INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Toward Extending Stylesheet Languages with Dynamic Document Rendering Features

Christoph Wieser

wieser@cip.ifi.lmu.de

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Abgabe der Arbeit: 12.08.2005
Betreuer: Prof. Dr. François Bry
Sacha Berger

Abstract

This thesis covers extensions to style sheet languages (like CSS [W3C98] or XSL [W3C01a]) toward statical and dynamical rendering for an easier visualization of semi-structured data.

Style Sheet Languages (like CSS) already offer dynamic rendering but do not have sufficient expressiveness for many dynamic applications like folding and unfolding of boxes. However, languages (like XSLT or ECMA Script [ECM99] having more expressive power are rather difficult to use. The static and dynamic extensions proposed here allow a declarative and, hence, easy definition of dynamic applications.

CSS serves as embedding language for the extension. As a test case for the extensions the visual query and transformation visXcerpt [Ber03, BBSW03] is implemented.

Zusammenfassung

Diese Projektarbeit behandelt Erweiterungen von Style Sheet Sprachen (wie CSS [W3C98] oder XSL [W3C01a]) um statisches und dynamisches Rendering zur einfacheren Visualiserung semi-strukturierter Daten.

Style Sheet Sprachen (wie CSS) bieten bereits dynamisches Rendering an, verfügen aber nicht über hinreichende Ausdruckskraft für viele dynamische Applikationen wie das Auf- und Zuklappen von Boxen. Ausdrucksstärkere Sprachen (wie XSLT oder ECMA Script [ECM99]) degegen sind meist komplex zu benutzen. Die hier vorgeschlagenen statischen und dynamischen Erweiterungen ermöglichen eine deklarative und somit einfache Definition von dynamischen Applikationen.

Als Sprache zur Einbettung der Erweiterungen dient CSS. Als Test-Fall für die Erweiterungen ist das Rendering der visuellen Anfrage- und Transformations-Sprache visXcerpt [Ber03, BBSW03] implementiert.

Acknowledgment

Several people have contributed directly or indirectly to the development of this project thesis. I want to express my gratitude to the members of the teaching and research unit "Programming and Modelling Languages" for giving precious advice and for offering a familiar atmosphere.

More than to any other person, I want to thank my supervisor Prof. Dr. François Bry for his patience, dedication, and thoroughness in reviewing the text and the content of this work. His encouraging motivation during the evolution of this thesis was beyond comparison.

I am also deeply indepted to my advisor Sacha Berger. Many of his visionary ideas enriched the work on this thesis. Without his refreshing and sometimes unconventional views, the thesis would lack of many interesting associations.

Contents

1 Introduction

Style sheet languages such as CSS [W3C98] or XSL-FO [W3C01a] have considerably gained in importance since the Web has become a mass medium. Such languages are widely applied for a sophisticated rendering of semi-structured data [ABS00] especially expressed using XML [W3C00b]. Controlling the appearance of Web pages in Web browsers has become the most frequent application of style sheet languages.

This thesis describes extensions to (Web) style sheet languages. Two types of extensions are proposed: extensions of the already existing 'static styling' features and extensions towards additional 'dynamic styling' features. Static styling refers to styling without action of a viewer. For instance, all portions of a structured text that are marked up as headings can be rendered using bold letters. In contrast, dynamic styling denotes styling as a consequence of a viewer's interaction. An example of dynamic styling supported by current style sheet languages is as follows: A viewer can change the styling of a portion of text in a Web browser by letting the mouse cursor hover over such a text portion. In other words, static styling refers to a styling that remains unchanged during viewing, whereas dynamic styling refers to a styling that might change as a consequence of a viewer's action during viewing.

Extensions to current style sheet languages (e.g., static and dynamic styling extensions) can be beneficial for several applications. Let us briefly discuss a few of the extensions proposed in this thesis and examples of applications that can benefit from these extensions.

A first extension addresses styling of subelements at arbitrary nesting depths like in the following HTML document. The document shows an excerpt of the highly nested 'Tree of Life'[1], a well known classification of living organisms.

Tree of Life

- **Eubacteria**
- **Eukaryotes**
  - *Animals*
    - **Echinoderms (sea urchins, starfish, sea cucumbers, etc)**
    - **Vertebrates (fish etc.)**
    - ...
  - *Green Plants*
    - **Ferns**
    - **Flowering Plants**

...

Figure 1: Tree of Life

The items of the tree are styled in alternating manner as follows: Items having an odd nesting depth are styled using bold font, and items having an even nesting depth are styled using italic font. Obviously, the alternating styling of fonts depending on the nesting depth helps

[1] `http://tolweb.org/tree/home.pages/popular.html`

to recognize the structure of the tree and helps to compare different parts of the document.

Such a styling is not possible with current CSS. Current CSS allows the styling of items on a certain depth only. Hence, for a correct alternating styling, the maximal nesting depth of a document must be known in current CSS. In concrete, one rule for each level of nesting depth until the maximum level is necessary as the following CSS program illustrates:

```
*             { font-weight : bold;   }
* *           { font-style  : italic; }

* * *         { font-weight : bold;   }
* * * *       { font-style  : italic; }
...

* * * ... *   { font-weight : bold;   }
* * * * ... * { font-style  : italic; }
```

Figure 2: Styling until a certain depth in CSS

As proposed in this thesis, cf. Section 4.3, extensions make an alternating styling independent of nesting depth possible. The extensions provide with styling rules that can be applied repeatedly depending on the nesting depth. Therefore the nesting depth of a document does not need to be known while writing its style sheet.

A second extension refers to dynamic styling of semantically related text portions. Footnotes are often used to annotate text portions. In the Web context, so called ‘side-notes’ are located beside the text like in the following example.


Cascading Style Sheets (CSS) is a simple mechanism for adding style (e.g. fonts[1], colors[1], spacing[1]) to Web documents. Tutorials, books, mailing lists for users, etc. can be found on the **“learning CSS” page**[2]. For background information on style sheets, see the **Web style sheets page**[2]. Discussions about CSS are carried out on the (archived) **www-style@w3.org mailing list**[2] and on **comp.infosystems.www.authoring.stylesheets**[2].

side-notes:
[1]Styling Aspects
[2]**Manuals**

Figure 3: Highlighting of semantically related text portions

Hovering with a mouse cursor over a side-note as in the example above can cause a highlighting of every semantically related text passage somewhere else in the document. Such a rendering would provide good support for an easier and better understanding while reading texts.

Current CSS or XSL-FO does not support such highlighting of related text portions. Helper tools beyond CSS and XSL-FO such as scripting languages (e.g., ECMA script [ECM99]) are needed to change the styling while viewing a document in a Web browser. A slight extension to CSS introduced in Section 4.5 allows a declarative way to define such dynamic highlighting of semantically related text portions within a style sheet language without additional tools.

A third extension is the static visualization of (parts of) the markup itself. Viewing XML data such as a timetable of trains in a text editor asks for a more concise representation (cf. left column of the following example). An extension proposed in this thesis (cf. Section 4.2) can be applied to render markup. Each XML-tag (such as <train>) can be rendered, e.g., in a Web browser as follows:

Source

```
<trains>
   <train number="ICE788">
      <departure>
        <station>Munich</station>
        <time>11:36</time>
      </departure>
      <arrival>
        <station>Hamburg</station>
        <time>17:45</time>
      </arrival>
...
```

Rendering

- trains
  - train (number ICE788)
    - departure
      - station Munich
      - time 11:36
    - arrival
      - station Hamburg
      - time 17:54

  ...

Figure 4: Rendering of markup

While current CSS allows to render known XML tags and known XML attributes only, XSLT allows the visualization also of unknown XML tags. However, such XSLT programs tend to get rather complex, as the following XSLT program transforming the train timetable to HTML illustrates. In this short version of the program, the rendering of attributes is not considered for simplicity reasons. (For the full version refer to Fig. 42.)

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>
  <xsl:template match="*">
    <ul>
      <xsl:for-each select=".">
        <li>
          <xsl:value-of select="name()" /><xsl:text> </xsl:text>
          <xsl:apply-templates />
        </li>
      </xsl:for-each>
    </ul>
  </xsl:template>
</xsl:stylesheet>
```

Figure 5: XSLT program rendering the XML file of Fig. 4

The extensions proposed in this thesis offer a more declarative (hence easier to use) way to render such marked up data as introduced in Section 4.1.

A fourth extension refers to dynamic styling of text portions by folding and unfolding them. As data like the train timetable of the latter example can become rather long, a better overview of the data is highly welcome. Folding undesired data and unfolding desired data helps to find information much easier.

- train (ICE 788)
  - departure
    - station Munich
    - time 11:36
  - arrival
- train (ICE 586)
- train (ICE 584)
- train (ICE 784)
  - departure
    - station Munich
    - time 15:44
  - arrival
- ...

Figure 6: Folded (e.g. `train (ICE 586)`) and unfolded data items (e.g. `train (ICE 784)`)

In the example above, the viewer has folded sub-items of the already known train timetable by clicking with a mouse cursor on the parent item like the train item. All items except of two train items are folded. Now, the viewer can compare the information about these two trains. Since the viewer is interested in departure times, all other information about the two trains are folded. This rendering makes the comparison of the departure times easy.

Note that all four kinds of extensions mentioned afore are highly useful in practice. For instance, working in an office, tasks like the orientation in multilateral documents, the analysis of spreadsheets, or the administration of databases should take rather short time. Hence, the rendering of such data should be optimized for human beings to permit a faster recognition of facts and their contexts. The extensions proposed in this thesis offer a declarative way to define such an optimized rendering.

The static and dynamic extensions to style sheet languages proposed in this thesis are in essence compatible with any style sheet language. In this thesis these extensions are worked out in CSS. CSS has been chosen because it is convenient to draw on Web standards to reduce implementation effort. CSS has been chosen instead of, e.g., XSL, for the following reasons:

- CSS is widespread because of its simple rule-based syntax. Thanks to this rule-based syntax only minor extensions of CSS are needed, while for style sheet languages such as XSL-FO, more significant extensions would be needed.

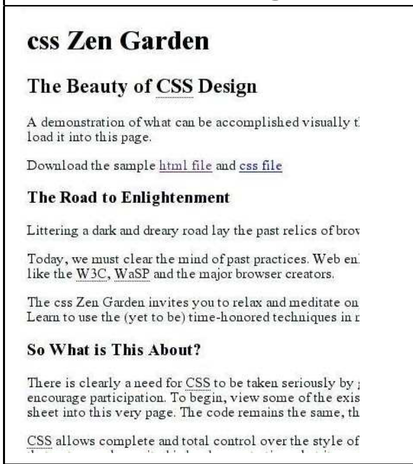
- CSS is implemented in almost every common Web browser and thus it can be easily used from different environments.

2 Cascading Style Sheets (CSS)

This section gives a brief introduction into the W3C's Style Sheet Language *Cascading Style Sheets (CSS)*. Concepts of CSS serving as basis for the extensions proposed in this thesis (such as CSS selectors) are addressed in detail. For details on the remaining concepts of CSS (such as font styling or positioning) refer to the full reference of CSS [W3C98].

CSS has been developed to simplify changes of the content as well as of the design of HTML documents by separating design from content. As a consequence of this separation minor changes on an HTML document (e.g., another styling for headings) require minor changes on a CSS style sheet. Without separation of design from content such minor changes cause a complete revision of a HTML document so as to modify, e.g., each heading in an HTML document.

HTML Rendering



CSS Rendering

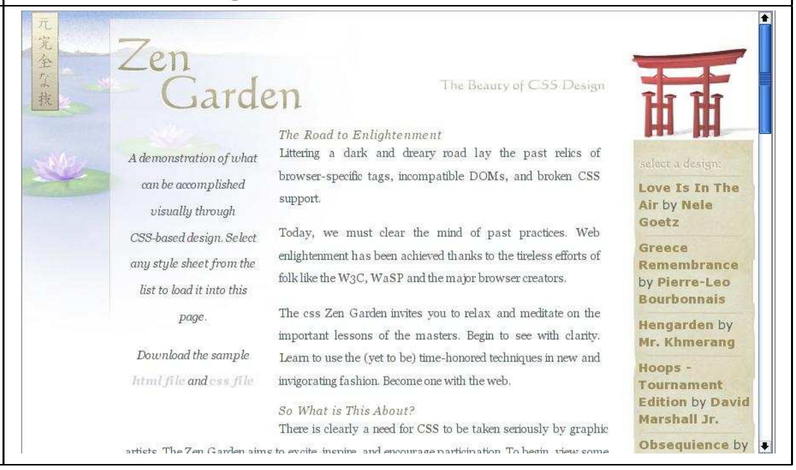


Figure 7: Demonstration of CSS Styling

As the figure above illustrates, CSS is capable of styling that is not possible with standard HTML styling. On the left side, the HTML page 'CSS Zen Garden'[2] is rendered without a user defined style sheet according to the guidelines of the HTML specification [W3C00a]. On the right side, the same HTML document is rendered using a CSS style sheet. (Note that the excerpt of the original rendering on left side shows less text by comparison to the right side only because of layout reasons.) Obviously, one HTML page can be styled by various style sheets. Currently, more than 600 additional style sheets[3] are available for the example given above. Voluntary designers wrote these style sheets on request of Dave Shea, a web designer from Vancouver (Canada), to demonstrate the advantages of CSS.

According to the W3C recommendation "Associating Style Sheets with XML documents" [W3C99a] CSS style sheets can also be used to render arbitrary XML documents and not only HTML documents, using the following XML processing instruction (PI):

[2] http://www.csszengarden.com/zengarden-sample.html
[3] http://www.mezzoblue.com/zengarden/alldesigns/

