An Algebraic Framework for Modeling and Processing Hyperdocuments

Volker Mattick
Lehrstuhl Logik in der Informatik, Fakultät für Informatik, TU-Dortmund
volker.mattick@tu-dortmund.de

1 Introduction

The most prominent languages for describing hyperdocuments are based on the Extensible Markup Language (XML[1]). This family of languages is normed by committees, initiated by the W3C[2], and became the quasi-standard in the World Wide Web. These xml-based languages are more the result of long discussion processes, influenced by industry, government and praticioneers, then of a rigoros language design. Maybe over time this standards may change, but a radical new design of hyperdocument description languages is not in sight and would be very time-intensive and expensive. So we need techniques, dealing with this, from the perspective of language design not optimal standard, that enables us to understand these standards better and tools that help developers to produce correct documents, that fulfill the users' needs.
A lot of theoretical research in this area comes originally from database theory. An xml-based document there is seen as a file of a semi-structured database, which is described by the according schema. So mainly structural aspects play a role. When seen as a hyperdocument also formatting aspects are crucial.

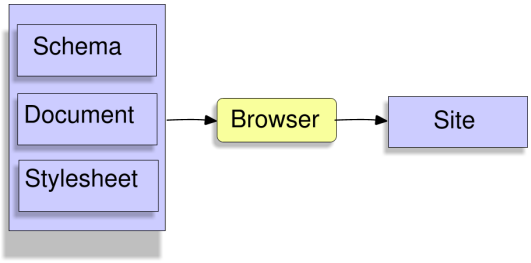

Fig. 1. *Simplified XML-Document Processing*

Abstracted from details (c.f. Fig. 1) a browser takes three different input files, a schema, a document and a stylesheet, each of them in a particular language. The browser parses the schema with an internally defined parser for the schema language and generates a parser for documents that are valid w.r.t. this schema. Then it reads the document itself and transfers it into an internal representation, called formatting object tree. In the next step the stylesheet is parsed with an internal predefined parser and transformed into functions, that modifies the formatting object tree into a refined formatting object tree. Finally this tree is interpreted by an area model, which is rendered by a layout engine (cf Fig. 2).
So in principle a browser is similar to an interpreter, parameterized with a grammar. An interpreter for a context-free grammar G can be defined by the according abstract syntax $\Sigma(G)$ and a parser for $L(G)$ and interpreted by a $\Sigma(G)$-algebra, which represents the target language. The common representation for both are the abstract syntax trees of G. Abstract syntax trees may not be the best form

[1] http://www.w3.org/XML
[2] World Wide Web Consortium, www.w3.org

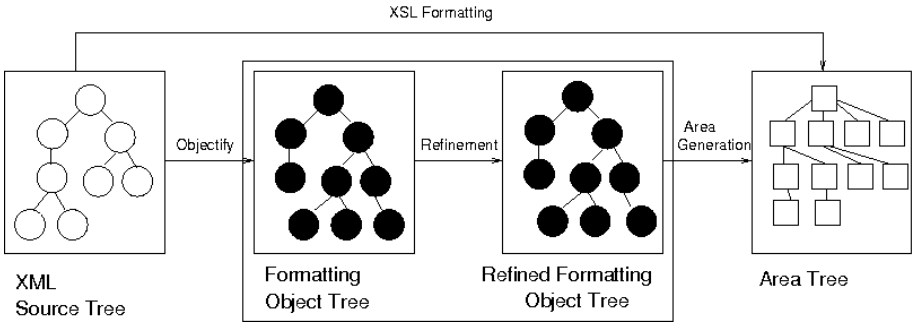

Fig. 2. *XSL Formatting*

to read for a human beeing, but it is the only unambigous form. So they can be used to represent and store xml-based hyperdocuments unambigously.

