Automated Extraction of Mixins in Cascading Style Sheets

Alan Charpentier, Jean-Rémy Falleri, Laurent Réveillère

Automated Extraction of Mixins in Cascading Style Sheets

Alan Charpentier, Jean-Rémy Falleri and Laurent Réveillère
University of Bordeaux
LaBRI, UMR 5800
F-33400, Talence, France
Email: {acharpen,falleri,reveillere}@labri.fr

***Abstract*—Cascading style sheets (CSS) is a language that describes the presentation of web documents. CSS is widely adopted in web development and it is now common for web projects to have several thousands of CSS lines of code. Because the language lacks advanced features to allow code reuse, several languages such as Sass and Less have emerged as extensions to CSS. They provide mechanisms such as mixins to enable reuse. However, when a developer wants to migrate her web project from CSS to one of these extension languages, identifying mixins is a challenging task. In this paper, we describe an automated approach to extract mixins from CSS code. We have developed a tool that identifies mixins in CSS files and automatically generates Sass code. Our technique enables a fine-grained control on the generated code tailored to developer needs. We evaluate our approach on more than a hundred CSS files and conduct several case studies to assess its real-world relevance.**

***Index Terms*—CSS; code duplication; mixin.**

I. Introduction

Cascading Style Sheets (CSS) is a language mainly used for describing the presentation of HTML documents. Along with JavaScript, CSS is a cornerstone technology used to develop web applications. The massive adoption of the language has led to a significant increase of the average size of CSS code. Indeed, nowadays, web applications often contain several thousands of lines of CSS code, and consequently a large amount of duplication [1]. Duplicated code in software systems is known to complicate code maintenance and evolution [2], [3], as fault fixes and changes must be propagated in multiple places. Therefore software developers tend to keep the amount of duplicated code as low as possible.

However, in contrast to most programming languages CSS does not provide advanced mechanisms such as variables or functions to allow code reuse. To alleviate this lack, several CSS preprocessors such as Sass[1] and Less[2] have recently emerged on top of CSS. They provide advanced mechanisms such as variables and *mixins* to ease the development and maintenance of CSS code. Developers using these languages invoke a compiler to generate the corresponding low level CSS code that can be embedded into their application.

Because they facilitate code reuse, CSS preprocessors are increasingly being used by developers of large well known projects such as *Bootstrap*[3] and *Foundation*[4], the two most popular CSS frameworks. However, many applications still rely on low level CSS code. A recent survey[5] with more than $13,000$ responses from web developers showed that almost half of them do not use CSS preprocessors.

In this paper, we argue that this low adoption of CSS preprocessors in practice is partially due to the complexity of migrating legacy CSS code and to the lack of tools to support it. To assess our claim, we have conducted an informal survey. We retrieved the list of trending repositories of the month for the CSS language from GitHub[6]. We used data of September 2015 to build a list of 43 repositories, and we retrieved 500 developers from these repositories. We sent a mail to each developer seeking its opinion on the usefulness of an automatic tool to extract mixins from CSS code. As a result, 64 developers answered anonymously our survey[7], leading to a response rate of 12.8%. We noticed that about half of the participants were interested by such a tool. From the developers that used both CSS and preprocessors, 65% were mostly interested.

Overall, this paper makes the following contributions:

1) We introduce an automated approach that identifies mixins from CSS code. We have implemented the approach as a piece of open source software called *Mocss*. Our approach enables a fine-grained control on the generated code tailored to developer needs.
2) We assess the efficiency of our approach with four case studies, demonstrating that our tool is able to help developers removing code duplication by extracting mixins from legacy CSS code.

The structure of this paper is as follows. Section II introduces the CSS language and its extensions. Our approach for the extraction of mixins from CSS code is described in Section III and an evaluation of its efficiency and real-world relevance is reported in Section IV. The related work are discussed in Section V. We conclude and mention future work in Section VI.

[1]http://sass-lang.com/
[2]http://lesscss.org/
[3]http://getbootstrap.coms/
[4]http://foundation.zurb.com/
[5]https://css-tricks.com/poll-results-popularity-of-css-preprocessors
[6]https://github.com/trending?l=css&since=monthly
[7]Results of the whole survey are available online: http://www.labri.fr/perso/acharpen/icsme16/materials.zip

II. Background

Cascading style sheets is a language used to tune the visual style of web documents. CSS enables the separation of document content and presentation. This separation of concerns eases the reuse of presentation code in different documents. In the remainder of this section, we first describe the mechanisms offered by CSS for describing the presentation of a document. Then, we discuss the reasons behind the presence of duplication in CSS.

A. The CSS Language

CSS code consists of a list of style rules. Each rule consists of one or more selectors and a list of declarations. Listing 1 shows the syntax of a CSS rule.

Listing 1: Syntax of a CSS rule.

```
selector_1, ..., selector_n {
  property_1: value_1;
  ...
  property_n: value_n;
}
```

Selectors specify the set of elements of a document on which a style applies to. There are three ways to identify an element in the document. First, an element can be selected by its name. For example, `h1` selects all `h1` elements in the HTML document. Second, elements can be selected based on attributes they define, such as a unique identifier (of the form `#id1`) or a class that groups multiple elements (of the form `.class1`). Third, elements can be selected depending on their relative position in the document tree. For example, `div > p` selects all `p` elements that are immediate children of a `div` element. In addition, selectors can be combined to obtain more specific selections. In that case, the order of the selectors is important. For example, `p .class1` selects all elements of class `class1` that are inside `p` elements, whereas `.class1 p` selects all `p` elements that are inside elements of class `class1`.

Declarations of a rule define the style of elements selected by this rule. A declaration consists of a property and a value. Each property has a finite set of possible values defined in the CSS specification.

Different style rules can select the same elements and apply values to the same properties. When several rules define conflicting properties to a same element, the selector with the highest specificity[8] wins. Last, if several selectors have the same specificity, the last selector defined in the CSS file wins.

The order of style rules in a CSS file therefore has an impact on the rendering of HTML documents, and thus is part of the semantics of the CSS file. Consider the HTML document in listing 2 that contains a `div` block using style rules from both classes `class1` and `class2`, and listings 3 and 4 that report two identical CSS files with the exception of the order of the rules. The use of one or another CSS file will impact the rendering of the HTML document. In the case that the first (resp. the second) CSS file is linked to the HTML document, ~~`content`~~ (resp. `content`) will be displayed.