```
<?xml-stylesheet type="text/css" href="stylesheet.css" ?>
```

Figure 8: XSLT program for rendering markup

Currently, the latest CSS version is CSS 2.1. The next version CSS 3 is still under development. Although CSS 3 is not yet a recommendation of the W3C, we will also draw on concepts of CSS 3 to reduce reimplementation efforts after CSS 3 is published.

2.1 Structure of CSS

A Cascading Style Sheet consists of a sequence of CSS rules. Each rule is separated into two parts: The head of a rule is called *selector* and the body of a rule is called *definition*. The main structure of a CSS rule is given in EBNF syntax as follows:

```
RULE       ::= SELECTOR "{" DEFINITION "}" .
DEFINITION ::= ( TUPLE )* .
TUPLE      ::= ( PROPERTY ":" VALUE ";" ) .
```

Figure 9: Main Structure of a CSS Rule in EBNF

The `SELECTOR` part of a rule matches (zero or more) XML nodes, and the `DEFINITION` part specifies the styling of the currently selected XML nodes. Concrete aspects of styling are addressed in the `DEFINITION` part as so called `TUPLE`s. Each of these `TUPLE`s consists of a `PROPERTY` (such as `font-weight` or `background-color`) combined with adequate `VALUE`s (such as `12pt` or `white`).

Note that comments in CSS are marked by “`/*`” as opening tag and “`*/`” as closing tag. Comments can appear everywhere in a CSS style sheet.

The following figure demonstrates, how the repository of a library written in XML can be styled using CSS:

Source

```
<bib>
  <book year="1994">
    <title>
      TCP/IP Illustrated
    </title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>
      Addison-Wesley, 1994
    </publisher>
    <price>65.95</price>
  </book>
  ...
</bib>
```

CSS Style Sheet

```
*             { display         : block; }
bib           { list-style-type : decimal;
                margin          : 1em 0;
                padding-left    : 40px; }
book          { display         : list-item; }
author        { font-style      : italic; }
first, last   { display         : inline; }
price::before { content         : "EUR "; }
```

Rendering

1. TCP/IP Illustrated
   Stevens W.
   Addison-Wesley, 1994
   EUR 65.95

...

Figure 10: Example: Rendering of XML data using CSS

The rule in the **first line** of the CSS style sheet above selects all XML nodes by the wildcard pattern (`*`). Hence, all XML nodes are primarily styled as specified in the definition part of that rule. In this case, all elements are arranged in blocks (and not as continuous text). In the **second line**, the `bib` element is defined as container for a numbered list with a concrete indentation of `40px` and a margin around the list. This numbered list consists of `book`s stated in the **fifth line**. In the **sixth line**, each of the `author`s is rendered in italic style but unlike the other elements the `first` and the `last` name are arranged in a row in **line seven**. The **eighth line** causes the String `"EUR "` in front of each price tag.

The following subsections introduce to selected concepts of CSS in detail. In particular, concepts being extended in section 4 like the CSS selector concept are addressed.

2.2 Simple Selectors

The basis of CSS selectors are so called *simple selectors*. As already applied in Section 2.1, simple selectors can select XML elements by pattern matching: The most general pattern is the star (`*`) which matches every XML element. A selection can be restricted to XML elements bearing the same name. For instance, the selector `author` instead of the star (`*`) selects `author` XML elements only. Further on, selections can be restricted to attribute names and values. The following rule, e.g., matches all `book` elements having an XML attribute `year` with `1994` as value:

```
book[year="1994"] { ... }
```

Figure 11: Example: Rendering of XML data using a Simple Selector

2.3 Combinators

Being able to select individual types of XML elements using simple selectors is a first step. The next step is to select XML elements being in a relationship to other XML elements. Two simple selectors can by related by the following CSS combinators:

E F	Matches any F element that is a descendant of an element E.
E > F	Matches any F element that is a child of an element E.
E + F	Matches any F element immediately following an element E in the so-called document order.
E ~ F	Matches any F element following an element E in the so-called document order.

Figure 12: CSS Combinators

The following rule, e.g., selects the `title`s of all `book`s. Other `title` elements having no ancestor node `title` are not selected:

```
book title        { ... }
```

Figure 13: Example: Rendering of XML data using a Combinator

2.4 Grouping

Obviously, XML elements of different types can be rendered using the same CSS `definition`s. Hence, each type of an XML element such as `first` or `last` can be rendered using its own rule like in the first and the second line of the following example:

```
1 first         { display : inline; }
2 last          { display : inline; }
3
4 /* Grouping of the previous rules */
5 first, last   { display : inline; }
```

Figure 14: Grouping of CSS rules

To simplify writing of CSS style sheets an to simplify changes on a set of elements, CSS allows grouping of elements. In the fifth line of the example above (cf. library example 2.1) the CSS rules of the lines one and two are grouped together.

Note, that the grouping operator "`,`" can be easily mixed up with CSS combinators introduced in Section 2.3.

2.5 Pseudo-selectors

Selecting, e.g., the first child of an XML element cannot be expressed by applying simple selectors and combinators only. Therefore, CSS offers pseudo-selectors for selections that go beyond the expressive power of simple selectors and combinators. In this section both types of pseudo-selectors are introduced: so called 'pseudo-elements' and 'pseudo-classes' are added to selectors. pseudo-elements address selections concerning the surroundings of one XML elements such as the first letter of a text node. Pseudo-classes refer to the context of an XML element that cannot be expressed by simple selectors and combinators only, such as selecting XML elements being the third child in XML document order [W3C00b].

2.5.1 Pseudo-elements

According to the specification of CSS [W3C98], pseudo-elements allow 'language specific' (e.g. HTML specific) selections such as the selection of the first letter or the first line of a paragraph. For instance, (e.g. ancient) texts can be rendered using majuscules, where the first character of a paragraph is rendered differently from the other characters. Since the first character of a paragraph is usually not surrounded by XML tags, it can not be selected by a simple selector. The pseudo-element `::first-letter` makes such a styling possible.

Beside selecting text portions inside an XML element like the first letter or the first line, pseudo-elements can also select 'virtual' XML text nodes before opening XML tags or after a closing XML tags. The following example renders the XML element `price` with additional text defined in the style sheet.

Source (extract)

```
  </publisher>
            <!-- virtual XML text node -->
  <price>65.95</price>
            <!-- virtual XML text node -->
</book>
```

CSS Style Sheet (extract)

```
price::before { content : "EUR "; }
```

Rendering

EUR 65.95

Figure 15: Selecting 'virtual' XML text nodes

The CSS rule on the right side (taken from the library style sheet given in Section 2.1) demonstrates the insertion of text: The currency `EUR` is inserted in front of the price of a book using the pseudo-element `::before`. Besides `::before`, the pseudo-element `::after` allows insertions of text behind a rendered XML element. In both cases, the inserted content is part of the style sheet and not part of the source document (like all other content such as `65.95`). Hence, another style sheet could insert the currency € instead of `EUR`.

2.5.2 CSS Functions

Besides the insertion of fixed text, CSS functions grant *limited access* to the markup of a source document. For instance, the rendering of HTML ordered lists (`<ol>...</ol>`) demands to insert the current number of each list item. CSS offers the following functions to afford insertions depending on the source document:

- `counter(name)`, `counter(name, style)`, and `counter(name, string, style)` insert the numbers of enumerations.
- `attr(X)` returns the value of an XML attribute `X` to the subject of the CSS selector as string. If the subject of the selector does not have an attribute X, an empty string is returned.

These CSS function are used within the `content property` of a CSS rule in combination with the pseudo-Elements `::before` or `::after`. Referring to the library example (cf. Section 2.1) the XML attribute `year` of a `book` could be rendered using the following additional CSS rule:

Source (extract)

```
<bib>
  <book year="1994">
    <title>
      TCP/IP Illustrated
    </title>
  <!-- ... -->
```

CSS Style Sheet (extract)

```
book::before{content:"published " attr(year): ";}
```

Rendering

1. published 1994: TCP/IP Illustrated
   ...

Figure 16: Example: Rendering of XML data using CSS

Note, that both of the offered CSS functions do not cover computational capabilities.

2.5.3 Structural Pseudo-classes

CSS introduces the concept of structural pseudo-classes to permit *"selections that are based on information lying in the document tree but cannot be represented by other simple selectors or combinators"* [W3C98].

The lines of a rendered list such as in the library example (cf. section 2.1) can be rendered using alternating colors to support a better orientation. Since the (even or odd) position of a list item cannot not be derived using simple selectors, Web designers use XML attributes to mark up even and odd items in CSS 2.1. CSS 3 offers structural pseudo-classes that can be used for a rendering depending on the relative position of a list item. The following CSS rule, e.g., renders the first item in a bold font (line one) and turns the background color of every second book item into gray beginning with the first one (line two):

```
1 bib:nth-child(2n+1)   { font-weight      : bold; }
2 bib:nth-child(2n+2)   { background-color : gray; }
```

Figure 17: Structural Pseudo-classes

The selector `:nth-child(An+B)` matches an XML element that has "$A \cdot n + B - 1$ siblings before it" [W3C05] in the XML document tree. In other words, the selector matches every `a`-th XML element beginning with the `b`-th sibling. While `A` and `B` stand for integers, `n+` is a reserved word. Beside the `:nth-child(An+B)` selector, other selectors of this category allow to match the XML document root and siblings beginning with the last XML element. We refer to the CSS reference [W3C05] of the W3C for a detailed reference on the remaining structural pseudo-classes.

2.5.4 Dynamic Pseudo-classes

In contrast to pseudo-elements and Structural pseudo-classes, Dynamic pseudo-classes denote all selections that cannot be deduced from the document tree, e.g. selections that depend on interaction of the viewer.

In a Web browser window, e.g., dynamic rendering of hyperlinks can be observed on many Web pages[4]. Hovering with the mouse cursor[5] over a hyperlink anchor can change its styling such as underlining the link. Removing the mouse cursor from the hyperlink anchor again activates the former styling. The following CSS rule causes such a dynamic rendering:

```
a:hover       { text-decoration : underline; }
```

Figure 18: Dynamic Rendering

Besides the `:hover` dynamic pseudo-class CSS offers the dynamic pseudo-classes `:active` and `:focus`. `:active` selects the hyperlink that points to the current Web page itself. Finally, the `:focus` pseudo-class selects the area where the text cursor is focused on. In XHTML this could be a text field of a form.

[4] cf. `http://www.ifi.lmu.de`

[5] A mouse cursor is controlled by *mouse devices* like 'mouses', 'touch pads', or 'track balls'.

2.6 Interpretation of CSS style sheets

The basic interpretation concept of CSS programs is based on the so called ‘cascading styling’. Similar to the concept of inheritance in object oriented languages (such as Java or C++) the rendering of the parent XML element is ‘inherited’ to each child XML element. Referring to the following example (cf. library example of Section 2.1) the `first` name and the `last` name of each `author` are rendered using an italic font-style, although no CSS rule defining the font style matches these XML elements.

Source (extract)

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

CSS Style Sheet (extract)

```
author   { font-style : italic; }
```

Rendering

Stevens W.

Figure 19: Cascading Styling

Further on, cascading styling allows to modify the scope of a CSS rule. To modify, e.g., the rendering of the `last` name in the example above, an additional CSS rule matching the `last` name can be introduced:

Source (extract)

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

CSS Style Sheet (extract)

```
1 author  { font-style : italic; }
2
3 /* Additional CSS-Rule: */
4 last    { font-style : normal; }
```

Rendering

Stevens *W.*

Figure 20: Overwriting CSS rules in depth

According to the figure above the `last` name of the `author` is styled in ‘normal’ font style and not in italic font style as defined in line one of the style sheet. Hence, the definition of the parent XML element `author` is shaded.

CSS rules can be shaded not only in depth such as in the latter paragraph but also in breadth. Referring the example in 21, a third rule could overwrite the styling of the `last` name as follows:

Source (extract)

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

CSS Style Sheet (extract)

```
1 author  { font-style : italic; }
2 /* Additional CSS-Rules: */
3 last    { font-style : normal; }
4 last    { font-style : italic; }
```

Rendering

Stevens W.

Figure 21: Shading CSS rules

The rule in line four of the style sheet frame above, shades the previous rule. As a consequence, the `last` name of the `author` is rendered in italic font style.

In CSS, the most specific rule is used for the rendering. Obviously, the most specific rule in depth is the XML element itself or the nearest ancestor XML element. In breadth, the most specific rule is the last one matching the same XML element.

2.7 Styling at Unknown Depth

CSS allows styling at unknown depth. Simple selectors allow to select XML elements independent of depth (cf. Section 2.2). Furthermore, according to the cascading styling paradigm (cf. Section 2.6) each XML element is styled depending on the most specific rule. For instance, the `author` rule of the library example (cf. Section 2.1) defines a default styling of all `author` XML elements independent of depth. This styling is inherited to all descendant XML elements independent of depth such as `fist` and `last`.

3 Limitations of CSS 2.1 and CSS 3

CSS 2.1 and CSS 3 both offers many facilities for a sophisticated rendering of semi-structured data (cf. Section 2: ‘CSS Zen Garden’). This section shows limitations of CSS, the extensions proposed later in this thesis try to overcome.

3.1 Transformation vs. Rendering

A widespread argument against CSS and in favor of XSLT is that CSS, in its current versions including CSS 3, does not allow to significantly *transform* the structure of a document and instead only offers primitives for adding style to structured data while, besides limited changes, keeping the document’s original structure unchanged.

The author believes that this limitation of CSS is one of its appealing features. The reason is that *transforming* up till significantly restructuring a (structured) document is a task that

1. basically is not part of *rendering* and
2. is not only needed for *rendering*.

It is the author’s conviction that if a rendering requires a significant restructuring, then it is preferable to specify this restructuring independently from rendering. Obviously, this leads to clearer programs and therefore to programs that are easier to maintain. (This is often called *separation of concerns* and is a key objective of ‘good’ programming.)

If significant restructuring should, in the authors’ opinion, be kept separated from rendering, *slight transformations* of a document’s structure are surely desirable within rendering specifications, e.g.

1. **“deleting”**, or merely “hiding” an **XML element** (and its sub-elements)
2. **“deleting”**, or merely “hiding” parts of **XML text nodes**

3. **adding** XML **text nodes** without iteration, and
4. **adding** XML **elements** without iteration.

Adding without iteration means that, e.g., recursively specified addition of XML elements should be precluded. This restriction averts, e.g., the specification of infinite documents in a CSS style sheet. Refer to Section 3.1.4 for more details.

Note that deleting an XML element does not affect the *source document* but the *visualization* of the XML element only. However, the view of deleting is applicable to a (virtual) 'intermediary step' of the rendering process. This intermediary step is the result of slight transformations caused by adding and deleting of markup. For instance, an XML element that is added by a CSS style sheet would appear in the intermediary step while the source document does not change. Consequently, the XML document of the intermediary step is rendered (instead of rendering the source XML document directly). In the following we will use deleting in this context.

As demonstrated in the table below, the rendering specifications 'deleting XML elements' and 'adding XML text nodes' are already possible in current CSS. However, deleting XML text nodes and adding XML elements is not possible in current CSS, and there seems to be no reasons for this.

	Adding	Deleting
XML text nodes	yes	no
XML elements	no	yes

One could expect that adding and deleting are permitted or prohibited line by line. The following sections address each value of the table above in detail:

3.1.1 Deleting XML elements

Deleting XML elements is supported in CSS 2.1 and CSS 3: The CSS definition `display:none` prevents the rendering of all XML elements that are matched by the corresponding CSS selector of a CSS rule, as demonstrated in the following figure:

Source (extract)

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

CSS Style Sheet (extract)

```
1 first   { display : none; }
```

Rendering

```
Stevens
```

Figure 22: Deleting an XML element

The `first` name 'W.' of the `author` in the Figure above is not displayed because of the rule in line 1 of the style sheet above. (Since CSS inherits styling definitions to sub-elements of a XML element, cf. Fig. 19, the sub-elements inherit the definition `display:none`. Consequently, these sub-elements are not rendered except another CSS rule overwrites this definition, cf. Fig. 20.)

3.1.2 Deleting parts of XML text nodes

Deleting parts of XML text nodes is not possible in current CSS. Although text nodes can be selected using pseudo-elements (cf. Section 2.5.1) such as `::first-letter`, XML text nodes can not be deleted because the CSS `display` property cannot be applied to *'pseudo-selections'* [W3C98]. Hence the CSS rule `author::first-letter { display: none; }` is possible in current CSS. The authors are not aware of any reason for this restriction because this transformation is more limited than *deleting XML elements*, which is supported by current CSS as discussed above.

Deleting XML text nodes would be an appealing feature of CSS. For instance, all text portions surrounded by parentheses could be deleted to get a compact version of a text, e.g. for rendering on a portable device with small screen. Such a selection would be possible using extended pseudo-elements to select arbitrary XML text nodes (instead of only selecting the first letter or the first line like in current CSS). A new pseudo-selector based on *Regular Expressions* such as (e.g. expressed in the POSIX[6] syntax) can afford such selections. This extension goes beyond the scope of this thesis and is therefore not addressed here.

3.1.3 Adding XML text nodes

Adding XML text nodes is possible in current CSS. The pseudo-elements `::before` and `::after` allow to specify the insertion of context. XML text nodes can be inserted like the currency, e.g., `Euro` can be inserted before or after an XML element (cf. Section 2.5.1).

3.1.4 Adding XML elements

Adding XML elements is not possible in current CSS (cf. Section 3.2). However, adding XML elements would be a winning feature of CSS because styling would become more flexible. For instance, tabs could be added to data items (such as `person`, `name`, etc., cf. Fig. 23). If realized with XML elements (instead of XML text nodes), CSS rules could be used to render the tabs. In particular, dynamic rendering could enrich the facilities of such tabs: For instance, a mouse click on a tab could fold the data item (and its subtree) and a second mouse click could open the tab again.

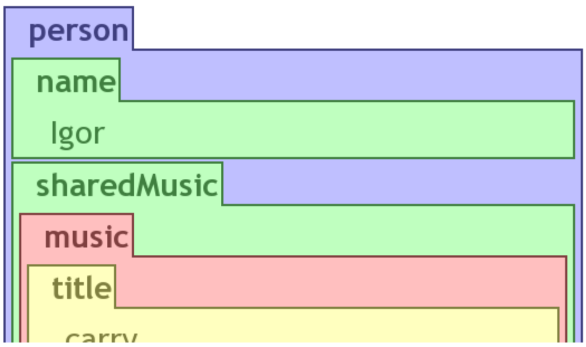


Figure 23: Adding XML Elements as Tabs

[6]`http://www.pasc.org/plato/`

Arguably, adding XML elements is not allowed in current CSS because of the following reasons:

As first reason, the *validity of the rendering* of an XML document would be dependant on the style sheet. (The rendering does not affect the source document, cf. Section 3.1.) For instance, an opening XML tag is inserted by a CSS rule but the corresponding closing XML tag is not inserted because there is no CSS rule inserting it:

Source (extract)

```
  </publisher>
           <!-- virtual XML text node -->
  <price>65.95</price>
           <!-- virtual XML text node -->
</book>
```

CSS Style Sheet (extract)

```
price::before {content:"<strong>";}

/* Not applied:
price::after {content:"</strong>";}
*/
```

Intermediary Step (extract)

```
  </publisher>
  <strong><price>65.95</price>
  <!-- missing closing tag -->
</book>
```

Rendering

not defined

Figure 24: Invalid insertion of XML elements

As a consequence, the source document could not be rendered like in the (hypothetic) example above. The intermediary step in the rendering process (including the added XML element) cannot be rendered because rendering of invalid XML documents is not defined in current CSS.

As second reason, an XML element could be *added iteratively* using recursive CSS rules. The following (hypothetic) example demonstrates the definition of such repetitively applicable CSS rules:

Source (extract)

```
  </publisher>
           <!-- virtual XML text node -->
  <price>65.95</price>
           <!-- virtual XML text node -->
</book>
```

CSS Style Sheet (extract)

```
1  price::before {content:"<price />";}
```

Intermediary Step (extract)

```
  </publisher>
  <price />...<price /><price>65.95</price>
</book>
```

Rendering

not defined

Figure 25: Recursive Insertion of XML elements

The XML tag `<price />` is added recursively because the CSS rule in line 1 of the CSS style sheet adds an XML tag, that is styled by the rule itself. Obviously, `<price />` is inserted infinitely. Therefore, the intermediary step would be an infinite XML document, and thus the rendering would be undefined. It is assumed that added XML elements are styled using the CSS rules of the same CSS style sheet[7].

As third reason, (if terminating) iterative rendering processes should not be allowed because primitives like *adding* and *deleting* of XML elements allow significant restructuring (as discussed above). For instance, XML elements can be grouped by criteria such as the content of an XML attribute or an XML element as demonstrated in Fig. 26. The XML document on the left side can be transformed to the XML document on the right side (and vice versa) using adding and deleting XML elements iteratively.

Trains Grouped by Time of Day

```
<trains timeOfDay="AM">
  <train id="ICE111">
    <departure>
      <station>Munich</station>
      <time timeOfDay="AM">11:36</time>
    </departure>
  </train>
</trains>

<trains timeOfDay="PM">
  <train id="ICE333">
    <departure>
      <station>Hamburg</station>
      <time timeOfDay="PM">19:47</time>
    </departure>
  </train>
  <train id="ICE888">
    <departure>
      <station>Munich</station>
      <time timeOfDay="PM">21:48</time>
    </departure>
  </train>
</trains>
```

Trains Grouped by City of Departure

