Editing with Style

Vincent Quint, Irène Vatton

Editing with Style

Vincent Quint
vincent.quint@inria.fr

Irne Vatton
irene.vatton@inria.fr

INRIA
655 avenue de l'Europe
38334 Saint Ismier, France

ABSTRACT

HTML has popularized the use of style sheets, and the advent of XML has stressed the importance of style as a key area complementing document structure and content. A number of tools are now available for producing HTML and XML documents, but very few are addressing style issues. In this paper we analyze the requirements for style manipulation tools, based on the main features of the CSS language. We discuss methods and techniques that meet these requirements and that can be used to efficiently support web authors in style sheet manipulation. The discussion is illustrated by the recent developments made in the Amaya web authoring environment.

Categories and Subject Descriptors

I.7 [**Document and Text Processing**]: Document Preparation—*Languages and systems, Markup languages, Standards*

General Terms

Design, Experimentation

Keywords

document authoring, web editing, style languages, CSS

1. INTRODUCTION

In the early days of the web, style was the first document technology that was developed after HTML. Since that time, the use of style sheets has become very popular for many different types of documents. This includes the traditional HTML page, but also all the documents that were made possible later by the rise of XML and the many languages based on it.

Style determine how documents are presented to readers. Having a strong impact on the perception of web resources, style is an important component of the web architecture [5]. Separation of presentation from content is an architectural principle that plays a key role in various facets of the web, such as accessibility, device independence, ubiquity and mobility.

To address the wide variety of documents and the different ways they are used on the web, two languages were developed for specifying style in XML documents, namely CSS [3] and XSL [2]. Although they have some common features (notably many style properties), both languages are quite different. In CSS, style properties are related directly to document elements, while in XSL, the formatting process first transforms the logical structure of a document into a presentation-oriented structure, then it associates style properties to this new structure and finally it formats it.

In this paper we focus on CSS, which is both simpler and more widely used on the web. While some research has already been done on the manipulation of XSL in authoring tools [11], it seems that CSS has received much less attention from the research community. As Marden and Munson note in [8], "style sheet languages are terribly underresearched". Some authors discuss the design and the key features of style sheet languages (see Håkon Lie's thesis [6] for a comprehensive review of languages, or [1] [7] for specific languages). However, the issue of manipulating style sheets from a user perspective is almost never addressed in the literature. Considerable efforts have been spent on tools for generating and editing XHTML and various types of XML documents, which have lead to a number of authoring tools. This is not the case for style sheets. A style language such as CSS may look simple at the first glance, but every web developer knows how complex it is to create and maintain a sophisticated and consistent set of style sheets for an entire web site.

Creating and editing style sheets is indeed a complex task. Many document editors for (X)HTML or XML simply ignore that document authors and designers may wish to design the style of a document at the same time as the document content. This is not contradictory to the architectural principle mentioned above. Although content and style are clearly separated in different resources using different languages, it is common practice to manipulate the content and the structure of a document through (or simultaneously with) its presentation. That is the basis of WYSIWYG and direct manipulation, two paradigms widely used in document editing.

In this paper, we discuss methods and techniques for manipulating CSS style sheets. The next section provides an analysis of the main issues that arise when manipulating

DocEng'07, August 28–31, 2007, Winnipeg, Manitoba, Canada.

style sheets. It also reviews the tools that are currently available. Then, the main issues identified are addressed in dedicated sections, starting with the creation and editing of style sheets. This is followed by the debugging process: when something goes wrong with the style of a document, how to efficiently locate the cause of the problem and fix it. The last section addresses the distributed nature of style resources on the web and how local and remote style sheets that affect the presentation of a document can be handled consistently. Finally the conclusion summarizes the main contributions and opens some perspectives.

2. ANALYSIS

CSS is a simple language. By writing `p{font-size:11pt}` in a style sheet, one can set the character size of all paragraphs (`p` elements) in a HTML document. Specifying simple things is indeed simple. But, in addition to this basic functionality, CSS also provides several powerful mechanisms. As a matter of fact, adding style rules to an existing style sheet is not the only task an author may need to perform. In this section we review the various aspects of CSS style sheets and the operations authors usually do.

We use the term author in a rather broad sense. In particular, we do not make a difference between the person who creates the content of a document and the person who specifies its style. Although this traditional separation of concerns still applies in many cases, including on the web, we have to recognize that the freedom brought by the web for publishing documents has led a number of people to play both roles at various degrees. So, we are interested in methods and techniques that may be used both by document authors and graphic designers. Users who are writing a document using existing style sheets should be able to change a few specific style aspects for the current document, without changing the style sheets themselves. Others should also be able to update a set of style sheets that are not handling well a particular case that they discover while writing. Some other users should be able to create from scratch the whole graphic form for a family of documents, and to make it available to a community, without writing a real document.

As a consequence, the manipulations an authoring tool should facilitate concerning style sheets cover a broad range, which includes creating, editing, debugging, maintaining, sharing, reusing, publishing, reorganizing, etc.

2.1 Issues to be addressed

Despite its apparent simplicity, CSS has some features that may be problematic for web authors when manipulating style sheets. A quick review of its main features may show where the issues stand.

Style rules are the basic components of CSS style sheets. A rule is constituted of three parts:

- a *selector* that selects the document elements that are concerned by the rule,
- a *property* (font size, color, margins, etc.),
- a *value* that is assigned to the property for all selected elements.