[8]http://www.w3.org/TR/selectors/#specificity

Listing 2: A `div` block using style rules from two classes.

```
<div class="class1 class2">content</div>
```

Listing 3: A sample CSS file.

```
.class1 { text-decoration: underline; }
.class2 { text-decoration: line-through; }
```

Listing 4: Another sample CSS file.

```
.class2 { text-decoration: line-through; }
.class1 { text-decoration: underline; }
```

B. Handling Duplication in CSS

Duplication can arise at different levels in CSS code. Listing 5 shows examples of duplication at the property level (`font-size`), at the value level (`red`) and at the declaration level (`margin:0`). CSS does not offer any mechanism to avoid duplication at the property and value levels. Eliminating duplication at the declaration level is usually done by grouping selectors, as shown in Listing 6. However, this mechanism may reduce code readability. Indeed, declarations of a particular selector might be placed in multiple rules, and the number of selectors for a particular rule might be very high. Another way to eliminate duplication at the declaration level is to introduce a new class instead of grouping selectors. This solution implies to modify HTML documents. Thus, the lack of advanced mechanisms to handle duplication has led to the emergence of CSS extensions languages.

Listing 5: Property, value and declaration duplication.

```
#id1 {
  font-size: 1.1em;
  background: red;
  margin: 0;
}

#id2 {
  font-size: 1.2em;
  color: red;
  margin: 0;
}
```

Listing 6: Eliminating duplication by grouping selectors.

```
#id1, #id2 {
  margin: 0;
}
#id1 {
  font-size: 1.1em;
  background: red;
}
#id2 {
  font-size: 1.2em;
  color: red;
}
```

Several languages have emerged to extend CSS by providing advanced mechanisms that are not available to the last CSS specification (CSS3). The most well-known extension languages are Sass, Less and Stylus[9]. Programs written in these languages are compiled into low level CSS code. They support variables to avoid duplication at the value level. In this paper, we focus on duplication at the property and declaration levels. Extension languages provide the notion of mixins to avoid these kinds of duplication. A mixin is similar to a macro function and can be parametrized by variables. An example of the use of mixins in Sass is depicted in Listing 7 whereas Listing 8 shows the corresponding generated CSS code.

[9]https://learnboost.github.io/stylus

Listing 7: A sample mixin in Sass factorizing two declarations.

```
@mixin config($s) {
  font-size: $s;
  margin: 0;
}
#id1 {
  @include config(1.1em);
  background: red;
}
#id2 {
  @include config(1.1em);
  color: red;
}
```

Listing 8: Generated CSS code from the Sass code in Listing 7.

```
#id1 {
  font-size: 1.1em;
  margin: 0;
  background: red;
}

#id2 {
  font-size: 1.1em;
  margin: 0;
  color: red;
}
```

Identifying mixins in existing CSS code is made challenging by the number of possible combinations of all the declarations. Because mixins can be combined as well, this number of possible combinations may explode. Consequently, extracting mixins from a CSS file becomes a cumbersome task.

III. Our Approach

Our approach applies formal concept analysis to automate the extraction of mixins from CSS files. Mixins are identified from a Galois sub-hierarchy (GSH) which is a Directed Acyclic Graph. In this section, we first describe how to generate a GSH from a CSS file and then we detail how to extract mixins from this structure.

A. Generating the GSH

Formal Concept Analysis [4] (FCA) is a branch of lattice theory [5], [6] that aims at automatically finding groups of *objects* that share in common a group of *attributes*. FCA works on a specific type of data, named a *formal context*, in which objects are described by several attributes. A formal context is a triple $K = (O, A, I)$, where O and A are finite sets of objects and attributes, respectively, and $I \subseteq O \times A$ is a binary relation associating objects with attributes: $(o, a) \in I$ if object o has attribute a.

When building a formal context from a CSS file, objects (O) are CSS selectors (e.g. `h1`) and attributes (A) are both CSS properties (e.g. `color`) and CSS declarations (e.g. `color:black`). Hence, duplication at the property and declaration levels can be easily detected. Let's consider the CSS code depicted in Listing 9 to illustrate the notion of formal contexts. This example defines five rules containing from one to four declarations. Table I shows the corresponding formal context $|O| \times |A|$.

Listing 9: A sample CSS code.

```
body {
  font-size: 1em;
  padding: 0;
}
a {
  color: black;
}
.info {
  font-size: 1.2em;
  color: black;
  font-weight: 100;
  margin: 5px;
}
.error {
  font-size: 1.3em;
  color: black;
  font-weight: 200;
  margin: 10px;
}
#content {
  font-size: 1em;
  padding: 0;
  color: black;
  font-weight: 100;
}
```

TABLE I: The formal context corresponding to the CSS code of Listing 9.

	font-size	font-size : 1em	font-size : 1.2em	font-size : 1.3em	padding	padding : 0	color	color : black	font-weight	font-weight : 100	font-weight : 200	margin	margin : 5px	margin : 10px
body	×	×			×	×								
a							×	×						
.info	×		×				×	×	×	×		×	×	
.error	×			×			×	×	×		×	×		×
#content	×	×			×	×	×	×	×	×				

Let $X \subseteq O$, $Y \subseteq A$, $f(X) = \{a \in A \mid \forall o \in X, (o, a) \in I\}$ and $g(Y) = \{o \in O \mid \forall a \in Y, (o, a) \in I\}$, then $f(X)$ gives all the attributes shared by the objects contained in X, and $g(Y)$ gives all the objects sharing the attributes contained in Y. For example, in the formal context of Table I, $f(\{body, .info\}) = \{font\text{-}size\}$ and $g(\{padding{:}0\}) = \{body, \#content\}$.

A *concept* is a pair of sets (X, Y) such as $X = g(Y)$ and $Y = f(X)$. In other words, a concept is a maximal collection of objects sharing common attributes. In the formal context of Table I, the objects $\{.info, .error, \#content\}$ and the attributes $\{font\text{-}size, color, color{:}black, font\text{-}weight\}$ constitute a concept because these attributes together describe only these objects, and these objects together share no other attributes. Conversely, the objects $\{a, .error, \#content\}$ and the attributes $\{color, color{:}black\}$ do not constitute a concept because these attributes describe also the object *.info*. For a concept (X, Y), the set of objects X is the *extent* and the set of attributes Y is the *intent*.

The set of all the possible concepts extracted from a formal context forms a complete partial order and can be ordered in a lattice. The subconcepts (resp. superconcepts) of a concept $c_i = (X_i, Y_i)$ are the concepts $c_j = (X_j, Y_j)$ such as $X_j \subseteq X_i$ or, equivalently, $Y_i \subseteq Y_j$ (resp. $X_i \subseteq X_j$ or $Y_j \subseteq Y_i$). The result of the application of FCA on a formal context is the *concept lattice*. Figure 1 shows the concept lattice corresponding to the formal context of Table I. The source of an edge is the superconcept and the destination is the subconcept. The *top* concept represents the attributes shared by all the objects and the *bottom* concept represents the objects that have all the attributes. In this example, the top concept has an empty intent and the bottom concept has an empty extent.

