# Visual Languages: A Matter of Style

Sacha Berger François Bry Tim Furche

Institute for Informatics, University of Munich, Germany
{sacha.berger,francois.bry,tim.furche}@ifi.lmu.de

Christoph Wieser

Salzburg Research, Austria
christoph.wieser@salzburgresearch.at

**Abstract**

Styling has become a widespread technique with the advent of the Web and of the markup language XML. With XML, application data can be modeled after the application logic regardless of the intended rendering. Rendering of XML documents is specified using style sheet languages like CSS. Provided the styling language offers the necessary capabilities, style sheets can similarly specify a visual rendering of modeling and programming languages. The approach described in this article considers visual languages that can be defined as a 1-to-1 visualization of (an abstract syntax of) a textual language. Though the approach is obviously limited by the employed style sheet language, its advantages are manifold: (a) visualization is achieved in a *systematic* manner from a *textual* counterpart which allows the same paradigms to be used in several languages and ensures a close conceptual relation between textual and visual rendering of a language; (b) visual languages are much *easier to develop* than in ad-hoc manners; (c) the capability for *adaptive styling* (based on user preference such as disabilities or usage context such as mobile devices) is inherited from Web style sheet languages such as CSS.

To make CSS amenable to visual rendering of a large range of data modeling and programming languages, this article first introduces limited, yet powerful extensions to CSS. Then, it demonstrates the approach on a use case, the logic-based Web query and transformation language Xcerpt. Finally, it is argued that the approach is particularly well-suited to logic-based languages in general.

## 1 Introduction

Styling has become a widespread technique with the advent of the Web and of the markup language XML. The success of style sheet languages such as CSS is based on the ability to separate the conceptual or logical structure of Web data (be it in HTML or XML format) from the visual presentation of that data. Such a separation is convenient for adaptive presentation of content based on user preferences or usage context (in particular, for human as well as machine users such as search engine bots), for agile management and rapid development of Web sites, and for separating the concerns of content and presentation.

Many of the reasons why styling has succeeded for visualizing data apply also to the visualization of *programs* (i.e., of data modeling and programming languages), though interactive features become possibly even more important. The advantages of styling for data are inherited: easy conception of new visual languages; adaptive styling allowing different presentations based on user, device, etc.; systematic relation between abstract concepts, visual, and textual rendering of the language limiting impedance mismatch when switching between different renderings of a language. A further advantage is that the approach inherently permits "round-trips": A program developed so far as text (visually, resp.) can be further developed visually (as text, resp.).

Obviously, this approach is limited by the capabilities of the style sheet language employed. We choose in this article CSS for its widespread use and impressive visualization abilities: recent developments in the area of Web design and rich interfaces for the Web as well as the development of CSS 3.0 demonstrate the versatility of CSS-based visualization. The days of strictly hierarchical visualization are over with features such as absolute positioning supported by all mainstream browsers. The only remaining limitations of CSS are the rather rigid box model (which makes, e.g., ad-hoc curves impossible) and the limited interactivity features. The first limitation is starting to get addressed by recent proposals to add free-style drawing to HTML and CSS (cf. `canvas` element). The resulting flexibility in visualization is demonstrated by applications such as Yahoo! Pipes[1].

A first step to address the limitations to interactivity is proposed in this article: a *limited, yet far reaching extension* to the style sheet language CSS that makes it better suited for the rendering of not only data but also programs where interactive behavior becomes even more central. This extension (as well as the entire approach) is demonstrated on a use case, the logic-based Web query and transformation language Xcerpt.

The visualization considered in this article is deliberately simple, so as to be realizable with a rather limited extension, called here CSS$^{NG}$, of the dominant Web style sheet language CSS. The generality of the approach should, nonetheless, become evident: Instead of CSS or CSS$^{NG}$ a style-sheet language offering other visualizations could be used.