```
<trains departure="Munich">
  <train id="ICE111">
    <departure>
      <station>Munich</station>
      <time timeOfDay="AM">11:36</time>
    </departure>
  </train>
  <train id="ICE888">
    <departure>
      <station>Munich</station>
      <time timeOfDay="PM">21:48</time>
    </departure>
  </train>
</trains>

<trains departure="Hamburg">
  <train id="ICE333">
    <departure>
      <station>Hamburg</station>
      <time timeOfDay="PM">19:47</time>
    </departure>
  </train>
</trains>
```

Figure 26: Grouping of XML Data (Intermediary Step)

In the left window the trains are grouped according to their time of departure. Hence, `ICE111` belongs to the first group (because it departs before noon) and the remaining trains `ICE333` and `ICE888` belong to the second group (because they depart after noon). Obviously, the XML document in the right window is significantly restructured by comparison to the left window. The restructuring is the result of a grouping by the city of departure instead of a grouping by the time of day. Therefore, `ICE111` and `ICE888` are grouped together on the right window instead of `ICE333` and `ICE888`.

If CSS would allow recursion such a significant restructuring could be implemented using adding and deleting XML elements as follows: For each group iterate over all XML elements of the source document and add each matching XML element (including its subelements) to a 'container XML element' of the group. As a consequence, if iterative rendering processes would admit such restructuring, CSS would be extended to the capabilities of Turing-complete transformation languages such as XSLT or XQuery [Kep04]. Hence, the posited *separation of*

[7]Obviously, adding XML elements that are not being styled would make no sense.

concerns (cf. Section 3.1) would be abandoned.

The restriction of forbidding iterative adding XML elements complies with the authors' conviction concerning *slight transformations* of semi-structured data instead of a significant restructuring (cf. Section 3.1). However, adding XML elements not iteratively should be allowed in CSS (as motivated by additive XML elements serving as tabs). Refer to Section 4.1 for concrete extensions.

3.2 Insertion of Markup

The insertion of XML elements using CSS pseudo-elements as demonstrated in Section 3.1.4 (Fig. 24 and 25) seems to be natural. Indeed, according to Section 2.5.1, CSS 2.1 allows to insert arbitrary characters (cf. encoding schemes UTF-8, ISO-8859-15, etc.) using pseudo-elements. Since markup is composed of XML 'control characters' such as $<$ or $>$, XML tags can be inserted, too, as demonstrated in the following example:

Source (extract)

```
  </publisher>
    <!-- virtual XML text node -->
  <price>65.95</price>
    <!-- virtual XML text node -->
</book>
```

CSS Style Sheet (extract)

```
price::before { content : "<new>&euro; "; }
price::after  { content : "</new>"; }
new           { font-weight : bold; }
```

Expected Rendering

€ 65.95

Actual Rendering

`<new>€ 65.95</new>`

Figure 27: Insertion of XML Control Characters

The rendering frame on the right side of the figure above illustrates the effect of the insertion of `<b>` and `</new>` on the rendering. Obviously, the characters $<$ and $>$ are not interpreted as expected (cf. rendering frame on the left side). Even the XML entity `€` is not rendered as expected. Hence, XML elements can not be added and rendered by CSS 2.1 and CSS 3 like in XML source documents because there is no other construct available in CSS to insert XML elements to an XML document.

3.3 Visualization of Markup

In current CSS, markup being inserted (as discussed in the latter section) as well as markup of a source XML document can be visualized. Pseudo-elements (cf. Section 2.5.1) and the CSS property `content` can define such a rendering using the selectors `::after` and `::before`. For instance, the markup of the train example (cf. Section 1) can be implemented in current CSS as follows:

In Fig. 28 the markup of the source document on the left side (e.g. `<trains>` or `<train id="ICE788">`) is rendered on the right side (e.g. trains or train (id ICE 788)). The definition of the rendering is given by the style sheet below (cf. Fig. 28):

Source	Rendering

<trains>
   <train id="ICE788">
      <departure>
        <station>Munich</station>
        <time>11:36</time>
      </departure>
      <arrival>
        <station>Hamburg</station>
        <time>17:45</time>
      </arrival>
...
	• trains
	   – train (id ICE788)
	      ∗ departure
	         · station Munich
	         · time 11:36
	      ∗ arrival
	         · station Hamburg
	         · time 17:54
	...

Figure 28: Rendering of markup

```
1 trains::before    { content : "trains" }
2 train::before     { content : "train" }
3 departure::before { content : "departure (id" attr(id) ")" }
4 station::before   { content : "station" }
5 time::before      { content : "time" }
```

Figure 29: CSS style sheet rendering the markup of the XML file of Fig. 28

Obviously in the style sheet (cf. Fig. 29), each XML element such as `train` as well as each XML attribute such as `id` in line 3 is addressed explicitly. Due to the lack of generic constructs to access the markup of the source document in the style sheet, the rendering of each markup construct (such as XML elements) needs to be defined separately. A generic construct like in XSLT (cf. Fig. 30) to insert the name of a currently selected XML element in a CSS rule is not available.

```
<xsl:value-of select="name()" />
```

Figure 30: Generic Access to the name of XML elements in XSLT

CSS functions such as `attr(X)` (cf. Section 2.5.1) offer a similar functionality by comparison to the XPath function `name()`. Contrary to expectations, the CSS function `attr(X)` does not offer generic rendering capabilities because the name of the parameter (selecting an attribute) must be known in advance while writing the style sheet. Note that the wildcard pattern `*` (cf. Section 2.1) is not applicable as parameter to the CSS function `attr(X)`.

As a consequence, CSS 2.1 and CSS 3 offer constructs to render XML elements and XML attributes. However, the rendering capabilities of CSS are restricted to XML elements and XML attributes that are known in advance. Styling of unknown XML elements and XML attributes such as covered by XSLT is not possible. An extension introduced in Section 4.2 overcomes this restriction.

3.4 Styling Depending on Depth

According to Section 1 an alternating styling of highly nested trees such as the 'Tree of Life' is highly welcome in practice. In current CSS, an XML element on a certain depth can be addressed using CSS combinators (cf. Section 2.3) as follows:

```
*             { /* DEFINITION A */ }                 /* level 1        */
* *           { /* DEFINITION B */ }                 /* level 2        */
* * *         { /* DEFINITION A */ }                 /* level 3        */
* * * *       { /* DEFINITION B */ }                 /* level 4        */
.
.
.
* * * ... *   { /* DEFINITION A */ }                 /* max(level) - 1 */
* * * * ... * { /* DEFINITION B */ }                 /* max(level)     */
```

Figure 31: Styling on a certain depth in CSS

Each nesting level in the XML tree needs its own CSS rule. Since CSS style sheets must be finite, the styling can only be written until a certain depth. Hence, the number of CSS rules defining an alternating styling in depth must be equal to the maximum depth of a source document. XML elements exceeding the maximum level of depth are styled depending on the most specific matching rule (cf. Section 2.7). Therefore, the styling stays the same until (theoretically) infinite depth. Hence, an alternating styling of XML elements on unknown depth is not possible using CSS. Note that this restriction is similar to the restriction of CSS concerning visualization of markup (cf. Section 3.3).

The parameterization of CSS selectors is available for direct child XML elements in CSS 3 (cf. Section 2.5.3). The authors are not aware of any reason, why a parameterized selection depending on depth is not offered by CSS 3. Although CSS 3 offers the parameterized selection for sibling XML elements, it does not offer the symmetric case of parameterized selection of nested child elements. For instance the `:nth-child(An+B)` structural pseudo-class allows an alternating styling in breadth. A corresponding construct for alternating styling in depth is not offered in current CSS. An extension introduced in section (cf. Section 3.4) will overcome this restriction.

3.5 Dynamic Rendering

Obviously, input devices are essential to dynamic rendering (cf. Section 1) because such devices serve as interface between the viewer of a document and the rendering engine (e.g., of a Web browser). In CSS, the mouse (cf. Section 2.5.4) is the only input device that can change the rendering of a source document directly. (Indirectly, the history of visited Web pages of a Web browser is an 'input device', too, cf. Section 2.5.4). Input provided by other devices such as the keyboard can not be selected.

A mouse can perform various actions like clicks or movements that can be received by the rendering engine of a Web browser as so called events. For instance, if the mouse is moved on the desk such that the mouse cursor enters the rendering of an XML element, the rendering engine receives a corresponding event message. In the case of the dynamic pseudo-class `:hover`, the rendering can be derived directly from the position of the mouse cursor because

:hover affects only the rendering of XML elements the mouse cursor is hovering above. For instance, tree structures can be rendered dynamically using the following style sheet:

```
1 *:root       { display:block; }
2 *:root *     { display:none;  }
3 *:hover > * { display:block; }
4 *:active     { display:block; }
```

Figure 32: Dynamic Rendering of Tree Structures

The CSS rule in line 1 of the CSS style sheet above renders the root XML element of a source document as block. Since the descendant XML elements of the root element should not be visible by default, the rule in line two defines that every child element of the root element is not displayed. Dynamic rendering in this example is defined in the rule of line 3: Whenever an XML element is hovered by the mouse cursor, its children become visible. Note that this rule works transitively because the rendering of each 'unfolded' XML element can be unfolded again, if the mouse cursor hovers the way down to it.

The CSS style sheet above can be used to define the 'navigation tree' of the Web-site on the right side in Fig. 32: If a hyperlink is active, it is displayed by default as defined by the rule in line 4. Otherwise an item, that was dynamically unfolded by the viewer, would be folded again as soon as the mouse cursor has left the rendering of an item.

However, a generic folding and unfolding like in Fig. 4 is not possible because the rendering state of each XML element (being folded or not) cannot be realized in CSS. Extensions of current CSS allowing extended dynamic rendering are introduced in Section 4.4.

3.6 Hyperlinks

The functionality of HTML hyperlinks is not subject of style sheet languages because navigating in the Web using hyperlinks changes the source document. Hence, such a navigation goes beyond pure rendering of a source document. (The whole Web is not interpreted as single document.)

Indeed, dynamic style sheets can simulate hyperlink-like behavior within Web pages. However, in those cases, the source document stays the same. Nevertheless the functionality of hyperlinks is not covered in this thesis, hyperlinks can, of course, be styled using style sheets.

4 Proposed Extensions

This section addresses the concrete extensions of this thesis that are motivated in Section 1. Each extension proposed in the following sections tries to overcome restrictions of current CSS (cf. 3). The extensions proposed in this section do not interfere with style sheets written in the latest CSS version CSS 2.1 because the semantics of existing constructs in current CSS was not changed. Hence, every style sheet of version 2.1 can be re-used.

4.1 Insertion of Markup

The insertion of markup (such as XML elements or XML attributes) for rendering issues permits more flexible rendering (cf. Section 3.2). For instance, a sophisticated rendering such as tabs for data items (cf. Fig. 23) can be defined.

The Insertion of XML text nodes is already possible. CSS 2.1 offers pseudo-elements (cf. Section 2.5.1) to insert XML text nodes, like in the following example:

```
price::before { content : "EUR "; }
```

Figure 33: Insertion of XML text nodes

Pseudo-elements are the only constructs offering insertion capabilities in current CSS. Hence, two alternatives are available to extent current CSS toward markup insertion capabilities. Either a completely new construct can be introduced or the existing pseudo-elements `::before` and `::after` can be extended.

The first alternative 'introducing a new construct' (instead of extending pseudo-elements) means extending CSS selectors as well as extending CSS definitions (cf. Fig. 9). (Extensions violating the rule based structure of CSS are not intended.) Consequently, CSS selectors would need a new construct allowing to select the area of insertion before or after an XML element to insert markup. Furthermore, a new CSS property allowing to insert XML elements would be needed, as demonstrated in the following example:

```
element::element-before { content : ... ; }
element::element-after  { content : ... ; }
```

Figure 34: Insertion of XML text nodes (first alternative)

Since CSS simple selectors are defined to realize basic selections, new pseudo-selectors `::element-before` and `::element-after` are used in Figure 34. The CSS property `content` is used analogously to the `content` property of the existing pseudo-elements `::before` and `::after`.