The selector is followed by curly braces that contain the property and the value, which are separated by a semicolon. The syntax of a rule is very simple. The selector identifies the structural context in which the rule must be fired. Its syntax is simple too (more details are provided in section 4.2). A property is identified by a name. This also is very simple, but there are many different properties (113 in CSS 2.1) and users often have difficulty to remember them all with the correct name. Providing help to choose properties and to generate valid names is something useful.

Depending on the property, values are either a name taken from a predefined list or a number followed by a unit. Units are also predefined, and the list of valid units depends on the property. A tool should help authors to follow these constraints and to enter a valid value for each property.

Many properties are independent from each other: the color of characters may be chosen freely whatever the font of these characters, for instance. The values of some other properties have to be chosen with some constraints: if the same color is given to characters and their background (two separate properties), the text will be unreadable, or hardly readable if the contrast is too low. There are also sets of properties whose values have to be chosen consistently to obtain the desired result. Positioning belongs to that category. To get the layout of Figure 1, the `width` of elements *A*, *B*, and *C* should be set to fill up the whole width of the containing element, and their `position`, `top`, `left`, `right` properties must be assigned precisely, as well as the `margin` property.

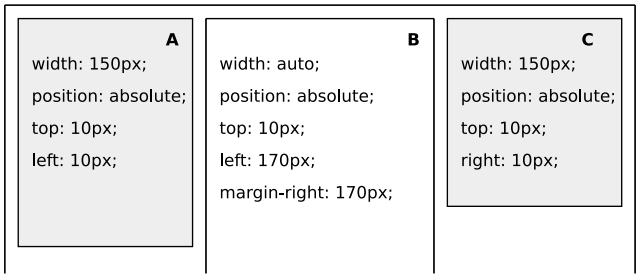

Figure 1: Positioning three elements

One issue with these inter-related properties is that setting all of them correctly requires some expertise. Another issue is that the values of these properties come from rules that are associated with different elements, and these rules are usually scattered across the style sheet, often across several style sheets. When modifying one rule, it is difficult to locate the other rules that have to be updated accordingly.

Grouping is another interesting aspect of CSS. Rules may be grouped in different ways:

- by selectors: Several property-value pairs applying to the same elements may be grouped in a *block* sharing the same selector. But a selector may appear several times in the same style sheet, which makes it difficult to find all the rules applying to an element.
- by properties: Several related properties may be subsumed by a single shorthand property, which then has a list of values, one for each of the atomic properties. For instance, properties `font-family`, `font-style` and `font-size` may be grouped under the name `font`. In that case the property may have up to three values, e.g. `font: times italic 10pt;`. This avoids verbosity, but makes it complicated to search a given

atomic property in a style sheet. It also makes it more difficult to enter a valid value for a shorthand property.

- by style sheets: Style rules and blocks are grouped in web resources called style sheets. A style sheet may contain all the rules that apply to a certain type of document, but a modular approach is also possible. With this approach, the style of a document type is specified in multiple style sheets, each one for a different aspect, or a different part of the document. Modular style sheets may be combined in different ways to easily change the document style. This flexibility has a downside: when handling the style of a document, multiple style sheets have to be considered, and on the web, these style sheets may reside on several remote servers or on local storage.

Most style sheets are not part of the document, but are linked to it. A general mechanism is available for all XML formats: a XML processing instruction of the form `<?xml-stylesheet href="foo.css" type="text/css"?>` associates the resource identified by `href` with the document where it appears. Some languages have their own syntax for associating style sheets. XHTML uses the `link` element for that purpose. In XHTML, the following syntax is equivalent to the PI above: `<link href="foo.css" rel="style sheet" type="text/css"/>`. Using both syntaxes, several style sheets may be linked to a single document.

A style sheet may itself refer to another style sheet, and thus import style rules from another web resource. This is achieved with the `@import` rule. As an example, the statement `@import url("bluish.css");` inserts the content of resource `bluish.css` at its position. The imported content may itself contain some `@import` statements.

Most of the style for a document is specified in separate style sheets linked to the document. However, several document languages such as XHTML or SVG provide means to embed CSS style in the document itself:

- Any element in the document may have a `style` attribute that contains a list of properties with their values. There is no selector: these properties apply to the element itself and only to it.

- A `style` element may be present at the beginning of the document. Like a style sheet, it contains a set of style rules, possibly organized in blocks. These rules apply to all elements in the document that match their selectors.

The external style sheets may have different *origins*. User agents (browsers for instance) have their own style sheets that are used by default, when no other style is available. Users may instruct their user agents to use their own style sheets, where they express personal preferences such as font size, color, font family etc., if the default style offered by the user agent is not convenient. Web servers have their style sheets that are shared by multiple documents to provide a consistent look and to express a graphic identity for the site. Document authors may wish to specify their own style for a given document, while keeping some aspects of the site look.

With so many origins for style (agent, user, site, author), conflicts often arise. For a given element in a document the same property may be assigned different values by different style sheets. The mechanism to solve these conflicts is known as the *cascade* (hence the name Cascading Style Sheets). The cascade is a set of rules that determine the priority between several conflicting style rules. Priority is based on three factors: origin, specificity of selectors and order of appearance. The three factors are ordered in the sense that the specificity is used only if the origin does not solve the conflict, and order of appearance is used only if origin and specificity are not sufficient. In addition it is possible to override the order of the cascade by setting an indicator `!important` for a particular style rule.

