Migrating Cascading Style Sheets to Preprocessors by Introducing Mixins

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

ABSTRACT

Cascading Style Sheets (CSS) is the standard language for styling web documents and is extensively used in the industry. However, CSS lacks constructs that would allow code reuse (e.g., functions). Consequently, maintaining CSS code is often a cumbersome and error-prone task. Preprocessors (e.g., Less and Sass) have been introduced to fill this gap, by extending CSS with the missing constructs. Despite the clear maintainability benefits coming from the use of preprocessors, there is currently no support for migrating legacy CSS code to preprocessors. In this paper, we propose a technique for automatically detecting duplicated style declarations in CSS code that can be migrated to preprocessor functions (i.e., *mixins*). Our technique can parameterize differences in the style values of duplicated declarations, and ensure that the migration will not change the presentation semantics of the web documents. The evaluation has shown that our technique is able to detect 98% of the *mixins* that professional developers introduced in websites and Style Sheet libraries, and can safely migrate real CSS code.

CCS Concepts

•Software and its engineering → Software maintenance tools; Maintaining software;

Keywords

Cascading style sheets, refactoring, duplication, migration

1. INTRODUCTION

Cascading Style Sheets is one of the three fundamental technologies in front-end web development (along with HTML and JavaScript), which is used for defining the presentation of documents written in a markup language (e.g., HTML). Currently, CSS is used in more than 91% of top 10 million websites in the Alexa ranking [42], and by more than 90% of front-end web developers [32]. Recently, CSS started being used in a wider spectrum of application domains, including mobile app UI design (e.g., Apache Cordova, Adobe PhoneGap), desktop app styling (e.g., WinJS, GTK+), and Integrated Development Environment theming (Eclipse4/CSS).

Despite its popularity, developing and maintaining CSS code is a cumbersome task [19, 15, 31]. Among others, two main reasons for this complexity are 1) the inherent limitations of the language to support reuse, and 2) the lack of well-documented and empirically validated best practices. As an example, in CSS there is very limited support for constructs that can facilitate the DRY (don't-repeat-yourself) principle, since functions and variables are not supported. Consequently, style declarations (i.e., statements in CSS that define a style, like `font-weight: bold`) are extensively duplicated in CSS code. In a previous work [30], we found that 40-90% of CSS style declarations are duplicated in modern web applications. This extensive duplication can definitely impose high development and maintenance costs.

CSS *preprocessors* have emerged as the de-facto solution to complement "vanilla" CSS by adding the missing constructs, which are available in traditional programming languages (e.g., variables and functions), and thus expedite the development and maintenance of Style Sheets. The use of preprocessors is a trend in the industry [9, 41], and leading web companies have already adopted them. Some examples of popular preprocessors are Less (used by Twitter), Sass, Google Closure Style Sheets, and Stylus. Despite the gradual adoption of preprocessors in the web development community, there is still a large portion of front-end developers and web designers using solely "vanilla" CSS. An online poll with nearly 13,000 responses from web developers [9] revealed that 46% of them develop only in "vanilla" CSS, mostly because they are not aware of preprocessors.

Therefore, there is a large community of web developers that could benefit from tools helping them to automatically migrate their "vanilla" CSS code to a preprocessor of their preference. In addition, the community of CSS preprocessor developers could benefit from tools helping them to utilize more effectively the features offered by the preprocessors. As a matter of fact, in this study we found several cases where professional developers under-utilize preprocessors.

This is the first work to investigate the automatic extraction of duplicated style declarations in CSS, into function-like constructs in CSS preprocessors (i.e., *mixins*), enabling the *reuse* of existing CSS code. Using *mixins* can also improve the readability of Style Sheets by assigning descriptive names to them. According to our previous study [29], developers introduce *mixins* mostly for reusing code in Style Sheets (63% of the *mixins* were called more than once in the

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970348

code base of each project), but also for breaking long style rules in smaller code fragments, or simply improving code readability. The proposed approach for abstracting duplicated style declarations to *mixins* is one of the fundamental requirements for developing a full-fledged recommendation system that can help developers to *migrate* existing CSS code to preprocessors, or even improve the maintainability of a preprocessor code base by eliminating duplication. This work makes the following contributions:

- We propose a method for detecting opportunities to automatically extract *mixins* from existing CSS code. The approach is preprocessor-agnostic, i.e., it is applicable for all CSS preprocessors supporting the notion of *mixins*;
- We propose a method for assuring that the *presentation semantics* of the CSS code are preserved after migration;
- We conduct an empirical study with real websites and Style Sheet libraries using preprocessors to verify the correctness and effectiveness of our approach.

2. PRELIMINARIES

2.1 The CSS Language

Style Sheets let developers apply styles to some *elements* of one or more *target documents* (e.g., making all hyperlinks in an HTML page blue and under-lined). CSS has a very simple syntax, as shown in Figure 1.

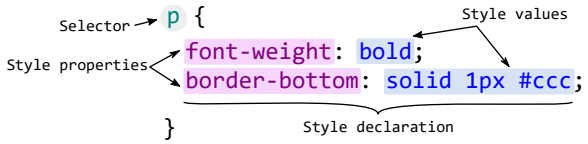


Figure 1: A style rule in CSS Syntax

A *style rule* (i.e., *rule-set*) is a fundamental building block of CSS files. It consists of a *selector*, which defines the elements of the target documents to be styled (e.g., `p` for selecting all paragraphs). Inside the body of a style rule there are some `style declarations`, each of which assigns some *style value* to a *style property* (e.g., assign style value `bold` to property `font-weight` to make the text of the selected elements bold-faced).

Specificity and Cascading: In CSS, when two or more different style declarations assign a value to the same property of a DOM element in the target document, the web browser has to decide which style declaration has a precedence over the others. In such a case, the style declaration belonging to a style rule with a more *specific* selector *wins* (i.e., its style value *overrides* the value of the other declarations). For instance, selector `a` selects all hyperlink elements, while selector `p a` selects only the hyperlink elements, which are children of paragraph elements (`a` and `p` are both CSS predefined selectors selecting the `<a>` and `<p>` tags in target HTML documents, respectively). Thus, the style declarations in `p a` will override the ones in `a`, because the latter is more specific than the former.

On the other hand, in the case that two style rules have selectors with the same specificity, the order of style rules will decide the *winning* style declaration, i.e., the style declaration in a style rule appearing later in the CSS file will override the style values of the previous declarations (the so-called *cascading* feature of CSS). However, developers can use the `!important` rule (e.g., `color: red !important`) to force a declaration to always override other declarations with the same property, regardless of the specificity and location of the style rule it belongs to. Using `!important` is a bad practice in Style Sheet development, but Style Sheet developers considerably use it [16]. As we will see, when detecting *mixin* migration opportunities, the use of `!important` has implications that should be taken into account.

2.2 CSS Preprocessors

As mentioned before, CSS preprocessors were introduced to address the limitations associated with CSS. Preprocessors are actually source-to-source compilers (i.e., *transpilers*). In other words, they compile the Style Sheet written in their syntax to "vanilla" CSS, allowing the clients to directly use the generated Style Sheets without requiring any server-side processing. This transpilation can be also performed on demand, i.e., when a user requests to view a web page.

