Automated Analysis of CSS Rules to Support Style Maintenance

Ali Mesbah
University of British Columbia
Canada
amesbah@ece.ubc.ca

Shabnam Mirshokraie
University of British Columbia
Canada
shabnamm@ece.ubc.ca

***Abstract*—CSS is a widely used language for describing the presentation semantics of HTML elements on the web. The language has a number of characteristics, such as inheritance and cascading order, which makes maintaining CSS code a challenging task for web developers. As a result, it is common for unused rules to be accumulated over time. Despite these challenges, CSS analysis has not received much attention from the research community. We propose an automated technique to support styling code maintenance, which (1) analyzes the runtime relationship between the CSS rules and DOM elements of a given web application (2) detects unmatched and ineffective selectors, overridden declaration properties, and undefined class values. Our technique, implemented in an open source tool called CILLA, has a high precision and recall rate. The results of our case study, conducted on fifteen open source and industrial web-based systems, show an average of 60% unused CSS selectors in deployed applications, which points to the ubiquity of the problem.**

***Keywords*-Cascading style sheets; CSS; dynamic analysis; software maintenance; web applications**

I. INTRODUCTION

One of the fundamental W3C standards for developing web applications is Cascading Style Sheets (CSS) [1]. CSS is a language for defining the presentation semantics of HTML elements, including their positioning, layout, colour, and fonts. The main driving force behind adopting CSS has been the separation of structure from presentation. Although this separation of concerns helps a web application's evolution as far as the structure and content is concerned, the CSS code itself is not easily maintainable [2], [3].

Writing CSS code is not trivial [4], [3]. It requires human computer interaction, graphic design, as well as web programming skills [4]. In addition, the language has a number of characteristics [5] such as *inheritance*, *cascading*, and *selector specificity*, which makes understanding how CSS properties are applied to Document Object Model (DOM)[1] elements at runtime a daunting endeavour for web developers. Therefore, developers are continuously faced with challenging questions during web development and maintenance tasks: Is my web application using all of the defined CSS rules? Which ones are obsolete? What will happen if this CSS rule is removed? Will it break the layout in some pages? Is this selector really effective or is it overridden by another rule at runtime? Which DOM elements does this rule affect? Are there any undefined classes in the HTML code? Consequently, as a web application evolves, unused rules or ineffective ones with properties that are always overridden start to accumulate over time. This accumulation of unused code has a number of negative consequences:

- All client-side code, including any unused CSS code, needs to be downloaded and parsed by the browser to load a web application; The larger the code, the higher the load on the network, server, and, browser.
- The web browser is a CPU-intensive program [6], [7]. Benchmarks of popular browsers such as Internet Explorer [8], Safari [9], and Firefox [7] reveal that visual layout consumes 40-70% of the average processing time. On every new page, the browser tries to match the parsed CSS rules against the DOM tree. Matching unused selectors is an unnecessary overhead, which can significantly increase the loading time of a web page [9] and decrease the level of responsiveness;
- Unused code adversely influences program comprehension, maintainability, and ultimately code quality by increasing the probability of introducing errors. In addition to financial costs, web applications that contain errors, such as pages that display incorrectly, result in loss of revenue and credibility [10].

Despite these challenges, CSS analysis has not received much attention from the research community and there is currently a lack of solid tools and techniques to support its comprehension and maintenance. To the best of our knowledge, analyzing CSS code from a maintenance perspective has not been addressed in the literature.

In this paper, we propose a technique that automatically (1) checks CSS code against different DOM states and their elements to infer an understanding of the runtime relationship between the two; (2) detects *unmatched* and *ineffective* rules, *overridden* declaration properties, and *undefined* class values. We have implemented the technique in an open source tool called CILLA. The results of our evaluation show that CILLA has a high precision and recall rate of retrieving unused CSS code. Our empirical study conducted on fifteen web-based systems indicates that, on average, 60% of CSS selectors is unused in deployed web applications.

[1]DOM is a standard object model representing HTML at runtime. It is used for dynamically accessing, traversing, and updating the content, structure, and style of HTML documents.

```
#news { background-color: silver; font: italic; color: black; }

.sports { color: blue; text-decoration: underline; }

H3, H4 { font—family: sans—serif; }

.latest { color: green;}

#news span { color: red; }

P.select { font—size: medium; }
```

Figure 1. Motivating CSS example (example.css). The CSS properties that are used at runtime are shown in bold.

The key contributions of this paper are:

- A discussion of challenges surrounding CSS rule comprehension and maintenance;
- A fully automated technique and algorithms for inferring knowledge about actual style code coverage and selector effectivity;
- An open-source tool implementing the analysis technique and algorithms;
- An empirical study to validate the proposed technique, demonstrating its efficacy and real-world relevance.

II. Background

A CSS rule is composed of two parts: *selector* and *declaration* [11]. Figure 1 presents a simple example of CSS rules and how they are linked to the elements of two DOM states in Figure 2. We will refer to these two figures as the 'motivating example' in the rest of this paper. In the first defined CSS rule, `#news` is the selector part. As the name suggests, a selector *selects* one or more elements from the DOM tree. The declaration part is code between curly brackets, which defines the styling properties to be applied to the selected element(s). In this case, for instance `color` is a declaration property and `black` is the value of that property. The language provides three mechanisms for selecting DOM elements:

Element selectors are defined by using DOM element types; e.g., `P {color: blue }` selects any `<P>` element with or without attributes.

ID selectors are defined by using the `'#'` prefix and are matched based on the value of the `ID` attribute of DOM elements; e.g., `#news {background-color: silver}` selects `<P id="news" ...>`.

Class selectors are defined by using the `'.'` prefix and are matched based on the value of the `class` attribute of DOM elements; e.g., `.sports {color: blue}` selects `<SPAN class="sports">`.

Both ID and class selectors can also be defined with an associated element type. For instance `P.select {font-size: medium}` selects `<P>` elements with a class attribute value equal to `select`. If no element type is defined, the rules apply to all matched element types.

Complex selectors can be written by combining multiple simple selectors separated by white space, e.g., `#news span` selects a DOM `SPAN` element that is the child of an element with an ID attribute equal to `news`. If the same declaration should be applied to different selectors, selectors can also be *grouped* together. Grouping happens by separating the selectors with a comma (e.g., `H3, H4 {font-family: sans-serif}`).

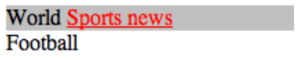
CSS also has the notion of *pseudo elements* (e.g., `P:first-line`) and *pseudo classes* (e.g., `A:visited`) to allow style formatting based on information that lies outside the DOM tree [12].