2 Preliminaries

A common method to define the *concrete syntax* of a formal language is via a grammar (c.f. [4]) that generates the language and which is often denoted in (extended) Backus-Naur-Form (BNF). Let now $L(G)$ be the language generated by the context-free grammar (CFG) G. Then the abstract syntax can be captured by a *many-sorted signature* $\Sigma = (S, F)$ (c.f. [2]), which consists of a set S of *sorts* and a $S^* \times S$-sorted set F of *function symbols*. Instead of $f \in F_{(e,s)}$, we write $f : e \to s \in F$. If $e = \epsilon$, then f is called a *constant*. In detail, if $G = (N, T, P, S)$ is a CFG, then the signature $\Sigma(G) = (N, CO)$ with $CO = \{c_p : X_1 \times \ldots \times X_n \longrightarrow X \mid \exists : p = (X \longrightarrow w_1 X_1 w_2 \ldots w_n X_n w_{n+1}) \in P, w_i \in T^*, 1 \leq i \leq n+1, X_j \in N, 1 \leq j \leq n\}$ is called the *abstract syntax of* G. A particular word of $L(G)$ is then represented by an *abstract syntax tree*. Let $\Sigma(G)$ be the abstract syntax of G, then $AST_{\Sigma(G)} = \cup_{n \in N} AST_{\Sigma(G)_n}$, where for all $e \in N^*, s \in N, f : e \longrightarrow s \in \Sigma(G)$ and $t \in AST_{\Sigma(G)_e}$ there holds $f(t) \in AST_{\Sigma(G)_s}$, is the set of all abstract syntax trees of G.

Each abstract syntax Σ can be assigned a semantic via a Σ*-structure*, a tuple (A, OP), where $\forall : s \in N$ there exists exactly one non-empty $A_s \in A$, $\forall : (co : s) \in CO$ there exists exactly one element $co^A \in A_s$ and $\forall : (co : e \to s) \in CO$ there exists exactly one function $co^A : A_e \to A_s \in OP$. Where possible without confusion, it is abbreviated as A. Mathematically, this structure is a Σ-algebra. One canonical algebra that exists for each Σ is the Σ-ground-term algebra, where each

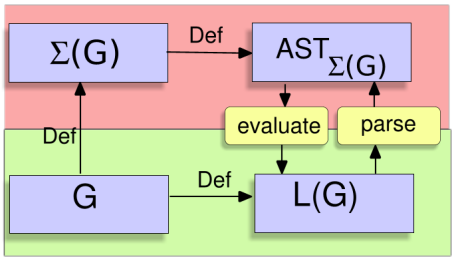

Fig. 3. *Concrete and abstract syntax*

constructor is interpreted symbolically. It can be shown that this is AST_Σ and it is initial in the class of all Σ-algebras. This means there exists an N-sorted Σ-homomorphism $eval^A : AST_\Sigma \to A$ for each Σ-algebra A. An s-sorted mapping h between Σ-structures A and B is called a *Σ-homomorphism*, if for all function symbols $f : w \longrightarrow s \in \Sigma$ there holds $h_s \circ f^A = f^B \circ h_w$. So each other semantic of the abstract syntax can be defined with a morphism from AST_Σ, called *evaluator*. If $AST_{\Sigma(G)}$ are the abstract syntax trees for a context-free grammar G, it can be shown that $L(G)$ is just one of these non-initial semantics of $\Sigma(G)$. Now for all $X \in N$, let $L(G)_X$ denote the subset of $L(G)$ that only contains sentences that are reachable from X, and $L(G) = \bigcup_{X \in N} L(G)_X$. So $L(G)$ is an N-sorted set that can be extended into a $\Sigma(G)$-algebra. To do that, define for each $c_p \in \Sigma(G)$, $c_p^{L(G)} : L(G)_{X_1} \times \ldots \times L(G)_{X_n} \longrightarrow L(G)_X$ as $c_p^{L(G)}(X_1[w_1/X_1], \ldots, X_n[w_n/X_n]) =_{def} X_1 \ldots X_n[w_1/X_1 \ldots w_n/X_n]$. Evaluator and parser are dual. A parser is a morphism between the two $\Sigma(G)$-structures$L(G)$ and $AST_{\Sigma(G)}$. But obviously there is no unique evaluator from $AST_{\Sigma(G)}$ to $L(G)$, whereas the parser is unambigous.
A more interesting feature is, that each semantic can be defined by a morphism and so a compiler in a target language can also be defined via a morphism or by defining an $\Sigma(G)$-structure and then showing that a morphism exists between $AST_{\Sigma(G)}$ and the new defined structure. So we have the possibility to give one abstract syntax tree different interpretations by only changing the $\Sigma(G)$-structure.

3 Algebraic Framework

An xml-based hyperdocument consists, beyond its name and some technical information, of three parts: the *schema*, the *stylesheet* and the structural *document* description. Each of this parts is in general described by a different language.

