links exist in many different attributes and HTML elements. Detection of these elements is crucial. Algorithm 2 presents the the algorithm for link detection. The two methods `FindLinkElements()` and `FilterLinks()` detect and filter the links, respectively. These are discussed next.

Algorithm 2 Link detection

```
 1: function DETECTLINKS()
 2:     $links \leftarrow$ FINDLINKELEMENTS()
 3:     remove duplicate links
 4:     $links \leftarrow$ FILTERLINKS($links$)
 5:     for all $link \leftarrow links$ do
 6:         if $link \notin pendingUrls$ AND $link \notin visitedUrls$ then
 7:             $pendingUrls.enqueue(link)$
 8:         end if
 9:     end for
10: end function
```

6.2 Finding link elements

Finding elements with URLs can again be done in multiple ways: using Regular Expressions (Regex) on the page's source, or by querying the DOM. Regexes are not used due to the relatively low accuracy and increased complexity. In some of our test cases the Regexes included part of the HTML element in the URL. Instead we query the DOM.

`FindLinkElements()` evaluates a function in the browser context (`document.querySelectorAll (selector)`) that selects all elements matching the selector. Of those elements the URL is returned. Table 6.1 on page 18 lists the elements and the attributes that possibly contain a URL.

Element	Attributes
a	href
applet	codebase
area	href
base	href
blockquote	cite
del	cite
frame	longdesc, src
head	profile
iframe	longdesc, src
img	longdesc, src, usemap
input	src, usemap, formaction
ins	cite
link	href
object	classid, codebase, usemap, data
q	cite
script	src
audio	src
button	formaction
command	icon
embed	src
html	manifest
source	src
video	poster, src

Sources: http://www.w3.org/TR/REC-html40/index/attributes.html,
http://www.w3.org/html/wg/drafts/html/master/index.html

Table 6.1: All elements with the attributes that could contain a URL

The element that is queried for is in the first column. The attributes we retrieve for each element is in the second column.

6.2.1 XML

Some web sites contain XML pages, like RSS feeds. These feeds could contain many links. Once an XML page is detected, it is parsed with an XML parser. It then searches for `link`, `comments`, and `loc` elements. The detector then proceeds as normal.

6.3 filtering URLs

An important step in the process of gathering links is filtering them. URLs locate resources and can be seen as an address for a resource. They consist of many parts, e.g. domains, hosts, hostnames, subdomains, queries, and fragments. The crawler must be able to determine what part is useful, and whether or not a certain URL matches one already in its queue. The better these duplicates can be detected, the less duplicate pages will be crawled, and thus the better the crawler's performance. The rules used to filter URLs are listed below:

1. **Valid URL**. Verifying the validity of a URL (i.e. valid URL, and not an email address) is the first step. If it fails the test it is dropped.
2. **Excluded extensions**. If the resource in the URL contains one of the excluded extensions it is dropped. Invalid extensions are: png, jpg, jpeg, gif, ico, css, js, csv, doc, docx, pdf. Files with these extensions are not web pages and thus not of any importance to the crawler.
3. **www. stripping**. Some web sites use URLs with or, without `www.`, whereas others use them interchangeably. Some web sites do not respond to a version with(out) the `www.`, or it points a completely different resource. Currently we have no way to detect to which case a web site belongs, so it has to be manually configured on a per-domain basis.
4. **domain checking**. The simplest check is ensuring that the URL is still within the domain that is being crawled. If the domain *example.com* is being crawled, then *donut.com* will be excluded. Sub domains are included by default right now, there is the option to exclude those though.
5. **query checking**. A possible cause for endless crawling is the query part of a URL, e.g. *?page=3*, *?page=3&ref=google*. Both examples may point to the same resource, or there could simply be an infinite amount of queries. We have therefore decided to disable queries, but once again these can be enabled.
6. **Fragment identifier/hash tags** are used as anchors to specific content on a web page. They can be used to set state (e.g. `:target` selector), but the page itself does not differ from the version without the hash tag. However, there is one major exception. With the rising popularity of AJAX some web sites emulate navigation without requiring reloads by changing the fragment. In these cases the fragment is used to link to certain portions of the web site [1]. For normal exploration hash tags are ignored, they might be used for dynamic state exploration though.

For completeness the algorithm is presented as algorithm 3.

Algorithm 3 Filtering links

```
Require: baseUrl be set
1: function FILTERLINK(link)
2:     if link is not a valid URL OR link is an email OR link has excluded extension then
3:         skip link
4:     end if
5:     strip www. and #
6:     if link has same host as baseUrl then return link
7:     end if
8: end function
```

[1] Fragment Identifier - https://en.wikipedia.org/wiki/Fragment_identifier

Chapter 7

Form detector

A large portion of the web is hidden behind web forms [12]. In order to visit those pages the crawler detects forms. Upon detection these forms are copied, including the URL of the page and the form's unique identifier, and sent to the web server. The web server then presents the form to the user where the form can be filled in. Once the crawler has exhausted all pages it will repeatedly requests newly filled in forms from the web server before browsing to the page to fill in and submit the forms.

7.1 Identifying forms

The initial approach was to identify forms by their ID. However, not all forms have an ID attribute. An alternative was to create a unique CSS selector, like Firefox can do in the development tools. It starts with the target element, then travels up the tree to the root, creating a unique selector along the way. Unfortunately this method results in rediscovering the same form numerous times due to small DOM changes.

So, using no identifiers, using IDs, and creating unique selectors all resulted in the crawler spamming the GUI with the same form. Instead we looked at what makes a form unique.

Firstly, two important attributes that contribute to a form's uniqueness are *action* and *method* of the form element.

1. **Action** defines the URL the form data is sent to. Up until HTML5 this attribute was mandatory and takes either an absolute or relative URL. If left out the form data is sent to the current page. Pre-HTML5 forms set action to # to indicate that data should be sent to the current page.

2. **Method** defines how data is sent. The value should be supported by the HTTP protocol, such as POST and GET. POST sends data, GET retrieves data. The default value is GET.

Forms that are capable of sending binary data (e.g. `<input type="file">`) are ignored due to difficulties with the two step process. In order to get this working the web server should accept the file, store it, then let the crawler pick that file. Additionally many forms expect the file to be of a specific type or have specific properties, which are nearly impossible to determine automatically. Due to the aforementioned reason and time and security constraints this is currently excluded from the prototype.

Secondly, a form's child elements. Two forms can be considered equal if the input elements are exactly equal (e.g. tags, IDs and attributes). Note that there is still the issue of two basically equal forms being different due to extra non-input HTML elements, attributes, or perhaps a different ordering of children.

For this prototype a form will be uniquely defined by its HTML. That method covers both the attributes and the form's children, while still being simple. Of course it will still fail on edge cases, mentioned one paragraph back, but it is sufficient for this thesis.

7.2 Form validation

Many sites contain some type of form validation; native HTML5, JavaScript, or server-side. The current prototype does not support checking the success of a form submission. It is also not feasible to copy this validation to the GUI, unless it is native HTML (e.g. `<input type="url">` only accepts valid URLs). Further iterations could verify submission of a form and perhaps detect changes in the form's DOM element.

Chapter 8

State machine

An increasingly common approach to create rich interactivity and responsiveness in web applications is through the use of JavaScript and dynamic DOM manipulation. As a result a single URL does no longer always point to a single document, or a single representation of that document. Static web crawlers depend on these URLs to crawl a web application/site. These crawlers are incapable of detecting dynamic states and are thus unable to detect portions of CSS rules.

Ali Mesbah et al. published a novel technique to crawl these dynamic web applications [12][2]. They have implemented their ideas in a tool called Crawljax[1]. The following content is heavily based on the ideas presented in their paper.

The initial idea was to re-implement how Crawljax detects states, which is by detecting clickable elements that change the state within the browser's dynamically built DOM. Crawljax opens a page in a browser, scans the DOM tree for candidate elements that are capable of changing the state, and finally fires events at those elements, while analyzing the DOM tree.

We add a slightly different method to detect additional states. PhantomJS offers the `page.onInitialized` callback, which is called as soon as the Page object is created, but before a URL is entered. By injecting a script that intercepts the creation and removal of events we get a list of events on a page at any given time.

Originally forms are also considered part of this section, but because they are detected and handled in a different manner they will not be discussed here. Instead, they were discussed in chapter 7.

Although we are capable of detecting some state transitions we must note that the software is not ready to crawl public web sites.

