Discovering Refactoring Opportunities in Cascading Style Sheets

Davood Mazinanian, Nikolaos Tsantalis
Computer Science and Software Engineering
Concordia University
Montreal, Canada
{d_mazina, tsantalis}@cse.concordia.ca

Ali Mesbah
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

ABSTRACT

Cascading Style Sheets (CSS) is a language used for describing the look and formatting of HTML documents. CSS has been widely adopted in web and mobile development practice, since it enables a clean separation of content from presentation. The language exhibits complex features, such as inheritance, cascading and specificity, which make CSS code hard to maintain. Therefore, it is important to find ways to improve the maintainability of CSS code. In this paper, we propose an automated approach to remove duplication in CSS code. More specifically, we have developed a technique that detects three types of CSS declaration duplication and recommends refactoring opportunities to eliminate those duplications. Our approach uses preconditions that ensure the application of a refactoring will preserve the original document styling. We evaluate our technique on 38 real-world web systems and 91 CSS files, in total. Our findings show that duplication in CSS code is widely prevalent. Additionally, there is a significant number of presentation-preserving refactoring opportunities that can reduce the size of the CSS files and increase the maintainability of the code.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Cascading style sheets, refactoring, duplication

1. INTRODUCTION

Cascading Style Sheets (CSS) [34] is a language for defining the presentation semantics of web documents. CSS is widely used in today's web development – over 90% of web developers use CSS [23] in 90% of the web sites [33]. CSS is also increasingly used in mobile app development through frameworks (e.g., PhoneGap, IBM Worklight) that generate hybrid mobile apps. As a result, CSS has become an important language with applications in many different domains.

While CSS has a relatively simple syntax, some of its complex features, such as inheritance, cascading, and specificity [15, 34], inherently make both the development and maintenance of CSS code cumbersome tasks [19] for developers. Furthermore, CSS development is far from a rigorous and disciplined process due to the lack of established design principles and effective tool support [9]. One instance of undisciplined development is the definition of new CSS rules by copying and modifying existing code instead of reusing already defined ones.

There is empirical evidence that duplicated code in software systems developed with procedural or object-oriented languages is associated with increased maintenance effort [16], higher error-proneness [11], and higher instability [22] in terms of change frequency and recency. We believe that the development and maintenance of CSS code is also subject to the same problems caused by code duplication. Moreover, the problem of duplication might even be more intense in CSS code, because the CSS language lacks many features available in other programming paradigms that could enable code reuse. For instance, there is no notion of *variables* and *functions* in CSS to build reusable blocks of code.

In addition, CSS code has to be transferred over the network from a server to many clients. Extensive code duplication increases the size of the transferred data, resulting in a large network load overhead. Once on the client side, the CSS code has to be processed by the web browser. Extensive code duplication increases the size of the CSS code that has to be processed by the browser, resulting in a computational overhead that might be significant taking into account the limited computation, memory, and power resources available in mobile devices. Previous studies [21] have shown that the visual layout of web pages, performed by analyzing the CSS code, consumes 40–70% of the average processing time of the browser.

In this paper, we propose an automated technique to (1) analyze and detect various types of CSS duplication, and (2) discover and recommend refactoring opportunities to eradicate duplicated CSS code. This work makes the following main contributions:

1. We define various types of duplication in CSS code and propose a technique for the detection of duplication instances.
2. We present a recommendation technique for refactoring opportunities that can eliminate CSS code duplication. Additionally, we provide a ranking mechanism based on

FSE '14, November 16–22, 2014, Hong Kong, China

the size reduction that can be potentially achieved by each suggested refactoring to help CSS developers prioritize their maintenance efforts by focusing on the refactorings with higher impact.
3. We describe preconditions that *preserve* the CSS styling after the application of a refactoring.
4. We perform an empirical study to assess the efficacy of our approach using 38 real-world web sites making use of 91 CSS files in total.

Our results show that the extent of duplication in CSS code is indeed very intense ranging from 40% to 90% for the vast majority of the examined CSS files. On average, we found 165 refactoring opportunities in the examined CSS files, out of which 62 could be applied by preserving the styling of the web pages. Finally, the average size reduction achieved by applying only presentation-preserving refactorings was 8%, while the highest reduction was 35%.

2. THE CSS LANGUAGE

CSS was one of the first standards approved by the World Wide Web Consortium (W3C) [34] along with the markup language HTML. CSS enables separation of concerns by disconnecting styling code from markup. Consequently, the same style sheet can be applied to different markups enabling the reuse of styling code. In this paper, we call the HTML pages on which the styles are applied as *target documents*. Style sheets can be written inside the target documents, or in separate external CSS files which are linked to the target documents. CSS files consist of a set of CSS rules, with a syntax shown in Figure 1.

```
selector {
    property_1: value_1;
    property_2: value_2;
    ...
    property_n: value_n[;]
}
```

Figure 1: CSS rules syntax.

The `selector` part in the CSS rule defines the elements of the target document on which the style rules should be applied. For example, a `P` selector selects every paragraph element `<p>` in the document. The `property: value` pair is called a *style declaration*. The declarations define style values for the specified properties of the element selected by the corresponding selector. For example, the declaration `color: blue;` will apply a blue color to the text of the selected element. There may be multiple style declarations in every selector.

2.1 CSS Selectors

It is possible to set an ID for elements in the target document, using the ID attribute. For example, the HTML tag `<div id="toolbar"/>` corresponds to a `div` element for which the ID is equal to `"toolbar"`. The `#toolbar` CSS selector, which is called an *ID selector*, can be used to select the element with this ID.

Also, a group of declarations could be defined as a *class* in CSS. It is then possible to apply the same class to many different elements thus avoiding the duplication of declarations. Figure 2 shows an example of a class selector.

There is also the possibility of *grouping* different selectors in CSS, using the "," character. For instance, if we want to

HTML	CSS

`<div class="class1">`
`  content`
`</div>`
`<span class="class1">`
`  content`
`</span>`
	`.class1 {`
	`  color: red;`
	`  font: 10pt tahoma;`
	`}`

Figure 2: Class selectors.

apply the same styles to all `h1` and `h2` HTML elements, we could use the `h1, h2 { /* declarations */ }` CSS rule.

We can also add *attribute conditions* to a selector. If selector `S` selects a set of elements S, a selector of the format `S[attr operator value]` selects the subset elements of S for which, in the target document, attribute `attr` is defined and set to a substring of `value`. The `operator` defines the condition for the substring [34]. For example, `a[target]` selects all `<a>` elements that have the `target` attribute set to any value, and `img[src $= ".png"]` selects all `<img>` elements that have the ".png" suffix in their `src` attribute value.

Pseudo classes can be used to filter the elements selected by a given selector. For example, `:not(div)` selects all elements except for `div` elements. There are also *structural pseudo classes*, such as `tr:nth-child(2n+1)` which selects every odd row in every table of the target document. *Pseudo-elements* create abstractions about elements in the target document, beyond those specified by the HTML standards [34]. The `p::first-line` selector, for example, selects only the first line of the text inside every `<p>` element.