Schema: The schema S is the generating device for the language of the hyperdocument, similar to a CFG G is a generating device for a context-free language $L(G)$. Schemas generate only special cases of context-free languages, namely regular tree languages (c.f. [6], [5]), a subclass for which easily can be defined a balanced version (c.f. [1]). This balanced version is called *markup language* and denoted by $ML(S)$. Regular tree grammars conists only of guarded rules, which makes the abstraction quite easy, because we have a canonical name for each constructor. So in principle each markup language could be also defined by a CFG. The problem is, that a schema not only contains information, that can be captured by a CFG, e.g. in a schema can be defined how often an element can occur. For simple special cases that it occurs zero or one times, exact once or infinitly often we can find a context-free represenation, but characterizing a range in between is not possible, at least not with reasonable work. Also the fact, that an attribute is required and even more that it has a default value, can not captured by a CFG. So simply converting a schema into a CFG and then building a parser and interpreter for it will not work. In practice the schema is usually

described by a Document Type Definition (DTD), an XML-Schema (c.f. [9]) or a RelaxNG-pattern (c.f. [7]). All these languages can be described by CFGs and so they also have an abstract syntax. So, the solution is to construct the abstract syntax for the schema language and build a schema parser. The abstract syntax then can be interpreted by three different $\Sigma(S)$-algebras. The first one is the $ML(S)$-algebra, which interprets $AST_{\Sigma(S)}$ as the concrete markup language for the author of a document. The second one is the $\Sigma(ML(S))$-algebra, which interprets the schema as the abstract syntax for the markup language. The third one is the document parser algebra, which interprets $AST_{\Sigma(S)}$ as a parser for documents in markup language, including all the constraints that cannot expressed by the language itself. The parser must not only reject non well-formed documents, but also documents, which not fulfill the additional constraints given by the schema.

Stylesheet: The stylesheet is the formatter for a hyperdocument. It defines path-expressions to locate a particular place in the document tree and actions, which modify the selected part of the tree. A *document tree* is an unranked, sibling-ordered, labeled tree with has two different kinds of leaves, called *content nodes* and *attribute nodes*. This document trees are not the abstract syntax trees, but another interpretation of $AST_{\Sigma(ML(S))}$. In practice a stylesheet is usually described by CSS (c.f. [8]) or XSLT[3], both context free languages, which can be captured by an abstract syntax. CSS stylesheets can only modify values of attributes of the document tree, but not the document tree itself. XSLT is a tree-transformation language that can change also the structure of the tree.
For both languages a parser can be constructed that reads the concrete word of the stylesheet language $L(Style)$ into an abstract syntax tree of $AST_{L(Style)}$. So when a hyperdocument is represented by an element $doc \in AST_{\Sigma(ML(S))}$, then a stylesheet must be interpreted by paths on doc and the action by tree-transformations at the located place.

Document: The document is described by a word of the markup language $ML(S)$ and transformed via the parser algebra into an abstract syntax tree $AST_{\Sigma(ML(S))}$. The resulting abstract syntax tree is not necessarily complete in the sense, that all attributes occur and have values and not minimal in the sense that attributes can occur more then once. Moreover this AST is not formatted, because the stylesheet is not yet applied. In practice most hyperdocuments are described by (X)HTML or sublanguages of that.
The interpretation of a $doc \in AST_{\Sigma(ML(S))}$ must have default interpretations for missing attribute values, which may be influenced by information in the schema, e.g. given default values. If an attribute occur more then once the interpretation must always interpret only the last occurence.

Browser: The browser (c.f. Fig. 4) starts with parsing the schema $s \in S$, where S is the schema language, with an internal parser into an abstract syntax tree