This chapter begins with a section on terms used throughout this chapter (section 8.1). The flow of the state machine is presented in section 8.2. section 8.3 Dynamic state transitions are explained in section 8.4. subsection 8.4.1 discusses the interception of events. The second group of event handlers is discussed in subsection 8.4.2. section 8.5 presents a data structure for storing the detected events. How to fire custom events to cause state transitions is discussed in section 8.6.

[1]http://crawljax.com/

8.1 Terminology

In the following sections certain terms will be used often. This section explains these terms and their relation to the whole.

Page object A page object refers not to a web page but to a PhantomJS page instance. A page can be seen as a tab in modern browsers. With PhantomJS you create pages explicitly, whereas you simply open a new tab in browsers.

State A state is the situation of (the DOM tree) of a web site at a point in time, possible after user interactions. It is represented by the internal structure of DOM tree. In this case it is defined by the elements, and not the textual content of the elements. See subsection 8.1.1 for the identification of a state. A new state is detected if it is not equal to a state in the state-flow graph (section 8.3).

A **state transition** happens when after interaction with the page the state differs from the previous state. It is the triggered once an event is fired at (or a user interacts with) an element on a web page. One can think of clicking a button, which then changes some other element.

8.1.1 State identification

To distinguish between different states and to compare states the DOM representation is converted to a simpler form. We ignore text in DOM states and instead only look at the DOM structure. Text is ignored due to not being crucial to distinguish different states and due to not having any importance with regards to CSS selector detection. The simpler representation consists of only the elements and their attributes. An example DOM is presented in Listing 8.1, its simplified version is presented in Listing 8.2.

```
<div id="header">
  <div>
      This is some text in the header, situated above the nav.
    </div>
  <nav id="nav">
      <ul>
          <li> button </li>
        </ul>
    </nav>
</div>
```

Listing 8.1: Example DOM tree

```
<div id="header"><div></div><nav id="nav"><ul><li></li></ul></nav>
   </div>
```

Listing 8.2: State representation of the DOM tree presented in Listing 8.1

8.2 Application flow

To get a general idea of how the state machine works this section explains its flow. The sections hereafter explain several of its phases in more details.

1. Inject event interceptor.
2. Open web page and create state machine.
3. Catch events.
4. Create state-flow graph.
5. Create and add initial state.
6. Process state:
7. Find DOM handlers.
8. For each event-element combination:
    (a) Fire event at element.
    (b) If state change: add state and transition to graph, reset the state.
    (c) If no state change: reset state.
9. Repeat 6-8 until all events have been exhausted

The first step is injecting the code that intercepts the creation of events. This step happens at prototype's booting phase and step 2 is part of the crawler. These do not happen during the state exploration phase. They are listed here to get a full picture of how it functions. The other phases all with the opening of a web page (step 2).

Once a web page is opened a new state machine is created. Immediately upon loading the event interceptor starts intercepting events. This happens automatically while the state machine is still idling.

When the web page is done loading the state machine is started. It creates a state-flow graph (thoroughly explained in section 8.3). The initial state is added to this graph and selected for processing. The state is loaded and starts gathering additional event handlers that could not be captured by the event interceptor. section 8.4 discusses all the event types and subsection 8.4.2 discusses the events being gathered here.

So, when a state is being processed it stores all the events from the event interceptor and the ones it gathered. How they are stored is discussed in section 8.5.

For each event-element combination in the event storage it fires the event at the element. After each event it checks whether this state it new or not. If it is new the state and the transition to that state are added to the state-flow graph. If it has been previously detected a state transition is created. The state is reset (back to the state currently being processe) and the process is repeated. Afterwards a new state is selected and the processed repeats itself until all states have been detected.

Finally, when the state machine is done it contains a graph with all possible states of a web page. These states can be revisited for other calculations, such as style detection.

8.3 State-flow graph

The state machine uses a state-flow graph internally to keep track of all visited states. The graph is defined as follows:

*DEFINITION 1: A **state-flow graph** for a page P website A is a three-tuple $< r, S, T >$ where:*

1. ***r** is the root node, which represents the initial state directly after **P** has fully loaded.*
2. ***S** is the set of states on **P**. Each state $s \in$ **S** represents a run-time state on **P**.*
3. ***T** is the set of transitions between the states. Each $(s_1, s_2, evt, sel) \in E$ represents a specific transition causing a state transition from s_1 to s_2 when firing event evt at selector sel.*

Initially the only state is the root node, other states and transitions are added incrementally as events are fired and new states are detected.

DEFINITION 1 lists several components. **Transitions** are explained in section 8.4. All detected events are then stored in a specific data structure (explained in section 8.5) which is attached to a **state**. Those events are then used to create new **transitions** between **states**.

Figure 8.1 presents an example of a state-flow graph, including the event data structure for two states. The example application contains four states (the initial root and three other states). The root has two state transitions ending in state 1 and 2. To reach state 1 from the root a custom event called **MyEvent** must be fired at the element selectable by *#call-to-action*. State 2 is reached by firing **click** at *#nav > div:nth-child(2)*, i.e. the second div element that is a direct child of the element with an ID of *nav*. State 2 has two transitions working in similar fashion.

In the example $r = root$, $S = \{$*root, state 1, state 2, state 3*$\}$, $E = \{$*(root, state 1, **MyEvent**, #call-to-action),(root, state 2, **click**, #nav >div:nth-child(2)),(state 2, state1, **MyEvent**, #call-to-action),(state 2, state 3, **keypress**, #lightbox)*$\}$.

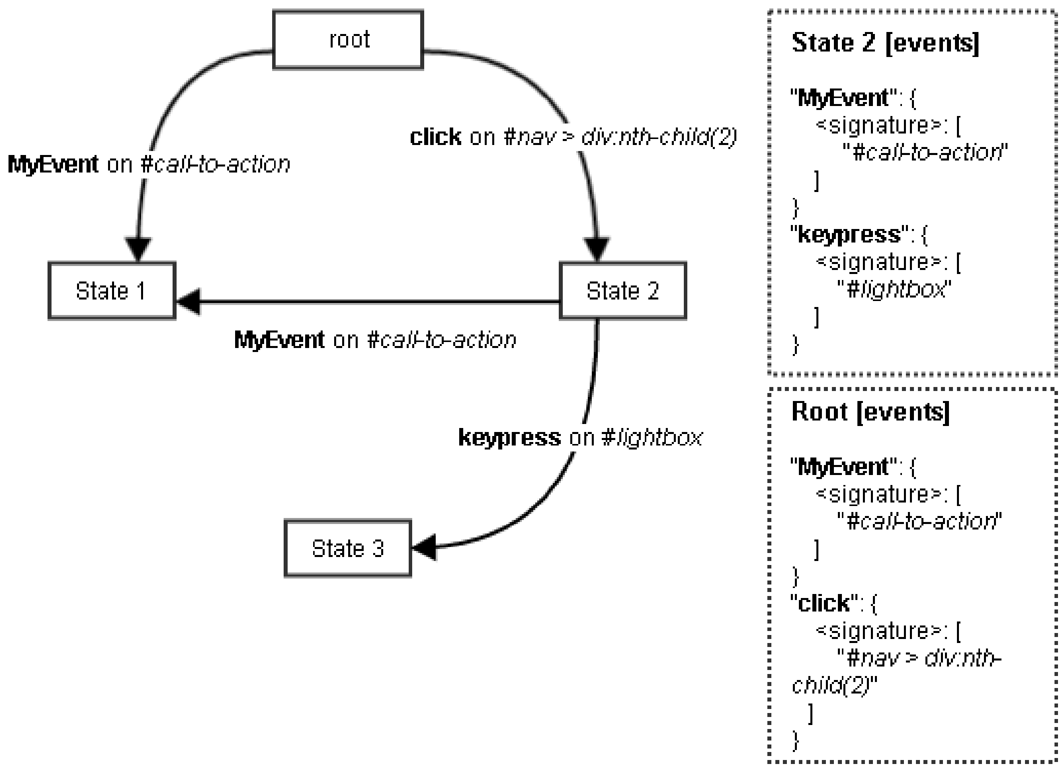

Figure 8.1: An example flow graph. The boxes represent states, the lines transitions. The dotted boxes represent the event data structure for two states.

8.4 Dynamic state transitions

A dynamic state is the state of an a web site caused by JavaScript, whereas a regular state is one caused by static crawling (following hyperlinks). We were able to identify five different ways in which JavaScript can cause dynamic state transitions.

1. `element.addEventListener(event, function)`. This is the current standard way to add event listeners.

2. `element.setAttribute('onclick', function)`, which can be used to set a HTML event attribute (e.g. `onclick`) of an element dynamically.