Most of the CSS preprocessors extend the syntax of CSS to make easier their adoption by CSS developers. There is a set of common features that almost all popular preprocessors support. Because of space limitation, we cannot describe all these features in this paper (the reader may refer to [29], or the documentation of CSS preprocessors [6, 37]). However, we need to briefly mention the two main features of CSS preprocessor languages that can be used for eliminating duplicated style declarations in Style Sheets:

Extend is a mechanism that enables the reuse of style declarations across style rules. The *extending* style rule inherits all style declarations of the *extended* style rule, and can optionally *override* some of the inherited style declarations in order to change their style values This mechanism is similar to inheritance in the object-oriented paradigm.

Mixin is a function-like construct containing a set of style declarations, optionally with parameterized style values. A *mixin* is usually called inside the body of a style rule or another *mixin* by passing style values as arguments for its parameters. A *mixin* parameter may have a default value, allowing to omit the corresponding argument.

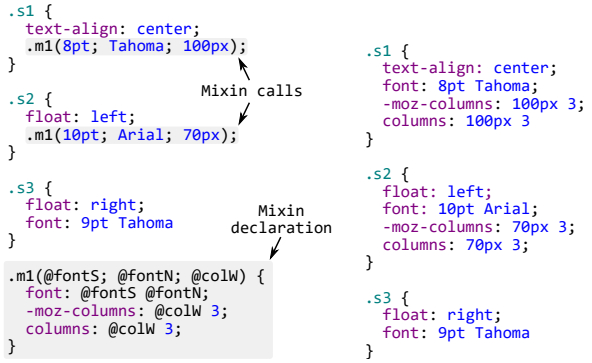
In the case of exactly duplicated style declarations across different style rules, one may use either the *extend* construct or a parameterless *mixin* for eliminating duplication. However, our previous study [29] revealed that developers prefer to use parameterless *mixins* over the *extend* construct (28% of the 100 analyzed websites exclusively used parameterless *mixins*, while only 9% used the *extend* construct). This preference to *mixins* is probably due to the *styling bugs* that may be caused by the incautious use of the *extend* construct [29]. Indeed, the grouping of selectors in CSS, or equivalently the use of the *extend* construct in preprocessors, might change the *order dependencies* that exist among the style rules, and thus change the presentation of the target document (i.e., introduce a styling bug) [30]. Additionally, *mixins* provide a more powerful reuse mechanism by allowing to parameterize the values of style properties. Therefore, in this work, we propose an approach for detecting *mixin* opportunities in CSS code that can help developers to easily migrate to a preprocessor of their preference.

2.3 Mixins

CSS preprocessor *mixins* are equivalent to functions in traditional programming languages, and usually contain a set of style declarations or other *mixin* calls. The values of these declarations can be literals (e.g., color or numeric

values), expressions involving variables, or calls to CSS preprocessor built-in functions. Just like functions, *mixins* can have one or more parameters, which can be used in the place of style declaration values. Figure 2a shows a preprocessor code snippet in LESS syntax, containing three style rules, namely `.s1`, `.s2` and `.s3`, and a *mixin* declaration `.m1`, which is called in style rules `.s1` and `.s2`.

When the piece of preprocessor code shown in Figure 2a is transpiled to CSS, the code shown in Figure 2b is generated. As it can be observed, the *mixin* calls are replaced with the style declarations of the called *mixin*, and the parameterized values are replaced with the arguments passed in the corresponding *mixin* call.



(a) Preprocessor Code (b) Genereated CSS

Figure 2: Mixin example

In Figure 2a, the *mixin* `.m1` contains three style declarations, assigning values for properties: `font`, `-moz-columns` and `columns`. The two latter properties are called *vendor-specific* properties, and are used to style the same property for different web browsers, e.g., the ones with `-moz-` prefix will only work for Mozilla Firefox. Web browsers usually support this kind of properties before they are fully standardized, because the standardization process is time consuming. As a result, developers have to repeat the declaration of the same property with different vendor prefixes to achieve a consistent presentation across multiple web browsers. Indeed, we previously found that developers tend to use *mixins* for grouping style declarations associated with vendor-specific properties [29].

3. AUTOMATIC EXTRACTION OF MIXINS

Our approach for detecting *mixin* migration opportunities is based on eliminating duplication at the level of style declarations, and consists of four main steps, which are explained in the following subsections.

3.1 Grouping Declarations for Extraction

The first step of our approach is to find sets of style rules sharing one or more style declarations styling the same properties. The tuple $\langle S, P \rangle$, where S is a set of style rules sharing a set of style properties P is considered as a *mixin migration opportunity*.

For example, to reverse-engineer the CSS code of Figure 2b to the preprocessor code of Figure 2a, we would need to group the declarations corresponding to `font`, `columns` and `-moz-columns` properties from style rules `.s1` and `.s2` and extract these declarations into *mixin* `.m1` after parameterizing the property values being different. In this case, $S_1 = \{$`.s1`, `.s2`$\}$ and $P_1 = \{$`font`, `columns`, `-moz-columns`$\}$.

There are, however, other *mixin* migration opportunities in the code fragment of Figure 2b; e.g., $S_2 = \{$`.s1`, `.s2`, `.s3`$\}$ and $P_2 = \{$`font`$\}$.

It should be emphasized that for a given property $p \in P$ the grouped style declarations should have the same *importance*. As mentioned, using the `!important` rule makes a declaration to have precedence over other declarations assigning a value to the same property. As a result, the importance of the involved declarations should be kept intact when forming a *mixin*, to make sure that the presentation of target documents will be preserved after migration. Consequently, we avoid the grouping of declarations having a different importance (i.e., either all or none of the grouped declarations may use the `!important` rule).

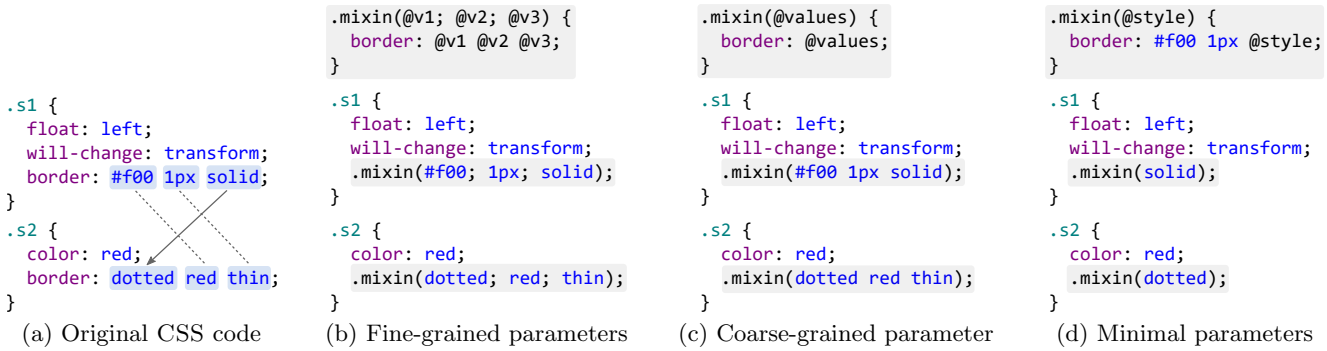
A naive approach for finding the repeated properties would be to exhaustively generate all possible combinations of style properties and then examine if they appear together in two or more style rules. However, this would certainly lead to a combinatorial explosion, due to the large percentage of duplication in CSS files. According to [30], on average, 66% of the style declarations in Style Sheets are duplicated at least once (i.e., they share the same property and value with another style declaration). For detecting *mixin* migration opportunities, the constraint that property values should be identical or equivalent is not necessary, since *mixins* allow the parameterization of differences in the property values. This results to an even larger percentage of declarations that can be potentially grouped to form a *mixin*, making the exhaustive generation inapplicable.