**XML source**

```
<bib>
  <book year="1994" id="42">
    <title>
      TCP/IP Illustrated
    </title>
    <author>
```

**Presentation**

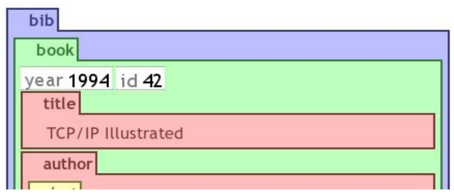

Figure 1: XML document (left side) and rendering using CSS$^{NG}$ (right side).

CSS$^{NG}$ is a novel extension of CSS 3, the latest version of CSS, introducing just a few novel constructs for *interactive or dynamic rendering* and for *markup visualization*. This limited extension of CSS 3 turns out to enable rather advanced visualization of programs. Even though CSS$^{NG}$ is a limited and conservative extension of CSS, it adds considerably to the power of CSS allowing (a) to specify many forms of (interactive or) dynamic styling; (b) to generalize markup visualization; (c) to integrate the keyboard as input device (where CSS 3 mostly treats only a pointer input device such as a mouse).

[1]http://pipes.yahoo.com/pipes/

Thus, CSS$^{NG}$ allows for a declarative, concise, and simple specification of dynamic document rendering in particular, when compared to current state-of-the-art techniques such as ECMA Script [10]. The same applies for markup visualization where currently far more complex technologies such as XSLT [13] must be employed.

## 2 CSS in a Nutshell

CSS 3 and its predecessors have been developed to simplify changes of the content as well as of the presentation of HTML and XML documents by separating content from presentation. It specifies formatting using rather simple guarded rules with formatting instructions. The following rule demonstrates a well-known *static* styling feature:

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

Intuitively, the rule reads as "**if** an element matches `a`, **then** format it by underlining its contained text". The left-hand, or *selector*, of the CSS rule selects HTML anchors (denoted as `a` elements). The *declaration* on the right-hand side assigns the styling parameter to XML elements matched by the selector of the rule.

Also, some *dynamic* styling features are offered in CSS 3. For instance, the background color of an HTML anchor can be switched to yellow while the mouse cursor is hovering (`:hover`) over it:

```
a:hover  { background-color: yellow; }
```

Markup especially in XML documents often conveys application relevant information (e.g., the role of a person associated with a book—author, editor, publisher, reviewer, etc.). Therefore, it might be useful to visualize it. However, CSS 2.1 and CSS 3 offer quite limited means for markup visualization which, in current Web application, often forces the use of other, less declarative technology to complement CSS such as ECMA script or server-side scripting languages. The following subsections 2.1 to 2.3 briefly introduce novel static CSS$^{NG}$ rules mainly aiming at visualizing XML markup. Finally Section 2.4 introduces the rule-based interface for dynamic document styling. Full details on how CSS$^{NG}$ extends CSS 3 can be found in [14].

### 2.1 Markup Insertion

CSS 3 allows the insertion of plain text specified in a CSS style sheet. The CSS emphpseudo-elements `::before` and `::after` cause insertion of text before and after a selected XML or HTML element.

CSS$^{NG}$ extends these pseudo-elements of CSS 3. In addition to inserting plain text in CSS 3, the CSS$^{NG}$ functions `element(NAME,ATTRIBUTES, VALUE)` and `attribute(NAME,VALUE)` provide in addition means for inserting XML elements and attributes before and after XML elements. The following example inserts a elements with a title-attribute of value "Tab" and content "element" before each element in an XML document. See Figure 3 how this can then be employed to visualize these new elements as "tabs" for hiding or unhiding information.

```
  *::before { content: element("a",
                            attribute("title","Tab"),
                            "element") }
```

## 2.2 Markup Querying

CSS 3 provides the function `attr(`$X$`)` for querying the content of a known XML attribute $X$ of an XML element. The name of an XML element and its XML attributes can not be queried. Implementing markup visualization as in Figure 1, i.e., where the name of an element is used as content of a newly created element to make the markup visible, without generalized markup querying means one rule for every XML element type like the XML `bib` element in Fig. 1.