3. **HTML attributes**. This covers the cases in which events are linked to HTML elements in the source code, e.g. `<button onclick="somefunction()"/>`. This method was specified in the DOM 2 Events specification[2].

4. **Event properties**. These are quite similar to number 3, but here the event attributes are set dynamically with JavaScript. The code `object.onclick = function (){};` adds a function as `object`'s `onclick` handler.

[2]DOM 2 Events - http://www.w3.org/TR/DOM-Level-2-Events

5. **JavaScript links**. Adding JavaScript to elements through `<a href="javascript:this.textContent = 'new Link'; return false;">link</a>`. The `javascript:` tells the browser that upon clicking this link a piece of JavaScript is to be executed. `return false;` signals the browser to not refresh the page. Even though this method is a bad practice it is still used on web sites today.

Number 1 and 2 can both be caught by intercepting the event creation. More on this in subsection 8.4.1. Number 3, 4, and 5 can be retrieved from the DOM. These three we call *DOM handlers*, due to being attached to the DOM (elements). We discuss these in depth in subsection 8.4.2.

8.4.1 Event interception

The first step in detecting which elements have event handlers is by intercepting the creation of these handlers. The interception is possible by overwriting four functions: `setAttribute()`, `removeAttribute()`, `addEventlistener()`, and `removeEventListener()` of the `window document`, and `Element.prototype` objects. What happens is that once events are about to be created they call one of the functions mentioned above, which have been overridden with our own. Our function grab the information from the events and then passes it onto the original function.

The script itself is injected into the page object in PhantomJS' `onInitialized` callback, which is called directly after the page object is created, but before a URL has been entered.

We came across an article[9] and a GitHub repository[3] explaining and using this method. We have not discovered this ourselves.

section 8.5 explains how these events are stored.

8.4.2 DOM handlers

We call number three to five of the list of possible transitions *DOM handlers*. We gave this name due to them being visibly and directly attached to DOM elements.

JavaScript links can be queried directly by supplying the `[href^='javascript']` selector to query function. This selector finds all elements whose href property starts with `javascript:`. The handler is simply the function after the colon.

HTML attributes can be found by querying the DOM for all elements and finally filtering those by the existence of attributes starting with **on**.

Event Properties Same as **HTML attributes** but instead of attributes we look for a non-null properties starting with **on**.

Appendix B presents the full list of event on-handlers we were able to find and which the crawler should ideally catch.

[3]https://github.com/fabiocicerchia/salmonjs

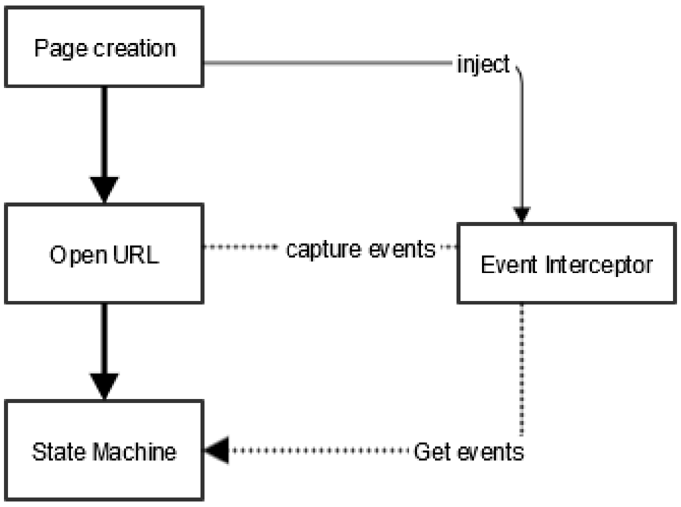

Figure 8.2: Workings of the event interceptor

8.5 Storing handlers

The next step in the process is storing the events and handlers to later simulate them in the exploration phase. So far we have the type of event, which is either a custom one or starts with **on**. In the case it starts with **on** we strip off that part (e.g. onclick becomes click). The reason for this is that the DOM (element) actually listens to the part without **on**. We also have the handler itself. When an event creation is intercepted the element to event is destined for is always supplied as an argument. For DOM handlers it is also simple, since we already access the element to see whether it contains certain properties and attributes.

However, the element and event type together are unsuitable to fully distinguish between different state transitions. One event handler may change the event handler of another DOM element, thus changing one of the transitions. In order to deal with these situations we create a signature based on the event handler. The signature is simply a SHA1 hash of the event handler function.

For the intercepted events the handler function is also given as an argument. For the DOM handlers it is directly accessible through the properties and attributes. The JavaScript links have the function after the colon. The `javascript:` part is stripped off, the rest is used to create the signature.

So there are three parts: event type, element, and the signature of the event handler. The final piece is the identification of the element so that we can find it again later to fire the event. subsection 8.5.1 explains the process.

With those three in place the data structure looks like:

```
{
  <event type>: {
    <signature>: [<element identifications>]
  },
  ...
}
```

Listing 8.3: Data structure for the events

8.5.1 Element identification

The state machine must be able to identify elements to fire events at, and the identification must always be the same for the same DOM. Two major options exist: XPath and CSS selectors. XPath is getting used less, in favor of CSS selectors, due to having less features. Though, both are good solutions. We chose for CSS selectors to not mix different technologies.

Firefox has a context menu button in its developer's tool screen where you can select an element and create a unique CSS selector path for that element. The paths resemble something like `div.col-sm-2:nth-child(1)> strong:nth-child(5)`. This means that it starts with the `div` element with the class "col-sm-2" that is the first child of its parent. Of that `div` it selects the `strong` element that is the fifth direct child of the `div`.

8.6 Firing events

Events must be fired to force state transitions. We can either inject a script that launches events directly at elements, or we use CasperJS to simulate actual user behavior. We have tested only the former one and found out that, due to changing specifications, some events work while others do not. The current webkit version used by PhantomJS is not fully up to date with the new event constructors. For each event we must find out which constructor to use. This has been done insufficiently and is currently a reason why the state machine does not yet fully function.

CasperJS offers a large set of user interactions one can simulate. A Mouse object is offered which can simulate six events; mouseup, mousedown, click, mousemove, mouseover, and mouseout. It simply takes the event type and selector as parameters. Sending keys is also supported, including modifiers (ctrl, alt, etc.).

A hybrid solution would be preferable: CasperJS for simulating mouseclicks and filling forms, and the browser context for custom and other type of events.

Part II

Cascading style sheets

Chapter 9

Cascading style sheets

9.1 Introduction

This chapter covers the basics of Cascading Style Sheets relevant to the rest of the detection process. Though CSS can be defined as a simple to use language, its ever-growing specification, its cascading nature, and infinitely many ways to combine selectors make it difficult to fully understand.

Section 9.2 presents the basics of CSS. The inclusion of CSS into web pages is described in section 9.3. In the second section the difficulties of CSS are described. The difficulties of CSS are discussed in section 9.4. Finally, in section 9.5 the selectors are categorized according to their properties.

9.2 CSS Basics

Cascading Style Sheets (CSS) is a style sheet language used for describing the look and formatting of a document written in a markup language. The most common application is the styling of websites, but CSS can be applied to any XML-like document. CSS is one of the three cornerstone technologies used to build web pages, along with HTML and JavaScript.

Up until version 2.1 the language was upgraded as a whole. However, since version 3 (CSS3) the specification is split into separate modules. Modules can now level independently, which is why some modules are already at level 4. Each level is backwards compatible and only extends the module with extra features [3].

A style sheet is a file containing multiple style rules. CSS Rules currently come in two forms: *qualified rules* and *atrules.*

Qualified rules add properties to elements and consist of two parts: a selector and a body of declarations. A declaration consists of a property and a value. In the example presented as Listing 9.2 *color* is the property and *orange* is the value. The general form is given in Listing 9.1.

```
<selector>[, <selector>] {
    [<property>: <value>;]
}
```

Listing 9.1: General form of a CSS rule

```
body, p.orange {
    color: orange;
}
```

Listing 9.2: example of a CSS rule

Atrules do not directly apply styles to elements, but instead dictate certain processing rules for the CSS document. These rules always start with a @ and have no standardized structure.

For the detection of rules we only require the selector. Mesbah and Mirshokraie listed three types of selectors to detect elements: **element selectors**, **ID selectors**, **Class selectors** [11]. Additionally there are **universal selectors**, **attribute selectors**, **pseudo-classes**, and **pseudo-elements**. These can be glued together with **combinators**[5]. The full list of selectors is presented as Appendix A on page 63.