Inheritance. Inheritance in CSS [5] allows a styling property value to be propagated from the parent to the descendent elements. Our motivating example shows a simple case of inheritance in working. The `SPAN` element inherits the background color property (`silver`) from its parent element `P`, through the CSS rule `#news`. An inherited property can also be overridden in CSS. For instance, the same `SPAN` element has a parent with a font color of black, but since there is a more specific color property defined for the element through `#news span`, the value of the inherited property is overridden by `red`.

Cascading and Specificity. The cascading notion in CSS ultimately determines which properties will be applied to a selectable DOM element. The cascading order is based on two main concepts: *specificity* and *location*. When there is a competition on an element from two or more CSS rules for the same property (e.g., color), the language applies these methods to determine which rule takes precedence.

DOM 1:

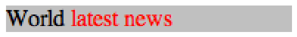
```
<HTML>
 <HEAD>
  <LINK href="example.css" rel="stylesheet" type="↩
       text/css"/>
 </HEAD>
 <BODY>
   <P id='news' style='font:normal'>World
     <SPAN class='sports'>Sports news</SPAN>
   </P>
   <DIV class='sport'>Football</DIV>
 </BODY>
</HTML>
```



World Sports news
Football

DOM 2:

```
<HTML>
 <HEAD>
 <LINK href="example.css" rel="stylesheet" type="↩
      text/css"/>
 </HEAD>
 <BODY>
   <P id='news' style='font:normal'>World
     <SPAN class='latest'>latest news</SPAN>
   </P>
 </BODY>
</HTML>
```



World latest news

Figure 2. Two DOM states of our motivating example. The visual effects of the CSS code from example.css are presented under each state.

1) Specificity: As we have seen, CSS provides different selector types to target DOM elements and each of these types carries a different weight of importance. The specificity weight of a selector is the sum of all its selector type's weights. The more specific a selector points to a DOM element, the higher its specificity weight.

In our motivating example, both `#news span` and `.latest` selectors compete for the `<SPAN class='latest'>` element's color property. `#news span` wins because it is more specific about the element (i.e., it specifies both the tag name and the parent's ID), and thus the DOM element receives a red color.

In case an element is targeted by multiple selectors carrying the same specificity weight, the selectors' location becomes the determining factor.

2) Location: Location is determined by the source of a CSS rule and its position. CSS rules can be defined in three different sources:

Inline declarations are defined on specific HTML elements using the `style` attribute;

Embedded rules are defined inside a `<STYLE>` element within an HTML document's header;

External style sheets are separate files, usually with a `.css` extension, referenced from HTML documents.

When multiple rules from various sources select the same element, the rule closest to the DOM element has the highest priority. Based on this definition, the priority scheme becomes from highest to least: inline, embedded, and external style rules. The language also allows authors to use the `!important` declaration on the property level. A property designated as important, receives the highest priority over competing rules.

Challenges and Motivation. Our motivating example is a very simple example of the kind of CSS code that is present on the web. The code snippet is showing the properties that are applied on the elements of the two DOM states at runtime in bold. This clear indication is, however, not readily available when developers write or maintain CSS code. Even in this simple example, with only two DOM states and a few lines of CSS code, it is not trivial to understand how the rules are applied to the DOM elements. Having to work on a web project with thousands of lines of CSS code and hundreds of DOM states can, thus, be very challenging. Our aim in this work is to provide a technique that can automatically provide the developer with information on CSS rule usage.

III. Related Work

We categorize related work into two groups, namely *CSS analysis* and *unused code detection*.

CSS Analysis. Despite its widespread adoption and maintenance challenges, CSS code has not received much attention from the research community. Lie [13] presents an in-depth discussion of style sheet languages and some of the design decisions behind CSS. Keller and Nussbaumer [4] compare user authored CSS code to generated CSS code and propose an abstractness factor for measuring code quality. They argue that manual CSS code has a higher abstractness factor than generated code. Quint and Vatton [3] provide an overview of techniques and tools for editing style sheets. They identify challenges CSS developers face and argue that robust CSS debuggers and rule analyzers are a necessity for web developers. Meyerovich and Bodik [9] propose algorithms for CSS selector matching, layout solving, and font rendering to improve page loading performance in browser layout engines.

Extending CSS to account for some of the limitations of the language has gained more attention. For instance, Badros et al. [2] propose a constraint-based style sheet model on top of CSS2, called CCSS, which allows a more flexible specification of the layout. A more recent language extension is Sass [14], which supports variables, nested rules, mixins, inline imports, and selector inheritance. The code written in Sass can be automatically translated into valid CSS code.

Current industrial tools for analyzing CSS code are mainly static analyzers concerned with either standard conformity, such as W3C CSS validator [15], or code formatting and optimizations, such as CSSTidy [16], CSSClean [17], and CSS Lint [18]. A few industrial tools exist, such as Dust-Me Selectors (DMS) [19] and CSSess [20], which target CSS rule analysis. All such tools are, however, still immature. In particular, they produce a high rate of false positives and false negatives (as shown and discussed in Sections VI-VII).

Unused Code Detection. Detecting unused code in conventional programming languages (e.g., Java, C++) has been explored in a number of different contexts.

Dead code and unreachable code elimination is addressed in compiler optimization techniques [21], [22]. The main focus in these techniques is optimizing code generation. Tip et al. [23] use program transformation and extraction techniques to detect and remove unreachable methods and redundant fields in Java. Their goal is to reduce the size of distributed applications deployed via the Internet. Tempero [24] presents an empirical study on unused design decisions in Java applications. The study indicates a high degree of unused code in open source software. Industrial tools such as PMD [25] and UCDetector [26] aim at spotting unused local variables, parameters, and methods in Java. Another related topic is clone detection and removal to improve program comprehension and maintenance. Roy et al. [27] provide an overview of clone detection tools and techniques.

To the best of our knowledge, analyzing CSS rule usage and effectivity has not been addressed in the literature.

IV. Our Approach

Our overall approach is based on dynamic analysis in which we automatically drive a given web application and infer the *runtime* relationship between the CSS rules and DOM elements of the navigated states. We use this inferred relational knowledge to spot unused CSS selectors.