[3] `http://www.w3.org/TR/xslt`

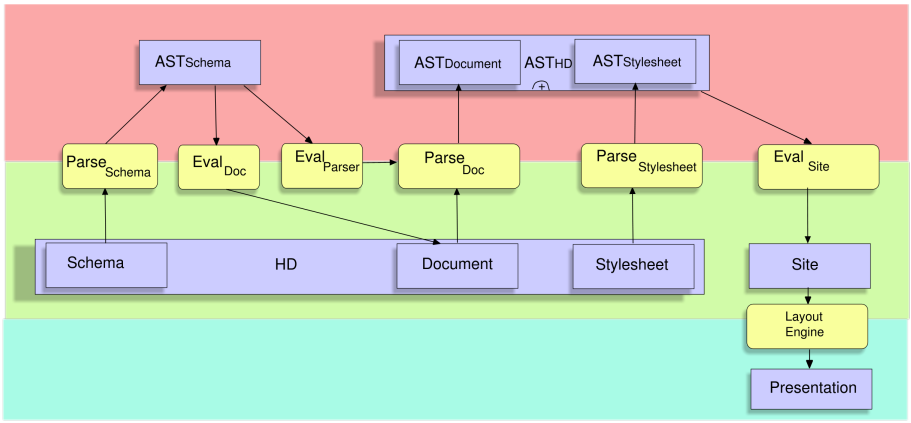

Fig. 4. *Complete Browser*

$schema \in AST_S$. In fact the abstract syntax trees here are not relevant, we are only interested in the signature $\Sigma(S)$ and in three special algebras. Firstly the markup language algebra, which interprets the signature as the according balanced regular tree language. Secondly the algebra, which interprets the signature as the abstract syntax. Thirdly the algebra, which interprets the signature as a validating parser for the markup language. The first one is needed for the author of the document, the second and third one is needed for the browser. All three algebras can be generated automatically, because they are constructed always by the same mechanisms. Then the document $d \in ML(S)$ itself is parsed by the parser algebra into $doc \in AST_{\Sigma(ML(S))}$. In this case we need the abstract syntax tree, because the parsed tree is not interpreted directly. Before we can add a semantic, we must parse the stylesheets. Here again the abstract syntax trees play no role, because we are only interested in one particular interpretation of the signature, namely that one, which interprets a stylesheet by a function on $AST_{\Sigma(ML(S))}$. Then the abstract syntax tree of the document is modified by the functions resulting from the stylesheet. This modified tree is then interpreted by a document algebra, we call *site* for short. For each document there can be different such algebras, depending on e.g. the later output media, the used layout engines or the reader's needs. Most browsers nowadays are graphical browsers, so the document itself is usually interpreted in a graphical way. But different browsers have slightly different interpretations, which leads to different output. For some special purposes, we need textual interpretation.

Editor: Usually hypertext editors work the way, that the author direct edits the markup language representation of the document. For pure textual editors this is not problematic, but for graphical editors it leads to vast problems, well known from WYSIWYG approaches. When a graphically designed document is directly stored in the markup language it is not necessarily interpreted the same way by another editor or browser (c.f. Fig. 5). For one graphical representation

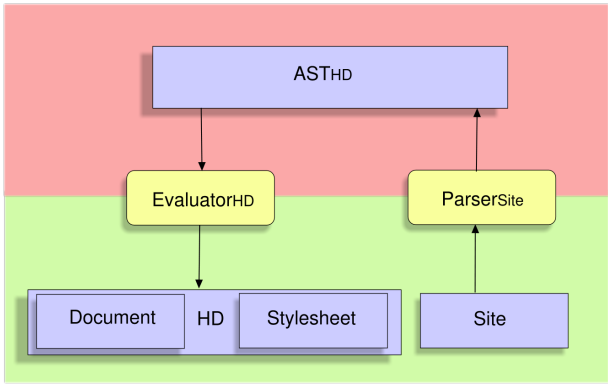

Fig. 5. *Editor*

it can exists more then one textual representation in a particular interpretation. In a even slightly different interpretation this may not be the same. If it is stored by the abstract syntax tree this problem is reduced. If the $\Sigma(ML(S))$-algebra is also stored this is unambigous.

Adaptable Hyperdocuments: Adaptable hyperdocuments (c.f. [3]) can be characterized by abstract syntax trees with variables. Usually abstract syntax trees are always ground terms, meaning terms without variables. If the interpretation of $AST_{\Sigma(ML(S))}$ is known, an abstract syntax tree with variables denotes a subset of $AST_{\Sigma(ML(S))}$. The second thing we have to know is a user profile, describing in terms of the $\Sigma(ML(S))$-algebra, which elements this algebra are allowed. Now the adaptable hypertext system must search for an assignment of the variables, so that the interpretion of the according variable-free abstraxt syntax tree fits the constraints of the algebra. It might be necessary that the original markup language $ML(S)$ must be enriched with new syntactical constructs for representing hyperdocuments with variabels, so called semi-documents.