9.3 Including CSS into web pages

Three methods to include CSS into a website exist. They as listed below and are of further significance in later sections.

Inline Adds styling to one specific element. `<div style="width: 100px;"></div>` sets the width of the `div` element to 100 pixels.

Internal/embedded The style sheet is included in the `<head>` of the document between `<style></style>` tags. Putting `div { width: 100px; }` between the tags sets the width of all `div` elements to 100 pixels.

External The style sheet is included through a link: `<link rel="stylesheet"href="stylesheet.css"/>`. No further difference with 2.

9.4 Difficulties

Due to the size and properties of CSS we had to deal with several difficulties, listed below.

9.4.1 Vast variety of selectors

Having access to a large amount of different selectors, and being able to combine them in many ways means the amount of selectors is basically unlimited. This significantly increases the difficulty of both the development and testing of the detector.

9.4.2 Pseudo classes/elements

Pseudo-elements are added to selectors and allow you to style certain parts of a document. The `::first-line` pseudo-element targets only the first line of an element specified by the selector. Pseudo-classes on the other hand refer to a state the selected element can be in: `:hover` styles the element when the mouse moves over said element.

Mesbah and Mirshokraie mentioned the exclusion of pseudo elements/classes as a limitation of their tool CILLA. They stated that the relation between DOM and pseudo elements/classes cannot be deduced from the DOM tree. This was before the widespread acceptance of HTML5 and CSS3 by browser vendors and is only partially true today.

It is still true that both pseudo classes and elements lie outside the DOM and therefore cannot be found by simply parsing the DOM. Some of the pseudos however can be selected by querying the DOM, such as `:matches()` and its vendor prefixed version, `:disabled`, and `:lang()`. Which can and cannot be selected by querying, and how we deal with those that cannot, is discussed in chapter 11.

Backwards compatibility

Prior to CSS3 pseudo-elements were made up of a single colon (:) followed by the name of the pseudo-element. In order to discriminate between pseudo-elements and pseudo-classes they are now made up of two colons (::). To ensure backwards compatibility user agents have to support the single colon notation as well. Additionally pseudos defined in Level 3 and upwards require double colons. This results in a mixture of notations for the coming years.

Multiple pseudos

According to the Selectors Level 3 specification only one pseudo-element may appear per selector. However, in future versions this may not be the case. As of Level 4's working draft on May 2013 this is still not the case anytime soon.

9.5 Categorizing CSS selectors

CSS selectors can be split into several categories based on how they can be detected by the crawler.

Simple Selector The most basic version is the **simple selector**, which simply refers to a element, class or id. These were all described by Ali Mesbah, as mentioned in section 9.2.

Always-present selector These are present on every single document, such as `:root` which points to root of the document. Or `:scope` which always falls back to referring to `:root` as a least resort.

Static-pseudos These exist simply when the element they are attached to exists. Well-known examples are `a:hover, a:visited, and a:active`, but also `p::before` `p::first-line`.

Combinations Combinations can be made by combining combinators and simple selectors. In some cases (e.g. `button:hover > span`) a combinator is combined with a pseudo. Simply trying to detect these directly does not work, as pseudos cannot always be detected directly. These selectors must first be cleaned and detected in parts. In some cases additional detection rules are applied, such as `:checked`, which can only be applied to a subset of elements. The cleaning process is discussd in chapter 11.

Chapter 10

Rule retrieval

As mentioned in section 9.3: Including CSS into web pages three methods exist: inline, internal, and external. Inline inclusion will be fully ignored due to (a) it being bad practice as it does not separate presentation and structure, and (b) the code is directly linked to the element and thus used.

Two methods to retrieve the rules have been investigated. The first utilizes the Document Object Model, which holds references to the included style sheets. The second method uses the source code to detect the ‘style‘ and ‘link‘ elements before downloading and parsing the content.

10.1 Extracting from the DOM

The DOM defines a parsed and rendered web page. It contains information like the DOM nodes, inheritance, and style sheets. The biggest advantage is that there’s no need to parse and locate CSS manually. The browser has already done this.

However, this method is not viable due to multiple reasons:

PhantomJS is not able to export information from the headless browser to the PhantomJS script if it is not serializable to JSON. The DOM contains cyclic references (parent to child and vice versa) which causes the serialization process to fail. Simply removing the cycli was unfeasible due to its large size (60K LOC for a simple website) and edge cases. An attempt has been made to implement a custom CSS Object Model by extracting the information. During this process it became apparent that the properties in that tree do not correspond to the actual CSS file. All the shorthand notations (border, background, etc.) are converted to their individual properties (background-color, background-image, etc.).

The rules’ selector text remained unchanged, which is basically all that was required. For the detection part this would be enough, with the added bonus of the rules being in the correct order. However, if the program were to provide feedback or find overridden properties it would be difficult to map the properties to the source file correctly.

Due to the altering of data and dependability on the browser’s DOM this method has been discarded.

10.2 Manual downloading and parsing

The chosen alternative is to acquire CSS resources and parse those to a CSS Object Model (CSSOM). This method depends on the quality of the CSS parser. Due to CSS's rapid evolvement and inclusion of rules such as `@media` and `@font-face` many parsers are now outdated. In case the parsing process fails the entire style sheet is dropped.

10.3 Retrieving style sheets

The retrieval process can be thought of as having two phases. The first phase contains the stylesheets referenced directly from the website. The second phase contains the stylesheets referenced by the style sheets from the first layer through the `@import` rule. The `@import` rule takes either a relative or absolute path to a different style sheet.

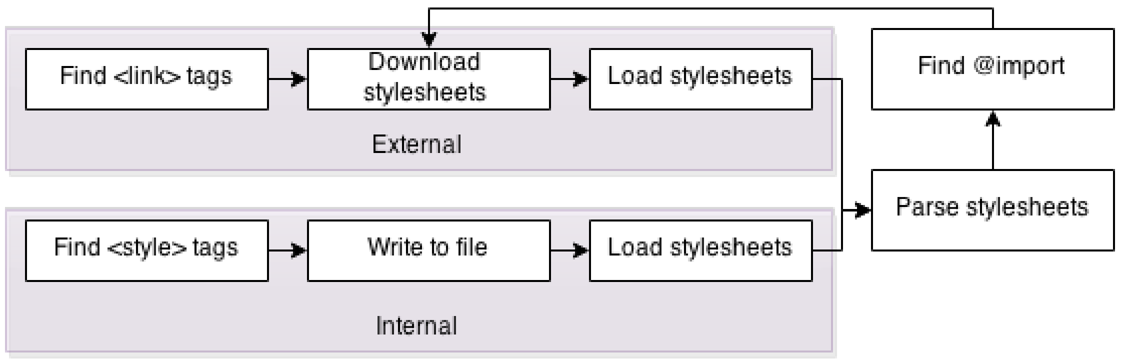

Figure 10.1: The retrieval flow for style sheets

10.3.1 Phase one: downloading initial sheets

Style sheets can be external (referenced by `<link>` elements) or internal (within `<style>` elements). Listing 10.1 presents examples for both methods.

```
External:
<link rel="stylesheet" media="all" href="layout.css"></link>
<link rel="stylesheet" href="http://example.com/css/layout.css"></
    link>

Internal:
<style> p { font-size: 20px; } </style>
```

Listing 10.1: Two ways to include style sheets in a web page

External style sheets are always referenced by `<link>` elements, but not all `<link>` elements refer to style sheets. The `<link>` element is used to refer to any kind of external resource, e.g. (fav)icons, license, author, alternative version[1]. Each `<link>` element has a *rel* attribute,

[1]Link types - https://developer.mozilla.org/en-US/docs/Web/HTML/Link_types

describing the resource it links to. This must be ”stylesheet”, so the program filters by this attribute’s value.

Internal style sheets are always enclosed in `<style>` tags. Before HTML5 these were only contained in the `<head>` of a document. However, since HTML5 these elements can also be part of the `<body>` as long as they contain the ”scoped” attribute. Scoped styles only apply to the sub tree of the element they are defined in. [2][3]. Due to the increased difficulty these are ignored.

10.3.2 Phase two: additional sheets

In order to reach the `@import` urls the stylesheets from the first phase must be parsed, hence this being the second phase. `@imports` can reference to an absolute or relative URL.

The imported style sheets can themselves also import other style sheets, thus this phase is done recursively until there are no more `@import`s to be found.

The resulting CSSOMs are stored and used to detect rules during crawling.

