Automated and Semantics-Preserving CSS Refactoring

Marti Bosch, Pierre Genevès, Nabil Layaïda

Automated and Semantics-Preserving CSS Refactoring

Martí Bosch
UPC & Inria
marti.bosch@estudiant.upc.edu

Pierre Genevès
CNRS
pierre.geneves@cnrs.fr

Nabil Layaïda*
Inria
nabil.layaida@inria.fr

ABSTRACT

The Cascading Style Sheets (CSS) language constitutes a key component of web applications. It offers a series of sophisticated features to stylize web pages. Its apparent simplicity and power are however counter-balanced by the difficulty of debugging and maintaining style sheets, tasks for which developers still lack appropriate tools. In particular, significant portions of CSS code become either useless or redundant, and tend to accumulate over time. The situation becomes even worse if we consider that more and more complex features are added to the CSS language (e.g. CSS3 powerful selectors). A direct consequence is a waste of resources such as bandwidth and CPU that are required to display web pages. Since each web page access may involve CSS code transfer and interpretation, the cost of ill-designed CSS code represents a significant amount of useless traffic and processing at web scale.

Style sheets are designed to operate on a class of documents (possibly generated). However, existing techniques consist in syntax validators, optimizers and runtime debuggers that operate for one particular document instance. As such, they do not provide guarantees concerning all web pages in CSS refactoring, such as preservation of the formatting. This is partly because they are essentially syntactic and do not take advantage of CSS semantics to detect redundancies.

We propose a set of automated refactoring techniques aimed at removing redundant and inaccessible declarations and rules, without affecting the layout of any document to which the style sheet is applied. We implemented a prototype of our analyses that has been extensively tested with popular web sites (such as Google Sites, CNN, Apple, etc.). We show that significant size reduction can be obtained while preserving the code readability and improving maintainability.

* Detailed affiliation of authors: M. Bosch[1234], P. Genevès[213], N. Layaïda[123] with: [1]Inria; [2]CNRS, LIG; [3]Univ. Grenoble Alpes; [4]Universitat Politècnica de Catalunya. Work partly supported by ANR project Typex (ANR-11-BS02-007).

Categories and Subject Descriptors

1.7 [**Document and Text Processing**]: Document preparation; D.2 [**Software Engineering**]: Testing and Debugging

Keywords

Web development, Style sheets, CSS, Debugging

1. INTRODUCTION

Cascading Style Sheets (CSS) [6, 14] is a style sheet language used to format documents written in markup languages, and nowadays it is widely used to style web pages. To suit the current context, where there is so much attention paid to the user experience in different kinds of devices, it includes a series of sophisticated features that offer a very wide range of possibilities to designers and developers.

Despite its widespread use and increasingly important role, CSS received very little attention from the research community [22, 15] with the notable exceptions of [7, 17]. Developers tend to be misled by the simplicity of its syntax, which hides the complexity of its most advanced aspects. This often results in a code that is very hard to maintain. This also leads to redundant declarations and inaccessible selectors, that unnecessarily (1) increase resources required to transfer the CSS files (and therefore web traffic at a global scale) as well as (2) slow down the browser rendering of the page layout [18] with redundant or useless computations, in particular, on mobile devices. Current syntax optimizers[1] perform syntactic analyses, while being unaware of the CSS semantics. As a result, they only partly solve (1), while losing the style sheet's readability. Recent CSS preprocessors[2] offer more advanced ways to deal with (1) but they accentuate (2), which is an issue nowadays, as handheld devices suffer especially from (2) due to their limited CPU and battery resources.

The use of empirical methods such as runtime debuggers is a common practice. While debuggers are useful in testing, they depend strongly on the document instance on which the style sheet is applied. This can induce unintended inconsistencies, as style sheets usually apply to a wider set of documents. Modifications achieved on a particular instance might undesirably alter the presentation the other

[1]Several tools available on line: http://www.cleancss.com/, http://www.codebeautifier.com/, http://csslint.net/, etc.
[2]Less http://lesscss.org/, Saas http://sass-lang.com/

documents. On the other hand, CSS code very often comes from different sources, such as external files, declarations stated under the HTML's `style` element, or inline styles set directly as attributes on specific elements. This makes it hard to spot the origin of bugs, and makes the developers spend significantly long dynamic debugging sessions in the browser to fix them.

Contributions. We propose a set of novel automated refactoring techniques aimed at removing redundant and inaccessible CSS declarations and rules, while preserving the layout of the documents to which the style sheet is applied. Our analyses take advantage of the semantics of CSS, and can be used in conjunction with other existing syntactic compressors to achieve additional size reduction without losing readability. Our goal consists in both reducing resources required for web traffic at a global scale, and saving device resources by making browser rendering more efficient. We implemented a prototype of our analyses and extensively tested them on popular web sites (such as Google Sites, CNN, Apple, etc.). We show that significant size reduction can be obtained while preserving the code readability and improving maintainability.

Outline. We first introduce the CSS language in Section 2 while emphasizing its main principles and most important features supported by our analyses. We then present a novel refactoring method in Section 3. We describe the main implementation techniques that we used for developing a full prototype in Section 4. We report on experimental results that we obtained on popular websites in Section 5. We finally review related works in Section 6 before concluding in Section 7.

2. THE CSS LANGUAGE

"Cascading Style Sheets was the first feature that was added to the initial foundations of the web (HTML, HTTP, and URLs)" [7]. It plays nowadays a key role in the web infrastructure and in enhancing the user experience. The simplicity of its syntax has attracted not only developers, but also graphical and web designers. CSS leverages on the principle of separation of content from presentation and a few rules are capable of producing impressively fancy presentations. A style sheet C can be seen as a sequence of rules, where each rule R consists of a selector S, and a set of declarations called declaration blocks. Selectors identify the set of elements that are styled by the declarations D_i. Each declaration D_i is a pair of a property P_i and its associated value V_i, that define how the elements selected by S will look like in the browser. The syntactic form of a CSS rule is illustrated in Figure 1.

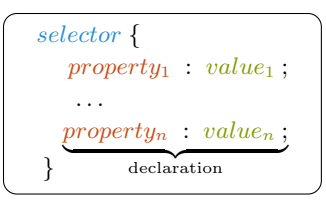

Figure 1: Syntax of a CSS rule.

2.1 Selectors

CSS selectors basically consist in a sequence of *simple selectors* separated by *combinators*. We review them below.

A **universal selector** matches any single element in the document tree. It is noted by an asterisk character `*`. If it is not the only component of a selector, it may be omitted. For example "`*.menu`" may be noted as "`.menu`".

A **type selector** matches all the instances of the element type. For example "`h1`" matches all `h1` elements in the document tree.

Together with *simple selectors*, CSS permits as well the styling of HTML elements based on the *attributes* that they carry.