4 Implementation

Here it is only given a short example using the GPS Exchange Format for exchanging geodata. This is not a typical markup language for hyperdocuments, but it has a comparatively short and understandable schema definition[4] and nearly all features can be demonstrated with this language too. The implementaion examples are coded with Haskell[5]
All elements and types of the schema are represented in the abstract syntax as constructors. The attributes, which belongs to a complex type are represented as constructors of the according attribute sort. Constraints like `minOccurs` or

[4] `http://www.topografix.com/gpx/1/1/gpx.xsd`
[5] `http://www.haskell.org`

`maxOccurs` have default value 1, so it means e.g. that `metadata` should be appear either exact one time or not at all. Basic types like `xsd:string` are assumed to have a predifined interpretation.
This following complex type from the schema S,

```
<xsd:element name="gpx" type="gpxType"/>}
<xsd:complexType name="gpxType">
<xsd:sequence>
<xsd:element name="metadata" type="metadataType" minOccurs="0"/>
<xsd:element name="wpt" type="wptType"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="rte" type="rteType"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="trk" type="trkType"
             minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="version" type="xsd:string"
               use="required" fixed="MiniGPX"/>
<xsd:attribute name="creator" type="xsd:string" use="required"/>
</xsd:complexType>
```

then can be represented by a signature $\Sigma(S) = (N, OP)$:

$$
\begin{aligned}
OP = \{\ & \text{root\_} :: \texttt{gpxType\_av} \times \texttt{gpxType} \rightarrow \texttt{gpx} \\
& \text{gpxType\_} :: \texttt{metadataAlt} \times \texttt{wptList} \times \texttt{rteList} \times \texttt{trkList} \rightarrow \texttt{gpxType} \\
& \text{gpx\_MiniGPX} :: \rightarrow \texttt{gpxType\_av} \\
& \text{gpx\_creator} :: \texttt{xsd:string} \rightarrow \texttt{gpxType\_av} \\
& \_ : \_ :: \texttt{gpxType\_av} \times \texttt{gpxType\_avList} \rightarrow \texttt{gpxType\_avList} \\
& [] :: \rightarrow \texttt{gpxType\_avList} \\
& \ldots
\end{aligned}
$$

This can be implemented straightforward in Haskell the following way:

```
data GpxSIG gpx gpxType gpxType_av ... =
     GpxSIG {root_mt :: [gpxType_av] -> gpx,
             root_ :: [gpxType_av] -> gpxType -> gpx,
             gpxType_ :: (Maybe metadata) -> [wpt] -> [rte] -> [trk] -> gpxType,
             gpx_MiniGPX :: gpxType_av,
             gpx_creator :: XSD_string -> gpxType_av, ...}
```

The algebra, which interprets a document as an abstract syntax tree of $AST_{\Sigma(ML(S))}$ looks like this in the Haskell implementation:

```
gpxALG :: GpxSIG Gpx GpxType GpxType_av ...
gpxALG = GpxSIG Root_Mt Root_ GpxType_ Gpx_MiniGPX Gpx_creator ...
```

```
data Gpx = Root_Mt [GpxType_av] | Root_ [GpxType_av] GpxType
data GpxType = GpxType_' (Maybe Metadata') [Wpt] [Rte] [Trk]
data GpxType_av = Gpx_MiniGPX | Gpx_creator XSD_string
```

And a second algebra, which interprets a document as a object of the Scalable Vector Graphics (SVG[6]) can look like this:

```
svgALG = GpxSIG gpx_ gpxType_ gpxType_av_ metadata_ metadataType_ wpt_ wptType_
                rte_ rteType_ trk_ trkType_ ... =
         where root_mt _ = "<svg />"
               root_ avlist gpxType = "<svg xmlns=\"http://www.w3.org/2000/svg\"
                     xmlns:svg=\"http://www.w3.org/2000/svg\"
                     xmlns:xlink=\"http://www.w3.org/1999/xlink\">"
                     ++ gpxType ++ "</svg>"
               gpxType_ metadata wpt rte trk = "<g>" ++ (conc wpt)
                        ++ (conc rte) ++ (conc trk) ++ "</g>"
               gpx_MiniGPX = "MiniGPX"
               gpx_creator c = c
```