The Venn diagram of Figure 3 depicts the total landscape of CSS selector coverage and what our approach targets. Set *AS* represents all the CSS selectors present in a web

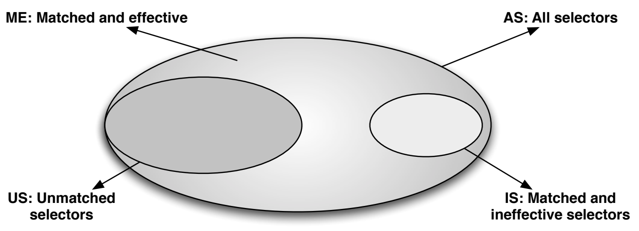

ME: Matched and effective
AS: All selectors
US: Unmatched selectors
IS: Matched and ineffective selectors

Figure 3. CSS selector coverage landscape.

application. *US* is the set of unmatched selectors, i.e., selectors with no DOM element counterparts. The set *AS* − *US* contains all the matched selectors, from which the set *IS* encompasses all the ineffective ones, i.e., the selectors that are matched but have no effect on the DOM elements. *ME* is the set of all matched and effective selectors. Ideally, after each development and maintenance cycle, the CSS code should only contain the set of selectors in *ME*.

Our technique operates in three steps to detect unused selectors and undefined class values. In the first step, we execute each selector against all DOM states to differentiate unmatched selectors (*US*). In the second step, the matched selectors (*AS* − *US*) and DOM elements are analyzed to detect the set of ineffective selectors (*IS*). The output of our technique as far as the total unused selectors is concerned is the set *US* ∪ *IS*. There is also the set of class values (*UC*) present on DOM elements that are not defined in CSS code (not shown on the diagram). In the last step, we use the matched selectors to spot these undefined class values. Each of these steps is further described in the following subsections.

A. Relation between CSS and DOM

The first step in our analysis is to determine the relation between CSS rules and their counterpart DOM elements.

Definition 1 (Matched Selector) *Let Σ denote the set of DOM states, then a CSS selector* S *is said to be matched if and only if there exists at least one DOM element $E \in \Sigma$ so that it is selectable by* S.

Based on this definition, a *selectable element* is a DOM element matched by a selector. Algorithm 1 shows our algorithm for analyzing CSS selectors and their counterpart DOM elements. For each detected DOM state (line 5), the algorithm extracts all the new CSS rules and adds them to the overall set of CSS rules (line 6). The utility function EXTRACTNEWCSSRULES scans each new DOM tree looking for unexamined embedded rules and external CSS files. Using the URL of each external CSS resource, the content is retrieved through a HTTP request. Then the extracted CSS code is parsed and transformed into an object model, containing all the information about the rules and declarations, as well as their resource (e.g., URL of the CSS file) and location (e.g., line number).

Since a CSS rule can have grouped selectors, we iterate over the set of selectors of the current rule (lines 7-9). Then,

Algorithm 1: Analyzing Selectors and DOM Elements

```
   input : $\Sigma$: set of visited DOM states
   output: Set of annotated CSS rules (R), unmatched selectors
           (US), selectable DOM nodes ($\Omega$)
1  begin
2      Set R ← ∅
3      Set $\Omega$ ← ∅
4      Set US ← ∅
5      foreach $dom \in \Sigma$ do
6          R ∪ EXTRACTNEWCSSRULES($dom$)
7          foreach $rule \in R$ do
8              S ← $rule$.GETSELECTORS()
9              foreach $selector \in S$ do
10                 $xpathExpr$ ←
                   TRANSFORMTOXPATH($selector$)
11                 $nodes$ ←
                   $dom$.EVALUATEXPATH($xpathExpr$)
12                 if $nodes \neq \emptyset$ then
13                     $selector.matched \leftarrow true$
14                     $selector.matchedNodes \leftarrow nodes$
15                     $\Omega \cup nodes$
16     foreach $rule \in R$ do
17         foreach $selector \in rule$ do
18             if $selector.matched = false$ then
19                 US ∪ {$selector$}
```

to check whether a selector matches any DOM elements, we transform the selector into a corresponding XPath expression (line 10) that can be evaluated on the current DOM tree (line 11). We have defined a mapping of the CSS selector syntax to the XPath language. For instance, the selector `#news span` is translated to `./descendant::*[@id = 'news']/descendant::SPAN`.

In case the XPath expression retrieves elements from the DOM (line 12), we annotate the selector as matched (lines 13-14) and add the retrieved elements to the overall set of selectable elements (line 15).

After all the DOM states are covered and having annotated all the matched selectors and selectable elements, the algorithm iterates over all the rules to find the set of unmatched selectors (lines 16-19). Going back to our motivating example, this algorithm returns `P.select`, `H3`, and `H4` as the detected unmatched selectors since these do not touch any of the elements in the two given DOM states. In addition, through this algorithm, we know exactly which elements in the two DOM states are selectable by which CSS selectors. We use this information in the following step of our analysis.

B. Effective Selectors and Declaration Properties

Not all matched selectors are used at runtime per se. For instance, although the `.latest` selector in Figure 1 matches `<SPAN class='latest'>` in state 2 of Figure 2, the selector is actually not *effective* due to its low specificity weight. Thus, to understand a selector's effect on DOM elements, a more specific definition is required.

Algorithm 2: Analyzing Declaration Properties

```
input : $\Omega$: set of selectable DOM nodes; R: set of annotated
        CSS rules
output: Set of ineffective selectors (IS) and properties
1  begin
2      foreach $node \in \Omega$ do
3          $overridden \leftarrow$ GETRANDOMINT()
4          S $\leftarrow$ GETMATCHINGSELECTORS($node$, R)
5          OS $\leftarrow$ ORDERSPECIFICITYLOCATION(S)
6          for $i \rightarrow 1$ to OS.SIZE() do
7              $selector \leftarrow$ OS$_i$
8              P $\leftarrow$ $selector$.GETPROPERTIES()
9              foreach $property \in P$ do
10                 if $property.status \neq overridden$ then
11                     $property.status \leftarrow EFFECTIVE$
12                     for $j \rightarrow i+1$ to OS.SIZE() do
13                         $selectorC \leftarrow$ OS$_j$
14                         PC $\leftarrow$ selectorC.GETPROPERTIES()
15                         foreach $propertyC \in PC$ do
16                             if property.name =
                               propertyC.name then
17                                 $propertyC.status \leftarrow$
                                   $overridden$

18 Procedure INEFFECTIVESELECTORS(R) begin
19     Set IS $\leftarrow \emptyset$
20     foreach $rule \in R$ do
21         foreach $selector \in rule$ do
22             $counter \leftarrow 0$
23             P $\leftarrow$ selector.GETPROPERTIES()
24             foreach $property \in P$ do
25                 if $property.status \neq EFFECTIVE$
                   then
26                     $counter++$
27             if counter = P.SIZE() then
28                 IS $\cup$ {$selector$}
```

Definition 2 (Effective Selector) *A matched CSS selector* S *is said be effective if and only if* S *has at least one declaration property that is applied to a selectable DOM element $E \in \Sigma$.*

Algorithm 2 shows our overall algorithm for analyzing the effectivity of selectors and declaration properties. The input to this algorithm consists of the set of annotated CSS rules as well as the set of selectable DOM elements from Algorithm 1.