To take into account the heterogeneity of the web, a style sheet may contain different sections for different kinds of devices (screen, projection, hand held, braille, TV, etc.). In a style sheet, an `@media` section gathers all style rules that apply only to a particular kind of device. Similarly, the link that associates a style sheet with a document may have a `media` attribute that indicates that the referred style sheet as a whole must be used only if the presentation device belongs to a given class. The `@import` rule also can specify which devices are concerned by the imported rules.

As the CSS language may be used for any XML document, it can be used for compound documents, i.e. documents that involve different languages to represent different kinds of content. A typical compound document uses XHTML for the main structure of the document and its textual parts, MathML for the mathematical expressions and SVG for the drawings. Each of these languages may use CSS. The issue of inheritance of style properties across language boundaries has then to be addressed. In a XHTML document, if the font size is set by a CSS rule for the `body` element, all elements in the document body inherit that font size, unless another style rule states differently for some elements. For a mathematical formula inside a paragraph, it will be the same. As a consequence, the character size of a MathML element, a fraction for instance, will be determined by a style rule from a style sheet intended for XHTML documents.

XHTML allows also external resources to be included by reference, through the `object` element. When a SVG drawing is included that way, it may inherit some properties from the surrounding XHTML structure, in the same way as the fraction in the previous example, but it may also come with its own style sheets. Some properties are inherited from the environment, some other are set by style sheets linked to the SVG resource itself.

To summarize, style information may be internal or external to the document. When it is external, several mechanisms (processing instruction, `link` element, `@import` rule) are used to refer to style sheets, which may be located on different servers or locally. In all these style sheets several rules may be conflicting. Conflicts are solved by the cascade. On the other hand, some rules should not be considered, depending on the device used to present a document. In addition, in compound documents, some properties are inherited from "foreign" style sheets.

While these mechanisms offer both flexibility and power, they make the task of an author very complex: when adding or changing a style rule somewhere it is often not clear how it will impact the presentation of a document. Conversely, when some aspect of a document has to be changed, it is often difficult to locate the rules that have to be modified. In addition, the number of resources involved, as well as their distribution, make their management a nightmare in many situations.

Finally, browser implementation differences are often mentioned among the problems style designers have to face with CSS. While this was certainly an important issue a few years ago, the situation has much improved. A new version of the language, CSS 2.1, was recently published to document all CSS features that are implemented interoperably. In addition, the latest versions of the most popular browsers have also fixed a number of discrepancies and provide a good implementation of CSS 2.1. Choosing this version of the language is a simple way to achieve interoperability.

Although the issues related to CSS syntax have to be considered (selector syntax, property names, valid values), it appears that the most difficult problems an author has to face are those related to the combinatorial aspects of CSS, such as the distribution of resources, cascading, media selection, inheritance, etc. After all, CSS is probably not as simple as it seems to be.

2.2 State of the art

Even if literature is not abundant with publications on these issues, some software tools try to cope with the needs of CSS authors. In this section, we review the different kinds of tools that are currently available and the service they provide.

The most common aid XML editors provide for manipulating style sheets mainly addresses syntax issues. Several editors (Oxygen, CSSEdit, Style Master) include a CSS validator that detects syntax errors. Some also assist users in entering source code. Oxygen, for instance, provides an assistant for entering CSS code. It proposes the valid values for a property or provides explanations about the use of a property. CSSEdit automatically completes the syntax, inserting colon, semicolon and braces where needed. Style Master and Nvu provide a number of dialog boxes that generate the CSS syntax for the user when she has selected properties and values from menus and input areas.

There are also "pure" CSS editors, i.e. tools that handle only style sheets (not documents) and the CSS language, such as CSSEdit, Style Master, and JustStyle. Most of them can edit only one style sheet at a time, which does not help much when multiple style sheets are linked to a document. To check the style sheet under construction, users may use a web browser and see the result on any HTML document (but not on XML or compound documents, with these editors). This may help authors to find the relevant rules in the style sheet when an element is selected in the HTML document. Some restrictions apply however. For instance, with CSSEdit, only one block is found, even if several blocks are concerned, which strongly reduces the advantage of the mechanism. Conversely, when a selector is clicked in the style sheet, the corresponding elements are highlighted in the HTML document, but the tool cannot edit the HTML document, which serves only as a frozen testbed.

Specialized CSS editors, as well as Nvu or Dreamweaver which are also HTML editors (but not XML editors), allow authors to see the actual value of each style property for any element the user selects in the HTML document. Some of them can even show the style rule that was applied to produce this value, but only if it is in the style sheet being edited or in a local style sheet: they are unable to edit remote style sheets.

Links play an important role in CSS, to connect style sheets to documents, to import style sheets within other style sheets, to associate background images, etc. Surprisingly, none of the CSS editor we have reviewed provide any help for entering correct URIs for all these links, while typos are very frequent when typing cryptic addresses. It should be noted however that Nvu provides some help to link style sheets to HTML pages (but not to XML or compound documents).

This review of the available tools clearly shows that only a small subset of the many issues presented above are addressed. These tools mainly (or even only, for some of them) consider issues related to syntax and basic editing. A few of them are addressing some aspects of debugging. Clearly, more has to be done to really support web authors when they are faced with real life issues related to style.