Attribute selectors can match elements in three main ways: (1) presence of the `foo` attribute "`[foo]`", (2) presence of the `foo` attribute with an exact value `bar`, "`[foo="bar"]`" or (3) certain substring matching in attribute values.

Class selectors might replace attribute selectors for dealing with the `class` attribute. They start with a dot, and are followed by the `class` attribute value. For example "`[class="foo"]`" might be rewritten as "`.foo`".

Id selectors select elements with an `id` attribute set to a certain value. In this case, the selector contains a sharp "`#`" symbol followed by the `id` attribute value. As an example, "`[id="bar"]`" might be replaced by "`#bar`".

To generate more complex navigational patterns, *combinators* describe structural relations between elements targeted by selectors.

A **descendant combinator** describes a descendant relationship between two elements, made by the whitespace sign, such as "`body p`".

A **child combinator** expresses a parent to child relationship between two elements. In this case, a greater than sign is used, for example "`body > p`".

A **sibling combinator** describes either an adjacent or a general sibling relationship, respectively denoted by a plus and tilde characters, for example "`p + ul`" and "`p` $\sim$ `ul`".

A *single selector* is any pattern that can be created by the combination of *simple selectors* with *combinators*.

When the same declarations are shared between different selectors, they might be *grouped* into a comma-separated list of selectors meant to reduce the code size. For example, the rules "`h1 { color: #666 }`" and "`h2 { color: #666 }`" can be merged into one equivalent rule, written as "`h1, h2 { color: #666 }`".

In more recent versions, CSS also includes a series of features that empower the language's styling capabilities, such as *pseudo-classes*, that permit the selection of an element based on information that cannot be expressed using the other selectors. They consist on a colon ":" followed by the pseudo-class name and optional value, for example "`a:nth-child(2)`" matches only an `a` element that is the second child of a given parent element. On the other hand, *pseudo-*

elements work like pseudo-classes, but they focus on styling only particular parts. They can start with a colon ":" or a double ":" colon depending on the specification. For example, "`p::first-line`" selects only the first line of the paragraph element `p`.

2.2 Cascading and Specificity

A given element might be affected by several CSS rules, depending on their respective selectors. For such elements, if the matching rules assign different values to the same element's property, CSS uses the *cascading* algorithm to identify which one will apply. The algorithm first determines the origin of the declarations that assign different values to the same element's property. There are three possible origins for a declaration, that, in decreasing priority order, are:

1. User: the user can define a custom style sheet to configure its web experience on all pages.

2. Author: most of the time, the styling that we see is the one that the developer (author) has set for that document.

3. User-Agent: each browser has a basic style sheet that provides a default presentation for any document.

If the *cascading* algorithm detects that the conflicting declarations have the same origin, it gives higher priority to the declarations that have the `!important` specifier. In case of equality, CSS will pick the declaration whose selector has a highest *specificity*. The *specificity* of a selector is represented by a four integer vector. In Selectors Level 3, [5] their components are, from most to less important, determined as follows: (1) whether the property is declared directly under the `style` attribute or not, (2) the number of references to the `id` attribute with the `#` symbol, (3) the number of class selectors, attribute selectors and pseudo-classes, and (4) the number of element type selectors and pseudo-elements. In case of selectors with the same specificity, the latter rule in the syntactic order gets precedence.

2.3 Inheritance

A CSS element with certain value set to a given property, might propagate *recursively* this styling to its descendant elements. This mechanism is called *inheritance*, and helps avoiding to write repetitious declarations. For a specific property, the value can be set to `inherit` to use it's parent (or other ancestor) value. In fact, an important set of properties have the initial value set to `inherit`, as for example `color`. Other properties where this behaviour is not desirable have different initial values set, like the `border` property.

2.4 Media Rules

A *media rule* starts by `@media` followed by a condition called *media query*. This defines a block that is enclosed by curly braces, and in which we can add a set of CSS rules that will only apply when the *media query* is satisfied. The *media queries* define the style sheet target according to *expressions* concerning devices' features, such as the color or dimensions. As an example, in Listing 1, the `text-indent` property will only apply for `p` elements if the device has a width greater or equal than 800 pixels. Such features are key for defining how web pages will look like on mobile devices.

Listing 1: Example of @media rule

```
1 @media (min-width:  800px) {
2   p {
3     text-indent: 5px
4   }
5 }
```

To adjust more precisely the style sheet scope, *media queries* might use four different logical operators to connect different *expressions*, which are: (1) *conjunction:* represented with the keyword `and`, (2) *disjunction:* stated with a comma `,` character, (3) *negation:* using the keyword `not`, and (4) *exclusivity:* with the keyword `only`. For example, "`(min-width: 480px) and hanheld`" would require the device to have a width greater or equal than 480 pixels *and* be handheld. It is also important to be aware of the fact that media queries do not alter the specificity.

3. CSS REFACTORING TECHNIQUES

In this section, we introduce series of refactoring techniques capable of identifying and deleting unaccessible declarations and combine rules in a way such that the same rendering semantics is preserved with lighter CSS files. These techniques do not require any information other than the style sheet itself, regardless of which instance is under consideration. For this purpose, our methods rely on the semantics of the CSS components described previously. In particular, our method aim at analysing and leveraging the fact that in CSS, rules use selectors that match the set of elements that are styled by the rule's declarations. Our main method consists in the detection of semantic relations between CSS selectors. When some of these relations are detected, further analysis of some CSS aspects might reveal that some declarations are in fact unnecessary and need to be deleted.

3.1 Containment Between Selectors

One fundamental relation between two selectors is the *containment* relation. For example we say that "`ul > li`" is *contained* in "`li`" since any "`li`" element with an "`ul`" parent is indeed a "`li`" element. Structural pseudo-classes and the set of *attributes* carried by some element might as well induce containment relations between selectors. A selector such as "`p.someclass`" is contained into "`p`", as any "`p`" element with "`class`" attribute "`someclass`" is indeed a "`p`" element. On the other hand, *grouping* selectors can also carry containment relations, as for instance "`p`" is contained in "`p, span`". These three types of containment relations are relatively straightforward, but they can be combined and form more sophisticated containment relations in general, such as "`#foo ul.bar li a, #foo ol.bar li a`" $\subset$ "`#foo a`". Containment relates sets of specific elements that are associated or related with selectors. Given two selectors S_b and S_p, S_b is contained in S_p iff any element pointed by S_b is also pointed by S_p. This section treats only *proper containment*, to clearly separate *containment* from *equivalence*.