We managed to overcome this issue by treating the initial problem as a *frequent itemset mining* problem [39]. In the data mining literature, there are efficient algorithms for finding the sets of *items* in a database (i.e., itemsets) that appear *together* in s or more transactions, where s is the *minimum support count*. In our approach, we treat a Style Sheet as a transactional database. Each style rule is a transaction, and each property corresponding to a style declaration is an item. Therefore, a *frequent itemset* is a set of properties P that appear together in a set of style rules S, where $|S| \geq s$. In our work, we set s to two (i.e., the smallest possible value). We selected this value because we have empirically shown that the median number of times a *mixin* is called in real-world CSS preprocessor code is two [29]. We adopted the FP-GROWTH [18] algorithm, because it is considered efficient and scalable. Applying the FP-GROWTH algorithm to the CSS code of Figure 2b will result in the output shown in Table 1.

Table 1: Frequent itemsets of style properties

$\|P\|$	S	P
1	{.s1, .s2}	{-moz-columns}
	{.s1, .s2}	{columns}
	{.s2, .s3}	{float}
	{.s1, .s2, .s3}	{font}
2	{.s1, .s2}	{columns, -moz-columns}
	{.s1, .s2}	{columns, font}
	{.s1, .s2}	{-moz-columns, font}
	{.s2, .s3}	{font, float}
3	{.s1, .s2}	{font, columns, -moz-columns}

Each row (itemset) in Table 1 constitutes a separate *mixin* migration opportunity. The *mixin* `.m1` in Figure 2a corresponds to the last itemset of the table, and it *subsumes* the first three itemsets with $|P| = 2$, which in turn *subsume* the two first itemsets with $|P| = 1$.



(a) Original CSS code (b) Fine-grained parameters (c) Coarse-grained parameter (d) Minimal parameters

Figure 3: Alternative ways for extracting a *mixin*

3.2 Detecting Differences in Style Values

For a given *mixin* migration opportunity $\langle S, P \rangle$, we need to check for every property $p \in P$, if the corresponding style declarations have different (i.e., non-equivalent) values. For each difference in the property values, a parameter should be introduced in the resulting *mixin*. However, the parameterization of differences can be achieved in several alternative ways. As an example, consider the CSS code shown in Figure 3a that contains two style rules for selectors `.s1` and `.s2`, respectively. Both selectors style property `border`, which is a well-known *shorthand* property allowing to set the values of several other CSS properties simultaneously, such as `border-width`, `border-style`, and `border-color`. In addition, some shorthand properties do not force a specific order for the values of the properties they group. As a result, in selector `.s1` the `border` shorthand styles first the `border-color` with `#f00`, then the `border-width` with `1px`, and finally the `border-style` with `solid`, while in selector `.s2` the order is different starting first with `border-style` followed by `border-color` and `border-width`.

In this particular example, and in all the cases involving shorthand properties, there are three possible ways to parameterize the differences in property values:

1. Introduce a separate parameter for each pair of different values in the order they appear in the corresponding shorthand property declarations (Figure 3b). This approach has two main limitations. First, it may introduce parameters for properties using different kinds of values. The variables and parameters in CSS preprocessors do not have a type, and thus it is possible to have parameters accepting arguments of different value kinds. However, it will be extremely difficult for a developer to understand and reuse a *mixin* having parameters that can take values for semantically different properties (e.g., a property that should be styled with color values, and a property that should be styled with dimension values). Second, such an approach may lead to *mixins* with an unnecessarily long parameter list, which is considered a code smell [13], since it makes more difficult to call and reuse such *mixins*. In a previous work [29], we showed that developers mostly tend to create *mixins* with zero or one parameter (68% of the *mixins* have either one or no parameters).
2. Introduce a single multi-value parameter for all individual properties grouped by the corresponding shorthand property (Figure 3c). Although this approach gives more flexibility to the developer when calling the *mixin*, it is also more error-prone, because the developer needs to know very well the CSS documentation regarding the individual property values that are mandatory and those that can be omitted (i.e., *optional* values), or the order of the individual property values. Inexperienced CSS developers would need to spend time studying the documentation in order to properly call a *mixin* with such parameters.
3. Introduce a parameter for each pair of *matching individual properties* having non-equivalent values in the corresponding shorthand property declarations (Figure 3d). In the example shown in Figure 3a, the matching individual properties between the two selectors are represented with arrows. The dashed-line arrows indicate properties with equivalent values (e.g., the *named color* `red` and the *hexadecimal color* `#f00` are not lexically identical, but are alternative representations for the same color). The solid-line arrows indicate properties with non-equivalent values that should be parameterized. This approach has two main advantages over the other approaches. First, it introduces a minimal number of parameters compared to the first approach, when some individual properties are styled with identical or equivalent values (regardless of the order they appear in the shorthand property declarations). Second, it allows to introduce parameters with more *semantically expressive* names compared to the second approach, since the names of the matching individual properties can be used as parameter names (e.g., `@style` parameter in Figure 3d corresponding to the individual property `border-style`).

3.2.1 Inferring Individual Style Properties

In our approach, we adopted the last parameterization strategy discussed in the previous section due to its advantages over the other two strategies. To implement this strategy, we first need to infer the *individual style property* (ISP) corresponding to each style value that appears within the style declarations for the set of properties P declared in the set of selectors S. An ISP represents the *role* of a style value in a style declaration, and corresponds to the actual individual style property this value is being assigned to. Table 2 shows the ISPs that are assigned to the values of the `border` style declarations in the example of Figure 3a. For instance, the pair of values corresponding to colors (`#f00` and `red`) are both assigned to the same ISP, which is the individual property `border-color`.

For the style properties that can accept only a single value (e.g., `color`, `float`), the ISP assigned to their values is the same as the style property name. For shorthand properties (e.g., `border`, `background`, `columns`), we refer to the CSS specifications [43] for assigning an ISP to each one of their

Table 2: Individual Style Properties (ISPs)

Declaration	Style Value	ISP
border: #f00 1px solid	#f00	"border-color"
	1px	"border-width"
	solid	"border-style"
border: dotted red thin	dotted	"border-style"
	red	"border-color"
	thin	"border-width"

values. In our implementation, we have coded ISPs for 47 multi-valued and shorthand CSS properties, which account for all major CSS properties used in Style Sheets. When comparing two declarations for parameterizing the differences in their values, we compare each pair of values corresponding to the same ISP. We follow the same approach used in [30] for examining whether two values are equivalent.

Optional (omitted) values: In CSS, some properties can have *optional* values. For instance, the property `font` can accept 7 style values, while developers may omit 5 of them. In the case of omitted values, web browsers follow the CSS specifications to compute them. In some cases, they assign *initial* (i.e., default) values to the omitted values. In other cases, the omitted value is calculated based on another explicitly given value. Following the same approach, for every omitted value we actually compute a *virtual value*, and also assign the appropriate ISP to it. This allows parameterizing declarations having an *unequal* number of style values, e.g., `font: bold 10pt Tahoma` and `font: 18pt Arial`. In this example, the first declaration is styling the `font-weight` ISP with the explicitly-defined value `bold`, while the second one styles the same ISP with the default value `normal`.