In the following sections, we present methods for helping authors in style manipulation on the web. This is based on the experience we have gained in developing Amaya [9], where these methods have been implemented. Amaya is a web authoring tool that allows users to create, publish and update multiple web resources such as XHTML pages, XML documents, SVG graphics, MathML equations, CSS style sheets, and compound documents that use a combination of these languages. Amaya closely follows W3C standards and produce standard compliant documents. It allows authors to directly manipulate the structure, content and presentation of documents, but strictly speaking, it is not a WYSIWYG editor: it takes advantage of the heterogeneous nature of the web and allows documents to be presented differently on different devices. Note that some aspects of style editing in Amaya are presented in [10]; they are not discussed in detail here.

3. CREATING AND EDITING

A CSS editing tool should be tightly integrated with a document authoring tool. With word processors and document formatting languages such as TEX, people are used to manipulate style and content in the same environment. This is an advantage whatever the task being performed. When focusing on style, it is very convenient to be able to check the impact of new style rules on different kinds of structures. Loading existing HTML or XML documents is not sufficient. It should be possible to make small changes to the loaded documents to check how a style sheet behaves with some particular details of the document structure that are not readily available in the loaded document. Changing these details immediately, in the same environment is clearly more efficient than editing a document separately and loading it in the style editor. Conversely, when editing a document (its content and its structure), it is often necessary to tweak the style sheets to cope with a few cases that were not addressed correctly in the original design of the style sheets. In this case too, using a unique environment for editing documents and style is necessary.

3.1 Entering and testing rules

Obviously, all the features offered by CSS-only editors should be available in a document editor. The first of them is assistance in style rule input. In Amaya we have chosen to offer an alternative: authors are free to either type in CSS syntax directly or to use dialogue boxes that generate the CSS code. They can enter CSS syntax as plain text wherever the CSS language is used. At any time during input, if they need assistance, they can use dialogue boxes

that generate CSS syntax for them at the current position, which may be in a CSS style sheet, in the `style` element of a XHTML or SVG document, or in the value of a `style` attribute. Each of these dialogue boxes concerns a consistent set of properties, such as Characters (font-family, font-style, font-size, text-decoration, letter-spacing, etc.), Colors (color, background-color, background-image, border-color, border-width, border-style, etc.), Format (float, clear, display, visibility, position, top, bottom, left, right). Grouping properties in dialogue boxes helps manipulating related properties consistently: the Format box, for instance, gathers all properties that have an impact on positioning. In the Color box, the colors of the foreground and the background are next to each other, to clearly show the contrast of the chosen colors.

In the dialogue boxes a different tool is used for entering the value of each property, both to cope with the type of value to be entered and to make sure that users enter only valid values. These tools vary from simple text input areas to file selectors and color creation palettes (see Figure 2).

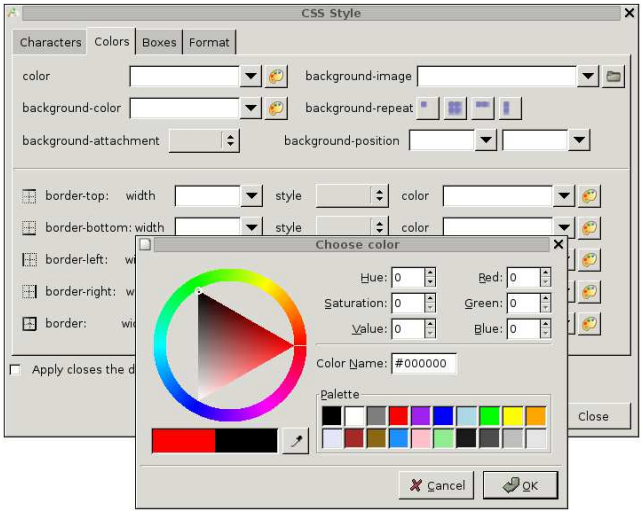

Figure 2: The color dialogue box

Rules entered through dialogue boxes are always syntactically correct, but as users are free to edit any part containing style information, including syntax generated by dialogue boxes, validity is not guaranteed. Therefore, modified CSS rules are checked on some specific events: when the value of a `style` attribute has been edited, when the author has made changes in a `style` element, when a style sheet is saved, and whenever the author wants to check the latest changes made to a style sheet.

When editing a style sheet, the user may decide at any time to apply this style sheet to some documents, to test the rules she has just created or modified. In that case, all documents that are currently loaded and that use the style sheet, are reformatted and redisplayed according to the current content of the style sheet. The style sheet is validated and errors are reported to the user, who can then check both syntax and semantics of her recent changes. All error messages can be clicked to go to the corresponding dubious statement.

It is useless, and even an hindrance, to apply a modified style sheet too often. Even when a style rule is complete, it may not make sense to apply it, because it is related to some other rule that has not been entered or changed yet. Therefore, the user should decide when the style sheet is consistent enough to be applied safely.

To help authors test the style sheets they write for different devices, Amaya allows them to select a media type that is then used for applying style rules. Amaya takes into account only the rules that are associated with the current media type. This feature combined with the previous one (applying the current style sheets to all loaded documents) has proven very efficient for developing style sheets with multiple `@media` sections or with documents using the `media` attribute in their links to style sheets.

The fact that Amaya can handle several documents and several style sheets simultaneously, be they local or remote, is an advantage. Different documents, presenting different structures can then be used simultaneously when validating a style sheet under construction, thus covering more test cases. Also, when several, inter-related style sheets have to be modified, editing is facilitated, and consistency is easier to check.