The simplified extent (resp. simplified intent) of a concept (X, Y) is the set $X' \subseteq X$ (resp. $Y' \subseteq Y$) of objects (resp. attributes) of this concept that are not in the extent of its subconcepts (resp. in the intent of its superconcepts). The simplified intent and extent are shown in bold in Figure 1. A simplified lattice is a lattice containing the simplified extent and the simplified intent of the concepts.

The GSH [5] is a simplification of concept lattice. Concepts with empty simplified extent and simplified intent are discarded. Figure 2 shows the GSH corresponding to the concept lattice of Figure 1. In this example, *top* and *bottom* concepts

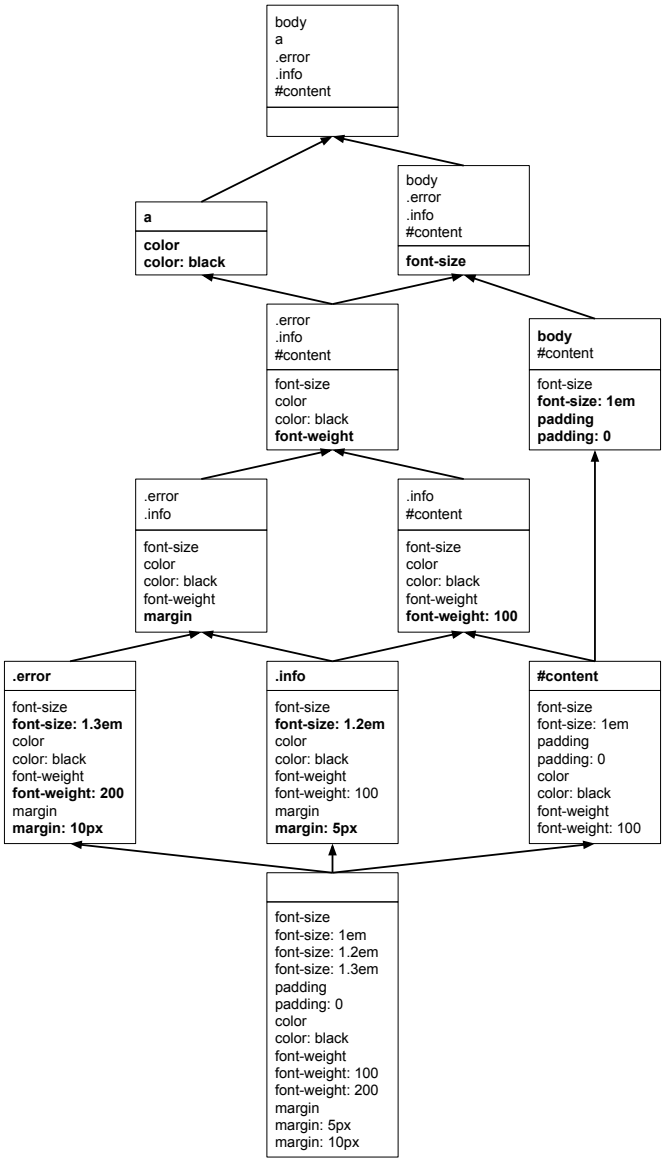


Fig. 1: Concept lattice corresponding to the formal context of Table I. Simplified intents and extents are shown in bold.

have been removed. While the number of concepts in a lattice can be exponential, it is at most equal to the number of objects and attributes ($|O| + |A|$) in a GSH [7]. Finally, Galois sub-hierarchies – which are Directed Acyclic Graphs – are the cornerstone for the identification of mixins.

B. Extracting mixins from the GSH

Our algorithm for mixins extraction takes as input a topologically sorted list of concepts extracted from the GSH, and generates as output both a set of mixins and a set of CSS rules. We define a filtering mechanism to discard irrelevant mixins before the generation phase.

1) Filtering irrelevant concepts: To discard irrelevant mixins, we remove concepts of the GSH. Removing a concept from the GSH modifies the simplified intent of its subconcepts.

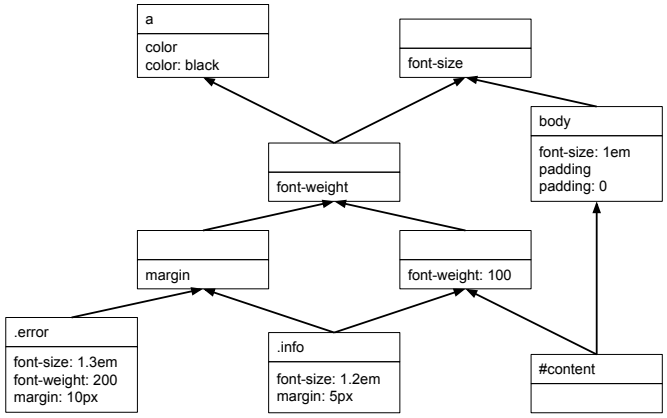


Fig. 2: GSH obtained from the concept lattice of Figure 1.

The content of the simplified intent of the removed concept is duplicated in the simplified intent of each of its subconcepts. Conversely, the simplified extent is not modified.

Four data are available to decide whether a concept has to be filtered out or not: its extent, intent, superconcepts and subconcepts. Any filtering function can be developed based on this information. Several filtering methods can be applied successively.

a) Our first filtering method: In this study, we consider a mixin irrelevant if (1) it is not used enough, (2) it does not introduce enough declarations, or (3) it takes too many parameters. For each concept, these criteria are checked using (1) the number of direct subconcepts, (2) the number of declarations and properties in the simplified intent, and (3) the number of properties in the intent. Thresholds are associated to each of these criteria and concepts that do not fulfill one of them are filtered out. Concepts with a non-empty simplified extent can not be discarded because the selectors they introduce would be lost. Consequently, when a concept with a non-empty simplified extent has to be filtered out, only edges with its subconcepts are suppressed if they exist.

