On the Analysis of Cascading Style Sheets

Pierre Genevès
CNRS
pierre.geneves@inria.fr

Nabil Layaïda
INRIA
nabil.layaida@inria.fr

Vincent Quint
INRIA
vincent.quint@inria.fr

ABSTRACT

Developing and maintaining cascading style sheets (CSS) is an important issue to web developers as they suffer from the lack of rigorous methods. Most existing means rely on validators that check syntactic rules, and on runtime debuggers that check the behavior of a CSS style sheet on a particular document instance. However, the aim of most style sheets is to be applied to an entire set of documents, usually defined by some schema. To this end, a CSS style sheet is usually written w.r.t. a given schema. While usual debugging tools help reducing the number of bugs, they do not ultimately allow to prove properties over the whole set of documents to which the style sheet is intended to be applied.

We propose a novel approach to fill this lack. We introduce ideas borrowed from the fields of logic and compile-time verification for the analysis of CSS style sheets. We present an original tool based on recent advances in tree logics. The tool is capable of statically detecting a wide range of errors (such as empty CSS selectors and semantically equivalent selectors), as well as proving properties related to sets of documents (such as coverage of styling information), in the presence or absence of schema information. This new tool can be used in addition to existing runtime debuggers to ensure a higher level of quality of CSS style sheets.

Categories and Subject Descriptors

I.7 [**Document and Text Processing**]: Document Preparation—*Languages and systems, Standards*; D.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*

Keywords

Web development, Style sheets, CSS, Debugging

1. INTRODUCTION

"Style sheet languages are terribly under-researched" [11]. This statement dates back from 1999, but it is still true. However, Cascading Style Sheets (CSS) [10] was the first feature that was added to the initial foundations of the web (HTML, HTTP and URLs). While style has become a key component of web user experience, development tools for style sheets have involved very little basic research. As a result, empirical methods are the only means available to web developers for implementing and maintaining style sheets.

The research presented in this paper addresses the issue of debugging CSS style sheets. At first glance, CSS appears to be a simple language, and from a syntactical perspective, it really is. Basically, a style sheet is simply a sequence of style rules. Each rule has a selector that specifies elements of interest in the document structure, and provides a value for a style property. The value is assigned to the corresponding property for all elements specified by the selector.

This apparent simplicity is contradicted by a number of combinatorial aspects, which bring a significant power to the CSS language, while making it a bit more complex. Style rules can be grouped to share the same selector, for specifying different properties that apply to the same elements. Style rules are also grouped by style sheets, and several style sheets may apply to a single document. A style sheet is usually external to the document it applies to, but it may also be embedded in the document, with the `style` element of HTML. Finally, several style rules may also be embedded within an element in a document with the `style` attribute. In addition, the same style property may appear several times in all these locations. The cascade sets the priority between several rules specifying the same property for the same elements.

As a consequence, when a style sheet does not work the way it was intended, it is very difficult to locate the origin of the problem. For this reason, the issue of debugging and maintaining style sheets is important to web developers. In this paper, we propose a novel approach to this issue, based on recent advances in theoretical tools that handle XML structures and query languages for these structures.

The paper is organized as follows. The next section reviews the methods and tools that web developers currently use to debug CSS style sheets. It is followed by an overview of the main features of CSS. The theoretical foundations on which the rest of the paper is based are then summarized. This is mainly a tree logic that is used in section 5 for modeling CSS style sheets. Based on this model, section 6 presents a software tool for the static analysis of style sheets which is illustrated by typical examples. The paper closes with some perspectives.

2. CURRENT PRACTICE

Developers use basically two kinds of tools to find errors in CSS style sheets: validators and debuggers.

Validators address only syntactic issues. They check that a style sheet strictly follows the CSS grammar. These tools

WWW 2012, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

perform *static* checking: they analyze a style sheet for itself, independently of any web page to which it could be applied. A typical example of this family is the W3C CSS validator.[1] While they are useful, validators do not address the difficult issue of locating rules that do not behave as expected.

As opposed to validators, debuggers are *dynamic* tools. They are coupled with a formatting engine that executes style sheets by applying them to web pages and displaying the result. They allow the user to see how the formatter applies style rules to the tested documents. All modern web browsers now include debuggers, such as Firebug (Firefox), Developer Toolbar (Internet Explorer), Dragonfly (Opera), or Web Inspector (Safari).

These tools do not address only style sheets. They deal with the many facets of a web page (DOM tree, scripts, style) [1], but they constitute the primary tool to debug style sheets. They help CSS debugging by providing a list of all style rules that apply to any element chosen by the user. All rules are displayed and any rule overridden by another through the cascade is struck through, thus helping developers to understand what style rules really apply to the chosen element. The origin of each rule (style sheet, style element, style attribute) is also presented. Rules can often be changed on the fly to quickly test alternative solutions.

Performances may be another issue. With complex style sheets, formatting may take some time. A tool such as the YSlow add-on for Mozilla may help to find performance issues, but it also addresses other aspects of performances in web pages, such as HTML and Javascript.

Other tools target CSS selectors specifically. Dust-Me Selectors, for instance, detects unused selectors dynamically, on a single HTML page or on a whole site.

Debugging style sheets after they have been written is not the only way to improve their quality. It could be done also at writing time. Two approaches are possible: generating style sheets automatically from some higher-level specification [8] [9] [13], and including debugging features in a CSS editor [12]. In the first case, the automatic tool is expected to generate bug-free style sheets, but the issue of debugging the higher-level specification remains. In the second case, the author gets assistance at the moment of creating the style rules, which helps her to create better style sheets.