3.2 Classes

The `class` attribute from XHTML and some other XML languages plays a key role in CSS. It is heavily used in selectors to make a distinction between elements that have the same name but are specialized in different ways, and then need to be presented differently. To fully take advantage of this mechanism, it is important for the author of an XHTML document to know what classes are available for the document being edited. For that purpose, Amaya builds a list of all classes usable in each loaded document. It collects the values of all `class` attributes that are carried by elements in the document, as well as the classes that appear in the selectors of all style sheets used by the document, including the `style` element. This list is updated whenever new selectors and new `class` attributes are created.

The list of available classes is displayed in a dedicated tool (see Figure 4) that is used to put `class` attributes on document elements. The user selects an element in a document, clicks a class in the Class tool, and the editor creates a `class` attribute on the selected element, assigns it the chosen name as a value, and redisplays the selected element, taking into account all style rules that are selected by the new class.

This feature greatly helps authors in using classes in their documents. This is important for using style sheets more efficiently, but classes are also heavily used in microformats. Therefore, this mechanism has also proven to be very useful for microformats. This is a typical example of the advantage of editing style sheets and documents in the same environment.

3.3 Borrowing style

A simple way to develop a style sheet, in the very web tradition, is to borrow from someone else's style. Some style designers copyright their work, but a lot of sites offer style sheets that are freely available to anyone. If an author finds on such a web site an attracting style that fits her own needs, she can copy the relevant style rules. The issue then is to identify the complete set of rules that produce the desired effect. Even if it is produced by a single rule, locating the exact rule that is applied may be tricky. If several rules concur to produce the effect (see Figure 1), it is much more complicated. This problem is actually the same as the de-

bugging problem discussed in the next section. The solution to the debugging problem contributes to make it easier to borrow style from existing style sheets.

Note that borrowing style is facilitated when the editor can act as a browser too. In Amaya, authors can comfortably access documents on the web, just by following links, and they can locate that way documents with the desired style. Like in any web user agent, the style sheets are downloaded at the same time as the document itself and they are immediately available in the editor. It is then easy to copy and paste style rules.

4. DEBUGGING STYLE SHEETS

An author may be faced with the debugging problem in several situations. We have just seen the issue of borrowing existing style, but it arises also while writing new style rules, when the result is not exactly what was expected. Basically the problem is to find what rules are involved in a particular effect, be it desired (borrowing style) or not (a bug when writing new rules).

4.1 Two levels

This problem may be approached at two levels, i.e. at the style sheet level or at the rule level, depending on how style sheets are organized. In a modular design, different style sheets are used for different concerns. For instance, a style sheet may set all the colors for a document, another is in charge of the layout and the third controls characters (font, style, size, etc.). This typically makes it possible to substitute the color style sheet by an alternate version, for changing only the colors consistently, or to adopt a different layout independently of colors or characters. But sometimes a monolithic or less structured approach is taken, and all rules fall in the same style sheet, or the logic of their distribution in different style sheets does not appear clearly.

In the clean, modular case, approaching the debugging issue at the style sheet level works well. By removing or disabling a single style sheet, a user can see if it is the style sheet she is looking for. Repeating that operation with a few style sheets allows an author to quickly find the style sheet of interest. With well designed style sheets, this may be enough, in particular when borrowing style.

To facilitate this approach, Amaya includes a feature that allows users to selectively disable or enable the style sheets associated with a document. Each style sheet that applies to the document may be enabled/disabled, including the internal style sheet contained in the `style` element and the remote style sheets downloaded through the network.

When identifying a style sheet that way is not enough, or when there is only one style sheet associated with a document, the rule level approach has to be taken. The goal is then to find a single rule or a limited set of rules, in one or several style sheets. The process, from the user point of view, is presented in detail in [10]. It allows the user to select an element in a document and to see all the style properties, with their values, that have been assigned to that element. It also allows the user to find, in a single click, the exact rule that has assigned each property to this element, as well as the style sheet it belongs to.

The naive way to apply style sheets to a document consists in traversing the whole XML structure and, for each element, checking all selectors in all style sheets to collect the rules that match this element. Then the cascade is applied to the collected rules to keep only one rule per property in case of conflict. While this method is easy and produces the right result, it does not scale. With multiple, large style sheets, browsing through all style sheets is a waste of computing power. In addition, it does not help debugging unless the origin of each rule applied to each element is recorded, which is a waste of memory. A more efficient algorithm has to be used to provide acceptable performances in an interactive editing environment.

4.2 Algorithms and data structures

The main issues that have to be addressed arise when

1. formatting and displaying documents: For each element of the document structure, the relevant rules have to be found quickly in the set of style sheets, before applying them all to the element.

2. editing style sheets: When a style rule has been created, deleted or modified in a style sheet, all elements affected by that rule have to be found before that rule to be applied to them.

3. debugging style sheets: For a given element in a document, all style rules that affect that element have to be found (see Figure 4).

Issues 1 and 3 are actually the same, as far as style sheets are concerned: given an element, find all relevant rules. We call that the *direct* case of rule application. Issue 2 is the opposite: given a rule find all affected elements. We call it the *reverse* case.

The relationship between elements and rules is defined by selectors. Basically, the selector of a rule expresses conditions that an element must satisfy to be affected by the rule. The most general form of selectors is the *contextual* selector, which consists of a sequence of *simple* selectors, separated by symbols representing structural relationships. These relationships are *descendant* (represented by a space), *child* (represented by `>`) and *immediate sibling* (`+`). A simple selector is the name of an element (or `*` which matches any element) optionally followed by one or several conditions on attributes, each of the form `[att]` or `[att="val"]`. Both forms mean that the element must have an attribute `att`, but the second requires in addition that this attribute have a given `val`ue (there are a few options about value comparisons, but they can be ignored id we consider only the main principles of algorithms). As attributes `class` and `id` are often used in selectors, they receive a special notation: `.foo` means `[class="foo"]` and `#bar` means `[id="bar"]`.