Shorthand vs. individual properties: Another possible scenario is having some selectors taking advantage of shorthand properties, while other selectors are instead declaring separately individual properties.

Consider the CSS example shown in Figure 4a. Style rule `.s1` contains four separate style declarations for the individual properties `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`, while style rule `.s2` contains a single style declaration for the shorthand property `margin`. Note that, the last value in the `margin` declaration is omitted (i.e., there are three values instead of four). This value corresponds to the `margin-left` ISP, and based on the CSS specifications, it should take the `margin-right` value (i.e., `5px`) when it is omitted. To enable the detection of *mixin* migration opportunities in such cases, we are collapsing the sets of individual property declarations that can be grouped into *virtual shorthand declarations*. In this way, it is possible to find *mixin* migration opportunities between selectors either having actual or virtual shorthand declarations, and thus the CSS example shown in Figure 4a can be migrated to use the *mixin* shown in Figure 4b.

3.3 Introducing a Mixin in the Style Sheet

In this section, we describe an algorithm that takes as input a *mixin* migration opportunity $MO = \langle S, P \rangle$ to generate a *mixin* declaration and update the style rules in set S to call the new *mixin* passing the appropriate arguments.

In Algorithm 1, we use the following helper functions. Function `getISPsWithNonEquivalentValues` returns the ISPs, which are defined in the style declarations (declared inside the style rules in set `S`) corresponding to property `p` and have non-equivalent values.

```
.s1 {
  ...
  margin-left: 5px;
  margin-right: 5px;
  margin-top: 3px;
  margin-bottom: 3px;
  ...
}
.s2 {
  ...
  margin: 2px 5px 1px;
  ...
}
```

(a) Original CSS

```
.mixin(@mtop; @mbottom) {
  margin: @mtop 5px @mbottom 5px;
}
.s1 {
  ...
  .mixin(3px; 3px);
  ...
}
.s2 {
  ...
  .mixin(2px; 1px);
  ...
}
```

(b) Extracted *mixin*

Figure 4: *mixin* for shorthand/individual properties

Function `generateStyleDeclarationTemplate` generates a style declaration template for property `p` with a placeholder (i.e., an unknown value) for each ISP defined in the style declarations (declared inside the style rules in set `S`) corresponding to property `p`. The template will essentially contain a mapping of ISPs to style values or variables (i.e., *mixin* parameters) after the execution of the algorithm.

Finally, function `getStyleDeclaration` returns the style declaration corresponding to property `p` declared inside style rule `s`, and function `getStyleValue` returns the style value corresponding to an ISP defined in style declaration `d`.

Algorithm 1: Algorithm for introducing a *mixin*

```
   Input  : A mixin migration opportunity MO = ⟨S, P⟩
   Output : A mixin declaration M = ⟨M_p, M_d⟩
            A mapping of selectors to lists of mixin arguments
1  M_p ← ∅ // the ordered set of mixin parameters
2  M_d ← ∅ // the ordered set of mixin style declarations
3  foreach s ∈ S do
4  |  M_a(s) ← ∅ // the list of mixin arguments for s
5  end
6  foreach p ∈ P do
7  |  differences ← getISPsWithNonEquivalentValues(p, S)
8  |  template ← generateStyleDeclarationTemplate(p, S)
9  |  foreach ISP ∈ template.ISPs do
10 |  |  if ISP ∈ differences then
11 |  |  |  param ← newMixinParameter(ISP)
12 |  |  |  M_p ← M_p ∪ param
13 |  |  |  ISP ↦ param // map ISP to mixin parameter
14 |  |  |  foreach s ∈ S do
15 |  |  |  |  d ← getStyleDeclaration(p, s)
16 |  |  |  |  arg ← getStyleValue(ISP, d)
17 |  |  |  |  M_a(s) ← M_a(s) ∪ arg
18 |  |  |  end
19 |  |  end
20 |  |  else
21 |  |  |  s ← S_0 // get the first style rule in S
22 |  |  |  d ← getStyleDeclaration(p, s)
23 |  |  |  value ← getStyleValue(ISP, d)
24 |  |  |  ISP ↦ value // map ISP to common value
25 |  |  end
26 |  end
27 |  M_d ← M_d ∪ template
28 end
```

Generating mixin declaration: In order to create the *mixin* declaration, the algorithm generates a style declaration template (line 8) for each property p in the set of style properties P. Function `generateStyleDeclarationTemplate` goes through all declarations styling p in the set of style rules S and finds the union of ISPs that are assigned with values. This approach can guarantee that all affected ISPs will be present in the template, even if some style declarations omit the definition of optional values. Next, for each ISP in the template, the algorithm checks if the assigned style values are equivalent or not. If the values are different, the ISP is mapped to a new *mixin* parameter, which is also added to the parameter list of the *mixin* declaration. Otherwise, the

ISP is mapped to the commonly assigned value in all style rules. Finally, the resulting template after the mapping of all ISPs is added to the list of style declarations inside the body of the *mixin* declaration. It should be emphasized that the order of the style declarations inside the *mixin* follows the relative order of the style declarations in the original style rules from which they were extracted. As it will be explained in Section 3.4, this is essential for preserving the presentation of the target documents, in the case where some style declarations have order dependencies with each other. In such a case, an ordering that reverses the original order dependencies between the style declarations would affect the values assigned to the ISPs, thus changing the presentation.

Adding mixin calls to style rules: Given the *mixin* migration opportunity $MO = \langle S, P \rangle$, for each one of the style rules in S, a call to the generated *mixin* should be added. Whenever the assigned style values for a given ISP are not equivalent, the algorithm goes through all style rules in S, and for each style rule s appends to the corresponding list of *mixin* arguments $M_a(s)$ the actual value assigned to the ISP by s (lines 14-18). At implementation level, in each style rule s the style declarations corresponding to the set of properties P are removed, and a *mixin* call with the argument list $M_a(s)$ is added.

3.4 Preserving Presentation

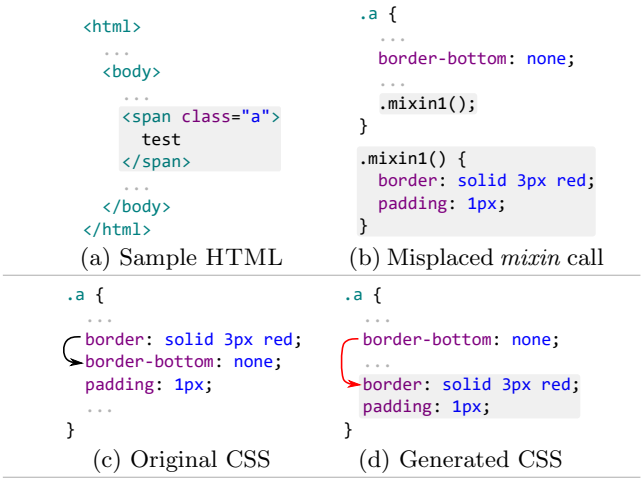
In refactoring [34] preserving the behavior of the program is very critical. The refactored program should have exactly the same behavior as the original program before refactoring. In a similar manner, any refactoring or migration operation applied to CSS code should preserve the *presentation* of the target documents (i.e., the style values applied to the DOM elements after refactoring should be exactly the same as before refactoring). Therefore, in the context of CSS, *program behavior* corresponds to *document presentation*, and any CSS refactoring/migration technique should make sure that document presentation is preserved.