[2]Scoped attribute - https://html.spec.whatwg.org/multipage/semantics.html#attr-style-scoped
[3]Scoped attribute - http://html5doctor.com/the-scoped-attribute

Chapter 11

Style detector

11.1 Introduction

Each time the crawler detects a new state it calls the style detector. The style detector is the component that actually detects whether the selectors on this page are used. It only checks the style sheets that are active on the current page, i.e. the ones that are included.

This chapter is split into different parts. Each section discusses different types of selectors, as discussed in section 9.5. We start with the simple ones in section 11.2 before discussing pseudo-classes and -elements in section 11.3.

11.2 Simple selectors

The simple selectors are listed below and are simple due to being easily detectable. Detecting these requires no additional actions or modifications.

Universal i.e. `*`
Type such as `h1, p, div, nav`
Class such as `.danger, .orange`, but also `div.large, button.green`
ID such as `div#nav, div#footer`
Attribute such as `h[title], h[class="example"], a[rel~="copyright"]`

An extensive list of selectors and examples can be found in Appendix A on page 63.

11.3 Pseudo-classes and -elements

As mentioned in subsection 9.4.2 certain elements can be detected with `document.querySelector()` while others cannot. We also mentioned the difficulty of backwards compatibility, always-present selectors, and static-pseudos.

11.3.1 Backwards compatibility

The first step is dealing with backwards compatibility by replacing the double colon notation with the single colon notation to normalize their notation. This simplifies further detection.

11.3.2 Always-present selectors

Always-present selectors exist, on matter the situation. Current these are limited to `:root` and `:scope`.

11.3.3 Static-pseudos

The biggest category is the static-pseudos. These exist if the selector they are attached to exists. It is safe to assume `a:active` exists if `a` exists. The classes and elements that possess this property are: `:target`, `:valid`, `:invalid`, `:indeterminate`, `:default`, `:hover`, `:active`, `:visited` , `:link`, `:any-link`, `:focus`, `:after`, `:before`, `:first-letter`, `:first-line`, `:selection`, `:backdrop`, and `:marker`, `in-range`, and `:out-of-range`.

A few special cases exist, these are either not truly static or require some additional detection rules.

One such pseudo is `:checked`. It is limited to checkboxes, radio buttons, and input lists. Thus, one cannot simply assume that `:checked` exists without first looking for said three elements. Once a selector containing `:checked` is being processed it will either search for the element it is attached to or, in case it is unattached, search for all three. `:in-range` and `:out-of-range` are added due to the difficulty one can actually detect them. Detecting their validity with sliders can be done by querying the validity property of the attached element. The rangeUnderflow and rangeOverflow must be false for in-range, and either of them true for out-of-range. When patterns are used, like a Regex pattern for text fields, complexity increases drastically.

`:valid` and `:invalid` are detected by default due to them being validation. If the attached element is present we assume it can be valid and invalid depending on its value. It is an assumption, but a safe one and it prevents removing too many selectors.

11.3.4 Remaining pseudos

Not all pseudos can be fully included in the lists above. `:dir(ltr)` for example, must first be translate to an attibute-selector (e.g. `[dir="ltr"]`) before it can be detected. The same goes for `:placeholder-shown`, which becomes `[placeholder]`. `:matches()` and `:-moz-any()` are converted to the webkit version (`:-webkit-any()`), as PhantomJS cannot detect the other two.

11.3.5 Excluded pseudos

We have decided to use the Selectors Level 4 specification due to some browsers already implementing some of its selectors, but with prefixes. This led to a few unimplemented selectors, which we thus cannot yet detect. Neither of the following is supported: `:focus-within`, drop, time-dimensional, `:user-error`, grid-structural, `::content` and `::shadow`.

11.4 Combinators

The detection process usually works for combinators, except when parts of the combinators contain pseudos.

We will explain this with an example. When you have a checkbox and next to that checkbox is text wrapped in span tags (like `<input type="checkbox"><span>I have a bike</span>`. If you want to change the text within the span once the checkbox is checked you could use a selector like: `input[type="checkbox"]:checked + span`. This selects the span element that is a direct sibling of a checkbox in its checked state. Directly selecting is not possible, due to the pseudo-class. Instead, we first have to split the selector and treat the parts with a pseudo just like we have done in section 11.3. Once that is done we will have a clean selector with a combinator which can be queried directly.

11.5 Algorithm

The identification process is rather simple thanks to being able to use the browser. Before the detection begins all CSS selectors are extracted from the retrieved style sheets. The algorithm is presented as algorithm 4.

Algorithm 4 Selector detection

```
1: for all $selector \leftarrow selectors$ do
2:     $selector \leftarrow$ convertRemainingPseudos($selector$)
3:     $selector \leftarrow$ stripStaticPseudos($selector$)
4:     if selectors exists then
5:         store link between selector and page
6:     end if
7: end for
```

The *selector exists* part is handled by CasperJS with the `casper.exists(selector)` function. Internally this calls `document.querySelector(selector)` within the browser's environment. This method is incapable of detecting a major part of the selectors due to the existence of pseudo-elements and -classes. Thus, some cleaning is done by `convertRemainingPseudos()` and `stripStaticPseudos()`. The former converts some undetectable pseudos to attribute alternatives (see subsection 11.3.4). The latter strips the selector from static pseudos, as explained in subsection 11.3.3.

Part III

Evaluation

Chapter 12

Empirical evaluation

One of the main questions of this thesis was to find out how many selectors on web pages remain unused today. Additionally, we define the effectiveness of the crawler and compare it to other crawlers.

The prototype has been testing on a random set of public websites, described in section 12.1. The comparison with other crawlers in terms of reach and effectiveness is covered in section 12.2. The style detector itself is manually tested and presented in section 12.3.

12.1 Experimental objects

Five web sites have been selected, ranging from personal blogs to large business websites. Table 12.1 lists the websites, together with basic information.

ID	Name	URL	Category	Technologies
1	G.SNEL	http://www.snel.nl/	Company website	
2	Emptymind	http://emptymind.me/	Personal blog	Ghost blogging platform
3	HeapAnalytics	https://heapanalytics.com	Company website	Ghost blogging platform
3	Geeking with Greg	http://glinden.blogspot.com	Personal blog	Blogspot blogging platform
4	Burger's zoo	http://www.burgerszoo.nl/	Company website	

Table 12.1: Experimental objects

G.SNEL is a very basic company website with about 270 pages. Other than an overlay script it does not have any special technologies. EmptyMind is a blog based on the Ghost blogging platform[1]. It is small, but has been chosen due to its many rules and hidden content. HeapAnalytics is a website with several subdomains, of which one is a Ghost blog. Geeking with Greg was chosen due to its large amount of pages and its platform being widely popular. Finally, Burger's Zoo was picked because of its many user interactions and graphical effects.

[1]https://ghost.org/

12.2 Crawler effectiveness

The results of the detector are strongly dependent on the effectiveness of the crawler. Like the rest of the thesis the crawler receives its own part. It is not attached to the style detector, thus can be tested separately.

12.2.1 Setup

For each website listed in Table 12.1 we compare the (number of) detected pages to five other crawlers. Two of the crawlers are well-known search engines. The other three are high-ranking sitemap generators. The crawlers are listed in Table 12.2.

ID	Crawler	URL	Page limit
Google	Google	www.google.com	-
Bing	Bing	www.bing.com	-
XS	XML-sitemaps	www.xml-sitemaps.com	500
SG	XML Sitemap Generator	www.xmlsitemapgenerator.org	-
WSM	Web-site-map	www.web-site-map.com	3500+

Table 12.2: Overview of the selector crawlers

Keep in mind that these comparisons may not be fully accurate due to multiple reasons.

We have no information on which filtering rules Google and Bing apply, such as ignoring RSS. What we might see as a new page, they might not. Furthermore we only have access to the data they present us. So, the amount of pages detected by these two search engines can be higher than what is returned. Additionally our prototype and the three sitemap generators use actual data, whereas the search engines may not have crawled the web site recently. This means they may contain recently removed pages and not yet contain recently added pages.

Both search engines also use links from other web sites that may result in discovering pages on the current web site. If there is no link to a certain page on the current web site our prototype and the three sitemaps will not detect that page.

For Google we turned off the filtering mechanism for possibly uninteresting results (according to Google) to give us as many pages as possible. We were unable to find such an option for Bing. Search queries on both these search engines were ran with "*site:example.com*" to expose all URLs belonging to *example.com*.

XML-sitemaps and Sitemap Generator do not automatically crawl sub domains. To cover those anyway we ran them both on sub domains manually and combined the results.