3.1.1 Refactoring Declaration Blocks

Once a containment relation has been identified between two selectors, refactorings will only be possible if they have at least one property in common (declared under both selectors). Considering the selectors S_b and S_p, where $S_b \subset S_p$,

with their respective sets of property declarations P_b and P_p, let us use $P_{b\wedge p}$ to denote the set of properties declared under both S_b and S_p, regardless of their values. In the containment context, refactorings require the existence of a non empty $P_{b\wedge p}$ set. If this condition is satisfied, further processing is needed depending on the specificities of the selectors. This leads to two possible refactoring procedures.

Procedure 1: Subset S_b more specific than superset S_p

foreach P_i *in* $P_{b\wedge p}$ **do**
  if P_i *has same value under* S_b *and* S_p **then**
    delete P_i statement from S_b
  end
end

Procedure 1 deletes from the subset S_b the property statements that are defined under both rules $P_{b\wedge p}$ *only if they share the same value*, as the value will already be defined on the superset S_p.

Procedure 2: Superset S_p more specific than subset S_b

foreach P_i *in* $P_{b\wedge p}$ **do**
  delete P_i statement from S_b
end

In Procedure 2, *all declarations* from $P_{b\wedge p}$ are deleted from S_b since values from the subset are *always* overridden by those defined in the superset S_p. It is clear that in general, we can only safely delete property declarations from the subset S_b, as S_b is overwhelmed by the superset S_p, but not the other way around, given the nature of the containment relation.

3.1.2 Example

Consider the CSS code shown in Listing 2. There are two rules such that there exists a containment relation between selectors: "`ul li > a.foo`" $\subset$ "`ul a`". Any `a` element with the `class` attribute set to "`foo`", and with a `li` parent who has a `ul` ancestor also matches the pattern of an `a` element with a `ul` ancestor. The subset selector "`ul li > a.foo`" is more specific than its superset, so under these conditions Procedure 1 can be safely applied.

Listing 2: Input for Containment Example

```
1  ul li > a.foo {
2    color: blue;
3    text-decoration: none;
4    font-weight: bold
5  }
6
7  ul a {
8    color: blue;
9    text-decoration: underline
10 }
```

In this case, `color` and `text-decoration` are declared under both selectors. When one element is selected by the subset (and implicitly by the superset as well), the value for `text-decoration` will be `none`, as "`ul li > a.foo`" is more specific. Consequently we cannot delete this declaration. However, for the same element, `color` is set to `blue` by both selectors, so the subset "`ul li > a.foo`" is not providing any additional style information, so it can be deleted while preserving Listing 2's semantics. As a consequence,the same formatting can be achieved with the lighter code shown in Listing 3.

Listing 3: Output for Containment Example

```
1  ul li > a.foo {
2    text-decoration: none;
3    font-weight: bold
4  }
5
6  ul a {
7    color: blue;
8    text-decoration: underline
9  }
```

3.2 Equivalence Between Selectors

An *equivalence* relation between two or more CSS selectors may also lead to some refactorings. In CSS, this may occur in several situations. First, it is very common to find more than one rule in the same file with the same selector. Some other times, we find a class selector "`.foo`" and also its corresponding attribute selector "`[class="foo"]`", or the id selector variant. Basic settings can be dealt with using some basic string processing. A complete semantic analysis such the one we are using here allow to detect more complex equivalences such as the one that exists between selectors "`p:nth-child(odd), p:nth-child(even)`" and "`p`", as every paragraph is either odd or even. In general, we define two selectors S_i and S_j as being *equivalent* iff any element pointed by S_i is also pointed by S_j and vice versa, no matter what are the syntactic forms used in selectors S_i and S_j.

3.2.1 Refactoring Declaration Blocks

Once an equivalence relation is detected between two selectors S_i and S_j, a deletion of declarations can achieved provided the existence of a non-empty set $P_{i\wedge j}$. The properties can be deleted in any of the two rules associated with the two selectors. It is the selector's specificity that will determine which statements are redundant. When S_i has higher specificity than S_j, for $P_{i\wedge j}$, the values set under S_i would then prevail, and since $S_i \Leftrightarrow S_j$, values set under S_j never apply. This yields refactoring procedure 3:

Procedure 3: $S_i \Leftrightarrow S_j$ with S_i more specific than S_j

foreach P_k *in* $P_{i\wedge j}$ **do**
  Delete P_k statement from S_j
end

3.2.2 Example

Listing 4 illustrates a case of two equivalent selectors. The grouped selector in line 7 is composed by three single selectors. Note that the first two "`div#nav a`", "`div#nav a.bar`" are both subsets of the third one "`div a`", so any element matched by their two first selectors will also be matched by the third. Therefore, the grouped selector only points to the set of elements matched by its third sub selector (this is equivalent to the selector in line 1). This results in the relation "`div a`" $\Leftrightarrow$ "`div#nav a, div#nav a.bar, div a`", as in fact both selectors point to a `a` element with a `div` ancestor.

From the CSS semantics point of view, the difference among the selectors in lines 1 and 7 is that the latter is *more specific*. Specificity cannot be determined for grouped selectors, but in this case each single selector that composes the group is *more specific* than the selector in line 1, so for any element matched by both "`div a`" and "`div#nav a, div#nav a.bar, div a`", the declarations from the grouped selector is chosen first. More details for the precedence rules determination are covered in Section 4.3.

Listing 4: Input for Equivalence Example

```
1  div a {
2    font-size: 14px;
3    border: 1px solid;
4    background-color: #fff
5  }
6
7  div#nav a, div#nav a.bar, div a {
8    font-size: 14px;
9    border: 1px dotted
10 }
```

Under these circumstances Procedure 3 applies with the "`div a`" as the less specific selector. Consequently we can remove "`font-size: 14px`" and "`border: 1px solid`" as such declarations will always be overridden by the ones stated under "`div#nav a, div#nav a.bar, div a`". As a result, they are unnecessary as they are always unreachable. The refactored code is shown in Listing 5, and obviously preserves the semantics from Listing 5.

Listing 5: Output for Equivalence Example

```
1  div a {
2    background-color: #fff
3  }
4
5  div#nav a, div#nav a.bar, div a {
6    font-size: 14px;
7    border: 1px dotted
8  }
```

3.3 Relations and Media Rules

The logic operators can now be applied to media queries described in Section 2.4 to detect certain inconsistencies. For example, consider Listing 6.

Listing 6: Relations and Media Rules

```
1  @media (min-width: 1000px) {
2    div { color: red }
3  }
4
5  @media (min-width: 800px) {
6    div { color: blue }
7  }
```

Observe that a device that satisfies the first media query will always satisfy the second one as well. And considering that media rules do not alter selector's specificity, in this case "`color: red`" will never be applied. Note that, if we reverse the order of the media rules, "`color: blue`" will be active for devices wider than `800px` but smaller than `1000px`, so the color of the `div` elements would be blue. One can reasonably guess that this was the behaviour that the developer intended.