We previously investigated the refactoring of duplicated style declarations directly in CSS by grouping selectors, which style properties with equivalent values [30]. We found that in order to preserve the presentation of the target documents, the order dependencies between the selectors declared in a CSS file should be preserved after refactoring. Formally, an *order dependency* [30] 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.

Practically, if two style rules have selectors with the same specificity [2], which select common DOM elements in the target documents and style the same property, then the style value initially assigned by the preceding style rule will be eventually *overridden* by the value assigned by the succeeding style rule. Therefore, the relative order of these style rules should be preserved after refactoring in order to obtain the same value for the property styled in both of them. As a result, *presentation preservation* in CSS can be achieved by preserving the order dependencies among all selectors declared in the file.

Within the context of *mixin* migration, the introduction of a new *mixin* does not affect the original order of the style rules. Therefore, it is not possible to break the existing inter-selector order dependencies by introducing a *mixin*. However, there might exist *intra-selector* order dependencies between style declarations in the style rules where the *mixin* calls will be added. Consider, for instance, the CSS code shown in Figure 5c. The `border-bottom` declaration in selector `.a` overrides the style values, which are defined by the `border` declaration. The `border` declaration is a shorthand declaration defining the border for all individual sides of an element (`top`, `right`, `bottom`, `left`). Next, the developer overrides with `none` the border styling for the bottom side of the element. CSS developers tend to override more generic style properties (e.g,. `border`) with more specific ones (e.g., `border-bottom`), because this approach requires less code than defining separately all specific style properties (e.g., `border-top`, `border-right`, `border-bottom`, `border-left`).



(a) Sample HTML (b) Misplaced *mixin* call

(c) Original CSS (d) Generated CSS

(e) Original presentation (f) Broken presentation

Figure 5: Intra-selector order dependencies

Applying style rule `.a` to the sample HTML code shown in Figure 5a will result to a web document rendered as shown in Figure 5e. Let us assume that we extract a *mixin* containing the `border` declaration, and we place the *mixin* call at the end of `.a`, as shown in Figure 5b. The preprocessor will then generate the CSS code shown in Figure 5d, where the `border` declaration is placed *after* the `border-bottom` declaration. This will invert the original overriding relation between the two declarations, resulting to the undesired presentation shown in Figure 5f (i.e., a styling bug). Therefore, placing the *mixin* call in an incorrect position can actually change the presentation of the target documents.

We define an *intra-selector order dependency* from style declaration d_i to d_j (both declared in the same style rule) due to individual property *isp*, denoted as $\langle d_i \rangle \xrightarrow{isp} \langle d_j \rangle$, iff:

a) declarations d_i and d_j set a value to individual property *isp* and have the same importance (i.e., both or none of the declarations use the `!imporant` rule),
b) declaration d_i precedes d_j in the style rule.

To ensure that the presentation of the target documents will be preserved, we define the following *preconditions*:

Precondition 1: The addition of a *mixin* call in a style rule should preserve all order dependencies among the style declarations of the rule.

The problem of finding an appropriate position for calling the extracted mixin m inside the body of a style rule can be expressed as a *Constraint Satisfaction Problem* (CSP) defined as:

Variables: the positions of the style declarations involved in order dependencies including m.

Domains: the domain for each variable is the set of values $\{1, 2, ..., N - M + 1\}$, where N is the number of style declarations in the original style rule, and M is the number of style declarations extracted from the style rule to m.

Constraints: Assuming that m contains style declarations assigning values to the set of individual properties $ISPs$, an order constraint is created in the form of $pos(d_i) < pos(d_j)$ for every order dependency $\langle d_i \rangle \xrightarrow{isp} \langle d_j \rangle$ where $isp \in ISPs$.

In the example of Figure 5c, there is one order dependency `border: solid 3px red` $\xrightarrow{\texttt{border-bottom-style}}$ `border-bottom: none`, resulting to the constraint $pos(\texttt{border}) < pos(\texttt{border-bottom})$. Based on this constraint, the call to `.mixin1` should be placed at any position before the `border-bottom` declaration to preserve the presentation of the target document in Figure 5a. If there are multiple conflicting order dependencies between the *mixin* call and declarations of the style rule, it might be necessary to reorder some style declarations in order to comply with the solution returned by the solver. On the other hand, if the CSP is unsatisfiable (i.e., no solution is found) the corresponding *mixin* migration opportunity is excluded as non presentation-preserving.

Precondition 2: The ordering of the style declarations inside a *mixin* should preserve their original order dependencies in the style rules from which they are extracted.

This precondition is checked by extracting the original order dependencies between the style declarations inside mixin m from each style rule where m will be called. Assuming that m contains style declarations assigning values to the set of individual properties $ISPs$, if there exist two style rules s_i and s_j, where an order dependency for the same $isp \in ISPs$ is reverse, i.e., $\langle s_i, d_k \rangle \xrightarrow{isp} \langle s_i, d_l \rangle$ vs. $\langle s_j, d_l \rangle \xrightarrow{isp} \langle s_j, d_k \rangle$, then there is an order dependency *conflict* between s_i and s_j, and the corresponding *mixin* migration opportunity is excluded as non presentation-preserving.

4. EVALUATION

To assess the correctness and usefulness of the proposed technique, we designed a study aiming to answer the following research questions:

RQ1: Does the proposed technique always detect *mixin* migration opportunities that preserve the presentation of the web documents?

RQ2: Is the proposed technique able to find and extract *mixins* that developers have already introduced in existing CSS preprocessor projects?

4.1 Experiment Design

Selection of Subjects: To be able to answer the aforementioned research questions, we need to create a dataset of CSS files, which actually contain opportunities for introducing *mixins* by grouping style declarations duplicated among different style rules. We relied on the dataset of our previous study [29], which was used to investigate the practices of CSS preprocessor developers by analyzing the code base of websites using two different preprocessors, namely Less and Sass. More specifically, out of 50 websites in which style sheets were developed using Less, we selected the preprocessor code base of 21 websites, in which at least one *mixin* declaration was called at least two times, since a *mixin* called more than once in the preprocessor code will result in duplicated style declarations in the generated CSS code. Our approach should be able to reproduce the original *mixins* declared in the preprocessor files, and possibly recommend other *mixin* opportunities that the developers might have missed. We further extended this dataset with the CSS code generated from the preprocessor source code of eight popular Style Sheet libraries. We expect that the selected libraries apply the best practices regarding *mixin* reuse, since they are developed by very experienced developers. The complete list of the selected websites and libraries, along with the number of Less files and CSS files (resulting from transpiling Less files), the number of developer-defined *mixins*, the total number of style rules and declarations (representing the size of the analyzed CSS files), and the number of migration opportunities detected by our approach for each subject are shown in Table 3. The collected data, in addition to the implemented tools are available on-line [1].