As an example, in a XHTML document, the selector `body div.sect > p[lang="en"]` selects all `p`aragraphs having a `lang`uage attribute with value `en`glish that are children (`>`) of a `div`ision with an attribute class=`sect`, the division being itself a descendant (space) of an element `body`, whatever the attributes of `body`.

Although all simple selectors in this kind of expression follow the same syntax, the rightmost one can be considered differently from the others. It is the main part of the selector: the selector above first and foremost selects `p`aragraphs in `en`glish. The other simple selectors in the expression are additional conditions that are used to refine the selection according to the structural context. Also, in a simple selector, the main part is the element name and the attribute expressions are additional conditions (except when the element

name is the universal selector *). So, a contextual selector is best parsed a simple selector after the other, right to left, and each simple selector is best parsed left to right (element name first, then attributes).

4.2.1 Direct application of rules

Based on this observation, an efficient way of handling the direct case of rule application is to associate the rules governed by a selector with the element of its rightmost part and to consider all the rest of the selector as additional conditions to be tested in the order indicated above. Note that all conditions of a selector must be met for an element to be selected, and their order is important. More precisely, the order of simple selectors (right to left) must be followed, because it reflects the order in which elements have to be tested in the document tree, but the order of conditions on attributes within a simple selector does not matter (in an XML document, the order of attributes of an element is not significant).

This principle is followed in Amaya. When a style sheet is loaded, it is compiled. For each style sheet, the CSS compiler creates an *element table* with an entry for each element defined in the DTD of the document, plus an entry for the universal selector * (which matches any element). This is illustrated by Figure 3. Selectors are parsed as indicated above, and all the rules they govern are associated with the entry of the element table corresponding to the element name of the rightmost part. As an example, all the rules controlled by the selector above are associated with entry for element `p` in the table. The rest of the selector is stored as a list of conditions attached to the rule and the specificity of the selector is added to each rule. This integer is computed according to the CSS specification. It will be used to determine the priority of rules in the cascade. To each style rule is also added its `!important` indicator, as well as its position in the source style sheet (identification of the style sheet, plus line number). This will be used when debugging a style sheet, to quickly display the relevant rule in its style sheet.

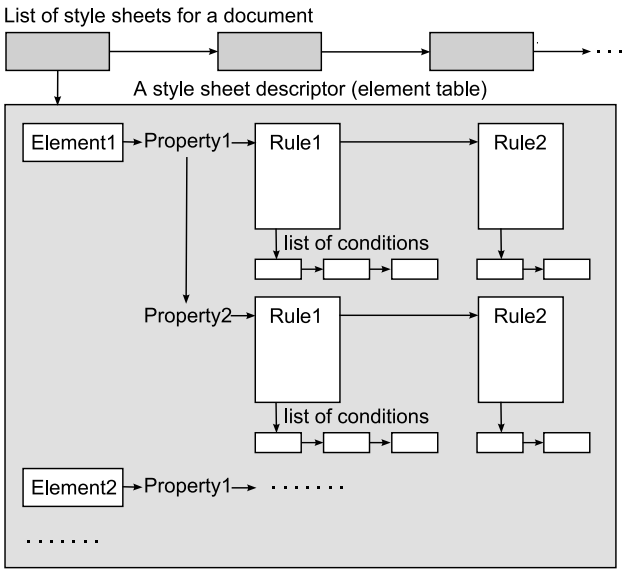

Figure 3: Data structures

When compiling a style sheet, all rules controlled by selectors that have the same element name in their rightmost part are linked to the same entry of the element table, as they are all candidate for application to the same elements (their conditions tell whether they must really be applied to a given element). These rules are sorted by properties: all rules for the same property are linked together. This speeds up the process of searching the rule that sets the value of a given property for a given element instance.

If a selector appears that has already been encountered in the current style sheet, the same entry of the element table is selected (same selector implies same rightmost element name). If a rule for the same style property is already present in that entry with a list of conditions corresponding to the selector, it is replaced by the new one, thus implementing at compile time the step of the cascade corresponding to the order of appearance of rules in a style sheet. The `!important` indicator is taken into account to avoid overriding a more important rule encountered earlier. This is an efficient way to merge blocks of rules that appear in several places in a style sheet with the same selector. This saves time when applying rules or when searching the rules that were applied to a given element (debugging).

The list of conditions attached to each rule for representing its selector is ordered as mentioned above. It is simply a linked list of simple tests, some of them complemented by a move in the document tree to prepare the next test:

- Is attribute `att` present on the current element?
- Is attribute `att` present on the current element and is its value `val`?
- Is the parent of the current element named `el`? Make it the current element in that case.
- Is there an element named `el` among the ancestors of the current element? Make that element the current element if found.

With that data structure, applying style rules to an element in a document is very simple. If there is only one style sheet, go to the element table entry corresponding to the element name and follow the linked list of properties linked from that entry. For each of these properties, check the lists of conditions of each rule. In a list of conditions, stop as soon as a test fails. If the end of the list is reached (all conditions are satisfied), keep the corresponding rule and link it to the *property table*. This table, unique for the whole system, collects links to the rules that are candidate for the current element. It contains an entry for each style property defined in the CSS language (the whole table is cleared each time a different element is considered). If a rule is already linked to the entry for a given property, the new rule replaces the existing one only if its specificity is higher; otherwise the new rule is ignored. This implements the specificity step of the cascade.