```
<p>This is<br />mixed<em>content.</em></p>
```

Figure 35: Mixed Content: XML Elements and XML Text Nodes.

If CSS would be extended by such new pseudo-elements that are able to insert XML text nodes only, consequently, the insertion of 'mixed content' (cf. Fig. 35), which is common e.g. in HTML, becomes impossible as follows:

Since inserting XML text nodes would need to be realized by another pseudo-element (like the existing CSS pseudo-elements), mixed content would be inserted using CSS rules inserting XML elements and CSS rules inserting XML text nodes alternately, like in the following example inserting mixed content before the XML element `element`:

```
1 element::element-before { content : "<p>" ; }
2 element::before         { content : "This is" ; }
3 element::element-before { content : "<br /> ; }
4 element::before         { content : "mixed" ; }
5 element::element-before { content : "</em>" ; }
6 element::element-before { content : "</p> ; }
```

Figure 36: Insertion of XML text nodes

Consequently, in the example above, the CSS rules in lines 1, 3, and 5 are shaded by the CSS rule in line 6. Analogously, the CSS rule in line 2 is shaded by the CSS rule in line 4 (cf. Section 2.6, Fig. 21). Therefore, `mixed</ p>` would be inserted instead of the mixed content of Fig. 24.

To avoid a change on the interpretation principles of CSS a construct allowing the insertion of XML elements and XML text node at the same time is needed. Thus, we decided in favor of extending the existing pseudo-elements and their properties toward insertion of markup (instead of introducing a new construct being able to insert mixed content).

The second alternative is already informally demonstrated by experiments in Section 3.1.4. The example demonstrates how `::before` and `::after` could be extended to insert markup by simply interpreting the `content` of the corresponding CSS definition as markup (such as in a source XML document). Consequently, invalid XML documents (cf. intermediary step, Fig. 24) could be specified, e.g., by inserting opening XML tags without closing tags. To avoid invalid XML documents that cannot be rendered we propose to introduce new CSS functions inserting valid markup only. Hence, the corresponding opening and the closing XML tag can be inserted by the rendering engine (e.g. of a Web browser) automatically such that no invalid insertions can be specified in the CSS style sheet.

Source (extract)

```
  </publisher>
            <!-- virtual XML text node -->
  <price>65.95</price>
            <!-- virtual XML text node -->
</book>
```

CSS Style Sheet (extract)

```
1 price::before { content: element(
2                 "strong",
3                 attribute("id","42"),
4                 "PRICE"); }
5 strong        { font-weight: bold; }
```

Intermediary Step (extract)

```
  </publisher>
  <strong id="42">PRICE</strong>
  <price>65.95</price>
</book>
```

Rendering

PRICE
65.95

Figure 37: Valid Insertion of XML elements and XML attributes

In the example above, a new XML element `strong` including its attribute `id` is inserted (cf. intermediary step). The opening and the closing tag for the XML element `strong` were inserted by the CSS function `element`. The rendering frame on the lower right side shows the rendering of the 'intermediary step'. While the attribute `id` does not influence the rendering according to the CSS style sheet above, the content of the XML element `strong` is rendered in bold letters because the CSS rule in line 5 is applied to the inserted elements.

Note that the CSS rule starting in line 1 of the CSS style sheet above is not written in one line to present the CSS rule clearly structured. The line break has no semantic in the rendering process.

Since CSS functions in current CSS yield XML text nodes, we propose to introduce a new class of CSS functions having markup (such as mixed content) as return type. For this new class of CSS functions we propose the following functions to allow the insertion of markup:

1. `element(NAME, ATTRIBUTES, VALUE)` inserting XML elements and
2. `attribute(NAME, VALUE)` inserting XML attributes.

Both CSS functions are discussed in the following sections in detail.

4.1.1 CSS Function `element`

The proposed CSS function `element` inserts XML elements to the 'intermediary step' as demonstrated in Fig. 38. The parameter `NAME` specifies the name of the XML element (such as `strong`). The type of the parameter `NAME` is String.

The parameter `ATTRIBUTES` denotes the possibility to add XML attributes that belong to the current XML element. Since an XML element can have zero or more XML attributes [W3C00b], we propose to substitute `ATTRIBUTES` by *variable arguments (varargs)* making `element` a variadic function[8]. The only allowed type of input parameters is an XML attribute. XML attributes can be constructed by the proposed CSS function `attribute` that is addressed in Section 4.1.2.

The last parameter `VALUE` of the proposed CSS function `element` can consist of one of the following alternatives:

value	**description**	`content` **property**	**intermediary step**
'no value'	empty element	element("em")	
String	element with content	element("em","PRICE")	PRICE
element(...)	nested element	element("em", element("br"))	
element(..., String)	mixed content	element("em", "Hello"element("br"))	Hello

Figure 38: Alternatives of the parameter `VALUE` of the CSS function `element`

4.1.2 CSS Function `attribute`

The proposed CSS function `attribute` offers to construct XML attributes that can be used in the proposed CSS function `element` (cf. Section 4.1.1). The first parameter `NAME`

[8]A variadic function is a function of variable arity, cf. ISO/IEC 9899:1999

specifies the name of an XML attribute such as `id`. The second parameter `VALUE` specifies the value of the XML attribute such as `42` (cf. Fig. 37).

Note that in both proposed CSS function, `element` and `attribute`, double quotes within String parameters must be paraphrased by XML entities such as `"` or `"` to avoid ambiguous expressions such as:

- `element("em", "The function "element("br")" inserts the element "br"".)`

This expressions must be paraphrased by

- `element("em", "The function "element("br")" inserts the element "br"".)`

4.1.3 Prevention of Iterative Insertion of Markup

As discussed in Section 3.1.4 iterative adding must be avoided. Since inserted markup and markup from the source document are rendered by the same style sheet, recursive non-terminating CSS rules can be defined (cf. Fig. 25).

To avoid iterative insertion of XML elements we propose that the pseudo-elements `::before` and `::after` can be applied only to XML elements of the source document and not to XML elements of the 'intermediary step'. As a consequence, inserted XML elements are not styled by CSS rule that have one of the pseudo-elements `::before` or `::after` as selector, as demonstrated in the following example:

Source (extract)

```
    </publisher>
                <!-- virtual XML text node -->
    <price>65.95</price>
                <!-- virtual XML text node -->
</book>
```

CSS Style Sheet (extract)

```
price::before {content:
                element("price",
                &euro;);}
```

Intermediary Step (extract)

```
    </publisher>
    <price>&euro;</price><price>65.95</price>
</book>
```

Rendering

€ 65.95

Figure 39: Restricted Insertion of XML elements

Obviously, this restriction prevents a recursive interpretation of CSS rules. Hence, iterative insertion of XML elements is not permitted introducing the extension of inserting markup.

4.1.4 Validity

Since XML elements can be inserted before or after each XML element in an XML document according to Section 4.1, XML elements can be inserted before or after the document root, too. Applied to the source document such a transformation would not be valid, because the source document would be disrooted, which conflicts with the specification of XML [W3C00b].

In current CSS, the insertion of XML text nodes can be applied to root elements (and in HTML to the `body` element). Such an insertion does not affect the validity of the source document because the markup is only inserted to the 'intermediary step' of the rendering process for rendering issues, while the source document is not changed.

To handle this phenomenon we propose to introduce a virtual root element that encapsulates the intermediary step. This virtual root can not be selected by CSS selectors. The virtual root serves for conceptual validity of the XML document in the intermediary step only. Consequently, the `:root` pseudo-class of CSS 3 does not match the virtual root but the root XML element of the source document. Due to the definition of the 'intermediary step' (cf. Section 3.1) all other CSS selectors such as `nth-child()` are applied on the intermediary step after adding and/or deleting XML elements.

4.2 Visualization of Markup

Current CSS allows to visualize known markup structures (such as XML elements and XML attributes) but a generic visualization of markup is not offered (cf. Section 3.3). The following extensions overcome this restriction:

Source	**Rendering**
<trains> <train id="ICE788"> <departure> <station>Munich</station> <time>11:36</time> </departure> <arrival> <station>Hamburg</station> <time>17:45</time> </arrival> ...	• trains – train (id ICE788) ∗ departure · station Munich · time 11:36 ∗ arrival · station Hamburg · time 17:54 ...

Figure 40: Rendering of markup

```
1 *::before    { content : element-name()
2                          id { content: "(" attribute-name() " " attribute-value() ")" }
3              }
```

Figure 41: CSS style sheet rendering the markup of the XML file of Fig. 40

The rule in Fig. 41 defines the rendering of the XML document on the left side of Fig. 40. The result of the rendering process is shown on the right side of Fig. 40. Each XML element is tagged by its name caused by the proposed CSS function `element-name` (cf. Section 4.2.1). Additionally, the XML attribute `id` is visualized by the *attribute rule* in line 2. The attribute rule consists of an *attribute selector* to select the attribute `id` and a rule definition to define the rendering of selected attributes (cf. Section 4.2.2).

The style sheet is a re-implementation of the style sheet of Fig. 29 (cf. Section 3.3). It reduces the programming effort from one rule per XML element name to one single rule for an arbitrary XML document in CSS. In XSLT the same rendering can be implemented like in by the XSL style sheet in Fig. 42. Obviously, the style sheet in Fig. 41 provides with a more declarative and easier definition of rendering by comparison to the XSL style sheet.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- generate HTML document -->
  <xsl:template match="/">
    <html>
      <body>
        <!-- render all XML elements -->
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="*">
    <ul>
      <!-- render XML elements in breadth -->
      <xsl:for-each select=".">
        <li>
          <xsl:value-of select="name()" />
          <xsl:text> </xsl:text>

          <!-- render XML attributes of current XML element -->
          <xsl:for-each select="@*">
            <xsl:text> (</xsl:text>
            <xsl:value-of select="name()" />
            <xsl:text> </xsl:text>
            <xsl:value-of select="." />
            <xsl:text>) </xsl:text>
          </xsl:for-each>

          <!-- render XML elements in depth -->
          <xsl:apply-templates />
        </li>
      </xsl:for-each>
    </ul>
  </xsl:template>

</xsl:stylesheet>
```

Figure 42: XSLT program providing the same rendering like the style sheet in Fig. 41

4.2.1 Visualization of XML elements

XML elements can be visualized by showing their names (cf. Fig. 40). However, current CSS does not allow to access the name of any XML tag for rendering issues directly. The only solution to visualize XML elements by their names is writing rules for every XML element with different name (cf. Fig. 28) indirectly. To allow access to the names of XML elements, we propose to extend CSS functions. Since the CSS function `attr` (cf. 2.5.1) already allows to access the values of XML attributes, extending CSS functions toward visualizing XML elements seems to be natural.

We propose to extend CSS functions by a new function `element-name()` for getting the name of the currently selected XML element in a CSS rule as a String. Like all other CSS functions [W3C98] `element-name()` can only appear in the context of a CSS `content` property (cf. Fig. 2.5.1). The name of the current XML element can be derived from the selection of the corresponding CSS selector because every CSS rule is applied individually to one XML element (cf. Section 2.2).

4.2.2 Visualization of XML attributes

In contrast to XML elements, XML attributes cannot be selected by CSS selectors. Selections can only be refined to XML elements having a special XML attribute configuration using square brackets (cf. Section 2.2). Consequently, extending CSS functions to a function `attribute-name()` (cf. `element-name()`, Section 4.2.1) is not sufficient because an XML element can have more than one XML attribute, and, therefore, no 'current XML attribute' can selected by default like a 'current XML element'. Therefore, we propose to extend CSS toward an extended selector concept capable of selecting XML attributes.