Figure 3 shows a concept filtering on the GSH of Figure 2. The numbering of concepts in this figure is only used to ease their identification in the discussion. In this example, we set the minimum number of children to 2, the minimum number of declarations to 1 and the maximum number of mixin parameters to 3. *Concept#1* and *Concept#4* had only one subconcept each. Hence, the edge with their subconcept is removed. Their simplified intent is thus duplicated in the simplified intent of their subconcept. Other concepts with a non-empty simplified extent are kept because they do not have more than three properties and introduce at least one declaration or property.

b) Our second filtering method: We propose another filtering function to meet developer needs, that can be applied in combination to the threshold-based method presented above. Concepts factorizing different kinds of properties are considered irrelevant and thus filtered out. Hence, mixins contain only related properties, such as ones describing *background*,

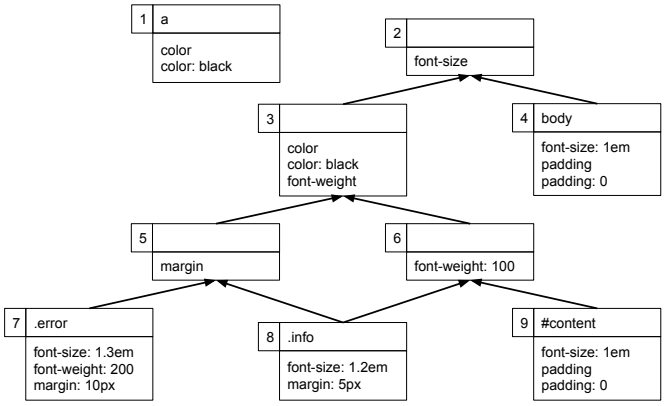


Fig. 3: Concept filtering on the GSH of Figure 2.

animation or *font*. In order to determine whether two properties are related or not, we rely on data from the *w3schools*[10] website which provides a classification of all the CSS properties in several groups. Thus, we consider two properties related if they belong to the same group.

2) Generating mixins and CSS rules: Before generating mixins and CSS rules, properties that have a corresponding declaration in the simplified intent of a concept are removed. For instance, in Figure 3, *Concept#1* has both a property `color` and a declaration `color:black`. The `color` property is thus removed.

A mixin is generated when a concept has at least one subconcept and a CSS rule is generated when a concept has a non-empty simplified extent. For instance, in Figure 3, a CSS rule is generated from *Concept#8* and a mixin from *Concept#6*. Listing 10 reports the Sass code generated with our approach from the GSH of Figure 3. Four mixins are generated corresponding to the concepts having subconcepts. When a concept has both a non-empty simplified extent and one or many subconcepts, a mixin and a CSS rule are generated. CSS preprocessors, such as Sass and Less, have a limited support of CSS rules inheritance: nested selectors can not be extended. For instance, it is not possible to extend the selector `.a #b`. This is why we generate both a mixin and a CSS rule for these concepts.

A mixin call is generated when there is an edge between two concepts. Calling a mixin requires to provide a value to each of its parameters. A track between parameter and the associated property is preserved. This allows to ensure the right order of parameters in mixin call. If a parameter has a matching declaration, i.e. a declaration with the same property, the corresponding value is used in the mixin call. Declarations that match a parameter are removed if they are in the simplified intent. If a parameter does not have a matching declaration, an *abstract* value is used in the mixin call. Abstract values will be set by subconcepts. Consider the mixin call between *Concept#2* and *Concept#3* on Figure 3. Since *Concept#3* does not have a declaration that matches the property `font-size`, an abstract value is used in the mixin call (line 5 in Listing 10). Conversely, the value `1em` is used for the mixin call between *Concept#2* and *Concept#4* (line 31 in Listing 10).

Generating a mixin from a concept is a three-step process. Firstly, a generic name is generated for the mixin. The list of all the selectors from which the properties and declarations are factorized is advised in comments ahead the mixin definition. Secondly, mixin calls are generated for each of the superconcepts, and the remaining declarations in the simplified intent are added to the content of the mixin. Thirdly, parameters list of the mixin is calculated. Properties in the simplified intent and abstract values in mixin calls constitute the parameters list. No particular order is used for the parameters list. For instance, in Figure 3, *Concept#2* generates a mixin with one parameter corresponding to its property `font-size` (line 1 in Listing 10). There is no other parameter since it does not have any superconcept. Conversely, *Concept#5* generates a mixin with three parameters (line 12 in Listing 10): one corresponding to its property `margin` and two others (`font-weight` and `font-size`) from its superconcepts.

Generating a CSS rule from a concept is a two-step process. Firstly, selectors are obtained from all the simplified extent's elements joined with commas. Secondly, as for mixin generation, mixin calls are generated and the remaining declarations in the simplified intent are added to the content of the rule. The generation is then finished since concepts with a non-empty simplified extent have only declarations in their simplified intent. For instance, *Concept#9* in Figure 3 generates a CSS rule with one declaration and one mixin call since it has one superconcept (line 23 in Listing 10).

It has to be mentioned that the compilation of Sass code into CSS code does not handle multiple inclusion of a particular declaration. Considering Sass code of Listing 10, the CSS rule `.info` (line 19) has two mixin calls and both of them includes mixin $m3$. Consequently, all declarations introduced in $m3$ will appear twice in `.info`. Sass does not check if declarations already exist before inserting them. The only issue of this behaviour is the size increase of the generated CSS code. Listing 11 reports the CSS code compiled from the Sass code of Listing 10 for the rule `.info`. In fact, the problem occurred when a concept has at least two paths to the same superconcept: it is known as the diamond problem in the field of object-oriented languages. To overcome it we compute a spanning arborescence on the GSH to turn it into a tree. As for the filtering step, the simplified intent of some concepts are duplicated because of the removal of some edges. We make this transformation optional since less refactorings are available.

Listing 11: Excerpt of the CSS code compiled from the Sass code of Listing 10.

```
.info {
  color: black; font-size: 1.2em; font-weight: 100;
  color: black; font-size: 1.2em; font-weight: 100;
  margin: 5px;
}
```

[10]http://www.w3schools.com/cssref/

Listing 10: Sass code generated from the GSH of Figure 3.

```
@mixin m2($s) {
  font-size: $s;
}
@mixin m3($w, $s) {
  @include m2($s);
  font-weight: $w;
  color: black;
}
@mixin m6($s) {
  @include m3(100, $s);
}
@mixin m5($m, $w, $s) {
  @include m3($w, $s);
  margin: $m;
}
a {
  color: black;
}
.info {
  @include m6(1.2em);
  @include m5(5px, 100, 1.2em);
}
#content {
  @include m6(1em);
  padding: 0;
}
.error {
  @include m5(10px, 200, 1.3em);
}
body {
  @include m2(1em);
  padding: 0;
}
```

C. Ensuring CSS rendering preservation

In Section II-A page 2, we highlight the importance of style rules order in a CSS file on the rendering of web document. To preserve the semantics of a CSS file, we refine the objects – the CSS selectors – by adding line numbers to the initial CSS file. Thus, objects with the same name can be distinguished by FCA.