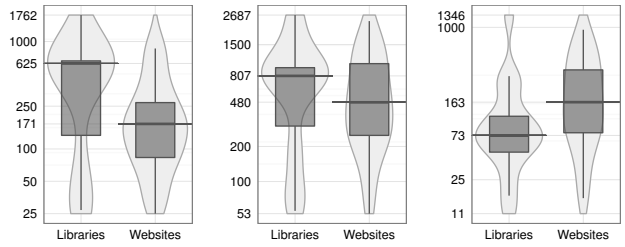
Table 3: Overview of the collected data

	Name	Less files	CSS files	Actual mixins†	Style Rules	Declarations	Detected Opportunities
Websites	aisandbox.com	3	2	6	113	443	188
	auroraplatform.com	2	1	1	83	247	100
	bcemsvt.org	17	2	2	163	327	70
	brentleemusic.com	13	1	13	861	2344	944
	campnewmoon.ca	2	1	3	218	527	162
	chainedelespoir.org	28	1	8	290	1081	528
	chunshuitang.com.tw	1	1	1	176	511	165
	colintoh.com	9	2	4	59	174	48
	first-last-always.com	16	1	6	339	1116	638
	florahanitijo.com	4	1	11	189	822	273
	greatlakeshybrids.com	1	1	3	104	373	168
	hotel-knoblauch.de	1	1	2	199	446	119
	intertelecom.ua	1	1	6	393	1329	708
	jutta-hof.de	2	1	3	171	560	245
	kko.com	1	1	1	98	255	84
	med.uio.no	80	1	6	762	1622	436
	naeaapp.com	14	1	8	828	1507	382
	neofuturists.org	12	3	11	522	1397	593
	paulsprangers.com	5	1	3	125	337	64
	schwimmschule-spawala.de	2	1	8	171	560	537
	summit.webrazzi.com	1	1	1	84	261	96
Libraries	base	20	1	2	489	736	114
	essence	119	7	11	4571	5939	615
	flatui	60	1	15	1139	2631	1346
	formstone	36	5	4	145	387	120
	kube	16	1	6	374	807	206
	schema	22	1	12	536	1524	284
	skeleton	1	1	2	95	222	45
	turret	83	1	34	1762	2687	307
Total		**572**	**44**	**193**	**15059**	**27811**	**9585**

† includes only the *mixins*, which are called at least two times

To better demonstrate the size characteristics of the examined CSS files, we show the distribution of the number of style rules and declarations defined in these files in Figure 6a and Figure 6b, respectively. The scale of the box plots and

the underlaid violin plots is logarithmic, and the horizontal bars correspond to the median values. The examined libraries tend to have more style rules and declarations than the examined websites. Figure 6c shows the plots for the number of *mixin* migration opportunities detected by our approach per CSS file in the dataset. As it can be observed, the median number of opportunities is 73 and 163, for the libraries and websites, respectively. In order to control for the CSS file size, and perform a fair comparison between the number of *mixin* migration opportunities in libraries and websites, we further normalized the number of opportunities detected in each CSS file by the number of style declarations defined in it. The normalized medians are 0.13 and 0.35 for the libraries and websites, respectively. This result shows that although the CSS code generated by libraries is larger in size than the code generated by the examined websites, the libraries tend to have less duplicated style declarations (and thus less *mixin* migration opportunities) than the examined websites. We can consider this as an indication that the preprocessor code of libraries is better designed.



(a) #Selectors (b) #Declarations (c) #Opportunities

Figure 6: Characteristics of the analyzed CSS files

4.2 Results

4.2.1 RQ1: Testing Presentation Preservation

Motivation: The goal of RQ1 is to ensure that our technique would never recommend the introduction of a *mixin* that could change the presentation of the target web documents (i.e., cause a styling bug).

Method: Let us assume that we have two CSS files C and C' (the latter resulting from transpiling the preprocessor code after the introduction of a new *mixin* in C) and we apply them on the same target documents. The target documents will have the same presentation iff:

1. There is a one-to-one correspondence (i.e., mapping) between the style rules defined in C and C'.
2. The relative order of the mapped style rules is the same in both C and C' (i.e., cascading is preserved).
3. Each pair of mapped style rules from C and C':
   (a) selects the same DOM elements in the target documents (i.e., specificity is preserved).
   (b) defines exactly the same set of individual style properties.
   (c) assigns equal or equivalent style values to each defined individual property.

Conditions 1, 2, and 3a are met by the nature of the transformation, because the introduction of a *mixin* does not add or remove style rules (i.e., one-to-one correspondence is met), does not change the order of the style rules (i.e., cascading is preserved), and does change the selectors specified in the style rules (i.e., specificity is preserved), respectively.

The only conditions that could be violated from an erroneous introduction of a *mixin* are 3b and 3c. Condition 3b could be violated, if the introduced *mixin* contains declarations for more, less, or different individual style properties than those that were removed/extracted from the style rule calling the *mixin*. Condition 3c could be violated if the parameterization of the differences in the style values is not correct, or if the *mixin* call is not placed in the appropriate position inside a style rule, or if the style declarations are not ordered correctly inside the *mixin* (Section 3.4).

Therefore, we developed a method to test conditions 3b and 3c that takes as input a CSS file C and a *mixin* migration opportunity MO, applies MO on file C to generate the corresponding CSS preprocessor code CP, then transpiles CP to obtain CSS file C', and examines the assertion:

> For every pair of matching style rules (s, s') defined in C and C', respectively, `style-map`$(s) \equiv$ `style-map`(s').

Two style rules are considered as matching if they have an identical selector. Function `style-map` takes as input a style rule and extracts a map in which the keys are the individual style properties (ISPs) defined in the style rule, and each key is mapped to the *final* style value assigned to the corresponding ISP, after all possible overrides. Two style maps are *equivalent* ($\equiv$) if their key sets are equal, and the style values corresponding to each key are equal or equivalent. Two style values are considered *equivalent*, when they are lexically different, but constitute alternative representations for the same style value (e.g., `red` $\equiv$ `#F00`, the first is the HTML named color representation for red, and the second is the hexadecimal representation for the same color).

Results: In total, we detected and applied 9,585 *mixin* migration opportunities and automatically tested them using the aforementioned method. It should be emphasized that a large portion of these opportunities overlap with each other (i.e., they affect common style rules and declarations), and thus it is not possible to apply them sequentially, since the application of an opportunity will make infeasible the opportunities it overlaps with. Therefore, we applied each one of them separately on the original CSS files.

We observed several cases where our testing method found styling bugs, which were due to our faulty implementation of style value inferencing. As an example, we found cases in which failing to assign correct ISPs to style values led to their incorrect parameterization and, consequently, the resulting preprocessor code was transpiled to a CSS file with different styling semantics than the original CSS file. For instance, when a shorthand property is assigned with the value `none`, only one of the ISPs is actually assigned with `none`, while the remaining ISPs are assigned with *default* values. Our implementation was not inferring correctly the default values, and this caused problems in preserving the presentation of the target documents.

Among the detected opportunities, there were 1227 cases, for which precondition #1 had to be examined, because there were order dependencies between style declarations extracted in the *mixin* and declarations remaining in the style rules where the *mixin* would be called. In one case, finding a satisfiable solution for positioning the *mixin* call was not feasible, and thus the migration was not performed. Moreover, there were 1190 cases, for which precondition #2 had to be examined. In all these cases, the original order of the declarations inside the extracted *mixin* could be preserved.

Overall, none of the issues found using our testing method was due to a flaw in the approach we proposed for detecting and extracting *mixins*. All issues were caused by implementation bugs that were eventually fixed, resulting in 100% of the tests being passed. Consequently, we can conclude that the *mixin* migration opportunities proposed by our approach are actually safe to apply.