CSS$^{NG}$ adds the function `element-name()` yielding the name of the currently selected XML element. Furthermore, one XML element can host several XML attributes. Therefore, CSS$^{NG}$ offers *attribute rules* selecting XML attributes instead of XML elements. The CSS$^{NG}$ functions `attribute-name()` and `-value()` query XML attribute names and values in the context of a selected XML element. The example in Figure 2 implements a tab in front of each XML element listing the XML element name and all of the XML elements' attributes including their values as shown in Figure 1.

**XML source (see Figure 1)**

```
... <book year="1994" id="42"> ... </book> ...
```

**CSS$^{NG}$ style sheet**

```
*::before { content:
  element("span",       element("span", element-name())
                    * { element("span", attribute-name() " "
                                        attribute-value() )
                      } )
}
```

**Resulting XML tree**

```
... <span>
      <span>book<span>
      <span> year  1994</span>
      <span> id  42</span>
    </span>
    <book year="1994" id="42"> ... </book> ...
```

Figure 2: Generation of tabs. The presentation in Figure 1 is obtained by rendering the resulting XML tree using further CSS 3 means.

## 2.3 Depth-dependent Styling

Styling depending on breadth (i.e., on position among siblings) is planned in CSS 3 [7]. Tables, for instance, can be styled using alternating background colors for each line. CSS$^{NG}$ additionally offers styling depending on the depth (i.e., position among ancestors) of an XML element in an XML document: `:nth-descendant(an+b)` restricts selections to XML elements having $an + b$ ancestors.

Figure 3 demonstrates the visualization of a highly nested XML document with colors repeating on every third level. On the left side this rendering is realized using

CSS$^{NG}$ and alternatively using CSS 3. Thanks to its depth-dependent styling features, the upper CSS$^{NG}$ style sheet needs only three rules. The CSS 3 style sheet below needs one rule for every level. Hence, styling in CSS 3 is possible up to a certain depth only as shown on the right side of Figure 3 using the CSS 3 style sheet on the lower right side of Figure 3. Such a styling is also useful for applications such as the visualization of threads in a discussion forum.

**CSS$^{NG}$**

```
*:nth-descendant(3n+1) { background-color: A; }
*:nth-descendant(3n+2) { background-color: B; }
*:nth-descendant(3n+3) { background-color: C; }
```

**CSS 3**

```
*                       { background-color: A; }
* *                     { background-color: B; }
* * *                   { background-color: C; }
* * * *                 { background-color: A; }
* * * * *               { background-color: B; }
* * * * * *             { background-color: C; }

* * * * * * *           { background-color: A; }
* * * * * * * *         { background-color: B; }
...
```

**Presentation using CSS 3**

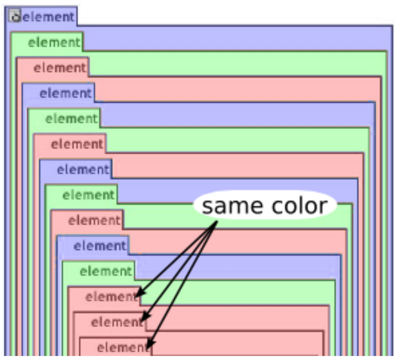

Figure 3: Comparing Depth-dependent Styling using CSS$^{NG}$ and CSS 3.

## 2.4 Dynamic Styling Generalized

Dynamic styling is necessary to support (basic) interactivity, i.e., to change formatting (position, color, font, etc.) based on user input such as mouse clicks or move. CSS 3 is limited to the dynamic pseudo-class `:hover`. This construct allows dynamic styling in the local context of the mouse cursor only as demonstrated in Section 2. This is not sufficient to implement a behavior like folding a tab as demonstrated in Section 5: when the mouse cursor moves away, the cursor does no longer hover over the selected XML element, and its tab would be automatically unfolded.