Consider the sample CSS file in Listing 12. When objects are only CSS selectors, the formal context contains one object (*.info*) with 4 attributes (*color*, *color:black*, *margin*, *margin:5px*) and generates only one concept: $(\{.info\}, \{color, color{:}black, margin, margin{:}5px\})$. At this step, we lose the information that the selector was defined in two different places. Finally, the concept generates only one CSS rule `.info` that contains two declarations: `color:black` and `margin:5px`. Hence, we can not guarantee that the semantics of the initial CSS file is preserved. Conversely, when adding line number information to CSS selectors, the formal context contains two objects: *.info1* and *.info4*. These object identifiers mean that two rules located at lines 1 and 4 share the same selector `.info`. This formal context generates two concepts $(\{.info1\}, \{color, color{:}black\})$ and $(\{.info4\}, \{margin, margin{:}5px\})$ Consequently, two distinct rules are generated. They are then sorted to be in the same order as in the initial CSS file. Finally all selectors grouped in a same rule by our approach are splitted across different rules, except if they were already in the same rule in the original file. Duplication introduced while splitting selectors can be avoided with mixins that are named differently.

TABLE II: *Mocss* options.

Option	Description
max-parameters	avoid mixins with too many parameters
min-children	avoid mixins not used enough
min-declarations	avoid mixins introducing not enough declarations
groups-filter	generates mixins factorizing common properties
no-duplicates-into-rule	avoid duplication as depicted in Listing 11
keep-semantic	determines whether semantics of the input file has to be preserved

Listing 12: Sample CSS file.

```
.info { color: black; }
.info { margin: 5px; }
```

The semantics preservation of the original CSS file is performed by default but it can be disabled. We apply the following procedure to check the semantics validity of the generated file. Once the file containing the mixins is generated, we compile it and compare the result to the original CSS file. Then, we check that exactly all CSS rules of the original file are present in the same order and contain exactly the same set of declarations.

D. Tool Implementation

We have implemented our approach to automatically identify mixins from CSS code in a Java open source tool, named *Mocss*, that we make publicly available on GitHub[11]. *Mocss* takes as input a CSS file and generates equivalent Sass code that uses mixins to reduce code duplication. The tool is currently distributed as a command-line program that accepts several parameters to control generated code, as described in Table II. We rely on the work of Mazinanian and Tsantalis [8] to define the default values of the first three options. They conduct an empirical study on the CSS preprocessor codebase of 150 websites and investigate the usage pattern of four language features including mixins. They find out that 63% of the mixins are called two or more times (*min-children* $= 2$), and 68% of the mixins have either one or no parameters (*max-parameters* $= 1$). We empirically evaluated that 3 seems a sensible value for the *min-declarations* option.

IV. Evaluation

To assess our approach, we perform two experiments and several case studies. First, we evaluate the time performance of our approach, to ensure that it is usable by developers on real projects. Second, we assess the effect of the thresholds described in the previous section on the number of generated mixins. Finally, we describe several case studies where our approach has been used by professional developers on CSS code coming from projects they work on.

A. Experiments

In the experiments described in the remainder of this section, we apply our approach on 108 CSS files. We use the data from the evaluation of Mazinanian *et al.* [1] which

[11] https://github.com/acharpen/mocss

TABLE III: Selected subjects.

Web app.	#CSS files[1]	#CSS files[2]	Web app.	#CSS files[1]	#CSS files[2]
Facebook	6	7	Pinteerst	2	3
YouTube	4	5	Reddit	1	2
Twitter	2	1	Tumblr.com	2	3
YahooMail	3	9	Wordpress.org	1	3
Outlook.com	6	11	Vimeo.com	3	2
Gmail	5	5	Igloo	2	2
Github	2	1	Phormer	1	1
Amazon.ca	3	3	BeckerElectric	1	0
Ebay	2	2	Equus	1	1
About.com	1	1	ProToolsExpress	1	3
Alibaba	3	0	UniqueVanities	3	3
Apple.ca	3	4	ICSE12	3	4
BBC	3	1	EmployeeSolutions	3	5
CNN	1	0	SyncCreative	3	3
Craiglist	1	3	GlobalTVBC	5	6
Imgur	2	1	Lenovo	1	1
Microsoft	1	1	MountainEquip	2	3
MSN	1	1	Staples	2	1
Paypal	1	3	MSNWeather	3	3

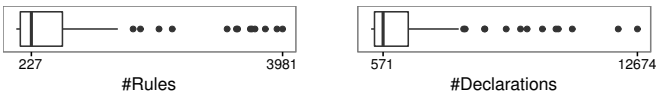
[1] Number of CSS files used in the study of Mazinanian *et al.* [1].
[2] Number of CSS files used in this study.

contains CSS files from 38 real-world web applications. They extracted 155 CSS files from these web applications. However some of them contain invalid CSS code and consequently they might raise parsing errors. Table III summarizes, for each of the 38 web applications, the number of CSS files analyzed by Mazinanian *et al.* and the number of CSS files successfully parsed by the parser we rely on[12]. As a result, we analyze 108 CSS files (69%) while Mazinanian did *et al.* 90 (58%).

1) Performance Evaluation: We evaluate our tool according to two CSS file characteristics: the number of rules and the number of declarations. Figure 4 shows the distributions of these two characteristics among the 108 CSS files of our study. One can note that 75% of these CSS files have less than 691 rules (resp. 1,768 declarations) and the median is 227 rules (resp. 571 declarations). The maximum number of rules (resp. declarations) in this set of CSS files is 3,981 (resp. 12,674).

We empirically evaluate that *keep-semantic* is the only option having an impact on the execution time. Indeed, by preserving the order of selectors, more objects are included into the formal context and thus more concepts are inserted in the generated Galois sub-hierarchy. For the sake of readability we only report the time performance of one configuration: the default one because it is likely to be the most used and it enables semantic preservation. Further, we provide all necessary data to analyze time performance of other configurations[13]. We run our tool ten times on each of the 108 CSS files and plot the median of these ten executions on Figure 5. The curves on the two figures represent the interpolation of the median times. Our test machine is a 2.10Ghz Core i7-4600U with 16GB of RAM. Overall, *Mocss* handles 63% of the CSS files in less than one second, and 83% in less than two seconds. Thus, in most of the cases, the results of *Mocss* are almost immediate.

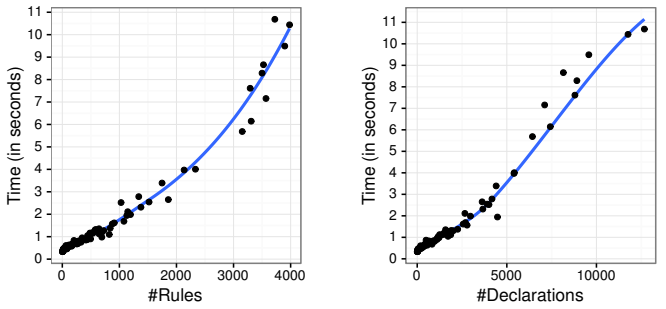
[12]https://github.com/phax/ph-css/releases/tag/ph-css-4.0.1
[13]http://www.labri.fr/perso/acharpen/icsme16/materials.zip



(a) Rules number of the 108 CSS files.

(b) Declarations number of the 108 CSS files.