Finally, we can *combine* various selectors to achieve more specific selectors, using different *combinators*. Assuming we have two selectors A and B, we can combine them as follows:

A B (descendant combinator) selects all elements selected by B, which are descendants of the elements selected by A.

A > B (child combinator) selects all elements selected by B, which are **direct** children of the elements selected by A.

A ~ B (general sibling combinator) selects all elements selected by B, which have an element selected by A as a sibling.

A + B (adjacent sibling combinator) selects all elements selected by B, which are directly preceded by a sibling element selected by A.

2.2 Inheritance, Cascading, and Specificity

Elements in the target document are organized in a hierarchical manner. For some specific element properties, CSS supports *inheritance* in values by taking advantage of this hierarchical structure. For example, if we apply `color: blue;` to the `<body>` element, all child elements of the `<body>` tag will be styled with the blue color.

It is possible to have different CSS rules that select the same HTML elements. If these selectors assign values to the same properties, the web browser can only apply one of the property values to the selected elements. In such cases, the web browser follows the *cascading origin rules*. Based on these rules, the *authored* style rules (i.e., the rules written by the web application developer and linked to the target document) will override the *user* style rules (i.e., the rules that the reader of a web site might have defined in the browser). Similarly, user style rules override the browser's default style rules [34]. In the case that the origin of the

selectors is the same (e.g., the conflicting selectors are in the same CSS file), the *specificity* determines the "winning" selector, i.e., the more specific selector has priority over the less specific ones [34]. If the origin and the specificity is the same for two selectors, the position of the selectors in the CSS file determines the winning selector (i.e., the last selector overrides the previous ones). It is also possible to add the `!important` annotation at the end of a declaration to allow for a selector to bypass the specificity rules and be always applied.

3. DUPLICATION IN CSS

Different types of duplication have been defined for procedural and object-oriented code [28] based on the textual or the functional similarity of two code fragments. The code duplication problem is expected to be more potent in CSS due to the lack of variables and functions that could be used to build reusable blocks of code. As a result, many common style declarations have to be repeated in multiple selectors.

3.1 Duplication Types

In this work, we focus on duplicated CSS declarations, which can only be avoided by changing the CSS code. Additionally, by eliminating this kind of duplication, we can reduce the size of the CSS code that has to transferred over the network and be maintained in the future. Therefore, we define three different types of duplication at the declaration level in this section.

Type I: *Declarations having lexically identical values for given properties.*

An extreme example of type I duplication can be seen in Figure 3, which is taken from the main CSS file of Gmail's inbox page. In this file, there are 23 declarations that are repeated in three selectors. Figure 3 shows only a subset of these declarations for two of the selectors.

```
.z-b-V {
    -webkit-box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    background-color: #fff;
    color: #404040;
    cursor: default;
    font-size: 11px;
    font-weight: bold;
    text-align: center;
    white-space: nowrap;
    border: 1px solid transparent;
    border-radius: 2px;
    ...
}
```

```
.z-b-G {
    -webkit-box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    background-color: #fff;
    color: #404040;
    cursor: default;
    font-size: 11px;
    font-weight: bold;
    text-align: center;
    white-space: nowrap;
    border: 1px solid transparent;
    border-radius: 2px;
    ...
}
```

Figure 3: Type I duplication in Gmail's CSS.

Note that the definition of type I duplication considers *only* the equality of the property values and disregards the value *order*. For instance, consider the two `border` declarations in Figure 3. If one of them was defined as `border: transparent solid 1px` (i.e., same values in different order), we would still consider them as an instance of type I duplication, because based on the CSS specification, the browser will interpret both declarations in the same way.

Type II: *Declarations having equivalent values for given properties.*

In CSS, we may have the same values for properties with alternative representations. Font size, color, length, angle and frequency values are representative cases. For instance, Table 1 shows alternative representations for the same color. We consider all these different representation values as *equivalent* values.

Table 1: Different representations for "Violet" color.

Representation	Value
HTML Named Color	Violet
Hexadecimal	#EE82EE
RGB	rgb(238, 130, 238)
RGBA	rgba(238, 130, 238, 1)
HSL	hsl(300, 76%, 72%)
HSLA	hsla(300, 76%, 72%, 1)

If two or more declarations have equivalent values for the same properties, we consider them as an instance of type II duplication. In Figure 5a, we can see an example of type II duplication in the CSS file of Gmail's inbox page. Note that the declarations with `color` property are duplicated.

In addition, there are some default values for certain properties, which are applied when explicit values are missing. For example, based on the CSS specification, the declaration `padding: 2px 4px 2px 4px;` can be also written in a shorter version as `padding: 2px 4px;` with the same effect. Such cases are also considered as equivalent declarations and thus constitute instances of type II duplication.

Type III: *A set of individual-property declarations is equivalent with a shorthand-property declaration.*

Some CSS properties, such as `margin`, `padding`, and `background` are called *shorthand properties*. With these properties, we can define values for *a set of properties* in a *single declaration*. For instance, the `margin` shorthand property could be used in order to define values for `margin-top`, `margin-right`, `margin-bottom` and `margin-left`, as shown in Figure 4.

```
margin: 3px 4px 2px 1px;  ⟺  { margin-top: 3px;
                               margin-right: 4px;
                               margin-bottom: 2px;
                               margin-left: 1px; }
```

Figure 4: Shorthand and individual declarations

If a selector contains a set of individual declarations, which is equivalent to a shorthand declaration of another selector, we consider those declarations as an instance of type III duplication. Figure 5b, shows an example of type III duplication in the CSS file of the Gmail's inbox page.

```
.fJ {
    ...
    color: white;
    ...
}
.Ik {
    ...
    color: #fff;
    ...
}
```

(a) Type II

```
.sG0wIf {
    border-bottom-color: #e5e5e5;
    border-bottom-style: solid;
    border-bottom-width: 1px
}

body .azewN {
    ...
    border-bottom: 1px solid #e5e5e5;
    ...
}
```

(b) Type III

Figure 5: Declaration duplication in Gmail's CSS

3.2 Eliminating Duplications

The aforementioned types of duplication can be eliminated directly in CSS code without changing the target documents by extracting a *grouping* selector. If a set D of declarations is duplicated (in the form of type I, II, III duplication) in a set of selectors $S_1, S_2, ..., S_n$, we can create a new selector for the group $S_1, S_2, ..., S_n$ and move D to this new selector. For instance, the CSS code snippet on the left side of Figure 6 could be refactored as shown on the right side of Figure 6.