CSS$^{NG}$ introduces dynamic pseudo-classes for *all* HTML intrinsic events [1] such as `onclick` or `onkeypress` (see [14] for sample applications). Instead of using HTML intrinsic event attributes like for scripting languages, CSS$^{NG}$ allows a standalone specification of dynamic styling in separate CSS$^{NG}$ files that can be applied for multiple documents. The following example in Figure 4 shows a rather simple dynamic CSS$^{NG}$ rule.

```
a:onclick(10) { background-color: green; }
```

Figure 4: Dynamic Styling of an adaptive hyperlink (CSS$^{NG}$).

The rule in Figure 4 implements an adaptive hyperlink. After 10 clicks on the hyperlink the background color changes to green meaning that the hyperlink on the Web page is often visited by a specific user.

This extension makes it possible to apply dynamic styling on different sections of an XML document at the same time. For instance if two hyperlinks were clicked ten times in a Web page, both will be presented with different background colors.

Similar extensions using HTML intrinsic events have been already proposed by the W3C [8]. The following paragraphs introduce the novel capabilities of CSS$^{NG}$:

**Recurrence Patterns.** All CSS$^{NG}$ dynamic pseudo classes support *recurrence patterns*, `an+b`, as parameters. For instance the CSS$^{NG}$ selector `*:onclick(3n+1)` detects the first, the fourth, the seventh, etc. click on an arbitrary XML element. More generally, a CSS$^{NG}$ selector fires, if $an + b$ events occurred before.

On one hand such recurrence patterns allow to reuse CSS$^{NG}$ rules for folding and unfolding as demonstrated in the following paragraph. On the other hand recurrence patterns allow to "delay" the application of rules up until a number of events, for instance clicks, as demonstrated in the previous Section (see adaptive hyperlink above).

**Dynamic Styling Combined.** A noticeable feature of the (novel) dynamic pseudo-classes of CSS$^{NG}$ is their compatibility with CSS 3 *combinators*, which allow to specify tree patterns.

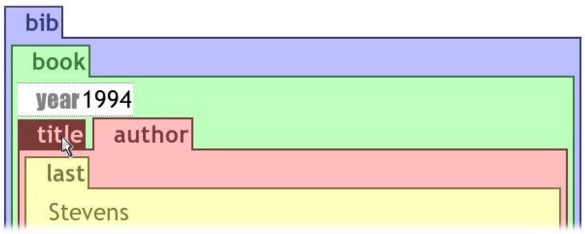

Figure 5: Folded visualization of an XML element `title`. The corresponding unfolded example is shown in Figure 1.

A CSS 3 selector is an alternating sequence of so-called *simple selectors* (already informally introduced in Section 2) and combinators. For instance, the combinator + means that the simple selector on its left side must be a preceding sibling of the simple selector on the righthand side. The CSS declaration (in curly braces) is only applied to the XML element matched by the matching simple selector.

The following example (see Figure 6) implements *alternating folding and unfolding* for the visualization of arbitrary (simple selector `*`) XML elements (see Figure 5). A click on a tab of a visualized XML element like `title` folds its visualization. Another click on a tab unfolds it (see `title` in Figure 1):

```
tab:onclick(2n+1) + * {display:none}        Fold on odd number of clicks.
tab:onclick(2n+2) + * {display:block}    Unfold on even number of clicks.
```

Figure 6: Combined dynamic styling in CSS$^{NG}$ (rendering in Figure 5).

In the example above, the lefthand *selector* of the first CSS$^{NG}$ rule above is composed of the two *simple selectors* `tab:onclick(2n+1)` and `*` combined with the CSS 3 combinator, +. The visualization of an XML element matched by the simple selector `*` disappears, if a mouse click was performed on its preceding sibling XML element, while its tab stays visible.

**Structure-Independent Styling.** A static CSS 3 styling rule is applied to all XML elements matching its selector. A dynamic CSS 3 styling rule is applied only to XML elements being in the context of an input device such as an XML element lying under the mouse cursor. CSS$^{NG}$ abolishes this restriction and allows (novel) so-called *monorama* and *panorama* selections as demonstrated in Figure 7: The `Author` element on the left side is highlighted, while the mouse cursor is hovering over the `Author` element on the right side.

```
Author                { background-color: black; }
Author:hover ? Author { background-color: white; }
```

Figure 7: Highlighting of Xcerpt variables.