Additionally, our prototype ignores the *robots.txt* file. Abiding the file results in less pages. Using the paths listed in file, but not abiding it, could result in more pages. Furthermore, our crawler does not attempt to detect a sitemap file at the default locations. Doing this may increase the number of detected pages.

Form detection and the State machine are disabled for this part.

Website ID	Prototype	Google	Bing	XML-sitemaps	Sitemap Gen.	Web-site-map
1	270	175	115	270	159	269
2	36	31	27	26	32	26
3	196	183	76	175	171	175
4[2]	1732		647	471	1	1731
5	169	208	178	88	78	141

Note: the numbers listed here may not be the actual number of pages detected. We have removed the URLs that went nowhere, i.e. no regular page, nor an error page.
1: This one continued almost endlessly (over 4000 pages) due to query strings. The given number is after sanitizing. XML-sitemaps hit the 500 page limit (471 after sanitizing). Sitemap Generator failed completely. Google's result was very inaccurate.

Table 12.3: The number of pages crawled by each of the crawlers

Website ID	U	$e_{prototype}$	e_{google}	e_{bing}	$e_{xmlsitemaps}$	$e_{sitemapgenerator}$	$e_{websitemap}$
1	273	0.989	0.641	0.396	0.989	0.582	0.985
2	43	0.837	0.721	0.605	0.605	0.744	0.605
3	196	0.903	0.843	0.350	0.806	0.788	0.807
4	1734	0.999		0.373	0.272[3]	0.001	0.998
5	243	0.695	0.856	0.733	0.362	0.321	0.580

2: XML-sitemaps hit the 500 page limit (471 after sanitizing). Sitemap Generator failed completely.

Table 12.4: The effectiveness of the crawlers

12.2.2 Results

The number of pages that are detected by each crawler is listed in Table 12.3. For each website and for each crawler we have calculated the effectiveness score, presented as Table 12.4. This score is between 0 and 1. 1 means complete coverage of all detected URLs, 0 means no coverage at all. Thus, the higher the effectiveness the more URLs a crawler is capable of finding. The results in Table 12.4 are presented as a box-plot in Figure 12.1.

The **effectiveness** e_x of crawler x on a website y is defined as

$$[H]e_x = \frac{|R_x|}{U}$$

where:

1. R_x is the amount of pages detected by crawler x
2. U is the set of all pages detected by the crawlers, so: $U = R_{prototype} \cup R_{google} \cup R_{bing} \cup R_{xmlsitemaps} \cup R_{sitemapgenerator} \cup R_{websitemap}$.

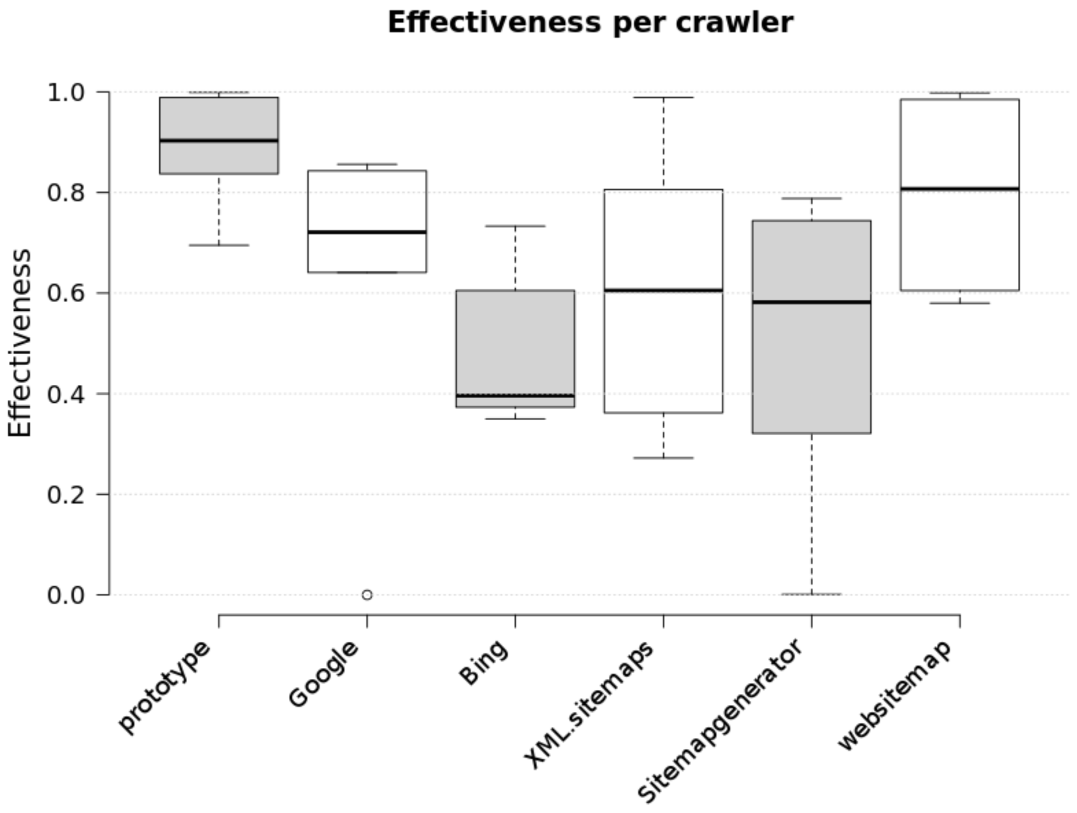

Figure 12.1: The effectiveness score per crawler

12.2.3 Findings

From analyzing the results we discovered that when ignoring the differences due to the *www.* prefix and encoding our prototype was consistently capable of finding almost all pages in U, with percentages between 69.5 and 99.9 (with an average of 88.5%). Thus we can conclude that our crawler is effective and is capable of detecting nearly all the static states of a web site. This gives us more confidence in the results of the Style detector. Figure 12.1 presents this clearly.

When our crawler was outperformed the majority of undetected URLs were caused by incoming URLs from different web sites. As we crawl only the internal links we cannot detect those.

We also discovered that both Google and Bing ignore URLs pointing to 404 error pages. Google completely ignores RSS documents, whereas Bing includes them sporadically.

For Emptymind (ID 2) and HeapAnalytics (ID 3) Google was able to find the */ghost/* path, which points to the login form of the Ghost blogging platform. Reaching this area opens up many new URLs. With form detection turned on this could have led to many more detected pages.

Our crawler also includes some useless URLs, such as *xmlrpc.php* which is used for "pingbacks". The page does not contain any information and is an endpoint for POST requests. Even though this accounts for only one out of hundreds of pages it still influences the effectiveness score.

ID	# Internal sheets	# External sheets	# Unique selectors	Used	Unused
1	2	3	279	151 (54.1%)	128 (45.9%)
2	1	5	1642	154 (9.4%)	1488 (90.6%)
3	6	8	4199	927 (22.1%)	3272 (77.9%)
4	3	3	1733	111 (12.8%)	755 (87.2%)
5	3	3	4354	735 (16.9%)	3619 (83.1%)

Table 12.5: Characteristics of the websites

ID	# pages verified	# used selectors verified	Correct	Incorrect
1	20	400	100%	0%
2	20	300	100%	0%
3	20	360	100%	0%
4	20	400	100%	0%
5	20	400	100%	0%

Table 12.6: Verification results of allegedly used selectors

12.3 Style detector effectiveness

Just like the crawler we test the style detector. During development controlled tests have been used. For this section we have used the same live websites as for the crawler. This distinction means that the scores in this section are influenced by the crawler's effectiveness.

12.3.1 Setup

The prototype is ran on each domain listed in Table 12.1 until it hits the page limit or has exhausted all pages. For each website we have randomly selected 20 of the detected pages. For each page we have randomly selected 20 selectors that were defined as used and verified their validity. We have not verified whether selectors marked as unused were correct, due to it being an even more time-intensive task and we had only limited time.

The verification process is done by opening the web page and its source. For each of the randomly selected selector we have browsed the source code looking for its existence. If the selector could not be found in the source we used the browser's developer tools, which includes the DOM influenced by JavaScript.

12.3.2 Results

Table 12.5 lists some of the characteristics of the websites we have crawled. For each website the number of internal and external style sheets is listed. The number of unique selectors of these style sheets combined is listed as well. The final two columns list how many of those selectors are used and unused, respectively. The percentage is also listed.

The manual testing results are presented as Table 12.6. The second column lists the number of pages we have checked and is always 20, unless there were no 20 pages. The same goes for the selectors. In some cases a page had <10 selectors. The final columns show how many of the selectors were accurately and inaccurately detected by the style detector. A score of 100% correct means no selector was wrongly marked as detected.