For each selectable DOM element, the algorithm starts by retrieving all the corresponding CSS selectors that matched that particular element (line 4). Then it sorts the list of matched selectors (line 5) according to their calculated specificity weight in decreasing order.

Based on the CSS specification [5], we calculate the overall specificity weight (*SW*) of a selector *S* composed of different selector types as follows:

$$SW(S) = concatenate(a, b, c, d)$$

where a, b, c, and d are calculated as follows:

- $a = \begin{cases} 1 & \text{if the declaration is inline} \\ 0 & \text{otherwise} \end{cases}$
- $b =$ number of ID types in *S*
- $c =$ number of class types in *S*
- $d =$ number of element types in *S*

Going back to our motivating example, `#news span` and `.latest` selectors have specificity weights of 0101 (1 ID attribute and 1 element type) and 0010 (1 class type) respectively. In this case `#news span` has a higher specificity.

If two selectors have the same specificity weight, then their cascading order in terms of their location is used for ordering (See Section II).

We know by definition that the declaration property of a selector that has a higher cascading order and specificity weight wins over all the competing selectors. Thus, once the selectors for an element are ordered based on their specificity and location, the algorithm starts annotating the properties in that exact order. For each property in a selector, the algorithm first checks whether the property has already been overridden by a higher order property (line 10). If that is not the case, the property is marked as *EFFECTIVE* (line 11). When an effective property has been identified, all the properties of the competing selectors that have the same property name are marked as *overridden* (lines 12-17). At the end of this step, we can distinguish between effective and ineffective properties.

Once all the properties are annotated, we can retrieve the set of ineffective (or effective) selectors. The INEFFECTIVESELECTORS procedure iterates over all the annotated rules and selectors (lines 20-21). If none of the properties of a selector is marked as *EFFECTIVE* (lines 24-27), then that selector is added to the set of ineffective selectors (line 28).

By applying this algorithm to our motivating example, `#news`, `.sports`, and `#news span` are identified to be effective, since they all have at least one effective property. This leaves `.latest` as ineffective.

Note that the total set of unused selectors that our approach returns is a union of unmatched selectors and ineffective selectors (e.g., {`P.select, H3, H4, .latest`}). Using the same principle, we can retrieve the total set of unused CSS properties (i.e., all the properties that are not marked *EFFECTIVE* plus the properties of unmatched selectors).

C. Detecting Undefined Classes

We reverse the process and analyze the DOM elements to examine whether all the class attribute values are defined as CSS rules.

Definition 3 (Defined Class Values) *A class attribute value C attached to a DOM element is said be defined if and only if there exists at least one CSS selector that matches C.*

There are at least four scenarios imaginable for a CSS class to be undefined: (1) the developer forgets to define

Algorithm 3: Analyzing Class Attribute Values

```
input  : $\Sigma$: set of visited DOM states; R: set of annotated CSS
         rules.
output: Set of undefined class values (UC)
1  begin
2      Se C $\leftarrow \emptyset$
3      Set DefC $\leftarrow \emptyset$
4      Set UC $\leftarrow \emptyset$
5      foreach $dom \in \Sigma$ do
6          N $\leftarrow$ $dom$.GETELEMENTSBYTAGNAME(*)
7          foreach $node \in N$ do
8              $attr \leftarrow node$.GETATTRIBUTE($'class'$)
9              CV $\leftarrow attr$.GETVALUES()
10             foreach $value \in CV$ do
11                 C $\cup \{value\}$
12                 foreach $rule \in R$ do
13                     S $\leftarrow rule$.GETSELECTORS()
14                     if $value \in S$ then
15                         $DefC \cup \{value\}$
16                         $break$
17     UC $\leftarrow \{C - DefC\}$
```

the CSS rule for the class value; (2) the CSS rule definition is removed making the class obsolete; (3) a mistake (e.g., typo) is made in attaching a defined class value to the DOM element (4) the class value is used for purposes other than styling such as querying the DOM through JAVASCRIPT. The first three scenarios require close inspection by the developers to either fix the error or clean up the unused code.

Algorithm 3 presents our method for detecting undefined CSS class values. This algorithm takes as input the set of visited DOM states along with the set of annotated CSS rules. For each DOM state, it first extracts all the nodes from the tree (line 6). Then, the values of each node's `class` attribute are retrieved (line 8). Each class value is then subsequently checked against all the selectors (lines 10-13). If a selector matches the class value, the algorithm adds the class to the set of defined classes (line 15) and continues with the next class value. At the end, the algorithm returns the set of undefined classes (line 17). When applied to our motivating example, the class attribute value `sport` from `<DIV class='sport'>` is returned as undefined.

V. Tool Implementation

We have implemented our CSS analysis technique in an open source tool called CILLA.[2] We use CSSPARSER [28] to parse CSS source code and transform the rules into a DOM 2 Style tree [29]. For automating the DOM state exploration phase, we use CRAWLJAX [30], [31], a dynamic web crawler capable of detecting DOM state changes of AJAX-based web applications.

When the initial index page of a web application is loaded, CILLA extracts all external and embedded CSS sources. Then it retrieves and parses the CSS code to create a map of all the CSS rules, selectors, and property declarations associated with each source. Each rule's relation to the elements of the current DOM tree is examined and the map is annotated with the findings (Algorithms 1-2). Then the class attributes of all the elements on the DOM tree are analyzed (Algorithm 3).

Our tool currently excludes pseudo elements/classes from the analysis process, because inherently their relation cannot be deduced from the DOM tree [12].

For each consecutive new DOM state, CILLA first examines all the CSS sources defined in that particular state. Any new CSS code that is identified is parsed and added to the map before the analysis is carried out for that particular state.