When multiple style sheets are associated with a document, each one is compiled and the element tables generated by the CSS compiler are linked in the order or their *origin*, the style sheet with the higher priority origin being linked at the end of that chain (see top of Figure 3). When gathering the rules to be applied to an element in a document, the element tables are used as explained above, but in the order of the chain built according to their origin. This implements

the step of the cascade related to the origin of style sheets: a higher priority rule replaces a lower priority rule in the property table.

Finally, when all style sheets have been handled, if the element has a `style` attribute, the style rules contained in this attribute are put in the property table: they have the highest priority. When this is done, the property table contains all the style rules that have to be applied to the element of interest.

If the goal is to format and display the document, these rules are immediately applied to the element and the property table is cleared, ready for processing the next element. If the goal is debugging, the property table is displayed in a human readable form in a small window, called the debugging window (see Figure 4). As the CSS compiler has associated with each rule the information about the style sheet it comes from and its position within this style sheet, the editor can access immediately the actual rule that has set each style property for the element of interest. The user can click any of the properties displayed in the debugging window. This opens up and displays the relevant style sheet, highlighting the rule that has set the chosen property. The user can then edit this rule, and she does so directly in the style sheet, with the full context provided by the surrounding rules.

Figure 4: Debugging a style sheet

The property table for an element contains only the properties that are set by the style rules applied to the element itself. However, it happens often that an element inherits style properties from its ancestors in the document structure, and it may be difficult for a user to figure out what element in the hierarchy has fired a style rule that affects the element of interest. Some CSS editors (Dreamweaver for instance) display all properties for a given element in a flat manner, and the user has to click each of them to see if it is inherited and from what element.

The mechanism implemented in Amaya is different: when the debugging window is open, like in Figure 4, it displays only the properties that apply to the selected element itself, not the inherited properties. This list is updated every time the user selects a different element. To find inherited properties, users select ancestor elements in the document tree. Travelling up the document structure is done in a single keystroke in Amaya. This allows authors to quickly check which of the ancestors sets a particular property. When the property appears in the debugging window, clicking it shows the corresponding rule in its style sheet, as we have seen above.

Allowing users to retrieve easily a particular style rule has an impact on the quality of style sheets. When authors are unable to locate the rule they would like to change, it is common practice to just add a new rule at the end of the highest priority style sheet, thus taking advantage of the cascade to override a faulty rule, which remains somewhere in the style sheets. After several additions of that kind, style sheets become bloated and unreadable, containing many unused rules.

4.2.2 *Reverse application of rules*

Now, consider the opposite problem: how to find the elements affected by a given style rule of a given style sheet? Based on the structures built by the CSS compiler, the answer to that question is easy: for each loaded document, go to the chain of element tables built by the CSS compiler and look for the one that corresponds to the style sheet of interest (the CSS compiler has associated each table with its style sheet). If this element table is present for the document, it means that the rule may affect the document. Otherwise, the document does not use the style sheet, and can not be affected. Take the rightmost element name in the selector of the rule of interest, and traverse the document tree, looking for elements of that name. It is not enough to check all the conditions of the selector for these elements, as the cascade may cause other rules, possibly in other style sheets, to supersede the rule of interest.

Given the structure already built by the compiler, the simplest way to solve the issue is to build, for each candidate element, the table of all rules to be applied, in exactly the same way as for a direct application. Then, it remains to check whether the rule of interest is in this table or not. This can be done quickly, because the table contains usually very few rules: inherited properties are not there.

Finally, the algorithm that builds the property table for a document element is the key point in editing and debugging style sheets, as it makes it possible to solve the main issues in a very efficient way that scales well with the size and the number of style sheets associated with a document.

5. STYLE SHEETS ON THE WEB

Many CSS style sheets are shared web resources, exactly like web pages. Style sheets can be accessed remotely. They can be linked to several documents. They do not need to reside on the same server as the documents that use them. They can import other style sheets from remote servers. They may include style properties that use images (properties `background-image` and `list-style-image`), which are external web resources. Manipulating style sheets requires manipulating many web resources. More precisely, the main actions authors should be able to perform on the web are:

- creating a new style sheet locally and publishing it on a web server,
- downloading a remote style sheet to edit it locally, before uploading it back,

- copying a remote style sheet and saving it locally or on another server after editing,
- importing local or remote style sheets in a style sheet,
- linking local or remote images to a style sheet,
- linking local or remote style sheets to a local or remote document.

To enable all these operations, four basic services are necessary:

1. downloading a remote resource (style sheet, document, image),
2. uploading a local resource on a remote server,
3. creating links to resources,
4. updating links when linked resources are moved.

Operations 1 and 2 are symmetrical and involve remote web access. The natural way to interact with resources on the web is through the http protocol. These two actions are based respectively on the GET and PUT methods of http.

Operation 3 covers the various links used in or to style sheets, which have different forms. A style sheet is linked to a document through a `link` element (HTML pages) or a XML processing instruction (XML documents). A style sheet is linked to another style sheet with the `@import` rule. An image is linked to a property in a style sheet using a `url()` value. Whatever the syntax used, there is always an URI. Entering an URI by "hand" is both tedious and error-prone. A comfortable and safe approach consists in designating the resource to be linked by clicking it on the screen, instead of typing its URI. This method has two advantages:

- by requiring the resource to be displayed on the screen, it allows the user to check that this is really the resource she means,
- by avoiding typing and using instead the address of a resource that was successfully loaded, it ensures that the URI is always correct.