12.3.3 Findings

Our two main findings are that (1) our detector has an accuracy of 100%, meaning that the selectors that are marked as detected are truly used, and (2) on average 77% of all included selectors is actually not used in a web application.

Now we cannot decisively say that the 77% is fully accurate. Our tests were limited to five web sites and were all done with the state machine and form detection disabled. In reality the value will probably be between 77% and 60%, the number Ali Mesbah et al. detected.

Additionally we found that the selector usage of Emptymind dropped significantly upon reaching a 404 page. The 404 page resulted in the retrieval an entirely new style sheet (of which the majority is used by the admin panel) for that one single page. As the admin panel has not been crawled, many remained undetected.

Chapter 13

Conclusions

From the research and the evaluation we can conclude several things. This chapter is split into two sections: detection, and crawler. Each section answers their respective research questions listed in chapter 2.

13.1 Detection

RQ1 How can unused CSS selectors be detected?

Unused CSS selectors can be detected by using PhantomJS' browser context, in which you have direct access to the DOM. The DOM can be queried with `document.querySelector(selector)`, which returns the element belonging to a selector. If this element is not null it can be marked as used.

RQ1.1 How to retrieve the CSS rules of a website?

The style rules themselves can be retrieved by using the same query method as above, but then for elements capable of containing style sheets. These are the `<style>` and `<link rel="stylesheet">` elements. The result is parsed to a CSS Object Model and used for detection.

RQ1.2 How to cross-reference the CSS rules to the DOM elements?

If a selector exists (**RQ1**) it always points to an element. If the selector contains any pseudo-classes or pseudo-elements we must first clean the selector. Depending on the type of pseudo we either remove it from the selector, execute some actions, or leave it unchanged.

13.2 Crawler

RQ2 How can websites be crawled?

A website can be crawler either statically or dynamically. The former means the source code is requested and analyzed with Regular Expressions or other means. The latter means the code is executed in a browser environment, in which the code is interpreted. We chose the latter, as it allows us to also crawl dynamic states caused by JavaScript.

Our prototype queries the DOM for elements containing URLs. It then looks for the attributes and whether they actually contain an URL. The detected URLs are filtered and cleaned.

RQ2.1 What elements cause state transitions?

We have found a set of 23 elements capable of carrying URLs. Each of these has a set of attributes with a URL as value. Table 6.1 presents the entire list or elements.

RQ3 How to improve the detection by taking dynamic features such as JavaScript into account?

Earlier research has shown that over half the web is a hidden and can only be research by interacting with a website, such as clicking elements and filling forms.

We have presented a method that intercepts the creation of events and additionally queries the DOM for elements containing events or event handlers. These are then simulated while the state machine builds a state-flow graph. Though, this has not been tested on public websites the current theoretical foundation and its prototypes proved to be effective.

RQ3.1 How can JavaScript influence the Document Object Model?

We have found five different ways in which JavaScript can cause dynamic state transitions. The first one is the creation of custom events. The second is the setting of attributes, which could contain an event handler. Both of these are intercepted and stored. The remaining three originate in the DOM, either are a HTML attribute (`onclick=""`), event property (`element.onclick=""`), and JavaScript links (`<a href="javascript:function()"></a>`).

RQ3.2 How to deal with content hidden behind forms?

Forms are dealt with by detecting them, sending a copy to the GUI, where the user can enter values manually. The form is sent back to the website and submitted. The new page is crawled as normal.

13.3 Evaluation

RQ4 How to use detected unused selectors for their safe removal?

Currently we do not detect overridden selectors and properties. If we did the detected unused selectors could be removed from the style sheets. If, after removal, the style rule has no further selectors we can remove the entire rule.
RQ4.1 What software combination is best suited?

Both Selenium and PhantomJS/CasperJS are suitable for crawling websites and detecting CSS selectors. However, we have only focused on PhantomJS/CasperJS due to personal preference and time constraints.

RQ4.2 What percentage of CSS rules are unused?

Tests done on five public web sites reveal that 45.9 to 90.6%, with an average of 77%, of selectors is used. We suspect the actual value will be a bit lower than 77%, due to the limitations of our crawler and value Crawljax found.
RQ5 What is the effectiveness of our crawler?

Our crawler is capable of covering on average 88.5% of all pages detected by other crawlers, such as Google. The majority of missed URLs are due to them being incoming links, rather than internal links. With these results it outperforms the comparison crawlers, though some do not count 404s.

RQ6 How accurate is the style detector?

Our detector has an accuracy of 100%, meaning that all selectors that are marked as detected are truly used within an application.
An attempt was made to compare our results to CILLA's but we were unable to get CILLA functional within the given time limit.

Chapter 14

Discussion

Limitations Due to multiple reasons not all intended functionality has been implemented, as has been mentioned in the respective sections. Some addition limitations exist:

The crawler starts with a single seed URL. This potentially leaves portions of a web site uncrawled, and thus misses potential selectors. We found that Google was able to find URLs we could not. These URLs opened entirely new areas of the web site. A very interesting improvement would be to first request Google for all URLs Google found.

Some stylesheets contain selectors, properties, and values with backslashes in them. This is a legacy hack for old IE versions, in other browsers these result in parser errors and are thus ignored.

Not all vendor prefixed selectors are currently implemented.

The crawler does not utilize *robots.txt* to detect new seed URLs, nor does it look for a sitemap in the default locations.

It does not attempt to crawl up the tree, e.g. from `/activities/1/` to `/activities/`.

Applications The most straightforward application is as assistance in CSS code maintenance. An option would be to include the prototype as a building step, visualizing code that is no longer useful. The removal of unused rules could greatly benefit both the business and the user. Amazon discovered that a 100ms delay in page load would result in substantial drops in revenue. Thus, remove unused code means smaller file sizes and thus a faster website. Users also benefit, especially those using mobile devices, in that they use less bandwidth.

Additionally, our prototype gives developers insight in the distribution of style sheets and their rules. During evaluation we ran into web sites that included large style sheets on pages that merely required a distinct subset.

Scalability and performance The current prototype is capable of running multiple crawl tasks (i.e. multiple websites) concurrently. However, a single crawl task only uses a single process and is not multi-threaded. The current architecture does not allow for this either. A next version could use this, as large portions of code do not directly rely on the main script.

Threats to validity It is a very difficult area to fully verify. The world wide web is vast and went through major changes in the last decade. Technologies spawned in rapid succession, and with it the hacks to achieve cross-browser compatibility due to lack of standards. Even though we have tested the prototype thousands of times on a variety of websites new edge cases kept arising. Uncertain is how many of these cases are left.

A relatively small set of experimental objects was used during evaluation. We are aware that many more objects are required. Currently there is not a large variety in the experimental objects. We have excluded very large websites (in excess of 1K pages) as the crawler tended the hang after several hundred pages with no apparent reason. We have been unable to find the cause in time.

For many of the difficult to detect selectors we have chosen the safe way, and with careful consideration, decided to mark certain pseudo classes as detected. If, in production, these selectors were falsely undetected harm could be done to the website's layout.

Another threat is the evaluation process. Though many selectors have been manually searched for (selected both randomly and by difficulty), the data set was vast and some unverified ones could have slipped through.

We have failed to get CILLA functioning properly within time and have thus not made any comparisons regarding the accuracy of our detector. Further work definitely needs this to have a more solid conclusion.

Chapter 15

Concluding remarks

This thesis presented a way to detect unused selectors in web applications. It was based upon CrawlJAX and the theory behind CILLA. The prototype is capable of crawling websites with a reach matching Google, Bing, and two site map generators.

The scope of this project turned out to be too big. An enormous amount of extra time has been spent on making a functional prototype. Thus, many more things can be improved. The biggest issue is its stability due to its architecture. In hindsight this architecture proved unsuitable for large websites. It relied on the browser to supply information to the database. A page with Content-Security-Policy activated rendered the crawler useless. Furthermore, the Form detection and dynamic state exploration remain unfinished upon submitting this thesis. However, a strong foundation has been laid for future work. Both of them prove to be very valuable, but still contain some minor issues we could not fix in time.

A comparison between CILLA and our prototype was planned. After spending a week to get CILLA to run, and failing. We have decided not to include this.

Even though the detection process has been tested thoroughly the web is extremely large and filled with edge cases. It has been troublesome to find websites to test on. We have limited us to websites with 30 to 250 pages during development. These websites consisted of several technologies and presented many hurdles, reducing its instability.