When the state exploration phase is done, the tool iterates over the entire CSS rule entries in the map and reports about the findings. The first entry in the report consists of the total number of (1) examined CSS resources, rules, selectors, and properties (2) detected matched and unmatched selectors (3) detected effective and ineffective selectors (4) detected used and unused properties (5) and detected undefined class values. In addition, details of the detected ineffective and unmatched selectors including their location and line number, and undefined class values are reported. The report also includes all effective and matched rules along with the list of corresponding DOM elements for each selector.

VI. Empirical Evaluation

To assess the effectiveness and real-world relevance of our approach, we have conducted a case study following guidelines from Runeson and Höst [32]. Our evaluation addresses the following research questions:

RQ1 What is the overall accuracy of CILLA in detecting unused CSS code?

RQ2 How does CILLA's detection rate compare to existing approaches?

RQ3 What percentage of CSS code is typically unused in online web-based systems?

A. *Experimental Objects*

During the implementation and testing phases of CILLA, we used two web applications with unused rule sets that were well-known to us. However, in the evaluation phase, our goal is to assess the effectiveness of the approach on real-world web applications. Our study includes fifteen web-based systems in total. Two are open source, namely, Phormer, which is a photo gallery written in PHP, and Igloo, a simple company website taken from the book 'Pro CSS for High Traffic Websites' [33]. Seven of the systems are randomly selected using the random link generator provided by Yahoo! [34], which has also been used in other studies [35]. The randomly selected web systems include Becker, Equus, PTE, Vanities, LCN, EmployeeSolutions, and Sync. We further include six online web applications in our list of experimental objects, namely those of the ICSE 2012

[2] http://salt.ece.ubc.ca/content/cilla/

Table I
EXPERIMENTAL OBJECTS.

ID	Exp. Object	Resource
1	Igloo	http://www.apress.com [33]
2	Phormer	http://p.horm.org/er/
3	Becker	http://www.beckerelectric.com
4	Equus	http://www.equuscap.com
5	PTE	http://www.protoolsexpress.com
6	Vanities	http://www.uniquevanities.com
7	LCN	http://www.centralfloridaphones.com
8	ICSE12	http://www.ifi.uzh.ch/icse2012/
9	EmployeeSolutions	http://www.employeesolutions.com
10	Sync	http://www.synccreative.com
11	GlobalTVBC	http://www.globaltvbc.com
12	Lenovo	http://www.lenovo.com/ca/en/
13	MountainEquip	http://www.mec.ca
14	Staples	http://www.staples.ca
15	MSNWeather	http://local.msn.com/worldweather.aspx

conference, GlobaLTVBC news, Mountain Equipment CO-OP, Staples Technology and Electronics, and MSN weather forecasts.

Table I shows each system's ID, name, and resource. The characteristics of these systems in terms of their size and complexity is shown in Table II.

B. Experimental Setup

Our study is composed of two parts. The first part (RQ1-2) focuses on the accuracy of CILLA and compares it to tools currently available to web developers for analyzing CSS selectors. The second part (RQ3) empirically explores how omnipresent unused CSS code is in web applications. The experimental data produced by CILLA is available for download.[2]

To run the experiment, we provide the URL of each experimental object to CILLA and choose the default crawling settings for event generation and DOM state comparison. Then we run the tool and save the generated output.

To evaluate the accuracy of reported unused CSS selectors (RQ1), we measure precision, recall, and F-measure as follows:

Precision is the rate of detected unused selectors found by the tool that are correct: $\frac{TP}{TP+FP}$

Recall is the rate of correct unused selectors that the tool finds: $\frac{TP}{TP+FN}$

F-measure is the harmonic mean of precision and recall: $\frac{2 \times Precision \times Recall}{Precision + Recall}$

where TP (true positives), FP (false positives), and FN (false negatives) respectively represent the number of unused CSS selectors that are correctly detected, falsely reported, and missed. To document TP, FP, and FN in a timely fashion while preserving accuracy, we randomly select 20% of the CSS rules from the first ten systems in Table I as follows:

$$\alpha = \begin{cases} \lceil 0.2 \times n \rceil & \text{if } n \geq 100 \\ 20 & \text{otherwise} \end{cases}$$

where, α is the number of randomly selected CSS rules and n is the total number of CSS rules. We manually examine the samples against the reported output. It is worth mentioning that manual checking of CSS rules is a labour intensive task. In this study, examining the ten samples took approximately 36 hours (in 4 days).

Table II
CHARACTERISTICS OF THE EXPERIMENTAL OBJECTS.

ID	# CSS Files	LOC (CSS)	# CSS Rules	# CSS Selectors	# Ignored Selectors	# CSS Properties	# Ignored Properties	# DOM states
1	2	832	150	321	76	864	159	2
2	1	854	114	120	13	384	13	58
3	3	2505	178	192	12	775	26	32
4	6	530	46	46	20	157	89	69
5	2	3642	353	405	20	1006	41	146
6	8	6764	706	754	24	2125	46	100
7	36	11504	1131	1452	133	3915	265	41
8	5	2506	253	336	10	655	21	47
9	3	2011	183	242	22	523	32	18
10	58	3705	361	372	2	1065	2	53
11	58	40802	6439	7822	671	18305	1785	182
12	11	11440	531	664	87	2077	189	235
13	14	9457	1434	1883	143	3987	368	154
14	39	14757	1052	1347	197	4504	794	205
15	63	6414	1535	2133	282	4516	534	253

To answer RQ2, we compare the results produced by CILLA to the results generated by two industrial open source tools, namely Dust-me Selectors (DMS) [19] and CSSESS [20]. Similar to CILLA, both these tools aim at spotting unused CSS rules. Since DMS is a Firefox plugin, we use Firefox for the state exploration phase in all our experiments to make comparisons possible (CILLA supports IE and Chrome as well). Note that the same set of randomly selected samples are used to conduct the comparisons through precision, recall, and F-measure as defined above. Since CILLA, DMS, and CSSESS are not able to analyze pseudo classes/elements, we ignore pseudo items in the evaluation of all the three tools (See Table II, # Ignored Selectors).

To address RQ3, we run CILLA on all fifteen experimental objects and calculate the percentage of the CSS code that is found to be unused.

C. Results

Table II presents the examined properties of each system, produced by CILLA. The table shows the number of examined CSS files, total lines of embedded and external CSS code, number of CSS rules, number of CSS selectors, number of ignored selectors (pseudo elements/classes), number of CSS declaration properties, number of ignored properties (belonging to a pseudo selector), and the number of DOM states examined.