This kind of inconsistencies can be detected too. Given two media rules, they will either:

a) Have no relation. For example "`(min-color: 4)`" and "`(max-width: 600px)`" have no relation whatsoever.

b) Have a *containment* relation, like in the example "`tv`" ⊂ "`tv and (scan: progressive)`"

c) Have an *equivalence* relation, like those introduced by patterns such as "`a, b and c`" ⇔ "`(a,b) and (a,c)`", being `a`, `b` and `c` in any media query expression. Or we might find the same media queries under different media rules, that would indeed be equivalent.

In order to perform the refactoring procedures proposed in Section 3.1 and Section 3.2 on media rules, we have to see how selectors' relations change according to their respective media context. Given two selectors S_a and S_b, and their respective media contexts represented by the media queries mQ_a and mQ_b, Table 1 describes all the possible cases.

	$mQ_a \subset mQ_b$	$mQ_a \supset mQ_b$
$S_a \subset S_b$	$S_a \subset S_b$	~~$S_a \subset S_b$~~
$S_a \supset S_b$	~~$S_a \supset S_b$~~	$S_a \supset S_b$
$S_a \Leftrightarrow S_b$	$S_a \subset S_b$	$S_a \supset S_b$

Table 1: Relations considering selectors' media queries.

The white cells correspond to cases where the relation among S_a and S_b does not change if their media contexts are considered. This would be the case for a selector S_a = "`a.foo`" under a media rule with the media query mQ_a = "`screen`" and S_b = "`a`" with mQ_b = "`screen, handheld`". Not only "`a.foo`" ⊂ "`a`", also "`a.foo`" only applies to a `screen` media, while "`a`" applies to both a `screen` and `handheld` media.

Now let S_a = "`div p`" with mQ_a = "`all`" and S_b = "`p`" with mQ_b = "`tv`". Note that "`div p`" ⊂ "`p`", but "`div p`" applies in `all` media contexts whereas "`p`" is only active for `tv` media types. So whenever we are not under a `tv` device, "`div p`" might point to elements that "`p`" would not, and thus it breaks the nature of the containment relation, and preventing our tool from performing the refactorings proposed in Section 3.1, which become no longer safe to achieve.

To conclude, consider again Listing 6. See how we have two identical selectors S_a, S_b = "`div`", in two different media contexts, mQ_a = "`(min-width: 1000px)`" and mQ_b = "`(min-widith: 800px)`", being true that $mQ_a \subset mQ_b$. Under these circumstances, any device satisfying mQ_a will also satisfy mQ_b, but not the other way around. So whenever S_a selects all the `div` elements, S_b will do too, but not conversely. In short, an equivalence relation is turned into containment, as described in the last row of Table 1.

Once Table 1 is taken into account, the refactoring procedures detailed for *containment* and *equivalence* between selectors can be carried out while considering media rules too.

3.4 Deleting Empty Rules

The procedures above are meant to delete declarations from CSS rules. Applying such procedures might result in an *empty* rule containing only a selector but with no declarations. This rule does selects a set of elements but does not define any styling for them and does not affect any other rule application. Consequently, it is safely deleted.

3.5 Merging Rules with Equivalent Selectors

Given two equivalent selectors S_a and S_b, a refactoring procedure that can reduce the style sheet's size consists in merging their respective rules R_a and R_b. By merging R_a and R_b we mean the following: as S_a and S_b affect the exact same set of elements, we group the declarations of both rules. Let S_a be shorter than S_b, then we would keep S_a (the shortest), and move all the declarations from R_b to R_a. If any declaration concerns the same property, and has different values set under R_a and R_b, we would keep the declaration set under the *most specific* of the two selectors, however Section 3.2 already takes care of this.

To exemplify the procedure, let's consider Listing 7. We have two equivalent selectors "`div#nav`" ⇔ "`div[id="nav"]`", so let's keep the rule with shortest one "`div#nav`", and move the declarations "`margin: 5px`" and "`text-align: left`" from "`div[id="nav"]`" to "`div#nav`". Instead of using two rules with 2 declarations each, after this refactoring we would use one rule with 4 declarations.

Listing 7: Merging Rules with equivalent selectors

```
1  div#nav {
2     padding: 0px;
3     text-indent: 2px
4  }
5
6  div[id="nav"] {
7     margin: 5px;
8     text-align: left
9  }
10
11 div div[id="nav"] {
12    margin: 10px
13 }
```

However, we cannot perform this refactoring as *it does not always preserve the style sheet's semantics*. The reason is because selectors confer its specificity to the declarations that they encapsulate. For example, in Listing 7, when we move "`margin: 5px`" from "`div[id="nav"]`" to "`div#nav`", if some element is pointed by "`div div[id="nav"]`", the declaration "`margin: 10px`" will always be inactive as "`div div[id="nav"]`" ⊂ "`div#nav`", and "`div#nav`" will confer its *higher* specificity to its declaration "`margin: 5px`", and so it would override "`margin: 10px`". Note that if we do not refactor Listing 7, "`margin: 5px`" has the specificity from its selector "`div[id="nav"]`", which is lower than the one from "`div div[id="nav"]`". Consequently, for the set of elements affected by "`div div[id="nav"]`", `margin` will be `10px`.

Applicability of the Procedure. In order to perform this refactoring without affecting the style sheet's semantics, we have to define its *applicability*. To determine when a pair of rules R_a and R_b with equivalent selectors $S_a \Leftrightarrow S_b$ can be merged, we need to guarantee that there is no selector S_c or S_d in the style sheet such that:

(i) $S_c \subset S_a$ with S_c *more specific* than S_a, and S_c *less specific* than S_b.

(ii) $S_d \supset S_a$ with S_d *more specific* than S_a, and S_d *less specific* than S_b.

Note that as $S_a \Leftrightarrow S_b$, the relation $S_c \subset S_a$ already implies that $S_c \subset S_b$, and the same applies to S_d. More generally, a pair of rules with equivalent selectors $S_a \Leftrightarrow S_b$ might be merged if *all* the selectors that hold a *containment* relationship with them, are either *more specific* than both S_a and S_b, or are *less specific* than both S_a and S_b. Observe that in Listing 7, the subset "`div div[id="nav"]`" is *more specific* than "`div[id="nav"]`" but *less specific* than "`div#nav`", and that is why the rules cannot be merged.

Nevertheless, if a rule R_x with a selector S_x fulfilling the conditions of (i) or (ii) was found, the rules R_a and R_b might still be merged only if the declarations we move among R_a and R_b *do not concern any of the properties* declared under R_x. Given the rules stated in Listing 7, there exists a problem in moving "`padding: 0px`" and "`text-indent: 2px`" from the rule with the selector "`div#nav`" to the one with "`div[id="nav"]`". Then we can delete the resulting empty rule "`div#nav { }`". In realistic style sheets, a much larger set of rules need to be considered in order to determine if this refactoring is valid or not. For our example, Listing 7, after merging "`div#nav`" and "`div[id="nav"]` with the right procedure, the smaller style sheet preserving the semantics is represented in Listing 8.