4.2.2 RQ2: Detecting Mixins Defined by Developers

Motivation: The goal of RQ2 is to investigate whether our technique is able to recommend *mixins* that a human expert (i.e., a developer with expertise in the use of preprocessors) would introduce.

Method: To evaluate this research question we first built an *oracle* of human-written *mixins* by extracting all *mixins* in our preprocessor dataset being called at least two times. These *mixins* are suitable for testing our approach, because they introduce duplicated style declarations in the style rules where they are called after transpiling the preprocessor code to generate CSS code. The *mixins* called only once in the preprocessor code cannot be detected by our approach, because they do not introduce duplicated style declarations. Next, we transpiled each preprocessor file and applied our technique to detect all *mixin* migration opportunities in the resulting CSS files.

A *mixin* m, created by applying the migration opportunity mo detected by our approach, matches with a *mixin* m' in the oracle, iff:

1. the set of ISPs styled by m is equal to or is a superset of the ISPs styled by m',
2. m is called in at least all the style rules where m' is called.

The first condition ensures that m styles the same set of properties as m'. This condition is relaxed, so that m could style more ISPs than m'. This relaxation is necessary to deal with cases where the preprocessor developer missed the opportunity to include additional style properties being duplicated in the style rules from which m' was extracted. The second condition ensures that m is called in the same style rules where m' was called. This condition is also relaxed, so that m could be called in more style rules than m'. This relaxation is necessary to deal with cases where the preprocessor developer missed the opportunity to reuse m' in additional style rules. If m' is matched by applying the first relaxed condition, then m' is not a *closed* frequent itemset, since there is at least one superset with the same frequency.

Results: Our approach was able to recover 189 out of the 193 *mixins* in the oracle. In particular, 3 *mixin* migration opportunities detected by our approach were exact matches, 17 contained additional properties (i.e., supersets with the same frequency), 29 were called by additional style rules (i.e., same sets with higher frequency), and 214 contained additional properties and were called by additional style rules (i.e., supersets with higher frequency). The large percentage of inexactly matched *mixin* migration opportunities ($260/263 = 98.8\%$) actually shows that in most of the cases developers under-utilize *mixins*.

We further manually investigated the 4 oracle *mixins* that our approach could not detect. In general, these *mixins* fall into two categories:

Mixins using property interpolation: *Interpolation* is an advanced preprocessor feature allowing to use variables to form property names. It is useful for styling different properties that have the same sub-properties (e.g., `margin-top` and `padding-top` can be interpolated as `@{property}-top`), or can take the same values. Figure 7a shows an example of an *interpolated* property name inside a *mixin*, and the resulting CSS code is depicted in Figure 7b. Our technique cannot detect such *mixin* opportunities, since it does not support the parameterization of differences in property names.

```
.inherit(@property) { @{property}: inherit; }
.s1 { .inherit(margin); }
.s2 { .inherit(padding); }
```

(a) Preprocessor Code

```
.s1 { margin: inherit; }
.s2 { padding: inherit; }
```

(b) Genereated CSS

Figure 7: Example of interpolated property name

Use of `!important` in arguments: As mentioned before, in our approach we avoid the grouping of properties having a different importance. However, the `!important` rule can be actually used in the arguments of a *mixin* call to parameterize properties having a different value and importance, as shown in the example of Figure 8a, resulting to the CSS code shown in Figure 8b. Our approach does not support for the moment such advanced parameterization of differences.

```
.m(@var){ color: @var; }
.s1 { .m(~'red !important'); }
.s2 { .m(blue); }
```

(a) Preprocessor Code

```
.s1 { color: red !important; }
.s2 { color: blue; }
```

(b) Genereated CSS

Figure 8: Example of `!important` use in arguments

4.3 Limitations

The success of a recommendation system is associated with the relevance of the recommendations to its users, often measured in terms of precision and recall. Assuming that the *mixins* introduced by developers (e.g., the oracle used in RQ2) constitute the gold standard, our approach can achieve very high recall with a small number of undetected actual *mixins* (i.e., false negatives), but it generates a large number of *mixin* opportunities, and some of them might be considered irrelevant by the developers (i.e., false positives). Although, it is not possible to determine the actual number of false positives without asking the developers' opinion about the recommendations, it is certain that the developers would like to inspect the smallest possible list of *mixin* opportunities that contains most of the relevant ones.

Therefore, we investigated whether it is possible to reduce the number of generated *mixin* opportunities (i.e., recommendations) without jeopardizing recall. To achieve this, we filtered out the *mixin* opportunities having a number of style declarations, or parameters above certain thresholds. The threshold values were automatically derived from the box plot *upper adjacent values* for the oracle used in RQ2. All data points above the upper adjacent value of the box plots are *outliers* that correspond to abnormal *mixins* introduced by developers. Indeed, defining thresholds based on box plot outliers is a statistical approach that has been also used in metric-based rules for detecting design flaws in object-oriented systems [27]. Table 4 shows the number of *mixin* opportunities obtained using different filters along with the number of recovered *mixins* from the oracle. The results show that it is possible to recover close to 90% of the oracle *mixins* with less than half of the original opportunities by applying appropriate threshold-based filters.

4.4 Threats to Validity

While the proposed approach for detecting *mixin* opportunities is preprocessor-agnostic, our actual implementation

Table 4: Threshold-based filtering of opportunities

	Filter	#Opp.	#Recovered	Recall (%)
I	None	9585	189	97.9
II	#Declarations ≤ 7	8686	180	93.3
III	#Parameters ≤ 2	4421	176	91.2
IV	II & III	4320	169	87.6

for the introduction of *mixins* currently supports only the Less preprocessor. This is because the source code transformations required to introduce a *mixin* are specific to the *abstract syntactic structure* of the targeted preprocessor. We selected to support Less, because it is slightly more popular among the developers [9]; however, the implemented tool is extensible enough to support any preprocessor. In addition, we have already shown that [29] code written in Sass (another popular preprocessor) has very similar characteristics with code written in Less. Thus, examining projects developed in Less does not limit the generalizability of our study. Nevertheless, we used Less files collected from a wide range of web systems, including libraries and websites, to mitigate the threat to the external validity of our study.

The ultimate approach for testing presentation preservation would be to compare the target documents, before and after applying migration transformations, as they are rendered in the browser. However, a visual comparison would be time-consuming and error-prone. Additionally, the state-of-the-art automatic techniques are computationally intensive, e.g., differentiating screen captures of web pages using image processing methods [7, 25, 36]. We instead considered all possible presentation changes a *mixin* can impose on a Style Sheet, and developed a lightweight static analysis method, based on preconditions derived from CSS specifications. This approach was able to reveal several styling bugs due to our faulty implementation, showing that the method is promising in testing whether presentation is preserved.

5. RELATED WORK

Maintenance of Cascading Style Sheets: CSS maintenance is a rather unexplored research area, despite the fact that CSS is extensively used in the industry [32]. Previous works proposed approaches for finding dead code in CSS files, in order to reduce their size, and thus the effort required for their maintenance. Mesbah and Mirshokrae devised a hybrid approach to detect unused style rules in CSS files, using dynamic and static analysis [31]. Having the same goal, Genevès et al. [15] proposed a technique based on tree logics. Similarly, Hague et al. [17] used Tree Rewriting techniques for removing redundant CSS rules. Finally, Bosch et al. [4] explored the possibility of CSS refactoring to remove unused style rules and declarations by statically extracting the relationships between CSS style rules.