Table III shows the results of our evaluation produced by CILLA. For each experimental object, the table shows the detected number of undefined classes (*UC*), unmatched selectors (*US*), ineffective selectors (*IS*), unused selectors (*US+IS*), unused declaration properties (*UP*), the percentage of unused selectors, and the percentage of unused properties.

Table III
EVALUATION RESULTS PRODUCED BY CILLA.

ID	UC	US	IS	US+IS	UP	% US+IS	% UP
1	16	80	23	103	90	42	13
2	0	20	5	25	83	23	22
3	2	49	9	58	223	32	30
4	0	2	5	7	14	27	20
5	13	127	6	133	288	34	30
6	15	433	70	503	1335	69	64
7	140	897	45	942	1906	71	52
8	42	211	8	219	334	67	53
9	18	158	4	162	341	74	69
10	43	207	2	209	551	56	52
11	95	5491	952	6443	14290	90	86
12	450	319	21	340	978	59	52
13	28	1237	103	1340	2447	77	68
14	31	1018	40	1058	3351	92	90
15	10	1509	17	1526	3199	82	80

Table IV
DESCRIPTIVE STATISTICS OF UNUSED SELECTORS AND PROPERTIES.

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
% IS	1.00	3.75	6.20	12.88	15.15	71.40
% US+IS	23.00	38.00	67.00	59.67	75.50	92.00
% UP	13.00	30.00	52.00	52.07	68.50	90.00
% Red.	10.00	23.00	32.00	33.07	44.50	58.00

We calculate the percentage of unused selectors (% (*US+IS*)) and unused properties (*%UP*) as follows:

$$\%(US+IS) = (\frac{US+IS}{\#CssSelectors - \#IgnoredCssSelectors} \times 100)$$

$$\%UP = (\frac{UP}{\#CssProperties - \#IgnoredCssProperties} \times 100)$$

Table IV shows descriptive statistics of the percentages of detected ineffective selectors (% *IS*), unused selectors (% (*US+IS*)), unused properties (% *UP*), and reduction in CSS code size if the unused code were to be removed (% *Red.*).

D. Findings

1) Accuracy: The precision and recall rates, measured for CILLA, DMS, and CSSESS, are presented in Figure 4. The F-measure is shown in Figure 5. As far as RQ1 is concerned, our results show that CILLA is highly accurate in detecting unused CSS selectors. The recall is 100%, meaning that our approach can successfully spot all unused selectors of type class, ID, and element, present in a web application. The precision oscillates between 80-100%, which is caused by a low rate of false positives (discussed in Section VII under Limitations). The F-measure varies between 90-100%. The comparisons in Figure 4 and 5 emphasize that the accuracy of CILLA is higher than that of DMS and CSSESS in recall, precision, and F-measure (RQ2). DMS performs better than CSSESS but it still suffers from a high rate of false positive and negative.

2) Unused CSS Code: Figure 7 depicts a bar plot of the percentages of unused CSS selector and declaration properties for all the fifteen systems. The numbers on each bar represent the number of unused entities. In order to obtain the results shown in Figure 7 in a reliable manner, we varied the number of DOM states (from 10 to 200) CILLA had to analyze from each system. By increasing the number of DOM states covered, we find out when the number of detected unused selectors as well as the percentage stabilize

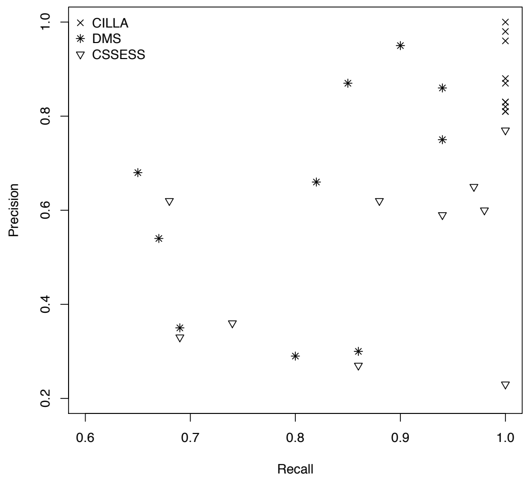


Figure 4. Plot of precision and recall for CILLA, DMS, and CSSESS.

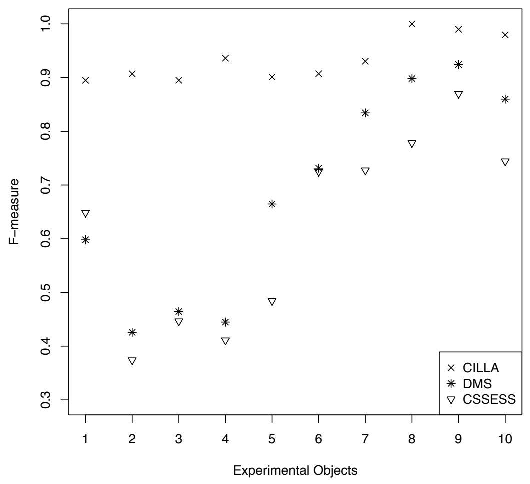


Figure 5. F-measure of CILLA, DMS, and CSSESS obtained for each object.

(i.e., remain unchanged). We depict the results in Figure 6 obtained from six systems with high numbers of DOM states. The percentage of unused selectors oscillates when we increase the number of explored DOM states from 10 to 50. After 50, the graph stabilizes, indicating that the process of detecting unused selectors has reached a steady state. The numbers in Figure 7 represent the percentages of unused selectors taken after stabilization.

Going back to RQ3, the high percentages reported in Figure 7 clearly indicate that there is a huge amount of unused CSS code in online web applications. As shown in Table IV, on average, there are 59.67% unused CSS selectors and 52.07% unused declaration properties. Staples has the highest unused percentages: 92% selectors and 90% declaration properties.

Our results show that ineffective selectors are present in web applications and they can form up to 71.4% of the total

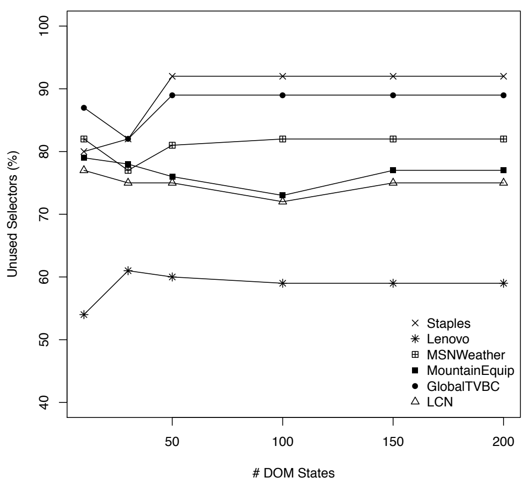


Figure 6. Percentage of unused selectors versus the number of DOM states.