To summarize, validators are the only tools available today that perform static analysis of a style sheet. The errors they report may potentially affect any web page the style sheet is applied to, and if they detect no errors, developers are sure that the style sheet will not have any syntactic issue whatever the page it is applied to.

Unfortunately, syntactic issues are only a small part of the debugging problem. To address the other issues, developers have only dynamic tools at their disposal. To get some confidence in their style sheets, they have to use these tools on a number of pages, but they can never get any complete assurance that these style sheets will not fail on some other page. The process is both painful and unsatisfactory. We believe that static analysis of the content of style sheets (not only their syntax) could considerably help developers in detecting errors and proving properties that are expected from style sheets, whatever the document they are applied to.

We have then developed a tool for the static analysis of CSS. After a brief review of the main features of CSS, we present a logical framework for modeling structured documents and selection of information in them, we show how CSS can be modeled in this logic, and we describe the tool based on this model.

[1]see http://jigsaw.w3.org/css-validator/

3. CSS: AN OVERVIEW

A style sheet C can be seen as a set R of rules, composed of simple rules R_i each composed of a single selector S_i and a set of pairs, each made of a property P_i and its value V_i. Selectors define which elements of a document the properties are applied to. Properties and their values define how those elements look like in the browser.

A selector is a chain of one or more sequences of simple selectors separated by combinators. Simple selectors considered here are of two types: the universal selector, noted `*`, and the type selector which is noted by the tag name of a given element, for example `h1`. For simplicity and without loss of generality, we consider that the rules are made of single selectors (the specification allows a comma separated list of selectors) which set a single property at a time (multiple properties are allowed for a given selector). It is easy to rewrite multiple selectors and property rules to a set of single selector and single property rules.

Selectors S_i, sometimes called patterns in the CSS specification [3], define boolean functions of the form:

$$expression \times element \rightarrow boolean$$

that define whether or not a given element is selected by the selector expression.

In the following, we explore the main vehicle for setting CSS properties on document elements, namely combinators, structural pseudo-classes, and property inheritance.

3.1 Combinators

CSS combinators define relations between elements of a document. In CSS3, they come in three variants according to the specification:

- Descendant combinator: a descendant combinator describes a descendant relationship between two elements. A descendant combinator is made of the whitespace sign, for example “body p”.
- Child combinator: a child combinator describes a childhood relationship between two elements. This combinator is made of the > sign, for example “body > p”.
- Sibling combinator: there are two different sibling combinators, the adjacent sibling combinator and the general sibling combinator. They are noted with the + and ~ signs respectively.

3.2 Structural pseudo-classes

Structural pseudo-classes permit to select elements based on positional information in the document tree. This positional information is based on calculating the position (via an index on sibling elements) of an element relatively to its parent. There are several pseudo-classes in the specification; we present just a few of them here. The others are similar with additional constraints on element types:

- :root pseudo-class: It represents an element that is the root of the document. In HTML 4, this is always the `html` element.

- :first-child and :last-child pseudo-classes: They represent an element that is the first child or the last-child of some other element respectively.
- :nth-child() pseudo-class : The :nth-child($an+b$) pseudo-class notation represents an element that has $an+b-1$ siblings before it in the document tree, for any positive integer or zero value of n, and has a parent element.
- :nth-last-child(): The :nth-last-child($an + b$) pseudo-class notation represents an element that has $an+b-1$ siblings after it in the document tree, for any positive integer or zero value of n, and has a parent element.

Other pseudo-classes are defined in the specification based on both the element type and position. Examples are :first-of-type, :last-of-type and :only-of-type pseudo-classes.

The positional pseudo-classes are very useful to set properties (like foreground and background colors, or fonts) in HTML structures such as tables. The following example alternates four colors (zebra striping when two) in table rows:

```
tr:nth-of-type(4n+1) {color: navy;}
tr:nth-of-type(4n+2) {color: green;}
tr:nth-of-type(4n+3) {color: maroon;}
tr:nth-of-type(4n+4) {color: purple;}
```

3.3 CSS properties and inheritance

CSS inheritance works on a property by property basis. The mechanism for assigning a value to each property for each element is based on the following steps, in order of precedence. If the cascade results in a value, this value is used. Otherwise, if the property is defined by the specification as inherited and the element is not the root of the document tree, the value of the property of the parent element is used (this situation also corresponds to a property with value `inherit`). Otherwise, the property's initial value is used.

The initial value is specific to each property and is indicated by the specification. The initial value for many properties is already `inherit`, and for most others (`border` for instance), inheriting the parent element's value is obviously not desirable. The allowed values for properties, their initial value, and whether they are inherited or not are summarized in the property table of the specification [2].

For example, with this style sheet and this HTML fragment:

```
div { background-color: white;
      color: blue;
      font-weight: normal; }
p {  background-color: inherit;
     color: inherit; }

<div>
  <p>
    Hello, world.
  </p>
</div>
```

the background color of the `div` element is set to white. The background color of the paragraph is also white, because its `background-color` property is set to `inherit` and the background color of the `div` parent element is white.

The `inherit` value does not require that the parent element have the same property set explicitly; it works from the computed value. In the above example, the `color` property of the paragraph has value `inherit`, but the computed value is `blue` because it inherits. The `font-weight` property of the `p` element is also set to `normal` since it is inherited by default.