Listing 8: After Merging Rules the right way

```
1  div[id="nav"] {
2     margin: 5px;
3     text-align: left;
4     padding: 0px;
5     text-indent: 2px
6  }
7
8  div div[id="nav"] {
9     margin: 10px
10 }
```

4. IMPLEMENTATION TECHNIQUES

4.1 Modeling & Analysis of Single Selectors

To detect relations between selectors, we build on and extend earlier works on the logical modelling of selectors described in [7]. We first translate selectors in a logic: each selector S_i is associated with a logical formula $\mathtt{F}(S_i)$ as described in [7], and then we use the logical satisfiability solver described in [8] to check for the existence of relations between logical formulas. For instance, the validity of a containment relation $S_i \subseteq S_j$ is checked by testing for the unsatisfiability of the negation of the logical implication $\mathtt{F}(S_i) \implies \mathtt{F}(S_j)$. Concretely, our prototype implementation issues external calls to the implementation of the logical solver of [8] for this purpose.

We review below the main extensions to the simple translation of selectors introduced in [7] as well as the developments that we introduced in our prototype implementation.

4.2 Grouped Selectors

A grouped selector is a list of $n > 1$ single selectors, separated by a comma character. The translation of a grouped selector S_g intro tree logic requires the translation of each one of the n single selectors. The comma character plays the role of a logical "`or`" between the different single selectors that compose the group, so the logical disjunction operator `|` will be used to connect the single selectors' translations, capturing this way the semantics of the grouped selector. Given a grouped selector S_g, composed by n single selectors $s_{g,i}$, we have:

$$\mathtt{F}(S_g) = \mathtt{F}(s_{g,1}) \mid \mathtt{F}(s_{g,2}) \mid \ \ldots \ \mid \mathtt{F}(s_{g,n})$$

4.3 Specificity and Grouped Selectors

Our tool considers static CSS files, so it operates on the *author* style sheets (described in Section 2.2). In this context the *specificity* determines which selector applies when more than one point to the same element, and in the case of a collision between selectors' specificities (several do have the same one), the one that appears the latest syntactically in the file is selected.

For a pair of single selectors, we calculate their *specificity* vectors, and compare their components. However, *specificity cannot be determined for grouped selectors.* Instead, for a pair composed by single selector S_s and a grouped one S_g, we check that S_s is either *more specific* or *less specific* than all the single selectors $s_{g,i}$ that compose the group S_g, and if so, S_s is considered *more specific* or *less specific* than S_g accordingly. Otherwise no refactoring will be possible. In the same way, if we have a pair of grouped selectors S_g and S_h, the same procedure is applied, but considering each $s_{g,i}$ against each $s_{h,j}$, and assuring that one is either always *more specific* or *less specific* than the other.

Nevertheless, when comparing specificities, not all the pairs $s_{g,i} : s_{h,j}$ are relevant to the global relation between S_g and S_h. When $s_{g,i}$ and $s_{h,j}$ represent *disjoint sets* ($s_{g,i} \wedge s_{h,j} = \varnothing$), there is no tree structure on which an element is pointed by both $s_{g,i}$ and $s_{h,j}$, so figuring out which one is more specific is not relevant. This means that we should only check that for *all the pairs that are not disjoint* ($s_{g,i} \wedge s_{h,j} \neq \varnothing$) $s_{g,i}$ is *always* either *more specific* or *less specific* than $s_{h,j}$ to conclude that S_g is respectively *more specific* or *less specific* than S_h. For example, for "`span.foo`" $\subset$ "`span, p#bar`" the fact that "`span.foo`" is *less specific* than "`p#bar`" is irrelevant, as there is no element that can be simultaneously matched by both selectors. Consequently, our tool will consider that "`span.foo`" is *more specific* than "`span, p#bar`" as "`span.foo`" is *more specific* than "`span`" and "`span.foo`" $\wedge$ "`p#bar`" $= \varnothing$.

4.4 Selectors and Media Rules

In our analysis, *every* selector will have a set of associated media types. For the selectors stated under a media rule, the set of associated media types is defined by the rule's media query. On the other hand, when a selector is not part of a media rule, it applies to the style sheet's *default* media type, which depends on the HTML specification. Nevertheless, when a CSS file is called under the `link` element, its `media` attribute can be stated with a media query as a value. If so, this media query will define the file's default media type.

4.5 Traversal of Selectors

Given certain style sheet with N rules, there will be N selectors S_i where $i = 1, 2, \ldots, N$. To detect all the possible relations between selectors, each S_i has to be tested for logical inclusion against the remaining $N-1$ selectors. This adds up to a total of $N \times (N-1)$ tests, which for relatively large style sheets (with more than 1000 rules) results in *millions* of tests. Since logical tests are the heaviest computation of our tool, several mechanisms can be used to avoid such explosion:

1. *If two selectors point to elements with different names, they will always represent disjoint sets*

2. *If a selector S_1 refers to one or more attributes that S_2 does not, S_1 will never contain S_2*

3. *If two selectors' translation into tree logic result in the same string, the selectors are equivalent*

4. *Given a pair of selectors such that $S_1 \Leftrightarrow S_2$, and a third selector S_3, if (a) $S_3 \subset S_1$, (b) $S_3 \supset S_1$ or (c) $S_3 \Leftrightarrow S_1$, then (a) $S_3 \subset S_2$, (b) $S_3 \supset S_2$ or (c) $S_3 \Leftrightarrow S_2$ due to the transitivity of equivalence and containment binary relations*

5. EXPERIMENTAL RESULTS

We now report on the experimental evaluation of our refactoring methods with real-world style sheets used in some of the most popular web sites.

5.1 Analysis of Single CSS Files

We have extracted the largest CSS file used in 20 different websites corresponding to various web applications types and having different complexity levels. They are presented in Table 2. For the sake of brevity, an identifier (integer) is used to identify each of them. The number of CSS rules is shown in this table. This number gives a clear idea of the style sheet's size and constitutes a relevant metric for the techniques described in this paper. The file sizes range from 10 to 320 Kilobytes.