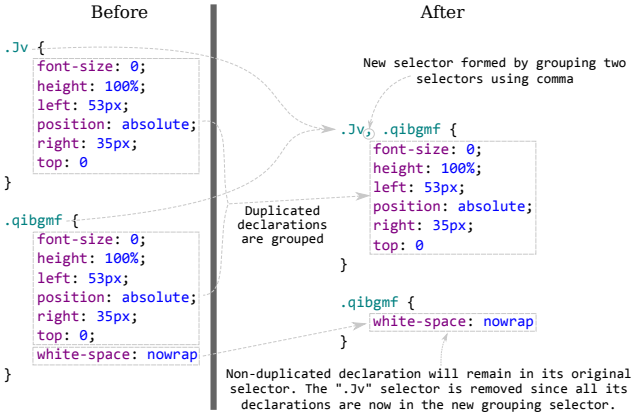


Figure 6: Grouping selectors to remove duplication.

Another possible solution, based again on grouping, is to create a common *class* for the repeated declarations and assign that class to the target elements. However, this solution requires also to update the target documents, so that they make use of the newly defined class.

4. METHOD

Our method for the detection of duplication in CSS and the extraction of refactoring opportunities is divided in four main steps discussed in the following subsections.

4.1 Abstract Model Generation

To look for duplications, we first need to parse all the CSS code of a given web application. In order to support all available CSS standards, we have adapted the W3C Flute Parser, so that it conforms with CSS3 specifications [34]. Our method then analyzes the code and generates an instance of the abstract model shown in Figure 7, which represents a high-level structure of the application's CSS code.

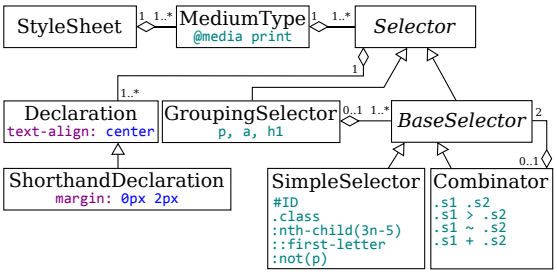


Figure 7: CSS Model

As shown in Figure 7, every style sheet may be bound to some *Medium type*. This identifies the target *presentation medium* for which the style sheet is defined. For instance, one may distinguish styles for printing and displaying in a mobile device [34]. It is also possible to define the same selectors for different media types within a given style sheet.

In this model, a `BaseSelector` represents selectors that do not perform grouping. A `SimpleSelector` represents *type selectors* or the *universal selector* (`*` selector, which selects every element). The class `SimpleSelector` has special attributes for specifying properties like the element ID, class identifier, pseudo class, pseudo element, and attribute conditions. Finally, a `Combinator` represents *combinator selectors*, which can be formed by combining two `BaseSelectors`. Examples of each of these selectors can be seen in Figure 7.

4.2 Preprocessing

The detection of Type I duplications does not require any preprocessing. However, to facilitate the detection of type II and III duplication instances, we perform three separate preprocessing steps.

Normalization of property values. In this step, we replace values that can be expressed in different formats or units (e.g., colors and dimensions) with a common *reference* format or unit. For example, every color in named, hexadecimal, or HSL format (see Table 1) is replaced with its equivalent `rgba()` value. Every dimension specified in centimeters, inches, or points is converted to its equivalent pixel value. All applied conversions are based on the guidelines provided by the CSS specification [32]. Replacing values with a common representation is known as *normalization* and has been used in traditional code clone detection techniques for finding clones with differences in identifiers, literals, and types [12, 29]. Our motivation is to find declarations using alternative formats or units for the same property value. Such cases constitute type II duplication instances.

Addition of missing default values. As we discussed in Section 3, CSS developers sometimes omit values for some of the multi-valued properties, in order to have shorter declarations. In this step, we have some predefined rules based on the CSS specification [32] that add the implied missing values to the properties in the model. For instance, `margin` property should normally have 4 values. We enrich the declaration `margin: 2px 4px` with the two missing implied values as `margin: 2px 4px 2px 4px`. This allows the comparison of declarations based on a complete set of explicit values enabling the detection of type II duplication instances.

Virtual shorthand declarations. Detecting type III duplication instances requires the comparison between shorthand declarations in one selector and an equivalent set of individual declarations in another selector. To facilitate this task, we add "virtual" shorthand declarations to the model. We examine the declarations of every selector to find sets of individual declarations that can be expressed as equivalent shorthand declarations. For every set of such individual declarations, we generate the corresponding shorthand declaration, and add it as a "virtual" declaration to the corresponding selector in the model. These virtual shorthand declarations will be compared with "real" shorthand declarations to detect type III duplication instances.

4.3 Duplication Detection

Duplication instances can be found by comparing every possible pair of declarations in the CSS model and checking

Algorithm 1: Detection of type I, II & III clones

```
Input  : A preprocessed style sheet styleSheet
Output : allClones including Type I, II & III clones
mediumTypes ← all medium types in the styleSheet
allClones ← ∅
foreach m ∈ mediumTypes do
    D ← all declarations in m
    clones_m ← ∅
    for i ← 1 to |D| do
        clone ← D_i
        for j ← i + 1 to |D| do
            if identical(D_i, D_j) ∨ equivalent(D_i, D_j) then
                clone ← clone ∪ D_j
            end
        end
        if |clone| > 1 then
            merged ← false
            foreach clone_k ∈ clones_m do
                if clone_k ∩ clone ≠ ∅ then
                    clone_k ← clone_k ∪ clone
                    merged ← true
                end
            end
            if not merged then
                clones_m ← clones_m ∪ clone
            end
        end
    end
    allClones ← allClones ∪ clones_m
end
```

whether they are equal (for type I) or equivalent (for type II and III). The notions of equality and equivalence were discussed in Section 3. Note that in our approach we consider the declarations present in the CSS model as they have been formed after the preprocessing steps to allow for the detection of type II and III duplication instances.

Our detection approach is summarized in Algorithm 1. The algorithm receives as input a preprocessed CSS style sheet and returns a set of *clones*, where each *clone* is a set of equal or equivalent declarations, $D = \{d_1, d_2, ..., d_n\}$, and each declaration belongs to a selector $s_i \in S = \{s_1, s_2, ..., s_n\}$ of the analyzed CSS style sheet.

Figure 8 depicts an example of a style sheet and the corresponding clones extracted from the application of Algorithm 1. The first two clones contain instances of type II duplication, while the last clone contains an instance of type III duplication.

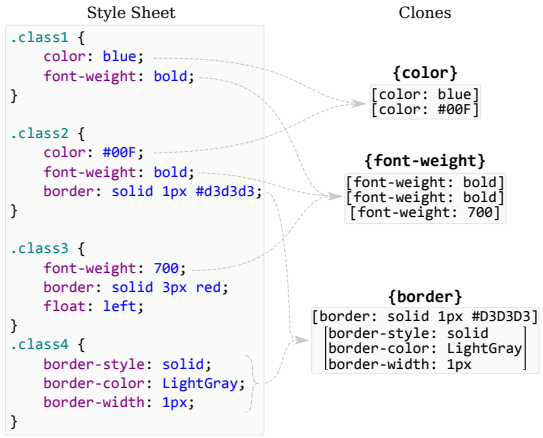


Figure 8: Clones extracted from a style sheet

4.4 Extracting Refactoring Opportunities

A *clone*, as defined in subsection 4.3, can be directly refactored by extracting a single declaration $d_i \in D$ to a new selector, which groups all selectors in S, and then removing all declarations in D from the original selectors they belong to. The refactored version of the CSS code will contain $|D| - 1$ less declarations, but should have exactly the same effect in terms of the styles applied to the selected elements. As such, the larger the clone (i.e., the cardinality of D), the more beneficial the corresponding refactoring is, since a larger number of declarations will be eliminated by the application of the refactoring.