When two selectors select the same element for a given property, the more "specific" one gets precedence. Specificity of selectors consists in counting a four integer vector corresponding to (1) whether the property is specified in a `style` attribute of not, (2) the number of `id` attributes in the selector, (3) the number of other attributes and pseudo-classes in the selector, (4) the number of element names in the selector. In our case, since we consider analyzing style properties on a possibly infinite set of HTML documents, we consider that selectors specificity is defined by the last integer corresponding to the number of element names. For example:

```
*             {}  specificity = 0,0,0,0 */
li            {}  specificity = 0,0,0,1 */
ul li         {}  specificity = 0,0,0,2 */
ul ol+li      {}  specificity = 0,0,0,3 */
```

Since specificity can be easily and statically computed before analysis, we consider that the corresponding number is provided for each selector by a function `Specificity`(S_i).

4. THEORETICAL FOUNDATIONS

In this section, we present the static analysis technology on which our tool is based, which relies on automated verification of properties that are expressed as logical formulas over trees.

4.1 Approach overview

We use a tree logic capable of capturing the semantics of CSS selectors as well as schemas. Schemas we consider are regular tree grammars which capture most of the XML Schemas, Relax NG schemas, and DTDs. Our approach consists in modeling element selection performed by CSS selectors and structural constraints described by schema information into the tree logic. We then use an algorithm to check satisfiability of formulas of the logic. Such an algorithm defines a partition of the set of logical formulas: satisfiable formulas (for which there exist at least one tree, among those defined by the schema, that satisfies the constraints expressed by the formula) and remaining formulas which are unsatisfiable (no tree satisfies the formula). Alternatively (and equivalently), formulas can be divided into valid formulas (formulas which are satisfied by all trees) and invalid formulas (formulas that are not satisfied by at least one tree). The use of a satisfiability-testing algorithm allows proving validity of a given logical statement P by testing its negation ($\neg P$) for unsatisfiability.

In the sequel, we progressively introduce the tree logic and explain how it captures schemas and CSS selectors. We first present the data model of the logic and then we introduce the syntax of logical formulas through examples.

4.2 Data model

A document is considered as a finite tree of unbounded depth and arity, with two kinds of nodes respectively named elements and attributes. In such a tree, an element may have any number of children elements, and may carry zero, one

or more attributes. Attributes are leaves with a value. Elements are ordered whereas attributes are not, as illustrated on Figure 1. The logic allows reasoning on such trees.

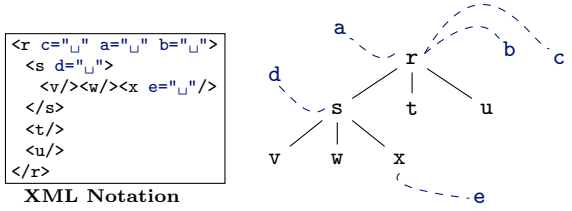

Figure 1: Sample XML tree with attributes

Unranked and binary trees.

There exist bijective encodings between unranked trees (trees of unbounded arity) and binary trees. Owing to these encodings binary trees may be used instead of unranked trees without loss of generality. The logic operates on binary trees. The logic relies on the "first-child & next-sibling" encoding of unranked trees. In this encoding, the first child of a node is preserved in the binary tree representation, whereas siblings of this node are appended as right successors in the binary representation. The intuition of this encoding is illustrated on Figure 2 for a sample tree. In the remaining of this paper, the binary representation of a tree is implicitly considered, unless stated otherwise. From an XML point of view, notice that only the nested structure of XML elements (which are ordered) is encoded into a binary form like this. XML attributes (which are unordered) are left unchanged by this encoding. For instance, Figure 3 presents how the sample tree of Figure 1 is mapped.

4.3 A gentle introduction to tree logic

Navigating in trees with modalities.

The logic uses two *programs* for navigating in binary trees: the program `1` for navigating from a node down to its first successor and the program `2` for navigating from a node down to its second successor. The logic also features *converse programs* `-1` and `-2` for navigating upward in binary trees, respectively from the first and second successors to the parent node. Some basic logical formulas together with corresponding satisfying binary trees are shown on Table 1.

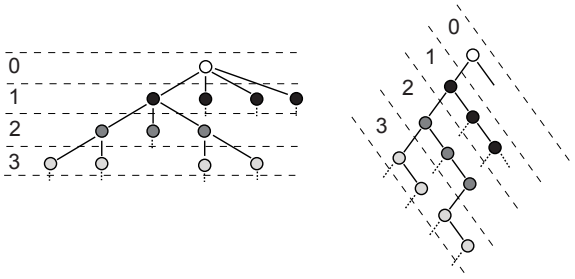

Figure 2: Binary encoding principle

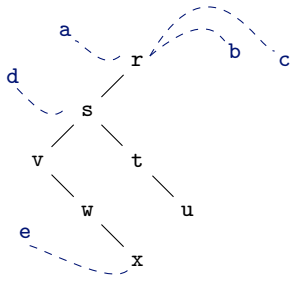

Figure 3: Binary encoding of tree of figure 1

Sample Formula	Satisfying Tree	In XML
`a & <1>b`	a / b	`<a>` `<b/>` `</a>`
`a & <2>b`	a \ b	`<a/><b/>`
`a & <1>(b & <2>c)`	a / b \ c	`<a>` `<b/><c/>` `</a>`
`e & <-1>(d&<2>g)`	d / e \ g	`<d>` `<e/>` `</d><g/>`
`f & <-2>(g&~<2>T)`	none	none

Table 1: Sample formulas using modalities

The set of logical formulas is defined by the syntax given on Figure 4, where the meta-syntax $\langle X \rangle^{\oplus}$ means one or more occurrences of X separated by commas. Models of a formula are finite binary trees for which the formula is satisfied at some node. The semantics of logical formulas is formally defined in [5, 6]. Table 1 gives basic formulas that use modalities for navigating in binary trees and node names, as well as sample satisfying trees in binary and XML notation.