ID	Web	# CSS Rules	ID	Web	# CSS Rules
1	ACM Digital Lib.	102	11	Iberia	729
2	Apple	784	12	Il Sole 24 Ore	833
3	Cambridge Univ.	845	13	Lamborghini	1472
4	CNN	2738	14	Microsoft	702
5	Coursera	3690	15	Opera	708
6	Ebay	1573	16	PayPal	1089
7	Facebook	2757	17	Salesforce	1158
8	Firenze Turismo	172	18	Shell	1111
9	Foundation[3]	800	19	WWW15	537
10	Google Sites[4]	1676	20	YouTube	1841

Table 2: Dataset for the experiments

In the remaining part of this Section, whenever global averages of percentages are calculated, we take the total deletions in all files and compare it to the total files' size. Our

[3]ZURB Foundation framework's default template
[4]Google Sites' default template

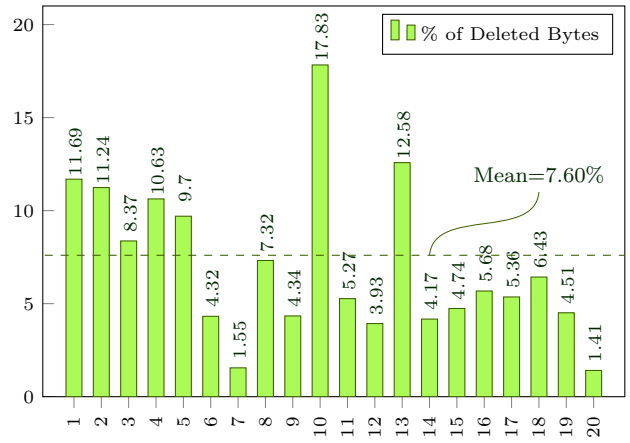

Figure 2: % of deleted bytes for each data set entry

tool prototype does not yet support all CSS3 selectors[5], so the percentages are determined *relative* to the part of the style sheet that our prototype supports.

For each file, the size reduction achieved by applying all the refactoring techniques from Section 3 is shown in Figure 2. We observe that the percentages of file size reduction in bytes vary dramatically among the style sheets. Our tool is capable of deleting up to the 17.83 % of a style sheet's bytes while preserving CSS semantics. There is a group of 5 CSS files, among which we find the style sheet form Apple, or Coursera, on which our tool reduces the size by 9.7 % and up to 12.58 %. For the majority of the items in the dataset, our tool can safely reduce the size between 4 % up to 8 %. Finally, for the CSS extracted from Facebook and YouTube, the tool can only reduce the size by 1.55 % and up to 1.41 % respectively.

For each one of the dataset entries, Figure 3 shows the percentage of deleted declarations, modified rules, and rules that became empty after analysis (with no declarations). On average, a 7.63 % of the declarations have been *safely* deleted, modifying a 13.70 % of the rules and leaving a 4.17 % of the total style sheet's rules empty. These refactoring are those leading to the size reduction shown in Figure 2.

Figure 4 illustrates the contribution of each refactoring introduced in Section 3.1. The deleted declarations will either be preceded by containment or equivalence between selectors, as explained in Sections 3.1 and 3.2 respectively. The containment test contributes to a 32.61 % of the size reduction in bytes, whereas the equivalence accounts for a 26.54 %. As naive as it might seem, deleting the rules that became *empty*, as proposed in Section 3.4, contributes to a 32.23 % of the size reduction. Merging rules as explained in Section 3.5 only contributes by an 8.62 % size reduction (in bytes). It is important to point out that the Media Rules are considered in this analysis, but they are in fact part of the containment and equivalence procedures, as explained in Section 3.3. An interesting conclusion that can be drawn from Figure 4 is that the effect of each procedure vary substantially from one case to the other in the dataset. For

[5]The tool supports 67.83 % of the selectors in the dataset shown in Table 2 (by November 2014)

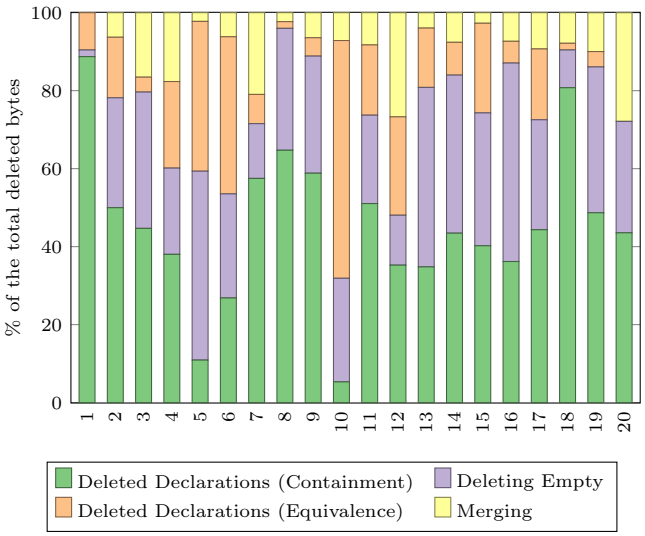

Figure 4: Each procedure's contribution to the total deleted bytes for each dataset entry

example, in some files the containment procedure accounts for up to 88 % of the total deletions, whereas in some others it only 5 %. The same is true for the other procedures. Real-world style sheets do not really share common development patterns but depend solely on the developers that builds them (this shows that there is no common practice in designing them from one site to the other). This illustrates the fact that CSS mechanisms remain generally unmastered by developers. We believe that this increases the relevance of automated tools like the one we propose here.

5.2 Style Sheets with Multiple Files

Modern web applications frequently use several CSS files to define their graphical presentation. Each file might correspond to the styling of distinct sections of the application, different parts of the same page, or simply some groupings which resembles libraries. Regardless of the specific instances of the HTML involved, our tool can achieve the refactoring procedures across several CSS files. In this case, the CSS cascade mechanism described in Section 2.2 defines which rule will apply when several rules from different files target the same element.

The information needed for the analysis is extracted from HTML documents as shown in Figure 5. It consists in exploring the document type declaration, and the `link` elements that are under the `head` element. For each `link` element with the attribute `rel="stylesheet"`, we retrieve the CSS file referenced by the `href` attribute, and if the `media` attribute is set, we extract its value. Such a value corresponds to a media query setting the style sheet's media type. If the `media` attribute is not set, then the style sheet's media type takes the default value as described in the specification: `all` in HTML5 [3] or `screen` in HTML 4 [10] and XHTML 1 [21]. This is why we need to extract the `!DOCTYPE` specification.