Fig. 4: Characteristics of the 108 CSS files used in the experiment.



(a) Running times according to the number of rules.

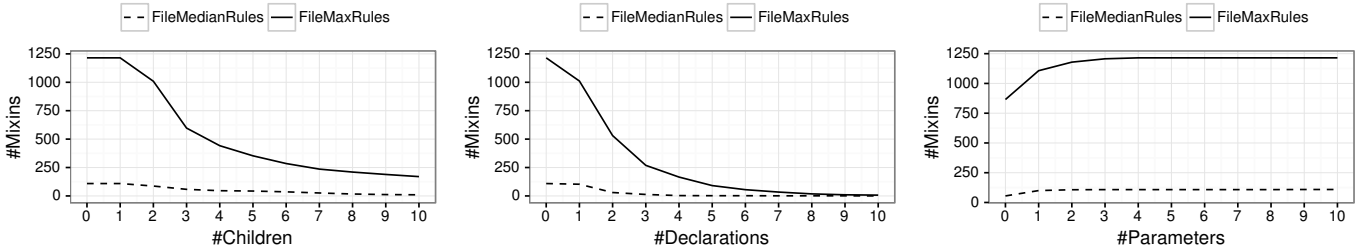
(b) Running times according to the number of declarations.

Fig. 5: Running times of our tool on the 108 CSS files according to the number of rules (left) and declarations (right).

More precisely, 85% of the CSS files that have a number of rules lower than 691 or a number of declarations lower than 1,768 are processed by *Mocss* in less than one second, and all these files in less than two seconds. For the largest files of our corpus, *Mocss* running time is about eleven seconds, which is still acceptable. To conclude, this experiment shows that our tool is usable by developers on real projects.

2) Thresholds Evaluation: *Mocss* has three thresholds that enable a fine-grained control on the generated mixins: *min-children*, *min-declarations* and *max-parameters*. We evaluate each of them individually on the set of 108 CSS files with values ranging from 0 to 10. When a threshold is evaluated the others are disabled in order to guarantee the measures validity. Therefore, for each threshold and CSS file, we obtain 11 values corresponding to the number of mixins generated by *Mocss* for a specific configuration on this CSS file. We report only results for two CSS files in Figure 6 for the sake of readability: the file having the maximum number of CSS rules and the one with a number of rules close to the median.

Overall, the thresholds allow to reduce significantly the number of generated mixins. Thresholds *min-children* and *min-declarations* have a larger impact on the number of generated mixins than *max-parameters*. The number of generated mixins decreases very quickly when setting a value greater or equal to 2 for *min-children* and *min-declarations*. Further, there are only few mixins with more than 3 parameters: only a few additional mixins are generated when setting a value greater than 3 for this threshold.



(a) *min-children* (b) *min-declarations* (c) *max-parameters*

Fig. 6: Evolution of the number of generated mixins according to each of the three thresholds: (a) *min-children*, (b) *min-declarations* and (c) *max-parameters*.

B. Case Studies

The case studies we conduct aim at getting some insight about the real-world relevance of our approach. Hence, we seek the opinion of developers that are more than familiar with CSS. We investigate the relevance of the thresholds and configuration options of our approach as well as its produced output.

1) Research Methodology: Four developers participated to the evaluation of our approach: Three are web developers from local companies. They all work everyday with CSS, and thus are more than knowledgeable about CSS and its extension languages, including Sass. The fourth one is a faculty member of our University. He is used to give lectures on web related technologies, including CSS and its preprocessors.

We conduct a survey that include open-ended questions through a Google form. Therefore, participants have unlimited time to complete the survey. They are asked to evaluate our tool on websites they designed. Each participant is allowed to perform the evaluation on several projects. For each of them, a two step process has to be conducted. First, she runs the tool on a CSS file and has to determine the number of relevant mixins from the whole set of results. In case she finds some irrelevant mixins, she is asked to explain the reasons. This step aims at investigating the default values of the configuration parameters of the tool. Similarly, the second step aims at evaluating the impact of the thresholds on the number of relevant generated mixins. This step can be skipped when the default configuration reports only relevant mixins. Each participant is asked to test different values for the parameters and to determine which configuration produces the best results, i.e. the highest ratio of mixins that are relevant and useful according to her. Once the survey is completed, participants are asked to give their feelings on the tool.

Table IV reports the raw results of these case studies, including the number of rules and declarations of the different CSS files. *Participant#1* evaluates our tool on three CSS files: two of them are in plain CSS while the last is written in Sass. One can note that the first two are medium sized while the last is large sized. For this last CSS file, the compiled version is used as the input of our approach. By evaluating our tool on CSS code generated from Sass, we can evaluate to what extend we can find mixins that were in the original Sass code. *Participant#2* and *Participant#3* conduct the survey on medium sized CSS files, while *participant#4*, who refers to the faculty member of our University, uses a small sized CSS file.

2) Results: The default configuration of our tool reports 203 mixins on the first CSS file considered by *Participant#1*. According to him, 198 (97%) were relevant and 5 were not used enough to be useful. He modified the value of the threshold *min-declarations* from 3 to 4 to obtain the best configuration. The latter reported 118 mixins that are all relevant. This first CSS file is the one generated from Sass code. *Participant#1* noticed that our tool found almost all of his mixins and a lot of new ones. For the two other CSS files he analyzed with our tool, the default configuration identified respectively 15 and 11 mixins. In both cases, the whole set of mixins were relevant. Overall, he found the tool easy to use and very interesting to migrate a website from CSS to Sass.

For *participant#2*, 12 mixins from the 37 (32%) reported by our tool were relevant. Irrelevant mixins had too many parameters and factorized properties that are not related. He did not identify a better configuration than the default one. According to him, the tool should have proposed to factorize related properties. We found his suggestion really interesting and discussed on it through emails. We have implemented this new way to filter mixins, as discussed in III-B1b (*Our second filtering method* paragraph). Nevertheless, he did not have time to test the new version of our tool.

For *participant#3*, the default configuration identified 25 mixins; 20 (80%) were relevant. Like *participant#1*, some mixins were irrelevant because they were not used enough. In order to reach the best configuration, *participant#3* increased the value of the threshold *min-children* from 2 to 3. This custom configuration leads to 19 mixins that are all relevant. Finally, she found out that the detection of mixins from legacy CSS code is useful and that our tool is able to identify interesting mixins.

Participant#4 found that all mixins reported by the default configuration were relevant. He considered the detection of mixins from CSS code interesting and found out that our

TABLE IV: Characteristics of the websites used in the case studies along with information about the mixins identified with the default and best configurations.