Recursive formulas.

The logic allows expressing recursion in trees through the use of a fixpoint operator. For example the recursive formula:

```
let $X = b | <2>$X in $X
```

means that either the current node is named `b` or there is a sibling of the current node which is named `b`. For this purpose, the variable `$X` is bound to the subformula `b | <2>$X` which contains an occurrence of `$X` (therefore defining the recursion). The scope of this binding is the subformula that follows the `in` symbol of the formula, that is `$X`. The entire formula can thus be seen as a compact recursive notation for a infinitely nested formula of the form:

```
b | <2>(b | <2>(b | <2>(...)))
```

φ	::=		formula
		`T`	true
	|	`F`	false
	|	l	element name
	|	p	atomic proposition
	|	φ `|` φ	disjunction
	|	φ `&` φ	conjunction
	|	φ `=>` φ	implication
	|	φ `<=>` φ	equivalence
	|	`(`φ`)`	parenthesized formula
	|	`~`φ	negation
	|	`<`p`>`φ	existential modality
	|	`<`l`>T`	attribute named l
	|	`<`l`>'`v`'`	attribute l with value 'v'
	|	`$X`	variable
	|	`let` $\langle$`$X` $= \varphi\rangle^{\oplus}$ `in` φ	binder for recursion
p	::=		program inside modalities
		`1`	first child
	|	`2`	next sibling
	|	`-1`	parent
	|	`-2`	previous sibling

Figure 4: Syntax of logical formulas

Recursion allows expressing global properties. For instance, the recursive formula:

```
~ let $X = a | <1>$X | <2>$X in $X
```

expresses the absence of nodes named `a` in the whole subtree of the current node (including the current node). Furthermore, the fixpoint operator makes possible to bind several variables at a time, which is specifically useful for expressing mutual recursion. For example, the mutually recursive formula:

```
let $X = (a & <2>$Y) | <1>$X | <2>$X, $Y = b |
                    <2>$Y in $X
```

asserts that there is a node somewhere in the subtree such that this node is named `a` and it has at least one sibling which is named `b`. Binding several variables at a time provides a very expressive yet succinct notation for expressing mutually recursive structural patterns (that may occur in DTDs for instance).

The combination of modalities and recursion makes the logic one of the most expressive (yet decidable) logic known. For instance, most DTDs and schemas (specifically regular tree grammars) can be expressed with the logic using recursion and (forward) modalities (see [5] or [6] for details). The combination of converse programs and recursion allows expressing properties about previous siblings of a node for instance, which happens to be very useful for capturing the semantics of CSS selectors.

5. A LOGICAL MODELING OF CSS

5.1 Capturing selectors

CSS selectors are systematically translated into the logic: Figure 5 shows how the main combinators found in CSS selectors level 3 [3] are mapped into their corresponding logical representation. The logical formula holds for elements that are selected by the CSS selector. Figure 6 presents how the structural and negation pseudo-classes of CSS level 3 are compiled into logical formulas. We have developed a general compiler that takes a CSS selector as input, systematically applies the translation rules, and outputs the corresponding logical formula. In the remaining part of this paper, we denote this compiler by a compilation function $\mathtt{F}(\cdot)$ so that we can refer to the logical translation of a selector S_i with $\mathtt{F}(S_i)$.

For example, the selector $S_1 =$ `ul li:nth-last-of-type(2)` selects any `li` element which is a second sibling of its type, counting from the last one, while being a descendant of some `ul` element. The corresponding logical formula is built in two steps. First, the translation of the descendant combinator (shown in Figure 5) is instantiated with the appropriate parameters `ul` and `li:nth-last-of-type(2)`, therefore the logical translation $\mathtt{F}(S_1)$ is as follows:

$$\varphi \texttt{ \& let \$X= <-1>(}\psi \texttt{ | \$X) | <-2>\$X in \$X}$$

where $\varphi = \mathtt{F}($`li:nth-last-of-type(2)`$)$ and $\psi = \mathtt{F}($`ul`$)$. As a second step, φ and ψ are computed:

- $\varphi =$ `li & let $X= <2>(li & ~let $Y=<2>li|<2>$Y in $Y) |<2>$X in $X` (see f_8 in Figure 6);
- $\psi =$ `ul` (see Figure 5).

Notice that the class attribute which very frequently used in style sheets is simply translated as an ordinary attribute (see class `foo` Figure 5).

5.2 Capturing Properties

In order to capture CSS properties, we consider that all elements in a schema, in HTML in particular, are augmented with the entire set of CSS properties encoded as attributes. For example, the following rule:

```
ul li:nth-last-of-type(2) {color:  green;}
```

is translated as $\mathtt{F}(S_1)$ `& <css:color>'green'` in the logic, with $\mathtt{F}(S_1)$ computed as explained above.