To analyze style sheets with different files, we have retained the web sites of our dataset whose main HTML page has

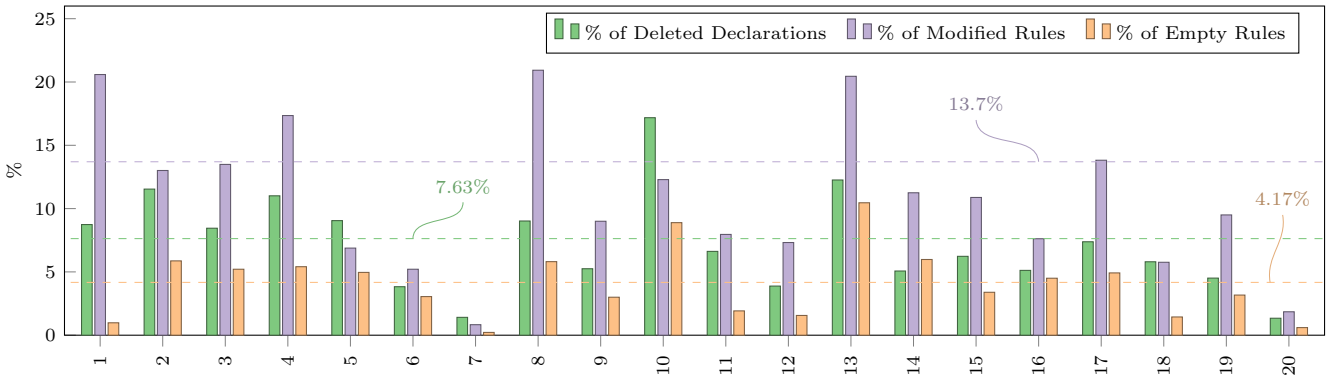

Figure 3: Modifications performed by our tool for each dataset entry.

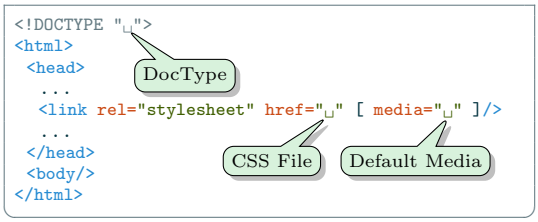

Figure 5: Information needed to analyze multiple CSS files.

more than one CSS file referenced in `link` elements. This new dataset is shown in Table 3, as well as the extra information extracted for this analysis and the total number of rules.

ID	Web	DocType	# CSS Files	Default Media	# CSS Rules
7	Facebook	HTML5	5	`all`	3543
10	Google Sites[4]	XHTML 1	6	`screen`	2046
11	Iberia	XHTML 1	6	`all`, `screen`[6]	3886
15	Opera	HTML5	2	`all`	890
16	PayPal	HTML5	3	`all`	1186
18	Shell	HTML5	6	`all`	1551
20	YouTube	HTML5	4	`all`	3615

Table 3: Dataset for the experiments.

One of the many CSS subtleties is that declarations from one file might be overridden by declarations stated in some other files. To see how this might affect the results of our refactoring procedures, in this section, we perform two experiments for each of the items listed in Table 3. The first experiment consists in analyzing all of the item's CSS files separately, as if they were used in isolation and without interaction between them. For the second experiment, we use the information extracted as shown in Figure 5 to emulate the global compounded style sheet that the browser will effectively use, and we then apply our refactoring over it.

The refactoring that our tool performs in this second experiment does not alter the presentation of any HTML instance, *as long as* the instance: (1) has the same `!DOCTYPE` declaration as the site's default (listed in Table 3), and (2) includes *at least the same* references to external CSS files (the `link` elements under `head`, shown in Figure 5). If (1) or (2) are not satisfied, then *our tool does not guarantee the semantic preservation of the presentation.*

[6]Some style sheets had `media` specified to `all`

For each item in Table 3, Figure 6 shows the percentages of deleted bytes for the two experiments. We can observe that for every item there exists some *interaction* between the different files, since more bytes are *always* deleted in the second experiment. This means that there is at least one declaration which is overridden by a declaration from another file. For item 15, there is almost no difference between the two experiments' results. Entries 7, 16, show that the percentage of deletions is around 25 % higher, whereas entries 11 and 20 remove the double amount of declarations in the second experiment. Item 10 raises the deletions from 14.24 % to 22.77 %, corresponding to a 60 % gain, which is smaller than the one seen in entries 11 and 20, however there are way more deletions in absolute terms. The biggest difference is seen in entry 18, whose results increase from 9.41 % to 32.16 %, representing a gain of 242 %.

Overall, Figure 6 shows that, in the setting of style sheets using several files, rules from different files tend to collide, and in some cases, this results in many more inaccessible and redundant declarations.

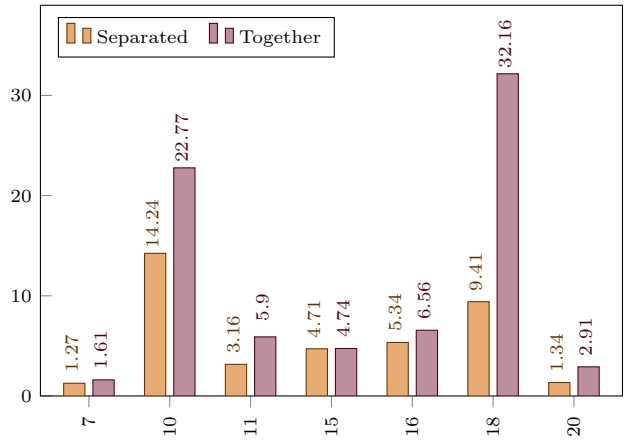

Figure 6: % of deleted bytes on multiple-file style sheets.

5.3 Performance of the Tool

The total elapsed time for each analyzed style sheet is plotted in Figure 7, depending on the number of analyzed rules.

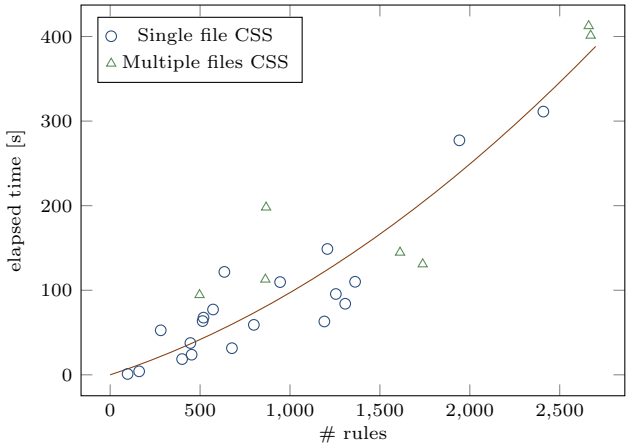

Figure 7: Tool's performance plot

We distinguish entries that correspond to one CSS file from those that correspond to multiple CSS style sheets. They are presented in Tables 2 and 3 respectively. The average time spent on our dataset is 120.46 s, very close to 2 minutes. Extreme cases are in the 1.03 s and 412.47 s figures. Given how the style sheets' rules are traversed, as described in Section 4.5, the obtained curve by fitting the data resembles a 2nd-degree polynomial. By making a simple regression, the coefficient $R^2 = 0.935$, yields a fair approximation of the empirically obtained results.

Each logical test using the logical solver takes an average of 48.94 ms. Tests are performed using the optimised traversal of selectors introduced in Section 4.5, that is we do not blindly check every selector against all the others. Without this optimization, the analysis time would have been much longer (21.7 hours) when compared to the one obtained here (120.46 s).