The detected clones constitute the "building blocks" for extracting more advanced and higher impact refactoring opportunities. For instance, there may exist selectors that have multiple declarations in common (i.e., selectors involved in multiple clones). A set of common declarations shared among a group of selectors constitutes a *clone set*. In that case, all declarations in the clone set could be extracted into a single grouping selector (or a class selector) reducing significantly the repetition of declarations. Figure 6 in subsection 3.2 presents an example of such a case. In general, the more clones are common in a larger set of selectors, the higher the impact of the corresponding refactoring opportunity in the reduction of repeated declarations.

In this work, we use a data mining metaphor to extract clone sets as refactoring opportunities from the initially detected declaration-level clones. Let us assume that the style sheet is a transactional dataset, in which every selector s_i is a transaction, and the clones corresponding to the declarations of s_i are the *items* of transaction s_i. Based on this mapping, Figure 9 shows the resulting dataset for the style sheet of Figure 8. Note that the clones are sorted according to their size (i.e., the number of duplication occurrences).

Transactions (Selectors)	Items (Corresponding clones)
.class1	{font-weight}{color}
.class2	{font-weight}{color}{border}
.class3	{font-weight}
.class4	{border}

Figure 9: Dataset for the style sheet of Figure 8

In the data mining domain, a set of items which is repeated in different transactions is called an *itemset*. If an itemset is repeated in more than a certain number of transactions, which is called the *minimum support count*, the itemset is known to be *frequent*. Our goal is to extract all frequent itemsets with a minimum support equal to 2 (i.e., the minimum size for a duplication instance), because a frequent itemset in our case represents a clone set that is repeated in more than one selector. Therefore, every frequent itemset is a potential *grouping refactoring opportunity*.

In our method, we use the FP-Growth association rule mining algorithm [10], which finds the frequent itemsets using a structure called *frequent-pattern tree* (FP-Tree) [31]. The FP-Tree is essentially a compact representation of the dataset, since every itemset association within the dataset is mapped onto a path in the FP-Tree. Figure 10 displays the FP-Tree for the dataset of Figure 9.

The FP-Tree has a *header table*, which includes all distinct items that exist in the FP-Tree. The items in this table are sorted in descending order based on their support count. There is a link between every item in this table to the first occurrence of that item in the FP-Tree (rep-

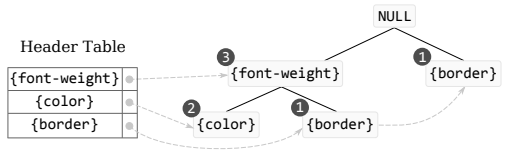


Figure 10: FP-Tree for the dataset of Figure 9

resented as a dotted arrow from the header table to the nodes of the FP-Tree in Figure 10). To enhance the traversal of an item in the header table to all nodes containing that item, nodes that contain the same item are also linked (e.g., the `border` nodes in the FP-Tree). The number next to a node represents the number of transactions (selectors in our case) involved in the portion of the path from this node to the root of the tree. For example, the path from node `color` to the root represents that there are two selectors that contain both items `color` and `font-weight` (i.e., selectors `.class1` and `.class2`). The path from node `border` nested under node `font-weight` to the root represents that there is only one selector that contains both items `border` and `font-weight` (i.e., selector `.class2`). Finally, the path from node `font-weight` to the root represents that there are three selectors that contain item `font-weight`.

Once the FP-Tree is constructed, the FP-Growth algorithm generates all frequent itemsets with the minimum support specified as input. Figure 11 shows all frequent itemsets (i.e., grouping refactoring opportunities) generated with a minimum support value equal to 2.

	Frequent Itemsets/ Refactoring Opportunities	Involved Selectors
1	[{border}]	.class2, .class4
2	[{color}]	.class1, .class2
3	[{color}, {font-weight}]	.class1, .class2
4	[{font-weight}]	.class1, .class2, .class3

Figure 11: Output of the FP-Growth algorithm for the style sheet of Figure 8

In the refactoring scheduling literature, two refactorings are considered as *conflicting* if they have a mutually exclusive relationship [18], i.e., the application of the first refactoring disables the application of the second refactoring and vice versa. Within the context of CSS, two refactoring opportunities are conflicting if their application affects a common subset of declarations. For instance, in Figure 11, if the last refactoring opportunity is applied, the third one becomes infeasible and vice versa, because these two refactoring opportunities affect two common `font-weight` declarations. In the same manner, the second and third refactoring opportunities are also conflicting, because they affect two common `color` declarations. However, in that case, the third refactoring opportunity *subsumes* the second one, since the set of declarations affected by the latter is a subset of the declarations affected by the former. Our approach filters out subsumed refactoring opportunities, if the ones subsuming them can be safely applied (Section 4.6). For the problem of conflicting refactoring opportunities, we provide a ranking mechanism explained in Section 4.5.

4.5 Ranking Refactoring Opportunities

Although a refactoring operation affects several quality aspects of the code, such as understandability, maintainability, and extensibility, in this work we focus on the size of the CSS code, because size is directly associated with the other aforementioned higher-level quality attributes (in general, a code with small size can be more easily maintained). Hence, in order to prioritize the refactoring opportunities and allow developers to focus on the most important ones, we define a ranking formula based on the number of characters that can be removed from the CSS code by applying a given refactoring opportunity.

Let RD_r be the set of duplicated declarations that will be removed from the style sheet by applying the refactoring opportunity r, S_r be the set of selectors that contain the duplicated declarations of set RD_r (i.e., the selectors that will be grouped after applying r), and AD_r be the set of declarations that will be added to the new grouping selector. It should be noted that AD_r contains the declaration with the *minimum* number of characters for each set of equal/equivalent declarations within RD_r. The size reduction (SR) achieved by refactoring opportunity r is calculated as follows:

$$SR(r) = \sum_{d \in RD_r} c(d) - \sum_{s \in S_r} c(s) - \sum_{d \in AD_r} c(d) \qquad (1)$$