5.3 Capturing Inheritance

The CSS property value `inherit` is a very particular value which is not related to style, but instead it indicates how the property value must be computed. Specifically, a computed value $v \neq$ `inherit` is *obtained* for a property p at a given element iff:

- value of p is explicitly set to v at the given element (intuitively this has been set by some custom selector);
- value of p is not explicitly set to v at the given element, it is not set to `inherit` either at that element, but the initial value for this property is v;
- value of p is set to `inherit` at that element, the given element is the root, the initial value for this property happens to be v.
- value of p is set to `inherit` at that element, the given element is not the root, and value v is obtained for the parent element (by applying this case analysis recursively);

Semantics	CSS	Tree Logic
Any element	`*`	`T`
Any 'p' element	`p`	`p`
Any child of some p element	`p > *`	`let $X= <-1>p \| <-2>$X in $X`
Any descendant 'b' of some 'a' element	`a b`	`b & let $X= <-1>(a | $X) \| <-2>$X in $X`
Any element with class 'foo'	`.foo`	`<class>'foo'`
Any element with attribute 'title'	`*[title]`	`<title>T`
Any 'p' element with an 'a' child	Not possible	`p & <1>let $X= a \| <2>$X in $X`
Any adjacent next sibling of a 'p' element	`p + *`	`<-2>p`
Any next sibling 'pre' of a 'h1' element	`h1 ~ pre`	`pre & let $X= <-2>h1 \| <-2>$X in $X`
Any 'e' whose 'foo' attribute value is 'bar'	`e[foo="bar"]`	`e & <foo>bar`

Figure 5: Main CSS combinators and corresponding logical formulas

Semantics	CSS	Tree Logic
An 'e' element, root of the document	`e:root`	`e & ~<-1>T & ~<-2>T`
Any first child of a 'p' element	`p > *:first-child`	f_1
Any 'li' element that is the last child of a 'ol' element	`ol > li:last-child`	f_2
Any odd row of an HTML table	`tr:nth-child(odd)`	f_3
Any even row of an HTML table	`tr:nth-child(even)`	f_4
Any 'foo', third child of its parent element	`foo:nth-child(3)`	f_5
Any 'e', second child of its parent, counting from the last one	`e:nth-last-child(2)`	f_6
Any 'e' element, second sibling among the 'e' 's	`e:nth-of-type(2)`	f_7
Any 'e', second sibling of its type, counting from the last one	`e:nth-last-of-type(2)`	f_8
Any 'e' element, first sibling of its type	`e:first-of-type`	f_9
Any 'e' element, last sibling of its type	`e:last-of-type`	f_{10}
Any 'e' element, only child of its parent	`e:only-child`	f_{11}
Any 'e' element, only sibling of its type	`e:only-of-type`	f_{12}
Any 'e' element that has no children	`e:empty`	`e & ~<1>T`
Any 'e' element that does not match simple selector s	`e:not(s)`	`e & ~s`

f_1 = `~<-2>T & let $X=<-1>p|<-2>$X in $X`
f_2 = `~<2>T & li & let $X=<-1>ol|<-2>$X in $X`
f_3 = `tr & let $X=<-1>T|<-2><-2>$X in $X`
f_4 = `tr & let $X=<-2><-1>T|<-2><-2>$X in $X`
f_5 = `foo & <-2><-2>(~<-2>T) & let $X=<-1>T|<-2>$X in $X`
f_6 = `e & <2>(~<2>T) & let $X=<-1>T|<-2>$X in $X`
f_7 = `e & let $X= <-2>(e & ~let $Y=<-2>e|<-2>$Y in $Y) |<-2>$X in $X`
f_8 = `e & let $X= <2>(e & ~let $Y=<2>e|<2>$Y in $Y) |<2>$X in $X`
f_9 = `e & ~let $X=<-2>e|<-2>$X in $X`
f_{10} = `e & ~let $X=<2>e|<2>$X in $X`
f_{11} = `e & ~<2>T & ~<-2>T & let $X=<-1>T|<-2>$X in $X`
f_{12} = `e & ~let $X=<2>e|<2>$X in $X & ~let $Y=<-2>e|<-2>$Y in $Y`

Figure 6: Structural and negation pseudo-classes (CSS level 3) and corresponding logical formulas

We model this inheritance mechanism for propagating values in logical terms. We introduce a predicate that logically describes each of those possible cases.

The predicate `inherit`(p, v) holds at a given element iff value v is obtained for property p at this element:

`inherit`(p, v) =

```
let $X = <-1>( <p>'v'
| ~<p>'v' & ~<p>'inherit' & initialvalue(p,v)
| <p>'inherit' & ~<-1>T & ~<-2>T & initialvalue(p,v)
| <p>'inherit' & $X ) | <-2>$X in $X
```

where `initialvalue`(p, v) is a predicate that holds iff property p has initial value v, as defined by the CSS recommendation (see [2]).

6. IMPLEMENTATION & EXPERIMENTS

We present here the tool we have developed based on the logical modeling presented in the previous section. Its architecture is outlined in Figure 7. It is composed of a set of parsers for reading the CSS and schema files (XML Schema, Relax NG, or DTD) together with a text file corresponding to problem description as a logical formula. Some compilers are used for translating schemas and CSS files into their logical representations. CSS files are first converted into the simplified form explained in Section 3. Then, the solver takes the overall problem formulation and checks it for satisfiability.