The CSS 3 rule in line 1 defines the standard background black for XML `Author` elements. In line 2 the CSS$^{NG}$ combinator `?`, called *if*, is applied as follows: If an XML `Author` element is hovered in an XML document, set the background color of all XML `Author` elements to white.

A proof-of-concept prototypical implementation of CSS$^{NG}$ was implemented as part of a diploma thesis [14] and presented [8].

## 3 Styling of Logic Languages

The approach described in the previous section to conceive a visual language as a rendering, or styling, of a textual language seems for the following two reasons especially convenient for logic languages:

- Logic languages are declarative, i.e. they focus on both the structural and conceptual organization of the data.
- Logic languages are often "answer closed" in the sense of query languages: queries or conditions resemble data and data (i.e., answers) can be used in place of queries. This makes style sheet languages developed for data visualization easily adaptable for program visualization since they are already able to visualize the data.
- Logic languages are often referentially transparent allowing mostly context-independent visualization of language constructs. In particular, this allows visual aids such as highlighting of related parts in a program or rule (e.g., variable occurrences or predicate symbols).
- Logic languages come in families that share traits, like e.g. modal languages, rule-based languages, logic programming languages, frame logic languages. With the approach proposed, "visualizations" can be rather easily developed and applied to various languages of a same language family.

For these reasons, it is the firm belief of the authors that the approach proposed in this article has the potential to boost the development and testing of visual languages, especially of visual logic languages.

## 4 visXcerpt — the Visual Twin Sibling of Xcerpt

As an example of the visualization of a textual language using the presented approach and CSS$^{NG}$, the Web query and transformation language Xcerpt [12] and its visual counterpart visXcerpt [3] are presented. Xcerpt is a rule based deductive language in the spirit of SQL or Datalog but for semi-structured data. As a textual language, it comes in two syntax flavors — an abbreviated syntax and an XML syntax. Rules consist of a head, also called construct pattern and a body consisting of logically connected query patterns. Query and construction share values by means of shared variables, rules query each other heads employing forward or backward chaining. Construct patterns may contain special grouping constructs to collect multiple variable bindings in one result, queries may consist of incomplete query patterns with incompleteness in breadth and/or depth and/or order, reflecting the incertitude about size and structure of documents on the web. Patterns are hence like *"examples"* of web data searched for in given documents.

The central part of visXcerpt, the visual rendering of Xcerpt, is the visualization of Web data, of XML documents. As Xcerpt itself comes in XML syntax, half the job is done by visualizing XML.[2] Further aspects, like partiality, grouping constructs and variables are then added to get a full featured visualization of query and construct patterns. Rules are just represented as horizontal aligned head and body, related by an arrow, though more involved visualizations (e.g., grouping by related root labels) can be realized with CSS$^{NG}$.

**Term Visualization.** Web data and patterns are considered to have a term like structure. Terms are rendered as boxes with their name as a tab on the top, the box contains all tabbed boxes of the subterms in the order they occur. The rendering is conceived to be suitable for most web browsers, considering that they are a wide spread technology with high adaptability to various screen sizes and resolutions. Order is given by a left-to-right and top-to-bottom flow layout, but the layout directions should be adapted to local writing habits of the user's culture. Width is given by the width of the display or browser employed. Nested boxes are further distinguished using colors, hence colors represent nesting depth. To be able to make a reasonable selection of well assorted, distinguishable and pleasant color themes, colors of upper levels are recycled for deeper nestings.

**Graph Visualization** On the Web, graph structures also need to be represented, e.g. RDF [11] data representing graph shaped structures or hyperlink structure. In textual representations of graph structures, references are used along a spanning tree of the graph. The presented approach of visualizing such graph structures is to model the references as hyperlinks in a web browser. This way, even very large graph structures can be represented and access to any references item is achieved by user interaction with constant complexity – a click on a hyperlink. While browsers often provide some means of navigating *back* along edges represented by hyperlinks, it is arguably useful to explicitly give hyperlinks for reverse traversal of edges, as hence the user is not restricted in his backward movement along edges he just visited.