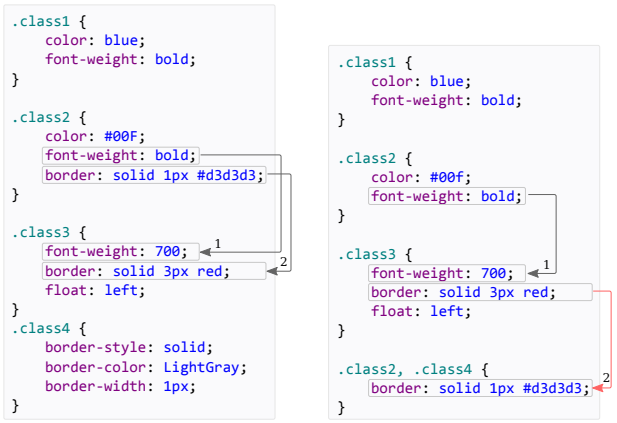
where the function c counts the number of characters of the declaration (or selector) passed as an argument. The higher the $SR(r)$ value, the higher the impact of r will be on reducing the size of the CSS code. A negative $SR(r)$ value indicates that the size of the CSS code will increase after the application of r. A negative value is possible if the textual size of the selectors being grouped is larger than the textual size of the declarations being removed. Of course, this would not be an issue if the duplicated declarations were placed under a newly defined class; however, this solution would require to update the target documents to make use of the new class. As mentioned before, in this work we aim to avoid modifications of the target document, i.e., all refactorings should be merely in the style sheets. Consequently, when size reduction is the objective, the refactoring opportunities should be applied in a descending order of SR value excluding those having a negative SR value.

Based on Equation 1, the size reduction values for the four refactoring opportunities shown in Figure 11 are 46, -3, 14, and 13 characters, respectively. The second refactoring opportunity would actually increase the size of the CSS code, if it was applied. The first refactoring opportunity corresponds to the highest size reduction, and the CSS code resulting after its application is shown in Figure 12b (the new grouping selector is appended to the end of the file).

4.6 Preserving Order Dependencies

Behavior preservation is a crucial property of refactoring [25]. The refactored program should have exactly the same functionality as the original program. Within the context of CSS, the notion of *behavior* corresponds to the *presentation* of the target documents (i.e., the property values that are eventually applied to the target document elements). Therefore, a refactoring can be considered as valid, if its application preserves the *presentation* of the target documents.

Let us assume that the CSS code of Figure 12a is applied to the target document shown in Figure 13a. As we can observe from Figure 13a, the second `div` element uses the style rules from both `.class2` and `.class3`. As we can see from Figure 12a, the declaration of the `border` property in `.class3` overrides the corresponding declaration in `.class2`



(a) (b)

Figure 12: (a) Order dependencies in the original CSS file, (b) Order dependencies after refactoring

and as a result, the second `div` element is styled with a red color border as shown in Figure 13b. Now, let us assume that the first refactoring opportunity shown in Figure 11 is applied to the CSS code of Figure 12a resulting in the CSS code of Figure 12b. In the refactored CSS code, the declaration of the `border` property in the extracted grouping selector `.class2,.class4` overrides the corresponding declaration in `.class3`. As a result, the second `div` element is no longer styled with a red color border as shown in Figure 13c, which is a clear indication that the applied refactoring did not preserve the presentation of the target document. This inconsistency is caused by the inversion of the original *overriding relationship* between selectors `.class2` and `.class3` after the application of the refactoring.

We define an *order dependency* from selector s_i containing declaration d_k to selector s_j containing declaration d_l due to property p, denoted as $\langle s_i, d_k \rangle \xrightarrow{p} \langle s_j, d_l \rangle$, iff:

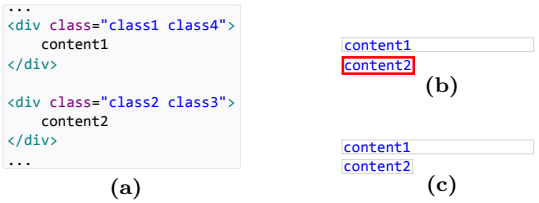
- *a*) selectors s_i and s_j select at least one common element having property p in the target document,
- *b*) declarations d_k and d_l set a value to property p and have the same importance (i.e., both or none of the declarations use the `!imporant` rule),
- *c*) declaration d_k precedes d_l in the style sheet,
- *d*) selectors s_i and s_j have the same specificity.

To ensure that the presentation of the target documents is preserved, we define the following *precondition*:

> *The extraction of a grouping selector should preserve all order dependencies among the selectors of the style sheet.*

The problem of finding an appropriate position for the extracted selector g in the style sheet can be expressed as a *Constraint Satisfaction Problem* (CSP) defined as:

Variables: the positions of the selectors involved in order dependencies including g.
Domains: the domain for each variable is the set of values $\{1, 2, ..., N+1\}$, where N is the number of selectors in the original style sheet.
Constraints: Assuming that g contains declarations for the set of properties P, an order constraint is created in the form of $pos(s_i) < pos(s_j)$ for every order dependency $\langle s_i, d_k \rangle \xrightarrow{p} \langle s_j, d_l \rangle$ where $p \in P$.



(a) (b) (c)

Figure 13: (a) Sample document, (b) Styling using the CSS code of Figure 12a, (c) Styling using the CSS code of Figure 12b

In the example of Figure 12a, the order dependencies are $\langle$`.class2, font-weight:bold`$\rangle \xrightarrow{font-weight} \langle$`.class3, font-weight:700`$\rangle$ $\langle$`.class2, border:solid 1px #d3d3d3`$\rangle \xrightarrow{border} \langle$`.class3, border:solid 3px red`$\rangle$ and we extract the constraint $pos($`.class2`$) < pos($`.class3`$)$. Based on this constraint, the extracted grouping selector `.class2,.class4` should be placed at any position before the selector `.class3` (i.e., `.class3` should be the last selector in the style sheet after refactoring) in order to preserve the presentation of that target document in Figure 13a.

If we assume that there is an additional order dependency from selector `.class3` to `.class4` due to property `border`, then the CSP would be unsatisfiable due to the new constraint $pos($`.class3`$) < pos($`.class4`$)$. In that case, the extracted selector `.class2,.class4` has to be placed before `.class3` to satisfy the first constraint and after `.class3` to satisfy the second constraint, and thus there is no solution satisfying both constraints. Refactoring opportunities leading to an unsatisfiable CSP violate the defined precondition, and therefore are excluded as non presentation-preserving.

5. EVALUATION

To assess the efficacy of our approach, we conducted a case study addressing the following research questions:

RQ1: What is the extent of declaration-level duplication in CSS files?
RQ2: What is the number of refactoring opportunities that can be potentially applied in CSS files and how many of them are actually presentation-preserving?
RQ3: What is the size reduction we can achieve by applying presentation-preserving refactorings in CSS files?

Our tool and empirical data are all available online.[1]

5.1 Experiment Design

Selection of subjects. In total, our study contains 38 subjects. In order to select representative real-world web applications, we adopted the web-systems included in the study conducted by Mesbah and Mirshokraie [19], in which they investigated the presence of unused CSS code. The list includes 15 (in total) open-source, randomly selected, and author selected online web applications. We included 14 subject systems from that list (one of them is not available online anymore). We extended the list with 24 more subjects including web applications developed by companies considered leaders in web technologies, such as Facebook, Yahoo!, Google, and Microsoft, in addition to a subset of the top-100 visited web sites based on Alexa ranking. The complete list of the selected systems is shown in Table 2. Figure 14 shows the size characteristics of the CSS code, selectors, and declarations of the subjects included in our study.