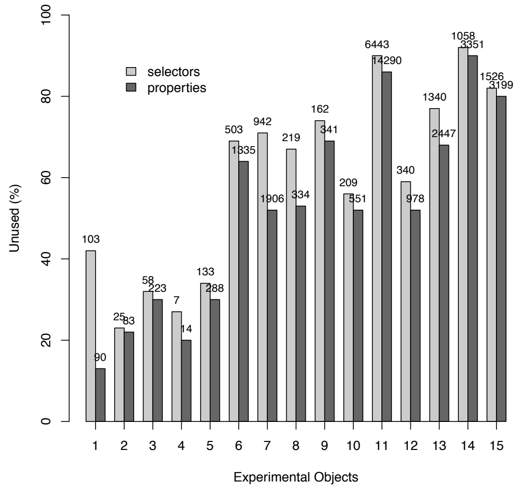


Figure 7. Bar plot of the percentage of unused selectors and declaration properties. The numbers on each bar represent the number of unused entities.

number of unused selectors. Unmatched selectors, however, constitute the largest portion of unused code. There are also a number of undefined class values in HTML code. The highest number reported in our study is 450 for Lenovo. As shown in Table IV, CSS file size can be reduced by up to 58% merely by eliminating unused selectors and properties.

VII. Discussion

Correlations. To examine the relationship between the number of unused selectors and the independent variables in Table II, we used R [36] to calculate the non-parametric Spearman correlation coefficients (r) as well as the p-values (p), and plotted the graphs. We present the combinations that indicate a possible correlation. Figure 8 depicts the scatter plot of the total number of selectors versus the number of detected unused selectors with the line of best fit. Clearly, there is a high linear correlation ($r = 0.964$, $p = 0$) between the two: The higher the number of CSS selectors, the more unused selectors are present in a web application.

More interesting is Figure 9, which shows the scatter plot of the percentage of unused selectors versus the average DOM size. The correlation coefficient ($r = 0.557$, $p = 0.017$) suggests that the variables are positively co-related. One reason might be that when there are more DOM elements present, it becomes more difficult for developers to have a clear understanding of their CSS code usage and effectivity. Another reason could be that when there are more DOM elements and selectors, developers become more reluctant to remove unused selectors in fear of breaking the style, since they do not know exactly if or where a certain selector is used.

Limitations. Similar to other tools, one of the limitations of our technique is that pseudo elements/classes are currently not supported. Table II presents the number of ignored pseudo selectors and properties. As mentioned earlier, we ignore pseudo items in the evaluation of all the three tools. Therefore, any unused pseudo item that is ignored could be regarded as a false negative. As shown in Figure 4, unlike DMS and CSSESS, CILLA achieves a 100% recall rate. This implies that in CILLA, false negatives could potentially be caused only by unused pseudo classes/elements.

Since our approach is based on dynamic analysis, the explored state space is only a subset of the entire state space. This limitation is in part inherent to all dynamic analysis techniques. Thus, if a CSS selector is used in a certain DOM state not present in the explored set, then that selector is mistakenly marked as 'unused'. A low rate of false positives produced by CILLA is caused by these CSS rules. The conclusive outcome from our evaluation, apart from the fact that unused CSS code is omnipresent on the web, is that within the automatically explored state space, our technique is capable of detecting unused selectors with a high level of accuracy.

Applications. The first direct application of our technique is in CSS code maintenance. CILLA can be incorporated into web development cycles through automated nightly runs. The reported unused code can, for instance, be deleted to maintain a clean code base and decrease the processing load on the browser. Specifically, mobile web applications could greatly benefit by avoiding the download of unnecessary CSS code to decrease bandwidth usage and user-perceived latency.

What we witness in the evaluation is that some unused rules are a result of reusing the entire CSS rule set of a parent website, while perhaps only a small subset is needed. Others are a result of copy-paste from other resources. CILLA can tell web developers exactly what the used subset is so that the remaining obsolete rules can be avoided. In addition, our technique can assist in understanding the relationship between CSS rules and DOM elements at runtime. For instance, CILLA provides all DOM elements affected by

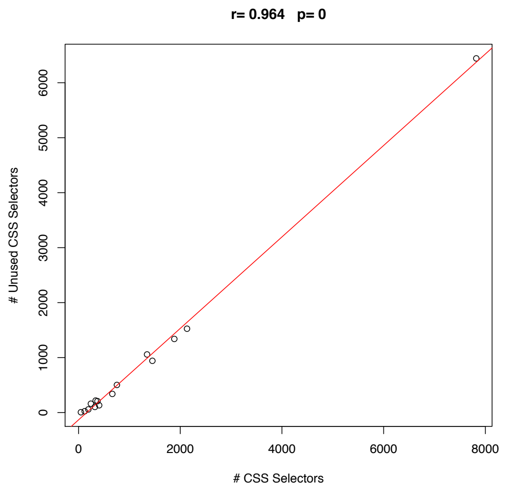


Figure 8. Scatter plot of the total number of CSS selectors versus the number of detected unused CSS selectors. r represents the Spearman correlation coefficient and p is the p-value.

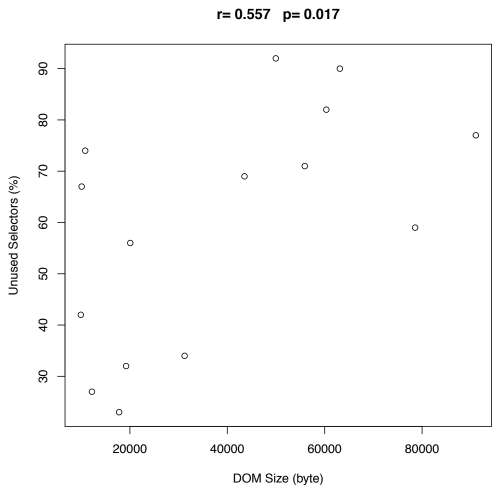


Figure 9. Scatter plot of the percentage of detected unused selectors versus the average DOM size. r represents the Spearman correlation coefficient and p is the p-value.

each selector, as well as all selectors affecting each element. This relational information can be valuable during program comprehension and refactoring tasks. Another interesting area where CILLA could support web developers in is cross-browser compatibility [35], [37] by providing an overview of how DOM elements are linked to CSS rules in different browsers.