Attribute Selectors. Since each XML attribute belongs to exactly one XML element, the 'XML element context' of an attribute can be selected by CSS simple selectors selecting XML attribute names instead of XML element names. Of course, attribute constraint defined with square brackets of CSS simple selectors are not applicable in this context. XML attributes are flatly structured and can be seen as a set of keys and values pairs [W3C00b].

The position of the proposed attribute selector is located in the `definition` part of a CSS rule. Obviously, the only reasonable place in the `definition` part is as value of a CSS `content` property because the extended selector concept is meant to realize insertions to the rendering of a source document (cf. Fig.41) using the pseudo-elements `::before` and `::after`.

Since there is no order defined on XML attributes, visualizing XML attributes can not be deterministic. Therefore, if using the wildcard pattern `*` for selecting all XML attributes (cf. Fig. 41), the XML attributes are rendered as given by the serialization of the source document.

Adopting CSS Grouping. Other selector concepts of CSS (cf. Section 2) can not be adopted reasonably except for grouping (cf. Section 2.4) and except for parts of the CSS structural Pseudo-classes (cf. Section 2.5.3).

In current CSS, grouping is syntactic sugar to integrate CSS rules having the same CSS `definition`. This concept can be adopted to attribute selectors. In this case the order of the rendering of XML attributes can be interpreted, like in the following example:

```
a::before { content : element-name()
              href, title { content: "(" attribute-name() " " attribute-value() ")" }
          }
```

Figure 43: Attribute Selector using Grouping

```
a::before { content : element-name()
              href  { content: "(" attribute-name() " " attribute-value() ")" }
              title { content: "(" attribute-name() " " attribute-value() ")" }
          }
```

Figure 44: Rule of Fig. 43 without Grouping

The attribute selectors in the example above use grouping indicated by the "`,`". Since `href` appears before `title` in the attribute selector, `href` is rendered before `title`.

Adopting CSS Structural Pseudo-classes. XML attributes have no order but adopting CSS structural pseudo-classes is reasonable for rendering issues. Structural pseudo-classes taking into account selection in breadth can be mapped to the serialized structure of XML attributes. The following example illustrates how to avoid the insertion of the last separator sign such as a comma, if listing XML attributes:

```
1 *::before { content : element-name()
2               *:not(:last) { content: "(" attribute-name() " " attribute-value() "), " }
3               *:last       { content: "(" attribute-name() " " attribute-value() ")" }
4           }
```

Figure 45: Attribute Selectors with Structural Pseudo-classes

In Fig. 45 a comma is inserted after each XML attribute according to line 2 of the style sheet. The rendering of the last XML attribute, however, is shaded by the rule in line 3, where no comma is inserted after the rendering of the XML attribute.

The following list itemizes all structural pseudo-classes of CSS 2.1 and CSS 3.0 that can be mapped to the structure of XML attributes:

CSS (2.1 and) 3.0	Mapping to Attributes	New Semantics
`:nth-child()`	`:nth-attribute()`	the n-th attribute
`:nth-last-child()`	`:nth-last-attribute()`	the last attribute
`:first-child`	`:first-attribute`	the first attribute
`:last-child`	`:last-attribute`	the last attribute
`:empty`	`:empty`	matches, if no attribute is defined
`:not()`	`:not()`	selects every attribute of the current element except of the attributes specified as parameter (cf. Fig. 45)

Figure 46: Mapping of Existing Pseudo-classes to Attribute Selectors

Attribute Definition. Additionally to the constructs of the constructs that are allowed in the value of a `content` property, new CSS functions for accessing the name and the value of XML attributes are needed. In analogy to the proposed function `element-name()` (cf. Section 4.2.1) we propose two new functions, as follows:

Proposed Function	Semantics
`attribute-name()`	Yields the name of the currently selected XML attribute.
`attribute-value()`	Yields the value of the currently selected XML attribute.

Figure 47: Mapping of Existing Pseudo-classes to Attribute Selectors

Note that CSS 2.1 already offers the CSS function `attr(X)` to access the value of a certain XML attribute (cf. Section 2.5.1). Nevertheless, we decided in favor of proposing a new function instead of extending the existing one because of symmetry reasons to the function names `element-name()` and `element-value`.

4.2.3 Validity of Insertions using Attribute Selectors

Introducing attribute rules reminds to extending CSS to the insertion of markup (cf. Section 3.2): Extending CSS to insertion of markup was not possible without restrictions to avoid, e.g., infinite and, hence, invalid insertion. Extending CSS toward XML attribute selectors, however, does not cause recursive insertion as follows:

Recursive insertion (like in Fig. 25) is not possible using attribute selectors. Indeed XML attributes and XML elements can be inserted using attribute rules. The inserted attributes, however, cannot be selected using the same attribute selector because the space for insertion is either before or after the corresponding XML element, and is, hence, not in the scope of the same attribute selector. The following example (cf. Fig. 48) illustrates why recursive insertion is not possible using attribute selectors.

The style sheet in Fig. 48 applied to the source document on the left side causes the intermediary step on the lower left side of the Figure as follows: The XML element `price` is matched by the CSS simple selector `price`. Afterwards, the attribute selector star (`*`) matches the only attribute of the XML element `newElement`. According to the body of the attribute rule a new element is inserted. The name of the new element is specified by the value of the

Source (extract)

```
  </publisher>
           <!-- virtual XML text node -->
  <price newElement="price">65.95</price>
           <!-- virtual XML text node -->
</book>
```

CSS Style Sheet (extract)

```
price::before { content:
 *{content:element(attribute-value(),
          attribute("price","42"));
  }
}
```

Intermediary Step (extract)

```
  </publisher>
  <price price="42" />
  <price newElement="price">65.95</price>
</book>
```

Rendering

€ 65.95

Figure 48: Insertion of XML Elements via Attribute Rule

currently selected XML attribute `newElement`. Hence, the XML element `price` (with the attribute `price`) is inserted as can be seen in the intermediary step. Since no new attribute can inserted to the currently selected XML element `price`, no recursive attribute can be specified.

4.2.4 Open Issues

The specification of XML [W3C00b] offers more constructs than XML attributes and XML elements. Comments, Processing Instructions, CDATA Sections, the prolog and the document type declaration are not covered in this thesis. Extensions towards visualizing the remaining constructs that are possible in XML documents can be done analogously introducing new functions to CSS.

4.3 Styling Depending on Depth

Styling depending on depth is currently not generically possible in CSS (cf. Section 3.4). However, documents that are highly nested like the tree of life (cf. Fig. 1) can profit from an extension making Styling depending on depth possible. Currently such structures can be styled only until a certain depth (cf. Fig. 31).

Since CSS 3 offers generic rendering in breadth by the structural CSS pseudo-class `:nth-child()`, we propose an analogous function for styling in depth. According to the terminology of XPath [W3C99b] styling depending on depth fits with the XPath axis descendant. Hence, combining the terminology of CSS and XPath the name of the proposed structural pseudo-class is `:nth-descendant()`.

Like in the structural pseudo-class `nth-element`, generic rendering can be defined using parameters. Therefore, also `:nth-descendant()` permits the parameters `(An+B)`. Using these parameters, XML elements having "$A \cdot n + B - 1$ ancestors before it" in the XML document tree can be selected. In other words, the selector matches every `a`-th XML element beginning with the `b`-th descendant. Where `A` and `B` can be substituted by integers, `n+` is fixed.

Using the proposed extensions as introduced above, a re-implementation of the CSS style sheet of Fig. 2 for the tree of life (cf. Fig. 1) can be given as follows:

```
1 *:nth-descendant(2n+1)       { font-weight : bold;   }
2 *:nth-descendant(2n+2)       { font-style  : italic; }
```

Figure 49: Generic Rendering Depending on Depth

The rule in line 1 of the Figure above renders the odd descendant XML elements (starting with the first descendant). The rule in line 2 renders the even descendant XML elements.

4.4 Extended Events

Dynamic pseudo-classes like `:hover` allow dynamic rendering in current CSS (cf. Section 2.5.4). For offering a more flexible rendering, we propose to extend CSS by the ability to handle additional input devices and not only the mouse, and we propose to extend CSS by supporting the selection of more actions of the input devices and not only hovering of the mouse cursor over an XML element.

4.4.1 Extended Input Devices

Mouse. Beside the pseudo-class `:hover` we propose a new dynamic pseudo-classes taking into account when a viewer clicks on the rendering. For clicking with a mouse button on the rendering of an XML element, we propose to introduce the dynamic pseudo-class `:onMouseClick(ITERATION)`. The parameter `ITERATION` is introduced in Section 4.4.2 but can be omitted.

The proposed pseudo-class admits `:onMouseClick(ITERATION)` new rendering capabilities. For instance, the menu items in a navigation tree (such as in Fig. 32) can be opened by a mouse click instead of being opened, if the mouse cursor hovers over it.

Obviously, the pseudo-class `:onMouseClick` is overloaded, if applied to a hyperlink because in Web browsers the navigation to other URLs via hyperlinks is performed clicking on a hyperlink with the mouse. Since the rendering of a Web page (e.g. in a Web browser) is the human interface to the Web page, and since a hyperlink can be rendered by a style sheet (and not the other way round), we decide in favor of the style sheet being able to shade the functionality of a hyperlink. Note that neither in XLink [W3C01b] nor in HLink [W3C02] the human interface for performing navigations via hyperlinks can be specified.

Analogously, to the pseudo-class `:hover`, we propose the dynamic pseudo-class `:onMouseOver (ITERATION)`. In contrast, to `:hover` this pseudo-class has the parameter `:onMouseClick (ITERATION)` (cf. Section `:onMouseOver(ITERATION)`). We decided in favor of introducing a new pseudo-class instead of extending the pseudo-class `:hover` for having similar names for all dynamic pseudo-classes.

Keyboard. Since in Web browsers such as Mozilla Firefox[9] or Microsoft Internet Explorer[10] browsing can be controlled by keys of the keyboard, too, we propose to admit changes of

[9] http://www.mozilla.org/products/firefox/
[10] http://www.microsoft.com/windows/ie/

the rendering caused by keys. The new dynamic pseudo-class `:onKeyDown(KEY,ITERATION)` detects if a key is pressed, and the new dynamic pseudo-class `:onKeyUp(KEY,ITERATION)` detects if a key is released.

The parameter `KEY` defines a single key that activates the selection of the corresponding dynamic pseudo-class (such as `:onKeyDown(KEY,ITERATION)`. The type of the parameter `KEY` is character. Hence a single letter such as 'a' can be the argument. Combinations of keys could be proposed, too, but are not addressed here. The parameter `ITERATION` is introduced in Section 4.4.2. It can be omitted.

Unlike with the mouse device, the keyboard can cause conflicts because two buttons can be pressed at the same time. As discussed above combinations of keys are not possible for selection in one rule but two different rules can select the state of two different keys (being pressed or not). Depending on the interpretation of CSS stylesheets (cf. Section 2.6) conflict resolution depends on the order of rule activation. In this case, the rendering of the rule of the second key shades the rendering of the rule of the first key.

These two pseudo-classes allow useful definitions of rendering. Since a key action such as hitting a key can show all renderings of XML elements on one level of a XML document.

```
1 *:root                                { display : block; }
2 *:root *                              { display : none;  }
3 *:root:nth-descendant(1):onKeyDown('1') { display : block; }
4 *:root:nth-descendant(2):onKeyDown('2') { display : block; }
5 *:root:nth-descendant(3):onKeyDown('3') { display : block; }
6 /* ... */
```

Figure 50: Dynamic Rendering using the Keyboard as Input Device

The rules in line 1 and 2 of the example above define the static rendering (without influence of the keyboard) as follows: The root XML element is rendered but all descendant elements are deleted. According to the rules in lines 3 to 5, only the XML elements on the level one, two or three are rendered depending on the currently pressed key '1', '2', or '3'.