[1] `http://goo.gl/q2ZvlB`

Table 2: Selected subjects

ID	Web app.	# CSS files	ID	Web app.	# CSS files
1	Facebook	6	20	Pinteerst	2
2	YouTube	4	21	Reddit	1
3	Twitter	2	22	Tumblr.com	2
4	YahooMail	3	23	Wordpress.org	1
5	Outlook.com	6	24	Vimeo.com	3
6	Gmail	5	25	Igloo	2
7	Github	2	26	Phormer	1
8	Amazon.ca	3	27	BeckerElectric	1
9	Ebay	2	28	Equus	1
10	About.com	1	29	ProToolsExpress	1
11	Alibaba	3	30	UniqueVanities	3
12	Apple.ca	3	31	ICSE12	3
13	BBC	3	32	EmployeeSolutions	3
14	CNN	1	33	SyncCreative	3
15	Craiglist	1	34	GlobalTVBC	5
16	Imgur	2	35	Lenovo	1
17	Microsoft	1	36	MountainEquip	2
18	MSN	1	37	Staples	2
19	Paypal	1	38	MSNWeather	3

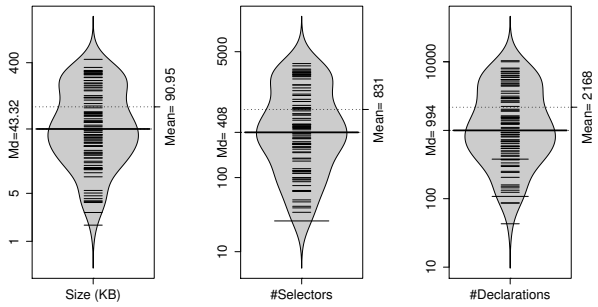


Figure 14: Characteristics of the analyzed CSS files

Extraction of CSS styles and DOM states. CSS styles can be directly embedded in the web documents, linked to web pages as external files, or dynamically generated at runtime through JavaScript code. For our experiments, we focus on the external CSS files, since the refactoring of the other sources of CSS code requires the modification of other web artifacts (such as HTML documents), which is not the focus of this paper. We take advantage of the dynamic analysis features provided by CRAWLJAX [20] and developed an external CSS file extractor plug-in. Additionally, we use CRAWLJAX to dynamically capture different DOM tree instances (i.e., DOM states) from the examined web applications and use them for the extraction of order dependencies between the CSS selectors.

Detection of presentation-preserving refactorings. In order to collect the set of presentation-preserving refactorings that can be safely applied on a CSS file f styling the set of DOM states S collected from a web application, we:

1. Extract the order dependencies between the selectors of f by analyzing the DOM states in S, as described in Section 4.6.
2. Extract the set of refactoring opportunities R that can be potentially applied to f.
3. Sort R based on size reduction (Formula 1) and remove the refactoring opportunities having a negative value.
4. Iterate through the elements of R and apply the first refactoring opportunity for which the CSP defined in Section 4.6 is satisfiable.
5. If step 4 results in the application of a refactoring, repeat steps 1-5 with the refactored CSS file f'.

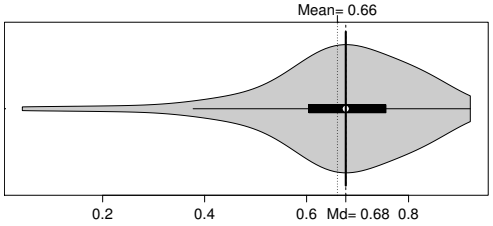
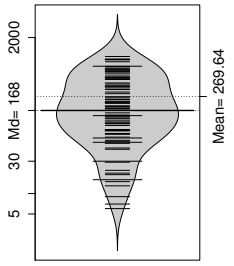


Figure 15: Percentage of duplicated declarations

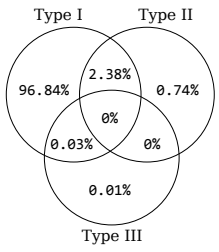
5.2 Results

Extent of duplication in CSS declarations (RQ1). The results of our empirical study confirm the expectation that duplication is more extensive in CSS code compared to procedural and object-oriented code (with 5–20% duplicated code [28]). Figure 15 displays a violin plot with the percentage of the declarations that are involved in at least one clone (i.e., they are at least once duplicated) in the analyzed style sheets. The median value for the percentage of duplicated declarations is 68%, while the average is 66%. The vast majority of the examined style sheets exhibits a duplication ranging from 40% to 90%. Note that in the reported results we have set the minimum support (i.e., the minimum number of selectors that should share a common declaration) to the lowest possible value (equal to 2); setting a larger minimum support value would lead to lower duplication rates.

Figure 16a shows the number of clones detected in the analyzed CSS files. On average, there are 270 distinct declarations being repeated more than once in the examined style sheets that could be used as building blocks for extracting more advanced refactoring opportunities. The Venn diagram shown in Figure 16b displays the percentage of the clones including different combinations of the duplication types defined in Section 3. As it can be observed, 97% of the clones include only type I duplication instances, while 2% of the clones include a combination of type I and II duplication instances. Furthermore, the existence of type III duplication instances within the clones is very rare.



(a) Total number of detected clones



(b) Duplication types in the detected clones

Figure 16: Statistics for the detected clones

Refactoring opportunities in CSS (RQ2). Figure 17 shows on top, a bean plot of the number of refactoring opportunities that were initially extracted from the original CSS files, excluding refactoring opportunities being subsumed and/or having a negative size reduction value. On the bottom of Figure 17, we can see a bean plot of the number of presentation-preserving refactorings, which we actually applied on the CSS files. As it can be observed, our approach

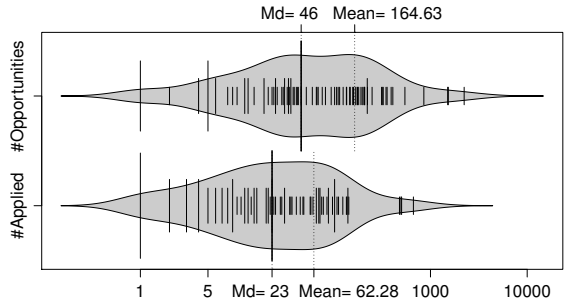
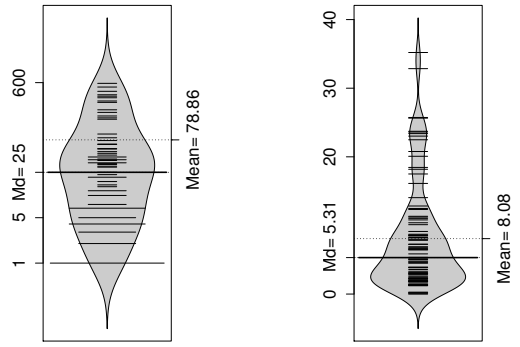


Figure 17: Initial refactoring opportunities vs. applied presentation-preserving refactorings

was able to detect, on average, 165 refactoring opportunities in the original version of the examined CSS files, while the average number of presentation-preserving refactorings was 62. Additionally, we found out that the examined CSS files had 79 order dependencies on average between their selectors, as shown in Figure 18a.