This approach works well in Amaya, because the tool is not only an editor for documents and style sheets, but also a browser, both features being seamlessly integrated. Thus, authors can access the resource of interest by browsing, using bookmarks, following links and using search engines. When the resource is on the screen, they create a link to it simply by pointing at it. The tool is in charge of generating the right syntax according to the type of link, and generating the valid URI (when loading a resource, Amaya keeps its URI; it can then easily return the URI when the user designates the resource).

Operation 4 is involved when a resource is published on the web for the first time (web page, style sheet, image) or when it is saved to a different location (locally or remotely). URIs in the various types of links have then to be created or updated. As an example, like every editor, Amaya offers a "Save As" command that can be used to save a resource to a different location (local file or remote web server). A frequent case is to save a web page to a new location, along with it associated resources, i.e. the style sheets and images it uses, but also the style sheets and images linked at multiple levels from the first level style sheets.

To execute this command, Amaya identifies all the resources involved, simply by following the relevant links recursively from the document. Then, it changes the URI of all these links, to take into account the new location where each resource will be saved. Finally, it saves all the resources with http PUT. When saving a style sheet that imports other style sheets and that uses images, the same processing is done.

This capability of working directly on the web is very specific to Amaya. All the tools we have reviewed require that (a part of) a web site be copied locally before editing it. Style sheets are no exceptions. This is not only tedious, but these local copies practically prevent users from working cooperatively, thus missing an important feature of the web.

6. CONCLUSION

In this paper, we have reviewed the major style issues a web author is faced with and we have presented solutions that can be offered by an authoring tool. These include methods for entering style properties, for retrieving the elements affected by a style rule and for retrieving the rules applied to a given element in a document, even when multiple style sheets are involved and when inheritance transmits style. We have also proposed solutions to address the difficulties related to the wide distribution of the multiple resources that define the style of a document.

The proposed solutions have been implemented in Amaya. By addressing most of the CSS issues and providing a comprehensive web authoring environment, Amaya constitutes an original tool. As opposed to "pure" CSS editors, it allows authors to work simultaneously on style, structure and content and to get a better understanding of the behaviour of their style sheets on real documents. As opposed to HTML editors that support style manipulation, it takes advantage of its double role of browser and editor to work directly on remote web resources. This makes it easy to share and reuse style resources. Another significant difference with HTML editors is that Amaya handles several XML formats, including compound documents. As a consequence, the style editing techniques discussed in this paper are usable not only for (X)HTML documents, but also for SVG, MathML, generic XML, and documents that use several of these languages.

Progress on style editing has a consequence for XHTML documents. Sophisticated CSS style sheets require that the XHTML pages to which they are associated be very precisely structured, to fully take advantage of the most complex selectors. Classes combined with "structuring" elements such as `span` and `div` play a key role in many style sheets. If they are not used correctly in the XHTML page, a number of style rules do not fire, and the document is poorly presented or even unreadable. Templates [4] constitute an efficient way to help authors to structure documents correctly and thus to fully exploit the features offered by advanced style sheets. Work on templates will then reinforce work on style.

Although some advances have been reported in this paper, the issue of editing style still requires more efforts. For instance, these authors are not aware of any technique that could efficiently help users in building CSS selectors. The issue of creating a consistent set of style rules for a few elements to correctly set their relative positions and sizes (see Figure 1) is not fully addressed. The cascade also needs more research.

7. ACKNOWLEDGEMENTS

The authors are grateful to W3C for their continuous support and contribution to the development and distribution of Amaya. The whole Amaya community is also acknowledged for their valuable contribution to the evolution of the software.

8. REFERENCES

[1] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 73–82, New York, NY, USA, 1999. ACM Press.

[2] A. Berglund. Extensible stylesheet language (XSL) version 1.1. W3C Recommendation, http://www.w3.org/TR/xsl/, 5 December 2006.

[3] B. Bos, H. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2. W3C Recommendation, http://www.w3.org/TR/CSS2/, 12 May 1998.

[4] F. Campoy Flores, V. Quint, and I. Vatton. Templates, microformats and structured editing. In *Proc. 2006 ACM Symposium on Document Engineering, DocEng 2006*, pages 188–197. ACM Press, Oct. 2006.

[5] I. Jacobs and N. Walsh. *Architecture of the World Wide Web, Volume One*. W3C Recommendation, http://www.w3.org/TR/webarch, Dec. 2004.

[6] H. W. Lie. *Cascading Style Sheets*. Phd thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, Feb. 2006.

[7] P. Marden and E. Munson. PSL: An alternate approach to style sheet languages for the world wide web. *Journal of Universal Computer Science*, 4(10):792, 1998.

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

[9] V. Quint and I. Vatton. Techniques for authoring complex XML documents. In *Proc. 2004 ACM Symposium on Document Engineering, DocEng 2004*, pages 115–123. ACM Press, Oct. 2004.

[10] V. Quint and I. Vatton. Towards active web clients. In *Proc. 2005 ACM Symposium on Document Engineering, DocEng 2005*, pages 168–176. ACM Press, Nov. 2005.

[11] L. Villard and N. Layaïda. An incremental XSLT transformation processor for XML document manipulation. In *WWW2002, the 11th International World Wide Web Conference*, pages 474–485. ACM Press, May 2002.