6. RELATED WORK

Little research effort has been dedicated to study the CSS language and fewer have been dedicated to the quality of widely deployed style sheets. The work found in [13] explores a visual approach to track the effect of source code modifications. This approach can be complementary to existing runtime debuggers available in current browsers [19, 9, 20]. [11, 12, 1, 23] focus on the reusability of CSS for different platforms, whereas [2] introduces a formalism to better separate CSS structural and styling components.

Among the most closely related works we find [17] that proposed an analysis whose purpose is to dynamically detect unused declarations. The most closely related research work can be found in [16]. The authors present presentation-preserving refactoring techniques for CSS based on the detection of code duplication. The approach is based on the detection of rules with very similar declaration blocks. Selectors are grouped and shorthands are used for recombining those rules. Their tool is tested with a large dataset and also obtains interesting size reductions. A fundamental difference with our approach is that their analysis is incomplete and conducted at runtime on a particular instance. In contrast, our static analysis technique needs to be performed only once and the size reduction is observed every time a page is fetched by a different user. Furthermore, the refactoring achieved by [16] can be used alongside those obtained in the present paper, as our tool is capable of actually *deleting* declarations rather than recombining or compressing them. One promising research work would consist in investigating a technique combining both static and dynamic analyses.

For the logical modelling of selectors we were inspired by the translation of simple CSS selectors into the logic introduced in [7]. We extended these translations to support more CSS features, as described in Section 4. One fundamental difference between [7] and the present work is that [7] focused on bug and inconsistency detection whereas the techniques we propose here aim at refactoring and compressing CSS files while preserving the rendering semantics.

A very preliminary version of the present work has been presented in a short paper at [4]. The present paper developed, generalised and extended the initial concept presented in [4], notably with CSS features such as e.g. media queries.

7. CONCLUSION AND PERSPECTIVES

In this paper, we present techniques and a tool capable of automatically refactoring CSS files in several ways with the aim of reducing the sizes of style sheets, while preserving their rendering semantics. In contrast with the existing tools which are mostly dynamic and operate with a particular document instance, our techniques are based on the static analysis of semantic relations between CSS selectors and media queries. Our refactoring applies on a given style sheet, independently of any particular document instance, and reduce the style sheet size once for all. Therefore, the application of our technique makes the browser's rendering of web pages more efficient and less demanding in terms of resources such as CPU and network traffic. Furthermore, the fact that our technique detects and removes unaccessible and useless declarations helps in improving the readability and maintainability of CSS code. Our technique can be used in combination with existing syntactic optimisers since it performs size reductions that the latter cannot do.

Experimental results show that our approach is capable of reducing significantly the size of CSS files currently used in some of the most popular and sophisticated web sites. For the CSS files tested separately, our tool has achieved an average size reduction of 7.60 %, with a maximum of 17.83 %. Furthermore, we showed that, with very little information about the documents that use a style sheet, the size reduction can increase dramatically, reaching 20 to 30 % in some cases.

One perspective for further work consists in taking constraints such as DTDs or schemas into account when performing analyses, and in particular constraints that define profiles for specific devices like mobile phones and other resource-constrained devices. The knowledge of additional structural constraints (such as those defined in e.g. XHTML Mobile Profile) would certainly be beneficial in detecting more relations between selectors. Supporting additional CSS features, such as pseudo-classes, would certainly make the tool perform even more important size reductions as well.

8. REFERENCES

[1] Edward Benson. Mockup driven web development. In *Proceedings of the 22Nd International Conference on World Wide Web Companion*, WWW '13 Companion, pages 337–342, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[2] Edward Benson and David R. Karger. Cascading tree sheets and recombinant HTML: better encapsulation and retargeting of web content. In *WWW'13*, pages 107–118, 2013.

[3] Robin Berjon, Steve Faulkner, Travis Leithead, Silvia Pfeiffer, Edward O'Connor, and Erika Doyle Navara. HTML5. Candidate recommendation, W3C, July 2014. http://www.w3.org/TR/2014/CR-html5-20140731/.

[4] Martí Bosch, Pierre Genevès, and Nabil Layaïida. Automated refactoring for size reduction of css style sheets. In *Proceedings of the 2014 ACM Symposium on Document Engineering*, DocEng '14, pages 13–16, New York, NY, USA, 2014. ACM.

[5] Tantek Çelik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, and John Williams. Selectors level 3. W3C recommendation, World Wide Web Consortium, September 2011.

[6] World Wide Web Consortium. CSS specifications, November 2014. http://www.w3.org/Style/CSS/current-work.

[7] Pierre Genevès, Nabil Layaïda, and Vincent Quint. On the analysis of cascading style sheets. In *WWW'12*, pages 809–818, 2012.

[8] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07*, pages 342–351, 2007.

[9] Google. Chrome Developer Tools, November 2014. https://developer.chrome.com/devtools/.

[10] Arnaud Le Hors, Dave Raggett, and Ian Jacobs. HTML 4.01 specification. W3C recommendation, W3C, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224.

[11] Matthias Keller and Martin Nussbaumer. Cascading style sheets: a novel approach towards productive styling with today's standards. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 1161–1162, New York, NY, USA, 2009. ACM.

[12] Matthias Keller and Martin Nussbaumer. CSS code quality: A metric for abstractness. In *Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 116–121, October 2010.

[13] Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. SeeSS: Seeing what i broke – visualizing change impact of cascading style sheets (CSS). In *UIST'13*, pages 353–356, 2013.

[14] Håkon Wium Lie. *Cascading style sheets*. Phd thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2005.

[15] Philip M. Marden and Ethan V. Munson. Today's style sheet standards: the great vision blinded. *Computer*, 32(11):123–125, nov 1999.

[16] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, page 11 pages. ACM, 2014.

[17] Ali Mesbah and Shabnam Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *ICSE'12*, pages 408–418, 2012.

[18] Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 711–720, New York, NY, USA, 2010. ACM.

[19] Mozilla. Firebug, November 2014. https://getfirebug.com/.

[20] Opera. Opera Dragonfly, November 2014. http://www.opera.com/dragonfly/.

[21] Steven Pemberton. XHTML^{TM} 1.0 the extensible hypertext markup language (second edition). W3C recommendation, W3C, August 2002. http://www.w3.org/TR/2002/REC-xhtml1-20020801.

[22] Vincent Quint and Irène Vatton. Editing with style. In *Proceedings of the 2007 ACM symposium on Document engineering*, DocEng '07, pages 151–160, New York, NY, USA, 2007. ACM.

[23] Nishant Sinha and Rezwana Karim. Compiling mockups to flexible uis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 312–322, New York, NY, USA, 2013. ACM.