					Default configuration		Best configuration[1]	
Participant	**Source**	**Generated file**	**#Rules**	**#Declarations**	**#Total mixins**	**%Relevant mixins**	**#Total mixins**	**%Relevant mixins**
1	professional	yes (with Sass)	2,819	6,133	203	97%	118	100%
1	personal	no	194	721	15	100%	-	-
1	personal	no	104	327	11	100%	-	-
2	professional	no	327	1,014	37	32%	-	-
3	personal	no	258	821	25	80%	19	100%
4	personal	no	97	255	5	100%	-	-

[1] When default and best configurations are the same, the last two columns are left blank.

approach fits his needs.

All participants found that the tool's running time is more than satisfactory. Furthermore, most participants did not find mixins' naming too hard. They reported that the comment inserted by the tool before a mixin simplifies the naming. This comment includes the name of the selectors from which declarations have been factorized.

3) Threats to validity: Our case studies bear one threat to internal validity: participants are contacts of our research group and thus their answers may have been influenced. Nevertheless, they are all professional and therefore we are confident they gave fair answers.

Threats to external validity refer to the generalization of our findings. First, the number of CSS files evaluated by each participant may not be sufficient. Case studies conducted by these participants on other CSS files may produced different results. To mitigate this threat, participants chose alone the CSS files on which they have evaluated our tool and were able to perform the survey on several CSS files. Second, only four developers have evaluated our tool. However, they all are experts on CSS and its preprocessors, and their participation to the survey was voluntary. Third, participants skipped second step of the case study when the default configuration reported only relevant mixins. But even in that case, testing different values for the parameters could have been valuable. Indeed, different values for the parameters could produce more mixins that are all relevant. This threat should be addressed in a replication study in order to complete our results. Finally, further validation with more participants should be performed to increase the generalizability of the findings. Nevertheless our tool seems to have a real-world relevance.

C. Discussion

We discuss here some limitations of current implementation of our tool. First, generating proper names for the mixins is important. Providing the list of all the selectors from which the properties and declarations are factorized ahead the mixin definition is not helpful enough. Second, our tool does not correctly handle shorthand properties. A shorthand property is a CSS property that allows to set the values of several other CSS properties. For instance, the declaration `border: 1px solid red` sets 12 different properties: `border-top-color`, `border-top-style`, `border-top-width` and so forth. Hence, some values are overriden when properties and their corresponding shorthand properties are in the same CSS rule. This overriding becomes an issue when it is not possible to determine the final value of a property. This occurs when the expansion of a declaration with a shorthand property is ambiguous. In that case, our tool can not guarantee the CSS rendering preservation of the original file. Considering the following valid CSS declaration: `border: solid`. In this example, the values of properties `border-color` and `border-width` can not be resolved by analyzing a CSS file because default values defined on the client side (web brower or user preferences) are used.

V. Related Work

We divide related work into two groups. First, we discuss the different issues investigated by the research community on CSS code. Then, we provide a brief overview of the many uses of formal concept analysis in software engineering.

Cascading style sheets

A number of studies have been conducted on developers' experience with CSS and how to improve it. Quint and Vatton [9] provide an overview of style issues a web author is faced with and solutions to address these difficulties. They provide a web authoring environment implementing the proposed solutions. They argue that some issues of editing style sheets have to be investigated. Keller and Nussbaumer [10] introduce an abstractness factor for measuring CSS code quality. They argue that a high abstractness factor represents a high reusability and maintainability of CSS code. They conduct an empirical study on human-written CSS code and generated CSS code, and they find that manual CSS code has a higher abstractness factor than generated code. Liang *et al.* [11] present a tool, called SeeSS, that aims at automatically tracking CSS change impact across an entire website and helping developers visualize them. Participants of their case study indicate fixing CSS problems more quickly when using SeeSS. Mesbah and Mirshokraie [12] present a technique to automatically detect unmatched and ineffective CSS code by analyzing the relation between CSS rules and DOM (Document Object Model) elements of web applications. They implement their approach in a tool called Cilla. Similarly, Genevès *et al.* [13] focus on static analysis for CSS style sheets. They propose a tool based on tree logics that is capable of detecting a wide range of errors in CSS code. Following on from the work of Genevès *et al.*, Bosch *et*

al. [14] present a tool of static CSS semantical analyzer and optimizer that aims at automatically detecting and removing unnecessary property declarations in CSS files.

Several studies have been conducted on the detection of duplication in web artifacts. Most of them focus on finding web pages with similar content [15], [16] or structure [17], [18]. Few studies focus on the detection of duplication in CSS code. Mao *et al.* [19] present an automated process for migrating websites from table-based layout to the style-based layout. They use clone detection to find the duplicated code across CSS files. The work of Mazinanian *et al.* [1] is the most closely related work to this paper. They define three types of CSS declaration duplication and propose an automated technique to eliminate those instances. Conversely to our approach, they identify only duplication at the declaration level because they rely on the mechanisms available in CSS (see Section II-B). Their technique eliminates duplicated CSS declarations by grouping selectors as depicted Listing 6, and uses DOM tree instances from web applications to find presentation-preserving refactorings, and thus to preserve the CSS semantics. They conduct an empirical study on 38 real-world websites and find that the extent of duplication in CSS code is extensive ranging from 40% to 90%. They also find that the number of presentation-preserving refactorings is significant. Differences with our work are due to the possibilities offered by CSS preprocessors over just the CSS.

Several extension languages have emerged to overcome the limitations of the CSS language. Badros *et al.* [20] propose a constraint-based style sheet model, called CCSS. Their model allows more flexible specification of layout and it is compatible with the CSS2 specification. Marden *et al.* [21] introduce another style sheet language, named PSL. More recently, CSS extension languages – such as Sass, Less and Stylus – offer among other things the support of mixins.

Formal concept analysis

Researchers apply FCA to debugging [22], [23], testing [24] and refactoring. The refactoring techniques of applying concept lattice include particularly class hierarchies refactoring [25], [26], module refactoring [27]–[29] and low-level to high-level programming refactoring [30], [31]. Our work is close to class hierarchies refactoring. Lienhard *et al.* [25] propose a semi-automatic approach using FCA for the identification of traits in inheritance hierarchies. Their tool enables a refactoring of class hierarchy with traits which preserves the initial behavior of each of the classes. They validate their approach with a case study and find that their refactorings are similar to the ones obtained manually. Snelting and Tip [26] present a method based on FCA for finding design problems in a class hierarchy by making the relationship between class members and variables explicit. They show their technique to be capable of identifying design anomalies such as class members that are redundant or that can be moved into a derived class.

To the best of our knowledge, using FCA to automatically extract mixins from CSS file has not been addressed in the literature.

VI. Conclusion and Future Work

The CSS language is widely adopted in web development and it is now common for web projects to have several thousands of CSS lines of code. Although its use is almost mandatory, the language lacks many of the advanced features programming languages provide to support code reuse. Thus, extension languages such as Sass have emerged to extend CSS capabilities. Programs written in these languages are then compiled into low level CSS code. However, migrating existing CSS code to Sass is challenging for web developers because of the lack of tool support.