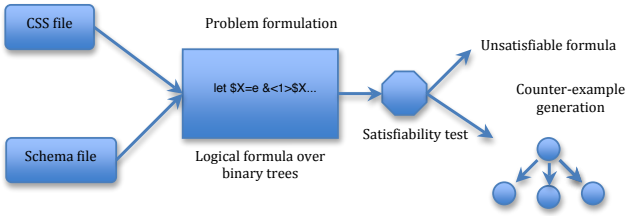

Figure 7: Overall architecture

The result of the analysis corresponds to two situations: either the formula is found unsatisfiable (meaning that the checked property holds for any tree), or it is satisfiable. In this case, the solver generates a counter-example document satisfying the formula (described in [7]).

6.1 Reasoning on style properties

In this section, we present some experiments highlighting how the analyzer works on some typical examples. These examples are simplified in order to make the analysis easier to understand (but the same kind of analyses can be applied to more complex cases). Notice that users are not asked to type these formulas as generic tests are provided as a set of macros in the tool (see Section 6.2). We just detail some of them enough to explain how they work. For the same reasons, we use the simplified HTML DTD shown in Figure 8.

The first example is the verification of the behavior of a style sheet when it comes to displaying text in different font sizes. Indeed, setting the `font-size` property to the value `inherit` can be error-prone. Specifically, a computed `font-size` value repeatedly obtained by inheritance from a relative value like `80%` or `smaller` may result in tiny or unreadable text. The goal of the test here is to check whether

```
<!ELEMENT html (head,body)>
<!ELEMENT head (title)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT body ((div|table)*)>
<!ELEMENT table (tbody)>
<!ELEMENT tbody (tr+)>
<!ELEMENT tr (td+)>
<!ELEMENT td (div*)>
<!ELEMENT div (#PCDATA|div)*>
```

Figure 8: Simplified HTML DTD.

the style sheet may yield such a bad rendering on some documents. This can be expressed logically by the following formula:

```
1. type("html.dtd","html") & ~<-1>T  & ~<-2>T
2. & let $CSS = (div => <font-size>'smaller')
3. & (~div => (~<font-size>T |<font-size>'normal'))
4. & (~<1>T | <1>$CSS) & (~<2>T | <2>$CSS) in $CSS
5. & let $Q = <font-size>'smaller'
   & ancestor(<font-size>'smaller'
   & ancestor(<font-size>'smaller')) | <1>$Q | <2>$Q in $Q
```

This formula is built from the sample style sheet of Figure 10. The first line allows translating the simplified schema of Figure 8 into a logical formula (omitted here). Notice that `~<-1>T & ~<-2>T` means that the element has no parent nor a previous sibling, i.e. it is the root element. Line 2 represents the logical counterpart of CSS rule 6 of the sample CSS of Figure 10. It corresponds to the logical implication "if an element is labeled `div` then the value of its `font-size` property must be `smaller`". Line 3 states that any other element than `div` (elements that are not concerned by the previous rules) may either carry no `font-size` property or if they do, then the value for `font-size` is set to `normal`. This models the default behavior of CSS for property `font-size` which is overridden by the rules for `div`. Line 4 in is charge of applying the style information to every element in the document.

Thus, lines 1 to 4 restrict the considered set of documents to those that are valid with respect to the DTD, and that have the style information defined by the style sheet.

Now, line 5 formulates the question: "*may the application of my style sheet render some valid document with unreadable text due to font-size too small because of a triple application of the relative font-size value 'smaller'?*" In this example, the predicate `inherit`(p, v) is simply substituted by the simpler ancestor predicate. When fed with this formula, the logical solver explores all possible situations and produces the following counter-example (which is displayed in the browser as the picture on the left in Figure 9):

```
<html xmlns:solver="http://wam.inrialpes.fr/xml">
 <head>
  <title/>
 </head>
 <body>Body text
  <div font-size="smaller">text in first level of div
    <div font-size="smaller">text in second level of nested div
      <div font-size="smaller">text in third level of nested
       div </div>
    </div>
  </div>
 </body>
</html>
```

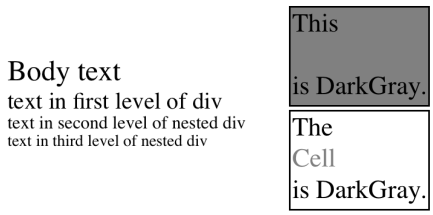

Figure 9: Counter-example layouts

```
1.  tr:nth-child(even) {background-color:LightGray;}
2.  tr:nth-child(odd) {background-color:DarkGray;}
3.  table td {font-size: 16px;}
4.  td {font-size: 14px;}
5.  div {color:gray};
6.  div {font-size:smaller;}
```

Figure 10: Sample CSS style sheet.

Notice that if we add the following rule

```
div div div {font-size:medium;}
```

in order to fix the style sheet, then the solver cannot find any counter-example anymore.

The second test consists in verifying that for a given style sheet, there is no document such that the style sheet generates text with the same color as the background color. For example, we consider the simple style sheet of Figure 10. The problem consists in testing wether rules that set the `color` and `background-color` properties together with the CSS inheritance mechanism may result in such a situation.