Keller and Nussbaumer [19] introduced metrics for measuring the abstractness of CSS files, and explored ways to make CSS files reusable for different target documents. Gharachorlu [16] investigated code smells in CSS code, and proposed a statistical model for predicting them. Liang et al. [24] designed a tool for tracking the visual impact of code changes in CSS across a website.

Previous studies also investigated the presence of duplication in web systems, either in the content [3] and structure [11, 12, 10, 35, 38, 8] of HTML pages, or their client- [22, 5] or server-side [33] scripts. We conducted the first study on CSS duplication [30], investigating the possibility of refactoring CSS files to remove duplication by *grouping* style rules that share *equivalent* declarations, i.e., declarations that result to the same presentation. The proposed approach in [30] eliminates duplications occurring between style declarations with *no differences* in style values, *directly in "vanilla"* CSS, while in this work, we proposed a technique for *migrating* CSS code to take advantage of *preprocessor mixins*, which allows the parameterization of differences in style values. Moreover, in contrast to this work, we did not use a testing approach in [30] for assuring that the presentation semantics of style sheets will be preserved after transformations.

Migration of legacy systems: There are numerous works in the literature proposing migration techniques for legacy systems in order to improve their maintainability. Several researchers developed techniques for migrating procedural code to the object-oriented paradigm, such as automatic or semi-automatic translators from C to C++ [44], Eiffel [40], or Java [28]. Migration is also performed when there is a lack of human resources for maintaining existing software systems written in an extinct language, e.g., migrating Lisp to Java [23]. Other works proposed approaches for detecting opportunities to use constructs introduced in a newer version of a programming language. For Java, there are techniques for introducing parameterized classes from non-generic ones [21], the enumerated type [20], and Lambda expressions [14].

Migration of web systems: Some of the studies that investigated the duplication in the content or structure of web pages, proposed techniques for migrating duplicated static web pages to dynamic, server-side web applications [3, 38]. Mao et al. [26] proposed an approach for the automatic migration of HTML pages having table-based structures to the style-based structure, by using clone detection tools to find duplicated code across different CSS files. Their approach can support only exact duplications, which are removed by creating a single CSS file for different web pages. Our work is the first one that supports the migration of existing CSS code to preprocessors and evaluates the correctness and efficacy of the detected *mixin* migration opportunities.

6. CONCLUSIONS AND FUTURE WORK

In summary, the main conclusions and lessons learned are:

1. Our approach facilitates the automatic migration of CSS code to preprocessors, by safely extracting duplicated style declarations from CSS code to preprocessor *mixins*.
2. Our approach is able to recover the vast majority (98%) of the *mixins* that professional developers introduced in websites and Style Sheet libraries.
3. We found that developers mostly under-utilize *mixins* (i.e., they could reuse the *mixins* in more style rules, and/or could eliminate more duplicated style declarations by extracting them into the *mixins*).
4. By applying appropriate threshold-based filters, it is possible to drastically reduce the number of detected *mixin* opportunities without affecting significantly the recall.

As future work, we are planning to develop a ranking mechanism in order to help developers prioritize the *mixin* opportunities based on their expected benefit on maintainability. We also plan to evaluate our recommendation system with actual web developers and find ways to further eliminate recommendations that are irrelevant to them, e.g., *mixins* containing semantically unrelated style properties that a developer would not normally group together.

7. REFERENCES

[1] Dataset and tool for reproduction and replication. https://git.io/v6eK6.

[2] CSS Cascading and Inheritance Level 3. Technical report, World Wide Web Consortium, October 2013.

[3] C. Boldyreff and R. Kewish. Reverse engineering to achieve maintainable WWW sites. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, pages 249–257, 2001.

[4] M. Bosch, P. Genevès, and N. Layaïda. Automated refactoring for size reduction of css style sheets. In *Proceedings of the 2014 ACM Symposium on Document Engineering (DocEng)*, pages 13–16, 2014.

[5] 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.

[6] H. Catlin. SASS: Syntactically Awesome Style Sheets. http://sass-lang.com/.

[7] S. R. Choudhary, H. Versee, and A. Orso. WEBDIFF: Automated identification of cross-browser issues in web applications. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

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

[9] C. Coyier. Popularity of CSS Preprocessors. http://css-tricks.com/poll-results-popularity-of-css-preprocessors/.

[10] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Understanding cloned patterns in web applications. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC)*, pages 333–336, 2005.

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

[12] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 481–486, 2002.

[13] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. 1999.

[14] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. LAMBDAFICATOR: From Imperative to Functional Programming through Automated Refactoring. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 1286–1289, 2013.

[15] P. Genevès, N. Layaïda, and V. Quint. On the analysis of cascading style sheets. In *Proceedings of the 21st International Conference on World Wide Web (WWW)*, pages 809–818, 2012.

[16] G. Gharachorlu. *Code smells in Cascading Style Sheets: an empirical study and a predictive model*. Master's thesis, University of British Columbia, 2014.

[17] M. Hague, A. W. Lin, and C.-H. L. Ong. Detecting Redundant CSS Rules in HTML5 Applications: A Tree Rewriting Approach. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 2015.

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

[19] M. Keller and M. Nussbaumer. CSS code quality: a metric for abstractness; or why humans beat machines in CSS coding. In *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 116–121, 2010.

[20] R. Khatchadourian, J. Sawin, and A. Rountev. Automated Refactoring of Legacy Java Software to Enumerated Types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 224–233, 2007.

[21] A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing java classes. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 437–446, 2007.

[22] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 379–386, 2003.

[23] A. Leitao. Migration of Common Lisp Programs to the Java Platform - The Linj Approach. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 243–251, 2007.

[24] 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 (UIST)*, pages 353–356, 2013.

[25] S. Mahajan and W. G. J. Halfond. WebSee: A Tool for Debugging HTML Presentation Failures. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, 2015.

[26] 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. *Proceedings of the 17th Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 12–26, 2007.

[27] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

[28] J. Martin and H. Muller. Strategies for migration from C to Java. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR)*, pages 200–209, 2001.

[29] 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 (SANER)*,

pages 168–178, 2016.

[30] 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 (FSE)*, pages 496–506, 2014.

[31] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 408–418, 2012.

[32] Mozilla Developer Network. Web developer survey research. Technical report, Mozilla, 2010.

[33] 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 *Proceedings of the 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6, 2013.

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

[35] D. C. Rajapakse and S. Jarzabek. An investigation of cloning in web applications. In *Proceedings of the 5th International Conference of Web Engineering (ICWE)*, pages 252–262, 2005.

[36] S. Roy Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 702–711, 2013.

[37] A. Sellier. LESS - The dynamic stylesheet language. http://lesscss.org/.

[38] N. Synytskyy, J. R. Cordy, and T. R. Dean. Resolution of static clones in dynamic Web pages. In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution (WSE)*, pages 49–56, 2003.

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

[40] M. Trudel, C. Furia, M. Nordio, B. Meyer, and M. Oriol. C to O-O Translation: Beyond the Easy Stuff. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 19–28, Oct 2012.

[41] U.S. General Services Administration. CSS coding styleguide. https://pages.18f.gov/frontend/css-coding-styleguide/preprocessor/.

[42] W3Techs. World Wide Web Technology Surveys. http://w3techs.com/technologies/details/ce-css/all/all.

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

[44] Y. Zou and K. Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference (APSEC)*, pages 390–399, 2001.