[2]To some extent, this applies to any language as we can always consider for styling the XML serialization of the abstract syntax tree of a program.

**Information focusing** For large documents, it is of vital necessity to give users the ability to hide temporarily unneeded information or to focus on relevant data. This is achieved by means of folding in or out terms behind their name tagged tabs. While elements are aligned vertically, tabs are first aligned horizontally and then vertically, saving even more space. The concept is strongly inspired by tree browser visualization as e.g. seen on the well known Windows file browser.

At this level, pure static visualization starts to merge with user interaction. A visualization with adequate support of user interaction, especially of editing, is indeed much more useful than a static visualisation.

**Textual Xcerpt Program, and** **visXcerpt rendering of it.**

```
CONSTRUCT
 results[
  all result[
   var Title,
    var Author
 ] ]
FROM
 in(resource="file:bib.xml")
 proceedings04[[
  papers[[
   paper[[
    var Title as title[[]] ,
    var Author as author[[]]
   ]]
  ]]
 ]]
END
```

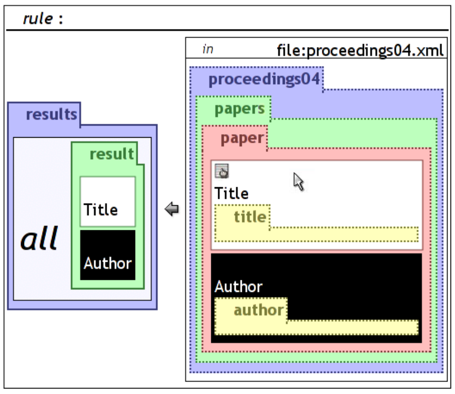

Figure 8: A single rule Xcerpt program (in abbreviated textual syntax) along with its visXcerpt rendering — the query part exploits a partial pattern (indicated by dotted lines in the visualization) to search for papers in a proceedings database, constructing title/author pairs all grouped in a list of results. All *Title* variables are highlighted as the mouse is hovering above one of them in visXcerpt.

**A Special Purpose Editor Model** For textual languages, copy-and-paste and text typing based editors are wide spread. Central to textual editing, is a cursor concept, that usually is a separator of the one dimensional program. For the presented visual approach, a separator seemed not intuitive, hence a context metaphor is used for editing: each box is a context, it is possible to cut, copy or delete it with or without its sub boxes, it is possible to paste the content of the cut/copy buffer into, before, after or around a context and hence term. The rich copy and paste model is accompanied by a template concept, giving access to all program constructs and possibly example terms or structures that can be altered, reduced or extended.

## 5 Realizing CSS$^{NG}$: CSS & XSLT

As a proof-of-concept, we chose to implement CSS$^{NG}$ by a combination of XSLT transformations and reductions to standard XHTML and CSS to allow for maximum portability and fast implementation.

All data formats and transformations except **CSS$^{NG}$ Parser** are based on W3C standards. Except for the CSS and CSS$^{NG}$ parsers, all other program transformations are implemented in XSLT [13]. The XSLT transformations essentially evaluate the (static) rules in the CSS$^{NG}$ stylesheet statically and adorn the XHTML elements to allow the use of standard CSS (and ECMA Script for the dynamic styling). The **Styler** is the heart of the system. It processes all XHTML elements in the document tree of an **(Un)styled Document** recursively. Each XHTML element passes through one test for each CSS$^{NG}$ rule in a CSS$^{NG}$ style sheet. If a test succeeds, the XHTML `style` attribute of the current XHTML element is modified. The tests are implemented in XPath [9]. Since tests are executed from the perspective of each XML element, CSS$^{NG}$ selectors need to be translated to XPath selecting XML elements in reverse direction as demonstrated in the following example (see Figure 9):

**CSS$^{NG}$**

```
div :onclick(2n+1) + *
```

**XPath**

```
self:: * /preceding-sibling:: div [
   span[@class='onclick'] mod 2 = 1 ]
```

Figure 9: Translation of CSS Selectors in XPath (CSS$^{NG}$).