The three dots in line 6 of the style sheet remind to 'Styling depending on depth' (cf. Fig. 31) and 'Visualization of Markup' (cf. Fig. 29) because the depth of the source document can be higher by comparison to the rules defined in the style sheet. However, the number of keys on the keyboard is limited, too. Therfore we decide in favor of not proposing a generic function to bind a key to a corresponding level in the document tree.

Note that the scope of keyboard events is the whole rendering of the source document. Hence, if a key event happens, the corresponding rules apply to all matching XML elements of the source document. In contrast, the mouse events introduced in the latter section rely on the XML elements in the scope of the mouse cursor.

The proposed pseudo-class `:onKeyUp(KEY,ITERATION)` allows to define a rendering similar to the CSS pseudo-element `::visited` (cf. Section 2.5.1). Data items (cf. Fig. 6) that have been folded and unfolded can be marked as visited, e.g., using another color.

4.4.2 Iteration of Events

The parameter `ITERATION` specifies, how often an event has happened before. In analogy to the structural the pseudo-class `:nth-child()` of CSS 3 (cf. Section 2.5.3), the parameter `ITERATION` has the same type of arguments of the form `(An+B)`. The semantics of that argument is as follows: The event selector matches, if the event has occurred "$A \cdot n + B - 1$ times in the past".

Permanent Changes on Rendering. In contrast to the dynamic pseudo-class `:hover`, the current state of the rendering can not be derived from the state of the input device. For instance, if a mouse cursor was hovering over the rendering of an XML element, it can be derived that the mouse cursor has hovered at least once over the rendering of the current XML element. However, it cannot be derived from the state of the mouse, how often this event occurred exactly. Therefore an accumulator for the rendering of each XML element is needed in the 'intermediary state' of the rendering process (cf. Section 3.1). The accumulator counts how often an rendering event already occurred. Since various events can be specified. An accumulator for each event such as `:onKeyDown(KEY,ITERATION)` or `:onMouseOver(ITERATION)` is needed, as demonstrated exemplarily in the following figure:

```
1  <trains event:onKeyUp="2" event:onMouseClick="5">        <!-- Events -->
2     <train id="ICE788" event:onMouseClick="4">            <!-- Events -->
3        <departure>
4          <station>Munich</station>
5          <time>11:36</time>
6        </departure>
7        <arrival event:onMouseClick="3">                   <!-- Events -->
8          <station>Hamburg</station>
9          <time>17:45</time>
10       </arrival>
11 ...
```

Figure 51: Event Accumulators realized using XML attributes

The XML elements in the lines 1, 2, and 7 in the example above contain XML attributes saving, how often events occurred during a rendering session. The XML attributes consist of a namespace `event` to avoid conflicts with other XML attributes. The name of each XML attribute of the namespace event corresponds to the exact name of the event such as `event:onMouseClick` for the event `onMouseClick`. Obviously, the attributes stating the number of occurred events are positioned in the 'intermediary step' of the XML document wherever an event was performed on the rendering of the corresponding XML element. Consequently, the default value of each XML attribute (if not stated) is `0`. Note that the attribute `event:onKeyUp` is located in the root XML element because keyboard events can not be assigned to a specific XML element by comparison to a mouse click being assigned to the XML element where the mouse cursor is hovering on.

Acyclic Events Avoiding the first part of the argument (`An+`) allows to specify acyclic events. For instance, an adaptive link changing its color depending on its usage can be rendered as follows:

```
1 hyperlink:onMouseClick(0) { color : black; }
2 hyperlink:onMouseClick(1) { color : green; }
3 hyperlink:onMouseClick(2) { color : red;   }
```

Figure 52: Acyclic Rendering: Adaptive Rendering of a Hyperlink

The rules in the style sheet above define the rendering of the XML element `hyperlink` depending on the number of mouse clicks on the XML element. According to line 1, the standard rendering displays the hyperlink in black letters. After one click, the font color changes to green. Finally, after the second click the color changes to red indicating a frequented hyperlink. More than two clicks on the rendering of an XML elements do not change the appearance of its rendering according to the rules above. (This behavior of CSS is discussed in Section 4.4.2.)

Cyclic Events Applications like folding and unfolding of XML elements quest for a cyclic evaluation of events. For instance, it should be possible to fold and unfold a data item several times (cf. 1). A cyclic matching of events can be defined as follows:

```
1 *:onMouseClick(2n+1) > * { display : block; }
2 *:onMouseClick(2n+2) > * { display : none;  }
```

Figure 53: Cyclic Rendering: Folding and Unfolding of XML elements

According to the rule in line 1 of the style sheet above, an XML element is unfolded (displayed) if an odd number of clicks were performed on the rendering of the parent XML element. Analogously, the rule in line 2 folds (displays not) an XML element, if an even number of mouse clicks were performed on the parent XML element.

Interpretation The proposed extensions on the dynamic pseudo-classes of CSS require an extension of the evaluation of CSS for events because in current CSS events are not permanent (cf. Section 2.5.4). In current CSS, XML elements are rendered taking account of the last and, hence, most specific CSS rule (cf. Section 2.6).

The same principle can be applied to the interpretation of extended dynamic pseudo-classes as follows: The most specific rule is applied to render an XML element. For instance, in Fig. 52, if more than two clicks are performed on the rendering of the `hyperlink` XML element, the rule in line 3 of the style sheet applies.

4.5 Extended Dynamic Selection

The selection of XML elements in Static Styling differs fundamentally from the selection of XML elements in Dynamic Styling (cf. Section 1) in current CSS as follows: The Static

Selection allows to select (and render) *all* XML elements matching the CSS selector. In contrast, one single XML elements can only be selected, if it can be identified unambiguously (for instance by an `id` XML attribute). The Dynamic Selection (`:hover`) in current CSS allows to select XML elements being *in the scope* of the mouse cursor only. For instance, dynamic highlighting of side-notes (cf. Section 1) is not possible because a side-note is usually not in the scope of an XML element but located somewhere else in the XML document (cf. Fig. 3).

	Certain XML Elements	Certain XML Elements
Static Selection	possible	yes
Dynamic Selection	yes	no

Figure 54: Selection of XML elements in CSS 2.1 and CSS 3

To enable applications like side-notes we propose to extend CSS by a new selector allowing to select XML elements beyond the convex scope of a CSS selector. Obviously, this extension does only apply to dynamic selections. Therefore, we propose to extend the dynamic pseudo-classes of CSS toward non-convex selections.

As keyword for the extension, we propose the word `panorama`, which means 'to see everything at the same time' (Greek). As standard value for selecting XML elements being in the scope of an event only, we propose as keyword `monorama`, which is a neologism meaning 'to see one thing at the same time' (derived from Greek). `monorama` is necessary to disable `panorama` in XML subtrees.

The following Figure demonstrates, how these keywords could be integrated into the CSS syntax:

```
1 *:onMouseClick(1):panorama { ... }   *:monorama:onMouseClick(1) { ... }
2 *:onMouseClick-panorama(1) { ... }   *:onMouseClick-monorama(1) { ... }
3 *:onMouseClick(1,panorama) { ... }   *:onMouseClick(1,monorama) { ... }
```

Figure 55: Proposals of the Integration of `panorama` and `monorama`

The rules in line 1 of Fig. 55 use the characteristic of CSS to handle more than one pseudo-class in one CSS selector, where the order of their appearance is arbitrary [W3C98]. Since `panorama` and `monorama` are only reasonable in combination with dynamic pseudo-classes, the proposal in line 2 of the Figure above assures a proper usage of both keywords and is, hence, more user friendly. The number of dynamic pseudo-class, however, increases by factor 2. Therefore, we propose to extend current CSS by the alternative in line 3. Here, both keywords are introduced as last parameters to all dynamic pseudo-classes.

To grant downward compatibility, the last parameter as introduced above is optional. If no value is given, the `monorama` applies as a standard value. Consequently, every dynamic pseudo-selector acts like in current CSS by default.

Source (extract)

```
<text>
  ... the <sidenote no="2">"learning CSS"
  page</sidenote>. For background
  information on style sheets, see the
  <sidenote no="2"> Web style sheets page
  </sidenote>. Discussions about CSS are
  carried out on the (archived) ...
</text>
<sidenotes>
  <sidenote id="1">Styling Aspects
  </sidenote>
  <sidenote id="2">Manuals</sidenote>
</sidenotes>
```

CSS Style Sheet 1 (extract)

```
sidenote[no="1"]:onMouseOver(panorama),
sidenote[no="2"]:onMouseOver(panorama)
   { font-weight : bold }
```

CSS Style Sheet 2 (extract)

```
sidenote[no!]:onMouseOver(panorama)
   { font-weight : bold }
```

Figure 56: Side-notes of Fig. 1 implemented using `panorama`

The style sheet 1 above implements the dynamic styling of Fig. 1. Using `panorama`, all occurrences of XML elements matching the selector (such as all `sidenote` elements having an `id` element of value 2) are rendered in bold font.

Since in style sheet 1 of the Figure above all side-notes need their own rule, we propose to extend the attribute constraint by a contruct selecting XML elements having attributes with the same name and the same value. We choose `!` as sign for this constraint. Using this extension, the rendering of side-notes, e.g., can be generically defined as demonstrated in style sheet 2 of the Figure above.

5 Test Case: Re-implementation of a visual query language as CSS-rendering

As a test case, the visualization of the visual query and transformation language *visXcerpt* [Ber03, BBSW03] is re-implemented using the extensions proposed in this thesis. visXcerpt can be seen as visual interface to the query and transformation language Xcerpt [SB04].

The re-implementation is realized as rendering instead of using scripting languages (such as ECMA script [ECM99]) or transformation languages (such as XSLT [W3C01a]) for rendering visXcerpt. Since the former implementation of visXcerpt [Ber03] partly already uses CSS, only those parts formerly implemented with other technologies, are given here.

As required in Section 3.1, the extensions allow a separation of concerns because the visualization of visXcerpt is realized by a style sheet language while editing capabilities are still implemented in ECMA script.

5.1 Data Terms

5.1.1 Elements including Attributes

Data terms are used to represent XML documents and the data items of a semistructured database. In visXcerpt each XML element is represented by a box which is labeled by a tab containing the name of the XML element. The parent child relation ship of XML elements is visualized by nesting child boxes into the parent box (cf. Fig. 23). XML attributes are visualized as first boxes inside the corresponding box of an element without a tab.

The following style sheet defines a generic rendering of Xcerpt data terms in visXcerpt:

```
1 *::before { content: element(tab,element(menu) element-name())
2                      * { content: element(attribute,
3                          attribute-name() attribute-value() );
4                        }
5           }
6 tab       { display:inline }
```

Figure 57: Rendering of Data Terms

The rule in line 1 of the Figure above inserts a tab on top of each XML element. According to the definition part of the rule, each tab contains the name of the corresponding XML element which is inserted by the proposed CSS function `element-name()`. (The meaning of `element(menu)` is discussed in Section 5.1.3). To render all attributes of an XML element, the attribute selector star (`*`) matches all possible names of XML attributes. The following body of the attribute rule defines the insertion of the name and the value of each selected XML attribute. The rule in line 6 specifies that tabs are rendered as `inline` elements for displaying the tabs of folded boxes in a row (cf. `title` and `author` in Fig. 58).