Scalability and Performance. The results in Table II show that our approach is scalable to deployed industrial web applications consisting of tens of thousands of CSS LOC and hundreds of DOM states. On average, it took CILLA 22 minutes in total (including state exploration and CSS analysis) to analyze 7K lines of CSS code and 100 DOM states. The results indicate that both scalability and performance of CILLA are acceptable.

Threats to Validity. To minimize selection bias, seven of our fifteen experimental objects were chosen randomly. We acknowledge the fact that more studies are required to draw more general conclusions. However, we believe the systems are representative of CSS code present in deployed web applications.

For measuring the rate of false positives and false negatives, we randomly select a sample of the entire rule set from the CSS code and check them manually against the output. Manual checking of CSS rules is a time intensive task done by the authors of the paper. Therefore, we acknowledge that it could be error-prone and biased towards our judgment, although we made every effort to mitigate these threats. The main reason behind selecting samples instead of the entire sets has been constraining the effort and time needed to manually form a baseline, since there are thousands of selectors and declaration properties along with hundreds of DOM states in the systems, as shown in Table II. Increasing the sample size or selecting other samples could change the assessment, positively or negatively. As a remedy for this threat, we created a script that randomly selects the samples.

With respect to reliability of our results, CILLA and all the fifteen web-based systems are publicly available, making the case study replicable. One threat with using online web applications in empirical studies is that they might change over time, making exact replications challenging. That is why we have also included two open source systems in the study.

VIII. CONCLUDING REMARKS

This paper presents an automated approach for analyzing the relation between CSS rules and DOM elements of web applications. Our technique, implemented in a tool called CILLA, is capable of detecting unmatched and ineffective selectors and properties as well as undefined class values.

The results of our empirical evaluation, on several open-source and industrial web applications, clearly point to the ubiquity of the problem: On average we found 60% unused selectors and 52% unused properties. Our results also demonstrate the efficacy of the approach in automatically detecting unused CSS code (100% recall and 80-100% precision).

Our work can be enhanced and extended in several ways. Improving the algorithms and implementation to further reduce the number of false positives and conducting larger case studies to obtain more empirical data form part of our future work. Other directions we will pursue are exploration of suitable techniques for analyzing pseudo classes and elements. Further, we will investigate how the analysis results can be used for detecting antipatterns and 'smells' in CSS code and suggesting refactoring steps for improving code quality and maintainability.

ACKNOWLEDGMENT

This research was supported in part by an NSERC Discovery grant, and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

REFERENCES

[1] W3C, "CSS," http://www.w3.org/Style/CSS/.

[2] G. Badros, A. Borning, K. Marriott, and P. Stuckey, "Constraint cascading style sheets for the web," in *Proceedings of the 12th annual ACM symposium on User interface software and technology*. ACM, 1999, pp. 73–82.

[3] V. Quint and I. Vatton, "Editing with style," in *Proceedings of the ACM symposium on Document engineering*. ACM, 2007, pp. 151–160.

[4] M. Keller and M. Nussbaumer, "CSS code quality: A metric for abstractness; or why humans beat machines in CSS coding," in *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC '10)*. IEEE Computer Society, 2010, pp. 116–121.

[5] W3C, "Assigning property values, cascading, and inheritance," http://www.w3.org/TR/CSS2/cascade.html.

[6] C. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodik, "Parallelizing the web browser," in *Proceedings of the First USENIX conference on Hot topics in parallelism*. USENIX Association, 2009, pp. 7–12.

[7] C. Badea, M. Haghighat, A. Nicolau, and A. Veidenbaum, "Towards parallelizing the layout engine of firefox," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association, 2010, pp. 1–6.

[8] C. Stockwell, "IE8 what is coming," Oreilly, June 2008, http://en.oreilly.com/velocity2008/public/schedule/detail/3290.

[9] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. ACM, 2010, pp. 711–720.

[10] M. Burnett, "What is end-user software engineering and why does it matter?" in *Proceedings of the 2nd International Symposium on End-User Development (IS-EUD'09)*. Springer-Verlag, 2009, pp. 15–28.

[11] W3C, "Grammar of CSS 2.1," http://www.w3.org/TR/CSS21/grammar.html#grammar.

[12] ——, "Pseudo-elements and pseudo-classes," http://www.w3.org/TR/CSS2/selector.html#pseudo-elements.

[13] H. W. Lie, "Cascading style sheets," Ph.D. dissertation, University of Oslo, 2006.

[14] "SASS: Syntactically awesome stylesheets," http://sass-lang.com.

[15] W3C, "CSS Validator," http://jigsaw.w3.org/css-validator/.

[16] "CSSTidy," http://csstidy.sourceforge.net.

[17] "CSSClean," http://www.cleancss.com.

[18] N. Sullivan, "CSS lint," http://csslint.net.

[19] "Dust-Me Selectors," http://www.sitepoint.com/dustmeselectors/, version 2.2.

[20] "CSSess," https://github.com/driverdan/cssess, version 1.0.

[21] A. Aho, R. Sethi, and J. Ullman, "Compilers: principles, techniques, and tools," *Reading, MA,*, 1986.

[22] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, pp. 378–415, 2000.

[23] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical extraction techniques for Java," *ACM Trans. Program. Lang. Syst.*, vol. 24, pp. 625–666, 2002.

[24] E. Tempero, "An empirical study of unused design decisions in open source Java software," in *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, 2008, pp. 33–40.

[25] T. Copeland, *PMD applied*. Centennial Books, 2005.

[26] J. Spieler, "UCDetector," http://www.ucdetector.org.

[27] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, pp. 470–495, 2009.

[28] "CSS Parser," http://cssparser.sourceforge.net.

[29] W3C, "Document object model (DOM) level 2 style specification," http://www.w3.org/TR/DOM-Level-2-Style/.

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

[31] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 35–53, 2012.

[32] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[33] A. Kennedy and I. de Leon, *Pro CSS for High Traffic Websites*. Apress, 2011.

[34] Yahoo!, "Random URL generator," http://random.yahoo.com/bin/ryl.

[35] S. R. Choudhary, H. Versee, and A. Orso, "WebDiff: Automated identification of cross-browser issues in web applications," in *Proc. of the 26th IEEE Int. Conf. on Softw. Maintenance (ICSM'10)*, 2010, pp. 1–10.

[36] R. Gentleman and R. Ihaka, "The R project for statistical computing," http://www.r-project.org.

[37] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11)*. ACM, 2011, pp. 561–570.