(a) #Order dependencies **(b) %Size reduction**

Figure 18: Order dependencies and size reduction

Size reduction (RQ3). In Figure 18b, we have depicted a bean plot with the percentage of the size reduction achieved by applying *only* presentation-preserving refactorings. In the examined CSS files, the average size reduction was 8%, while the maximum achieved value was 35%. Overall, in 12% of the examined CSS files (11 out of 91) the size reduction was over 20%, while in 27% (25 out of 91) the size reduction was over 10%.

In order to determine the factors that influence the applicability of refactorings in the examined CSS files, we decided to build a statistical regression model. Regression models are mostly used for the purpose of prediction, where the values of one or more *predictor variables* can be used to predict the value for the *response variable*. However, a multiple linear regression model can be also used to assess the impact of one predictor on the response variable, while controlling the other predictors [8]. Using regression, we estimate a coefficient for each predictor, which shows the magnitude and direction of the effect of the predictor on the response variable. We built a model with the *number of applied refactorings* as the response variable, and *size* and *number of order dependencies* as predictors. Intuitively, we expect a positive relationship between the number of applied refactorings and the size of the CSS files, since larger files exhibit more duplication and thus offer more opportunities for refactoring. On the other hand, we expect a negative relationship between the number of applied refactorings and the number of order dependencies detected for a given CSS file, since a larger number of order dependencies implies a higher probability for a precondition violation and thus rejecting a candidate refactoring opportunity. To this end, we created a generalized linear model using the *Poisson* distribution function [35].

Table 3: Statistical model's estimated parameters

Parameter	Estimate	p-value
Intercept*	2.989	<2e-16
Size coefficient	8.149e-03	<2e-16
#Order dependencies coefficient	-1.195e-03	<2e-16

* The intercept is the constant term in the regression model, which makes the residuals have a mean of zero.

As it is shown in Table 3, all estimated coefficients are statistically significant, and as we expected, the coefficient for the size of the CSS files is positive, while the coefficient for the number of order dependencies is negative. From this result, we can conclude that for CSS files with a similar size, the number of applicable refactorings decreases as the number of order dependencies increases. Additionally, we can conclude that our approach is more effective in terms of size reduction for large CSS files with a limited number of order dependencies.

5.3 Discussion

CSS duplication and refactoring opportunities. Our case study shows that CSS code duplication is prevalent in today's web systems. The majority of the clones we found pertain to type I duplication instances, and type II and III duplications are relatively less common. This indicates that developers use the same representation for property values consistently throughout their style sheets. Additionally, they make use of shorthand-property declarations consistently within different selectors. The results of our evaluation also show that our method is able to successfully detect many CSS refactoring opportunities that remove duplications and preserve the initial presentation of the target documents. These refactorings, when applied, allow for a much cleaner CSS code and considerable size reduction.

Size reduction. In our method, we focus on refactoring opportunities that extract the same set of equivalent declarations in a grouping selector. An alternative approach would be to extract and group the declarations in a new *class* selector, instead of a grouping selector. By selecting an appropriate name for the class selector, we can reduce even further the size of the CSS file, i.e., by replacing a set of selector names with a single class name, and at the same time improve its understandability (the class name could represent a common concept being extracted). However, this approach requires to make use of the new class in the DOM elements of the target document. From the refactoring point of view, this approach should update the corresponding HTML documents for static web sites, or even the source code that generates the HTML elements for dynamic web sites.

Limitations. In our current implementation, we have only considered CSS files linked to the HTML documents. In order to provide complete CSS refactoring support, in the future, we will also include in our analysis CSS styles embedded inside the `<style>` tags of the web pages as well as dynamically generated CSS styles through JavaScript.

Threats to the validity. A threat to the *internal validity* is that the DOM states collected from each web application may be insufficient to extract all possible order dependencies between the selectors of the examined CSS files, since for some dynamic web applications the number of DOM states is practically infinite. Missing order dependencies from unvisited DOM states could make some of the applied refactoring opportunities to be non presentation-preserving for this particular set of unvisited DOM states.

To avoid selection bias, we selected 14 subjects from the list of web sites analyzed in a related CSS study[19]. To mitigate threats to the *external validity* and make the results of the experiment as generalizable as possible, we included 24 additional web sites developed by leading companies in web technologies applying the current state-of-the-art CSS development practices. Finally, the developed tool and the collected data are all available online[1] to enable the replication of the experiment by other researchers.

6. RELATED WORK

Despite the fact that CSS is widely used in practice, it has not received much research attention. Mesbah and Mirshokrae [19] propose a technique to automatically detect unused and ineffective CSS code. Having a similar goal, Geneves et al. [9] use tree logics to detect unused CSS code. Keller and Nussbaumer [13] conclude that human-written CSS code has a higher abstractness (i.e., higher reusability) compared to generated code. To the best of our knowledge, our work is the first to analyze CSS with respect to code duplication and provide refactoring opportunities.

Several researchers have developed techniques for the detection of duplication in web artifacts. Most of the studies in this area have focused on the detection of duplicated content in web pages [1], or finding web pages with similar structure [6, 7]. Boldyreff et al. [1] replace the content of the web pages (i.e., the text inside different tags) with hash values and compared them to find duplicated content in web pages. Lanubile and Mallardo [14] propose a semi-automatic approach to find *function clones* in the source code of web applications. Their approach first compares the names of the functions written in either JavaScript or VBScript. If the names are the same, they compute various size metrics and report the functions with similar metric values as candidate clones. In their follow-up work, they evaluated this approach on four web applications and found out that the 21% to 80% of functions were duplicated and could be refactored [2].

De Lucia et al. [5] use the Levenshtein edit distance to quantify the structural similarity between different web pages. Rajapakse and Jarzabek [26] use CCFinder, a clone detection tool which finds the clones by applying token-to-token comparison [12], to find code clones in the source code of web applications written in various languages. They examined 17 web applications and found out a duplication rate of 17% to 63%. Synytskyy et al. [30] use an *island grammar* in order to define smaller portions of the HTML syntax for elements, such as forms and tables, that might be cloned across different pages. The grammar is used to extract those structures from web pages and examine whether their structure is repeated in other pages.

Cordy et al. [3] propose an approach that is language-independent and can detect exact and near-miss clones using island grammar extraction, pretty-printing and textual differencing of the clone candidates. In their study, they implemented this approach for HTML. This work led to the introduction of NiCad [29], which is an exact and near-miss clone detector. Later, researchers used [24] NiCad to detect clones and to find clone patterns in the PHP code of two industrial systems.