In this paper, we have presented a new approach to automatically identify and remove code duplication in CSS code. Our approach relies on formal concept analysis (FCA) to automate the identification and extraction of mixins. We have implemented our approach in a tool named *Mocss* that automatically generates Sass code from existing legacy CSS code.

We perform an experiment on 108 CSS files from real-world web applications and find that our tool enables a fine-grained control of the generated code and has good performance as well. Further, we conduct several case studies to evaluate the real-world relevance of our approach. We demonstrate that our tool helps developers removing code duplication by extracting mixins from existing CSS code. Developers that tested our tool appreciate the opportunity it provides to automatically migrate legacy CSS code to Sass.

We are now exploring a number of research avenues. We are working on some improvements of our tool: we are adding support for media queries, variables in order to avoid duplication at the value level, and nesting since developers widely use it according to Mazinanian and Tsantalis [8]. We are also investigating the use of FCA to identify other refactoring opportunities of CSS rules. From now, mixins generated with our approach contain only declarations. Nevertheless, extension languages enable mixins to factorize full CSS rules. This offers an additional way to reduce the amount of duplicated code in CSS files. Further, we plan to work on an extension of our approach: a step-wise approach that recommends a ranked list of mixin opportunities and allows developers to preview each one of them in order to assess their impact on maintainability and understandability. Following this step-wise approach, developers would give appropriate names to the mixins.

Acknowledgments

The authors would like to thank all the participants to the survey and the case studies.

References

[1] D. Mazinanian, N. Tsantalis, and A. Mesbah, "Discovering Refactoring Opportunities in Cascading Style Sheets," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 496–506. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635879

[2] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, Sep. 2008, pp. 227–236.

[3] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings of the 8th International Symposium on Software Metrics*, ser. METRICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 87–. [Online]. Available: http://dl.acm.org/citation.cfm?id=823457.824038

[4] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[5] M. Barbut and B. Monjardet, *Ordre et classification algèbre et combinatoirs*. Hachette, 1970.

[6] G. Birkhoff, G. Birkhoff, G. Birkhoff, and G. Birkhoff, *Lattice theory*. American Mathematical Society New York, 1948, vol. 25.

[7] A. Berry, M. Huchard, A. Napoli, and A. Sigayret, "Hermes: an efficient algorithm for building Galois sub-hierarchies," in *CLA'2012: 9th International Conference on Concept Lattices and Applications*, U. P. Laszlo Szathmary, Ed. Fuengirola (Málaga), Spain: Universidad de Malaga, Oct. 2012, pp. 21–32. [Online]. Available: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00743882

[8] D. Mazinanian and N. Tsantalis, "An empirical study on the use of css preprocessors," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER 2016, 2016.

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

[10] M. Keller and M. Nussbaumer, "CSS Code Quality: A Metric for Abstractness; Or Why Humans Beat Machines in CSS Coding," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, Sep. 2010, pp. 116–121.

[11] H.-S. Liang, K.-H. Kuo, P.-W. Lee, Y.-C. Chan, Y.-C. Lin, and M. Y. Chen, "Seess: Seeing what i broke – visualizing change impact of cascading style sheets (css)," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '13. New York, NY, USA: ACM, 2013, pp. 353–356. [Online]. Available: http://doi.acm.org.gate6.inist.fr/10.1145/2501988.2502006

[12] A. Mesbah and S. Mirshokraie, "Automated Analysis of CSS Rules to Support Style Maintenance," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 408–418. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337272

[13] P. Geneves, N. Layaida, and V. Quint, "On the Analysis of Cascading Style Sheets," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 809–818. [Online]. Available: http://doi.acm.org/10.1145/2187836.2187946

[14] M. Bosch, P. Genevès, and N. Layaïida, "Automated Refactoring for Size Reduction of CSS Style Sheets," in *Proceedings of the 2014 ACM Symposium on Document Engineering*, ser. DocEng '14. New York, NY, USA: ACM, 2014, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2644866.2644885

[15] C. Boldyreff and R. Kewish, "Reverse engineering to achieve maintainable www sites," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 2001, pp. 249–257.

[16] F. Calefato, F. Lanubile, and T. Mallardo, "Function clone detection in web applications: A semiautomated approach," *J. Web Eng.*, vol. 3, no. 1, pp. 3–21, May 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=2011138.2011140

[17] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, "Understanding cloned patterns in web applications," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, May 2005, pp. 333–336.

[18] G. Di Lucca, M. Di Penta, and A. Fasolino, "An approach to identify duplicated web pages," in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, 2002, pp. 481–486.

[19] 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," in *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '07. Riverton, NJ, USA: IBM Corp., 2007, pp. 12–26. [Online]. Available: http://dx.doi.org/10.1145/1321211.1321214

[20] G. J. 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*, ser. UIST '99. New York, NY, USA: ACM, 1999, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/320719.322588

[21] P. M. Marden Jr and E. V. Munson, "Psl: An alternate approach to style sheet languages for the world wide web." *J. UCS*, vol. 4, no. 10, pp. 792–806, 1998.

[22] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, "Debugging temporal specifications with concept analysis," in *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA: ACM, June 2003, pp. 182–195. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=72000

[23] P. Cellier, "Formal concept analysis applied to fault localization," in *Companion of the 30th international conference on Software engineering*. ACM, 2008, pp. 991–994.

[24] S. Khor and P. Grogono, "Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 346–349. [Online]. Available: http://dx.doi.org/10.1109/ASE.2004.71

[25] A. Lienhard, S. Ducasse, and G. Arévalo, "Identifying Traits with Formal Concept Analysis," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 66–75. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101921

[26] G. Snelting and F. Tip, "Reengineering class hierarchies using concept analysis," in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '98/FSE-6. New York, NY, USA: ACM, 1998, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/288195.288273

[27] R. Al-Ekram and K. Kontogiannis, "Source code modularization using lattice of concept slices," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, March 2004, pp. 195–203.

[28] H. Kim and D.-H. Bae, "Object-oriented concept analysis for software modularisation," *IET software*, vol. 2, no. 2, pp. 134–148, 2008.

[29] P. Tonella, "Concept analysis for module restructuring," *Software Engineering, IEEE Transactions on*, vol. 27, no. 4, pp. 351–363, Apr 2001.

[30] A. El Kharraz, P. Valtchev, and H. Mili, "Concept analysis as a framework for mining functional features from legacy code," in *Proceedings of the 8th International Conference on Formal Concept Analysis*, ser. ICFCA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 267–282. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11928-6_19

[31] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, Nov 2004, pp. 112–121.