As for the chosen software we would not use CasperJS for future iterations. Even though the functions it supplies are useful, there are too many for our needs. A better solution would be to re-implement those functions that specifically suit our needs and go with pure PhantomJS. One of the last changes we made to the prototype was checking for style sheets on every single page. With some minor improvements to both the crawler and the web server it may be possible to go fully multi-threaded. Each crawler would simply receive an URL and extract information. The web server would be responsible for ensuring proper synchronization of data.

Bibliography

[1] HTML & CSS. `http://www.w3.org/standards/webdesign/htmlcss`, June 2015. 1

[2] Zahra Behfarshad and Ali Mesbah. Hidden-web induced by client-side scripting: An empirical study. In *Proceedings of the 13th International Conference on Web Engineering*, ICWE'13, pages 52–67, Berlin, Heidelberg, 2013. Springer-Verlag. 23

[3] Bert Bos. CSS current work & how to participate. `http://www.w3.org/Style/CSS/current-work`. 33

[4] Pierre Geneves, Nabil Layaida, and Vincent Quint. On the Analysis of Cascading Style Sheets. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 809–818, New York, NY, USA, 2012. ACM. 2

[5] CSS3 Working Group. Selectors Level 3. `http://www.w3.org/TR/css3-selectors/`, September 2011. 34, 63

[6] CSS3 Working Group. Selectors Level 4. `http://dev.w3.org/csswg/selectors-4/`, May 2015. 63

[7] Matthew Hague, Anthony Widjaja Lin, and Luke Ong. Detecting Redundant CSS Rules in HTML5 Applications: A Tree-Rewriting Approach. *CoRR*, abs/1412.5143, 2014. 2

[8] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanović, and Rastislav Bodík. Parallelizing the Web Browser. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 7–7, Berkeley, CA, USA, 2009. USENIX Association. 1

[9] Florian Mller Matthias Christen. JavaScript Event Madness! Capturing *all* events without interference. `https://css-tricks.com/capturing-all-events/`, July 2014. 28

[10] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering Refactoring Opportunities in Cascading Style Sheets. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 496–506, New York, NY, USA, 2014. ACM. 2

[11] Ali Mesbah and Shabnam Mirshokraie. Automated Analysis of CSS Rules to Support Style Maintenance. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 408–418, Piscataway, NJ, USA, 2012. IEEE Press. 1, 2, 34

[12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012. 2, 21, 23

[13] Vincent Quint and Irne Vatton. Editing with Style. In *Proceedings of the 2007 ACM Symposium on Document Engineering*, DocEng '07, pages 151–160, New York, NY, USA, 2007. ACM. 1

[14] Tail Garsiel and Paul Irish. How browsers work. `http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/`, August 2011. 1

Appendix A

List of existing selectors

Below is an extensive list of all pseudo-classes and elements to have existed up until the Selectors Level 4 draft of may 29th, 2015[5] [6].

A.1 Elemental selectors

1. Type selector
   (a) `E`
2. Universal selector
   (a) `*`

A.2 Attribute selectors

1. Attribute presence and value selectors
   (a) `E[foo]`
   (b) `E[foo="bar"]`
   (c) `E[foo~="bar"]`
   (d) `E[lang|="en"]`
2. Substring matching attribute selectors
   (a) `E[foo^="bar"]`
   (b) `E[foo$="bar"]`
   (c) `E[foo*="bar"]`
3. class selector
   (a) `E.warning`
4. ID selector
   (a) `E#warning`

A.3 Logical combinations

1. Negation pseudo-class
   (a) `E:not(s)` Version 3: single simple selector
   (b) `E:not(s[,s])` Version 4: a selector list
2. Matches-any pseudo-class
   (a) `E:matches(s[,s])` Takes selector list
3. Relational pseudo-class :has
   (a) `E:has(s[,s])` Takes relative selector list

A.4 Linguistic pseudo-classes

1. Language pseudo-class
   (a) `E:lang(c)`
2. Directionality pseudo-class :dir()
   (a) `E:dir(ltr)`
   (b) `E:dir(rtl)`

A.5 Location pseudo-classes

1. Hyperlink pseudo-class
   (a) `E:any-link`
2. Link history pseudo-class
   (a) `E:link`
   (b) `E:visited`
3. Target pseudo-class
   (a) `E:target`
4. Reference element pseudo-class
   (a) `E:scope`

A.6 User action pseudo-classes

1. Pointer hover pseudo-class
   (a) `E:hover`
2. Activation pseudo-class
   (a) `E:active`
3. Input focus pseudo-class
   (a) `E:focus`
4. Generalized Input focus pseudo-class
   (a) `E:focus-within`
5. Drag-and-drop pseudo-class
   (a) `E:drop`
   (b) `E:drop(active)`
   (c) `E:drop(valid)`
   (d) `E:drop(invalid)`

A.7 Time-dimensional pseudo-classes

1. Current-element pseudo-class
   (a) `E:current`
   (b) `E:current(s)`
2. Past-element pseudo-class
   (a) `E:past`
3. Future-element pseudo-class
   (a) `E:future`

A.8 Input pseudo-classes

A.8.1 Input control states

1. Enabled/disabled pseudo-classes
   (a) `E:enabled`
   (b) `E:disabled`
2. Placeholder-shown pseudo-classes
   (a) `E:placeholder-shown`
3. Mutability pseudo-classes
   (a) `E:read-write`
   (b) `E:read-only`
4. Default-option pseudo-classes
   (a) `E:default`

A.8.2 Input value states

1. Selected-option pseudo-classes
   (a) `E:checked`
2. Indeterminate-value pseudo-classes
   (a) `E:indeterminate`

A.8.3 Input value-checking

1. Range pseudo-classes
   (a) `E:in-range`
   (b) `E:out-of-range`
2. Validity pseudo-classes
   (a) `E:valid`
   (b) `E:invalid`
3. Optionality pseudo-classes
   (a) `E:required`
   (b) `E:optional`
4. User-interaction pseudo-class
   (a) `E:user-error`

A.9 Tree-structural pseudo-classes

1. `E:root`
2. `E:empty`
3. `E:blank`

A.9.1 Child-indexed pseudo-classes

1. `E:nth-child(n)`
2. `E:nth-last-child(n)`
3. `E:first-child`
4. `E:last-child`
5. `E:only-child`

A.9.2 Typed child-indexed pseudo-classes

1. `E:nth-of-type(n)`
2. `E:nth-last-of-type(n)`
3. `E:first-of-type`
4. `E:last-of-type`
5. `E:only-of-type`

A.10 Combinators

1. Descendant combinator
   (a) `E F`
   (b) `E >> F`
2. Child combinator
   (a) `E > F`
3. Next-sibling combinator
   (a) `E + F`
4. Following-sibling combinator
   (a) `E ~F`

A.11 Grid-structural selectors

1. `F || E`
2. `E:nth-column(n)`
3. `E:nth-last-column(n)`

A.12 Pseudo-elements

1. ::first-line pseudo-element
   (a) `E::first-line`
2. ::first-letter pseudo-element
   (a) `E::first-letter`
3. ::before pseudo-element
   (a) `E::before`
4. ::after pseudo-element
   (a) `E::after`
5. ::selection pseudo-element
   (a) `E::selection`
6. ::content pseudo-element
   (a) `E::content`
7. ::shadow pseudo-element
   (a) `E::shadow`

Appendix B

List of on-handlers

- onclick
- oncontextmenu
- ondblclick
- onmousedown
- onmouseenter
- onmouseleave
- onmousemove
- onmouseover
- onmouseout
- onmouseup
- onkeydown
- onkeypress
- onkeyup
- onabort
- onbeforeunload
- onerror
- onhashchange
- onload
- onpageshow
- onpagehide
- onresize
- onscroll
- onunload
- onblur
- onchange
- onfocus
- onfocusin
- onfocusout
- oninput
- oninvalid
- onreset
- onsearch
- onselect
- onsubmit
- ondrag
- ondragend
- ondragenter
- ondragleave
- ondragover
- ondragstart
- ondrop
- oncopy
- oncut
- onpaste
- onafterprint
- onbeforeprint
- onabort
- oncanplay
- oncanplaythrough
- ondurationchange
- onended
- onerror
- onloadeddata
- onloadedmetadata
- onloadstart
- onpause
- onplay
- onplaying
- onprogress
- onratechange
- onseeked
- onseeking
- onstalled
- onsuspend
- ontimeupdate
- onvolumechange
- onwaiting
- onerror
- onmessage
- onopen
- onwheel
- ononline
- onoffline
- onshow
- ontoggle
- onwheel