The extracted duplication information can be used to re-engineer web applications, i.e., create dynamic pages from static ones [1, 30], generate more-generalized dynamic web pages to minimize the duplication [4, 27], or find similar functionalities across different web pages [5]. None of the aforementioned works investigated the existence of duplication in CSS code, or developed a technique specialized on the detection of duplication in CSS code.

The most closely related work to this paper, regarding the detection of duplication in CSS code, is the approach proposed by Mao et al. [17] on the automatic migration from table-based structure to the style-based structure for web pages. In their work, they used traditional clone detection approaches in order to find the duplicated code across different CSS files and remove the duplications by creating a single CSS file, which could be applied to different web pages. Although this kind of detection and analysis might be sufficient for finding type I duplications, it cannot find type II and III duplication instances, as defined in this work.

7. CONCLUSIONS AND FUTURE WORK

In this work, we developed a technique for the detection of refactoring opportunities that can eliminate duplicated CSS declarations safely, i.e., without side-effects in the styling of the web documents. We performed an experiment on 38 real web applications and found that (1) code duplication is extensive in CSS files; on average 66% of the style declarations are repeated at least once, (2) there is a significant number of presentation-preserving refactoring opportunities (62 on average) that is associated positively with the size of the CSS files and negatively with the number of order dependencies between the selectors of the CSS files, and (3) on average a 8% reduction in the size of the examined CSS files can be achieved by applying the detected refactoring opportunities.

As future work, we plan to investigate the refactoring of CSS code into CSS Preprocessor code. CSS preprocessors, such as Sass[2] and Less[3], allow the definition of variables and functions to build reusable and parameterizable blocks of code, adding essentially features of procedural programming languages. Eventually, the preprocessor code is processed to generate pure CSS code. The research goal is to find repeated CSS styles and refactor them into functions by parameterizing possible differences in property values.

8. ACKNOWLEDGMENTS

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds de Recherche du Québec – Nature et Technologies (FRQNT), and the Faculty of Engineering and Computer Science at Concordia University.

[2] `http://sass-lang.com/`
[3] `http://lesscss.org/`

9. REFERENCES

[1] C. Boldyreff and R. Kewish. Reverse engineering to achieve maintainable WWW sites. In *Proc. of the 8th Working Conf. on Reverse Engineering*, pages 249–257, 2001.

[2] F. Calefato, F. Lanubile, and T. Mallardo. Function clone detection in web applications: a semiautomated approach. *Journal of Web Engineering*, 3(1):3–21, 2004.

[3] J. R. Cordy and T. R. Dean. Practical language-independent detection of near-miss clones. In *Proc. of the 14th Conf. of the Centre for Advanced Studies on Collaborative Research*, pages 1–12, 2004.

[4] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Reengineering web applications based on cloned pattern analysis. In *Proc. of 12th IEEE Int'l Workshop on Program Comprehension*, pages 132–141. IEEE, 2004.

[5] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Understanding cloned patterns in web applications. In *Proc. of the 13th Int'l Workshop on Program Comprehension*, pages 333–336. IEEE, 2005.

[6] G. Di Lucca and M. Di Penta. Clone analysis in the web era: an approach to identify cloned web pages. In *Proc. of the 7th IEEE Workshop on Empirical Studies of Software Maintenance*, pages 107–113, 2001.

[7] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *Proc. of the 26th Annual Int'l Computer Software and Applications Conf.*, pages 481–486, 2002.

[8] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for research*. Wiley-Interscience, 3rd edition, 2004.

[9] P. Geneves, N. Layaida, and V. Quint. On the analysis of cascading style sheets. In *Proc. of the 21st Int'l Conf. on World Wide Web*, pages 809–818, 2012.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate gention. *SIGMOD Rec.*, 29(2):1–12, 2000.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of the 31st Int'l Conf. on Software Engineering*, pages 485–495, 2009.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, 28(7):654–670, 2002.

[13] M. Keller and M. Nussbaumer. CSS code quality: a metric for abstractness; or why humans beat machines in CSS coding. In *Proc. of the 7th Int'l Conf. on the Quality of Information and Communications Technology*, pages 116–121, 2010.

[14] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *Proc. of the 7th European Conf. on Software Maintenance and Reengineering*, pages 379–386, 2003.

[15] H. W. Lie. *Cascading Style Sheets*. Ph.D. Thesis, University of Oslo, Norway, 2005.

[16] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. of the 24th IEEE Int'l Conf. on Software Maintenance*, pages 227–236, 2008.

[17] A. Y. Mao, J. R. Cordy, and T. R. Dean. Automated conversion of table-based websites to structured stylesheets using table recognition and clone detection. *Proc. of the 2007 Conf. of the center for advanced studies on Collaborative research*, pages 12–26, 2007.

[18] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Softw. and Sys. Modeling*, 6(3):269–285, 2007.

[19] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proc. of the Int'l Conf. on Software Engineering*, pages 408–418, 2012.

[20] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. on the Web*, 6(1):3:1–3:30, 2012.

[21] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proc. of the 19th Int'l Conf. on World Wide Web*, pages 711–720, 2010.

[22] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on clone stability. *Applied Computing Review*, 12(3):20–36, Sept. 2012.

[23] Mozilla Developer Newtork. Web developer survey research. Technical report, Mozilla, 2010.

[24] T. Muhammad, M. F. Zibran, Y. Yamamoto, and C. K. Roy. Near-miss clone patterns in web applications: An empirical study with industrial systems. In *Proc. of the 26th IEEE Canadian Conf. on Electrical and Computer Engineering*, pages 1–6, 2013.

[25] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[26] D. C. Rajapakse and S. Jarzabek. An investigation of cloning in web applications. In *Proc. of the 5th Int'l Conf. of Web Engineering*, pages 252–262, 2005.

[27] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proc. of the 29th Int'l Conf. on Software Engineering*, pages 116–126, 2007.

[28] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queens University, TR, Kingston, Canada, 2007.

[29] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. of the 16th IEEE Int'l Conf. on Program Comprehension*, pages 172–181, 2008.

[30] N. Synytskyy, J. R. Cordy, and T. R. Dean. Resolution of static clones in dynamic Web pages. In *Proc. of the 5th IEEE Int'l Workshop on Web Site Evolution*, pages 49–56, 2003.

[31] P. Tan, M. Steinbach, and V. Kumar. *Introduction to data mining*. Addison-Wesley, 2005.

[32] W3Schools.com. CSS reference. `http://www.w3schools.com/cssref/`.

[33] Web Technology Surveys. Usage of css for websites. `http://w3techs.com/technologies/details/ce-css/all/all`.

[34] World Wide Web Consortium. CSS specifications. `http://www.w3.org/Style/CSS/current-work`.

[35] A. Zeileis, C. Kleiber, and S. Jackman. Regression models for count data in R. *Journal of Statistical Software*, 27(8):1–25, 7 2008.