## 6 Outlook and Conclusion

The presented approach — obtaining a visual language by mere rendering or styling of a textual language — has been explored with the textual query language Xcerpt. To the largest extend, this has been achieved using standard CSS, for the most salient features however an extension of CSS has been conceived.

### 6.1 Conclusion

visXcerpt has been prototypically implemented and successfully applied for the presentation of Xcerpt [6] [5], widely easing the comprehension of the concepts of Xcerpt. visXcerpt's editor model turned out to be convenient for Xcerpt programming tasks from the area of HTML content extraction, creation and wrapping, over XML data transformation to Semantic Web and hybrid Web and Semantic Web reasoning [4].

CSS$^{NG}$ as an extension of CSS turned out to be easily realizable without heavy computational overhead compared to CSS 2 and CSS 3. It proved itself to be not only a tool for the implementation of visXcerpt, but especially for sophisticated visualization of XML data with easily realizable domain specific behavior.

The approach of conceiving a visual language based on a textual back-end turned out convenient in both cases, for the creator of the visual language as well as for the programmer using the language — creating a visual language as a rendering of a textual one was reasonably easy, and programmers using it where pleased to be able to switch between textual and visual representation.

To the best of the knowledge of the authors similar generic approaches of developing visual languages as mere rendering using CSS and extensions have not been proposed so far.

## 6.2 Outlook

Further interesting research in the area of Xcerpt/visXcerpt is to investigate about type support, not only in the textual language for checking and validation of programs [2], but also in the editing process. This could help novice users to by just providing editing features that lead from one valid program to another, as well as providing a type based template approach over the example based approach. In the area of generic visualization of textual languages, it is needed to systematically investigate further features/functionalities that would be desirable for visual languages and what existing styling languages would be a convenient basis for adding these features. It would be interesting to develop a few style-sheet languages which could render various textual modeling and/or programming languages as visual languages after various visualization paradigms. The Semantic Web logic languages RDF, OWL and the new Rule Interchange Format (RIF) would be promising candidates for such investigations.

## References

[1] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. *HTML 4.01*. W3C, 1999.

[2] Berger, Coquery, Drabent, and Wilk. Descriptive typing rules for xcerpt. In *Proc. of 3rd Workshop on Principles and Practice of Semantic Web Reasoning*, 2005.

[3] S. Berger. Conception of a Graphical Interface for Querying XML. Diploma thesis, Institute for Informatics, LMU, Munich, 2003.

[4] S. Berger, F. Bry, O. Bolzer, T. Furche, S. Schaffert, and C. Wieser. Querying the standard and Semantic Web using Xcerpt and visXcerpt. In *Proc. of European Semantic Web Conference*, 2005.

[5] S. Berger, F. Bry, and T. Furche. Xcerpt and visXcerpt: Integrating Web Querying. In *Proc. of Programming Language Technologies for XML*, 2006.

[6] S. Berger, F. Bry, S. Schaffert, and C. 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*, 2003.

[7] B. Bos. *Cascading Style Sheets Under Construction*. W3C, 2005.

[8] F. Bry and C. Wieser. Web Queries with Style: Rendering Xcerpt Programs with CSS-NG. In *Proc. of 4th Workshop on PPSWR*, 2006.

[9] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. W3C, 1999.

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

[11] O. Lassila and R. R. Swick. *Resource Description Framework*. W3C, 1999.

[12] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. of Extreme Markup Languages*, 2004.

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

[14] C. Wieser. CSS$^{NG}$: An Extension of the Cascading Styles Sheets Language (CSS) with Dynamic Document Rendering Features. Diploma thesis, Institute for Informatics, LMU, Munich, 2006.