The rule in line 3 defines the rendering of XML attributes. The XML attributes are inserted before the first child of each XML element. The following figure demonstrates the rendering of the library example (cf. Fig. 10) as data terms in visXcerpt:

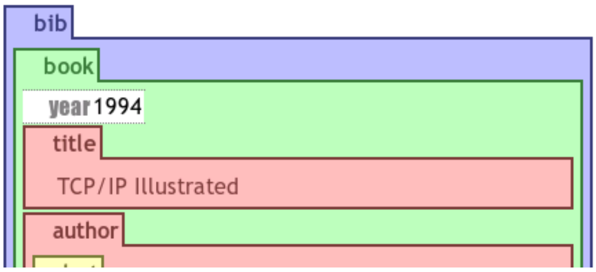


Figure 58: Library Example (cf. Fig. 10) rendered with the style sheet in Fig. 57

5.1.2 Folding and Unfolding

Data terms in visXcerpt can be folded and unfolded (cf. Section 1) by clicking on its tab. The example in Fig. 60 shows the `title` XML element of Fig. 58 after being folded by a mouse click on their tabs. Such dynamic rendering can be defined by the following style sheet:

Tabs stay visible independently from the state of its XML element according to the rule in line 1 of the style sheet above. The XML element beneath, however, can be folded by clicking on its tab as defined by the rule in line 2. A second mouse click unfolds the XML element again

```
1  tab                         { display:inline; }
2  tab:onMouseClick(2n+1) > * { display:none;   }
3  tab:onMouseClick(2n+2) > * { display:block;  }
```

Figure 59: Folding and Unfolding of Data Terms

(cf. line 3). By default, all XML elements are unfolded because the rule in line 3 matches for the initial case because no mouse event happened yet.

The following Figure demonstrates how a mouse click on the tab of the XML element `title` changed the appearance of the rendering in Figure 59:

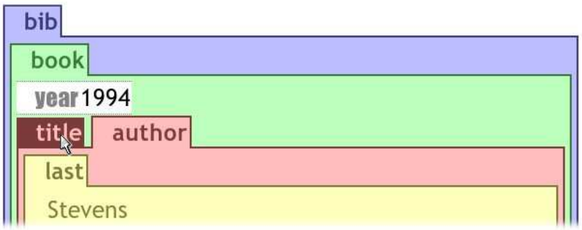


Figure 60: Folding of Data Terms (cf. Fig. 58)

5.1.3 Context Menus

Context menus are used in visXcerpt to manipulate the structure of data terms. As shown in Fig. 62 various operations such as *copy*, *paste*, or *paste before* are available. The rendering of such context menus can be defined in a style sheet using the proposed extensions as follows:

```
1  *                               { position:relative; }
2  .menu                           { display :none;     }
3
4  // alternative 1
5  menu:onMouseClick(panorama)     { display :block;    }
6  .menu:onMouseClick(panorama)    { display :none;     }
7  .menu                           { position:fixed;    }
8
9  // alternative 2
10 menu:onMouseClick(2n+1) > *     { display :block;    }
11 menu *:onMouseClick(2n+2)       { display :none;     }
```

Figure 61: Embedding Context Menus

According to line 2 of the style sheet above, context menus are invisible by default. Two alternatives are given, to visualize such context menus. In both alternatives the context menus can either be defined in the source document or in the style sheet using insertion of markup (cf. Section 4.1).

Alternative 1 uses `panorama` to show the context menu after a mouse click. Consequently, the context menu occurs only once in the intermediary step (cf. Section 3.1) of the rendering process, for instance in the end of the document. This technique is similar to side-notes (cf. Fig. 1 but in this case the context menu has a fixed position in the browser window independently from scrolling according to line 7). The fixed position is not comfortable because the menu does not appear next to the corresponding XML element by default (like in Fig. 62) but on a fixed position for every XML element in the Web browser window. To cope with that problem a new value for the property `position` could be introduced allowing to render XML elements matched by a `panorama` selector relatively to the XML element where the event was initiated. Otherwise the second alternative admits relative positioning of context menus as follows:

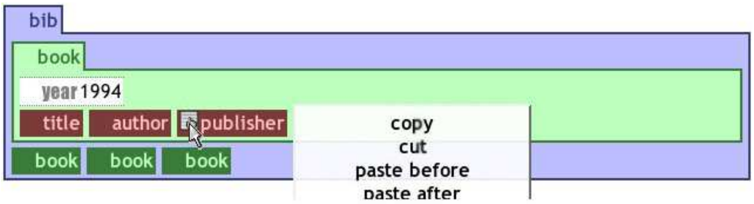


Figure 62: Relative Positioning

Alternative 2 assumes that the context menu appears in the tab of every XML element. In this case, the context menu can be inserted using the style sheet of Fig. 57 extended by the menu items in line 1 in `element(menu)`.

```
1 <tab>
2   <menu>
3     <item>copy</item><item>cut</item><!-- ... -->
4   </menu>
5   bib
6 </tab>
7 <bib>
8   <!-- content of bib -->
9 </bib>
```

Figure 63: Intermediary Step of the `bib` XML Element

Note that the style sheets only define the rendering of the context menus. Their features can be implemented using a scripting language such as ECMA script [ECM99].

5.1.4 Color Scheme

The `:nth-descendant` pseudo class proposed in Section 4.3 allows to overcome the definition of colors only until a certain depth. As can be seen in Fig. 65, the colors have a fixed order which is repeated twice. Afterwards the colors stay the same since the CSS rule on depth 12 matches for all XML elements on deeper levels:

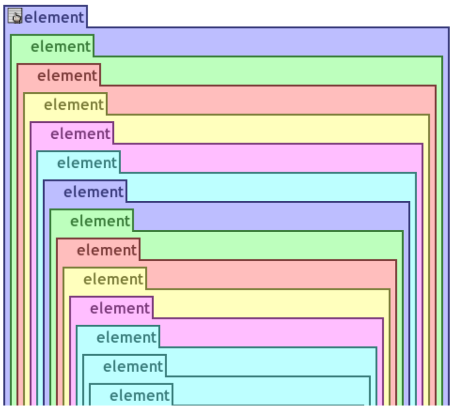


Figure 64: Acyclic Definition of Colors in visXcerpt

The following style sheet, however, uses only six rules instead of twelve and can cope with coloring XML elements on arbitrary depth:

```
1 *:nth-descendant(6n+1) { background-color:#bfbfff; color:#3f3f7f; }
2 *:nth-descendant(6n+2) { background-color:#bfffbf; color:#3f7f3f; }
3 *:nth-descendant(6n+3) { background-color:#ffbfbf; color:#7f3f3f; }
4 *:nth-descendant(6n+4) { background-color:#ffffbf; color:#7f7f3f; }
5 *:nth-descendant(6n+5) { background-color:#ffbfff; color:#7f3f7f; }
6 *:nth-descendant(6n+6) { background-color:#bfffff; color:#3f7f7f; }
```

Figure 65: Cyclic Definition of Colors in visXcerpt

Since six different colors are used in visXcerpt to visualize XML elements on diverse levels, six rules are needed to define the color on each class of levels. According to the rule in line 1 of the style sheet in Fig. 65, each XML element on every sixth level starting with the first level gets violet as color. The rule in line 2 of the style sheet starts coloring every sixth element starting on the second level and so on.

5.2 Construct-Query Rules and Goals

A visXcerpt program consists of zero or more construct-query rules, one or more goals and zero or more data terms. Both rules and goals have the following form:

```
<rule>
 <construct><!-- construct term --></construct>
 <from>     <!-- query term     --></from>
</rule>
```

Figure 66: Skeleton of a Rule (or a Goal) in Xcerpt

Rules (or a goal) in visXcerpt can be rendered using the proposed extension ‘insertion of markup’ (cf. Section 4.1) as follows:

```
1  *                   { display: block }
2  rule                { margin: thin solid black}
3
4  // positioning
5  rule, construct     { position:relative }
6  from                { position:absolute; right: 50% }
7
8  // arrow
9  from::before        { content: element(arrow)}
10 arrow               { background-image:url(arrow.png);
11                       background-repeat:no-repeat }
```

Figure 67: Style Sheet defining the rendering of Rules (or Goals) (cf. Fig. 68

The style sheet in the Figure above defines the basic rendering of a visXcerpt rule (or goal). In line 2, a margin around the rule is added. Since the construct term and the query term of a visXcerpt rule (or goal) are supposed to be arranged in a row, the rule in line 6 lets each query term appear relatively (line 5) on the right side of a construct term. As can be seen in Fig. 68, the construct and the query terms are separated by an arrow. This arrow is defined in line 9, where its element is inserted, and in line 10, where the arrow picture is defined as background picture of the `arrow` XML element.

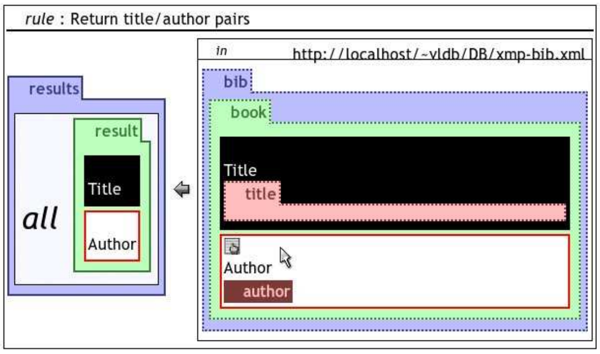


Figure 68: Rendering of an visXcerpt Rule using the Style Sheet of Fig. 67

Note that the rendering of construct terms and query terms (cf. Fig. 66) can be defined analogously to the rendering of data terms (cf. 57) with slight extensions supporting for instance for `descendant` or for `and`. The style sheets are not given in this thesis.

5.2.1 Highlighting of Variables

In visXcerpt variables having the same name can be highlighted, if hovering over its rendering with a mouse cursor (e.g. in a Web browser). For instance, the variable `Author` in Fig. 68 is highlighted because the mouse cursor hovers over an appearance of the `author` element.

Highlighting of variables such as demonstrated in Fig. 68) can be defined by the style sheet given in Fig. 69. By default variables are visualized by a box using a black background color and white font. If a variable is highlighted, all occurrences are highlighted by inverting the colors of their renderings.

```
1 variable                               { color: black;
2                                          background-color: black; }
3 variable[name!]:onMouseOver(panorama) { color: white;
4                                          background-color: white; }
```

Figure 69: Style Sheet Defining Highlighting of Variables

Variables are rendered as discussed above by the style sheet of Fig. 69. The rule in the first line defines the default rendering of variables. Dynamic rendering is defined by the rule in line 4: If a mouse cursor hovers over the rendering of a an XML element `variable`, all renderings of `variable`s having the value of the XML attribute `name` are highlighted.

6 State of this project

The extensions proposed in this project are not implemented. For developing issues, the dynamic extensions were simulated using existing constructs from CSS such as the dynamic pseudo-class `:hover`.

An implementation of the extensions could be realized, for instance, using the Java Programming Language [Sun05], XUL [Moz01] or ECMA Script [ECM99]. All of these alternatives are based on a platform independent language to ensure widespread accessibility of an implementation.

References

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[BBSW03] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Proceedings of 29th Intl. Conference on Very Large Databases, Berlin, Germany (9th–12th September 2003)*, 2003.

[Ber03] Sacha Berger. Conception of a Graphical Interface for Querying XML. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2003.

[ECM99] ECMA. *Standard ECMA-262, ECMAScript Language Specification*, 1999.

[Kep04] Stephan Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Proceedings of Extreme Markup Languages 2004, Montreal, Canada (2nd–6th August 2004)*, 2004.

[Moz01] Mozilla Foundation. *XML User Interface Language*, 2001.

[SB04] Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.

[Sun05] Sun Microsystems. *Java Programming Language*, 2005.

[W3C98] W3C. *Cascading Style Sheets (CSS), level 2*, 1998.

[W3C99a] W3C. *Associating Style Sheets with XML documents*, 1999.

[W3C99b] W3C. *XML Path Language (XPath) Version 1.0*, 1999.

[W3C00a] W3C. *Extensible HyperText Markup Language (XHTML) 1.0*, 2000.

[W3C00b] W3C. *Extensible Markup Language (XML) 1.0, 2nd Edition*, 2000.

[W3C01a] W3C. *Extensible Stylesheet Language (XSL) 1.0*, 2001.

[W3C01b] W3C. *XML Linking Language (XLink) Version 1.0*, 2001.

[W3C02] W3C. *Link recognition for the XHTML Family*, 2002.

[W3C05] W3C. *Cascading Style Sheets Under Construction*, 2005.