This is expressed in logical terms is as follows:

```
1. type("html.dtd","html") & ~<-1>T  & ~<-2>T
2. & let $CSS =
   ((tr & let $V=<-1>T|<-2><-2>$V in $V)
   => <background-color>'LightGray')
3. & ((tr & let $X=<-2><-1>T|<-2><-2>$X in $X)
   => <background-color>'DarkGray')
4. & (~tr => (~<background-color>T |<background-color>'white'))
5. & (div => <color>'DarkGray')
6. & (~div => (~<color>T |<color>'black'))
7. & (~<1>T | <1>$CSS) & (~<2>T | <2>$CSS) in $CSS
8. & let $Q = (<color>'DarkGray'
       & ancestor(<background-color>'DarkGray'))
     | <1>$Q | <2>$Q in $Q
```

Line 2 and 3 are the logical counterparts of CSS rule 1 and 2 of the sample of Figure 10. They correspond to the logical implication "if an element is labeled `tr` and is at an even position (odd position respectively) among its siblings, then its background color must be 'LightGray' ('DarkGray' respectively)". Line 4 says that any other element than `tr` (elements that are not concerned by the previous rules) may either carry no `background-color` property or if they do, then its value is set to`white`. This models the default behavior of CSS for the `background-color` property which is overridden by the rules for `tr`. Similarly, line 5 is the logical implication that corresponds to the CSS rule on line 5 of Figure 10. Line 6 models the default behavior for the property color. Line 7 is in charge of applying the style information to every element in the document.

Line 8 formulates the question: "*may the application of my style sheet render some valid document with unreadable text because it is displayed in the same color as the background?*" For the sake of simplicity, the ancestor predicate in this line models the default CSS inheritance behavior for the `background-color` property. A more general statement should use the predicate inheritedValue. When fed with this formula, the logical solver explores all possible situations and ends up with this counter-example (which is displayed in the browser as the picture on the right in Figure 9):

```
<html xmlns:solver="http://wam.inrialpes.fr/xml">
  <head>
    <title/>
  </head>
  <body>
    <table>
      <tbody>
        <tr background-color="LightGray">
          <td>
          This <div color="DarkGray"> Cell</div> is DarkGray.
          </td>
        </tr>
        <tr background-color="DarkGray">
          <td>
          The <div color="DarkGray"> Cell</div> is DarkGray.
          </td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

The disclaimer text contained in the second cell of the table has both properties `color` and `background-color` set to `DarkGray`. This is caused by the default inheritance rules for these properties which are set to `inherit`. The disclaimer text has inherited its values from the enclosing `div` resulting in this color collision.

Now, we check the consistency of selectors in the style sheet of Figure 10. The test amounts to comparing the selectors for a given property. For example, if we focus on the `font-size` property for table element selectors `table td` and `td`, the test consists in checking the precise relation between selectors, equivalence or containment, against the HTML DTD for instance.

The problem formulation for the solver is as follows:

```
1.  type("html.dtd","html") & ~<-1>T  & ~<-2>T
2.  & let $Q =
      ~(td & ancestor(table) <=> td)
      | <1>$Q | <2>$Q in $Q
```

The formula is found unsatisfiable which means that the two selectors are equivalent in the presence of the DTD. Intuitively, here, both selectors are equivalent since under HTML schema constraints `td` always occurs under a `table` element. However, for `td` elements, CSS rule precedence gives higher priority to rule 5 which has specificity 2 compared to rule 6 with specificity of 1. Therefore, rule 6 will never be reachable by any HTML document. As a consequence, rule 6 can be safely removed from the style sheet.

When generalized to all selectors for a given property, this mechanism allows to clean up style sheets from such inapplicable rules, enhancing their readability, as seen in the next Section.

While in the case of HTML such situations can be detected by an expert designer, things become much harder when considering CSS for general XML documents. In particular, CSS rules (see selectors of [15]) for very structured schemas like Docbook [14] or DITA [4], tend to be much

more involved as they use complex compositions of combinators with type elements.

6.2 Identifying and verifying generic issues

In this section, we investigate the logical formulation of generic issues that correspond to useful questions for CSS developers and that can arise in many style sheets. The tests described here can be checked in the absence or in the presence of a schema. In the latter case, we use the logical translation of the schema that we insert as the initial context for the translation of selectors[2].

Emptiness of selectors.

This test is the generalization of the example presented above. The test consists in extracting every selector and testing its satisfiability against a given schema. We check $\mathtt{F}(S_i)$ for unsatisfiability. If $\mathtt{F}(S_i)$ is unsatisfiable then the selector is inconsistent and the corresponding style rule is always inactive.

Equivalence of selectors.

We check the validity of $\mathtt{F}(S_i) \Leftrightarrow \mathtt{F}(S_j)$ for $i \neq j$ pairwise by checking for the unsatisfiability of $\neg\,(\mathtt{F}(S_i) \Leftrightarrow \mathtt{F}(S_j))$. If two selectors are equivalent and if one has a lower specificity, i.e. $\mathtt{Specificity}(S_i) < \mathtt{Specificity}(S_j)$ then the rule for S_i is always inactive. If both have the same specificity then the first rule in the lexical order in the style sheet is always inactive since CSS favors the last one in this case.

The emptiness and equivalence tests for selectors can be used for tuning a CSS style sheet for a particular schema by pruning inactive CSS rules automatically. In addition to improve readability, this reduces the size of the CSS file and therefore the amount of time needed to download the webpage materials.

Coverage without properties nor inheritance.

We check the validity of $\mathtt{T} \Rightarrow \bigcup_i S_i$, that is, we check the unsatisfiability of $\neg\left(\bigcup_i S_i\right)$. If this formula is unsatisfiable, then it means that some elements are not covered by any style sheet selectors. In other terms, the style properties set by the CSS developer do not cover all the possible elements of a document. If a rule with selector `*` exists then obviously all elements are covered. This test can be performed on the style sheet except `*` selectors, in order to capture the coverage of CSS properties other than those defined for all elements (`*`).

Coverage with inheritance for a given property.

We want to determine, whether a given property p is set to some value v for all elements of a document, while taking into account the propagation of values defined by the inheritance mechanism of CSS. We define the predicate $\mathtt{customset}(p, v)$ as the disjunction of all selectors that set the value v for the property p. We check for the validity of the following formula ψ:

$$\texttt{<}p\texttt{>'}v\texttt{'} \Rightarrow (\mathtt{inherit}(p, v) \,|\, \mathtt{customset}(p, v))$$

or in other terms we check for the unsatisfiability of $\neg\psi$

[2]The notion of context is explained in details in [6]. The acute reader may notice that the resulting formula is large, however it can still be decided efficiently, because common subformulas can be shared, following the result of [7].

where p is the property and v is the value for which we check the coverage. For example, we can think of a web designer building a style theme restricted to a limited set of colors. She may be willing to test whether all possible HTML documents and their respective elements do have only these colors without reverting to default ones.

$$\neg(\bigcup_{i=1,2} \texttt{<color>}c_i \Rightarrow (\bigcup_{i=1,2} \mathtt{inherit}(\mathtt{color}, c_i) \,|\, \mathtt{customset}(\mathtt{color}, c_i)))$$

Is this formula is satisfiable, this means that there exist some document instance for which the style sheet renders some elements with another color than the c_i's, breaking the intended design.

The tool is available at: http://wam.inrialpes.fr/websolver

7. CONCLUSION

In this paper, we introduce the concept of static analysis for CSS style sheets. To the best of our knowledge, this is the first attempt at statically analyzing CSS style sheets. We propose an original tool based on recent advances in tree logics. The tool is capable of statically detecting a wide range of common errors, as well as proving properties related to sets of documents, such as coverage of styling information, in the presence or absence of schema information.

From a theoretical perspective, CSS selectors could be related to XPath queries, for which an extensive static analysis has been conducted in [6]. In this paper, we deal with the peculiar combinators and pseudo-classes found in CSS selectors. In particular, we have extended the logical solver, initially developed for XPath, to be able to reason about attribute values, by introducing an equality test that compares an attribute value to a constant. This is a worthy extension since it is sufficient for supporting CSS while preserving decidability for the extended logic (it is known that extending the logic with equality tests with variables results in undecidable logics, but this feature is not needed for CSS). In addition, we deal with style properties and the propagation of values defined by the inheritance mechanism of CSS, which do not have any XPath counterpart.

From a practical perspective, there exists a whole class of dynamic analyses. Most of this technology relies on runtime debuggers that check the behavior of a CSS style sheet on a particular document instance. However, the aim of CSS is to be applied to an entire set of documents, usually defined by some schema. The existing runtime debugging tools help reducing the number of bugs. However, compared to our approach, they do not allow to prove properties over the whole set of documents to which the style sheet is intended to be applied. Therefore, our new approach and tool can be used in addition to debuggers to ensure a higher level of quality of CSS style sheets.

There are several directions for future work. One is to characterize erroneous patterns in the box model and in positioning. As seen in the examples, such errors may also be captured by logical descriptions regardless of values such as sizes, paddings, etc. Another direction is to extend the analysis to complex and very large style sheets such as those for Docbook [14] or DITA [4]. Another perspective consists in taking into account in the analysis the program that generates the document to which a CSS style sheet is applied. Programs usually generate restricted HTML or XML struc-

tural patterns. This subset could be inferred by some procedure and combined with the analyses proposed here.

Acknowledgments. The authors acknowledge the partial support of ANR project Typex (ANR-11-BS02-007), BLANC SIMI 2 Program.

8. REFERENCES

[1] J. J. Barton and J. Odvarko. Dynamic and graphical web page breakpoints. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 81–90, New York, NY, USA, 2010. ACM.

[2] B. Bos, T. Çelik, I. Hickson, and H. W. Lie. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. W3C recommendation, World Wide Web Consortium, June 2011.

[3] T. Çelik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams. Selectors level 3. W3C recommendation, World Wide Web Consortium, September 2011.

[4] K. J. Eberlein, R. D. Anderson, and G. Joseph. Darwin information typing architecture (DITA) version 1.2. Oasis standard, OASIS, December 2010.

[5] P. Genevès. *Logics for XML*. PhD thesis, Institut National Polytechnique de Grenoble, December 2006. http://www.pierresoft.com/pierre.geneves/phd.htm.

[6] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 342–351, 2007.

[7] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types (extended version). Research Report 6590, INRIA, July 2008.

[8] M. Keller and M. 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.

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

[10] H. W. Lie. *Cascading style sheets*. Phd thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2005.

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

[12] V. Quint and I. 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.

[13] M. Serrano. HSS: a compiler for cascading style sheets. In T. Kutsia, W. Schreiner, and M. Fernández, editors, *PPDP*, pages 109–118. ACM, 2010.

[14] L. M. N. Walsh. *DocBook: The Definitive Guide*. O'Reilly & Associates, 1999.

[15] H. Werntges. A CSS for docbook, November 2011. http://www.cs.hs-rm.de/ werntges/proj/wysiwyg-dbk01.html.