Then a monadic parser not only parses the document, but also transforms it directly into a given interpretation.

```
parseGpx :: GpxSIG gpx gpxType gpxType_av metadata metadataType wpt
    wptType wptType_av rte rteType trk trkType name time
    bounds boundsType_av ele sym number rtept trkseg trksegType trkpt
    -> MParser Char gpx
parseGpx alg = do result <- (parseE `parM` parseC); return result
        where parseE = do avlist <- (opencloseAV "gpx" (parseGpxType_av alg));
                          return (root_mt alg avlist)
              parseC = do avlist <- (openAV "gpx" (parseGpxType_av alg))
                          content <- (parseGpxType alg)
                          close "gpx"
                          return (root_ alg avlist content)

parseGpxType ::  ... -> MParser Char gpxType
parseGpxType alg = do result1 <- qmM' (parseMetadata alg)
                      result2 <- starM (parseWpt alg)
                      result3 <- starM (parseRte alg)
                      result4 <- starM (parseTrk alg)
                      return (gpxType_ alg result1 result2 result3 result4)

parseGpxType_av :: ... -> MParser Char gpxType_av
```

[6] http://www.w3.org/Graphics/SVG/

```
parseGpxType_av alg = parL [parseGpx_MiniGPX alg, parseGpx_creator alg]
     where parseGpx_MiniGPX alg = do isTag "version"
                                     isChar '='
                                     result <- parseDQ
                                     return (gpx_MiniGPX alg)
           parseGpx_creator alg = do isTag "creator"
                                     isChar '='
                                     result <- parseDQ
                                     return (gpx_creator alg result)
```

If `parseGPX` is called with `gpxALG` and applied on a document from $L(ML(S))$,

```
<gpx version="MiniGPX" creator="JOSM GPX export">
  <metadata>
    <bounds minlat="51.4813624" minlon="7.385542"
            maxlat="51.5019492" maxlon="7.4255655"/>
  </metadata>
  <trk>
    <name>H-Bahn</name>
    <trkseg>
      <trkpt lat="51.4921644" lon="7.4166625">
        <time>2008-03-28T17:02:06Z</time>
      </trkpt>
      <trkpt lat="51.4922462" lon="7.4167966">
        <time>2008-03-31T22:29:55Z</time>
      </trkpt>
      <trkpt lat="51.492271" lon="7.4168691">
        <time>2008-11-27T12:47:33Z</time>
      </trkpt>
    </trkseg>
  </trk>
</gpx>
```

then the result is an abstract syntax tree from $AST_{\Sigma(ML(S))}$.

```
Root_ [Gpx_MiniGPX,Gpx_creator "JOSM GPX export"]
  (GpxType_
    (Just (Metadata_ (MetadataType_
       Nothing
       Nothing
       (Just (Bounds_mt [Bounds_minlat 51.48136,Bounds_minlon 7.385542,
                         Bounds_maxlat 51.50195,Bounds_maxlon 7.4255657])))))
    []
    []
    [Trk_ (TrkType_ (Just (Name_ "H-Bahn"))
      [Trkseg_ (TrksegType_ [
       Trkpt_ [WptType_lat 51.492165,WptType_lon 7.4166627]
```

```
      (WptType_ Nothing (Just (Time_ "2008-03-28T17:02:06Z")) Nothing Nothing),
   Trkpt_ [WptType_lat 51.492245,WptType_lon 7.4167967]
      (WptType_ Nothing (Just (Time_ "2008-03-31T22:29:55Z")) Nothing Nothing),
   Trkpt_ [WptType_lat 51.49227,WptType_lon 7.416869]
      (WptType_ Nothing (Just (Time_ "2008-11-27T12:47:33Z")) Nothing Nothing)])])))
```

If it is called with `svgALG`, the the result is a SVG-term:

```
<svg>
 <g>
  <line fill="none" stroke="#000000" stroke-width="2" x1="51.492165"
   x2="51.492245" y1="7.4166627" y2="7.4167967"/>
  <line fill="none" stroke="#000000" stroke-width="2" x1="51.492245"
   x2="51.49227" y1="7.4167967" y2="7.416869"/>
 </g>
</svg>
```

5 Conclusion

The presented algebraic approach gives a precise and clear model to understand xml-based hyperdocuments and hyperdocument processing. It shows new techniques for developing and implementing browsers and editors, as both can be seen as constructing compilers. Known techniques and results from this area can be adapted. The abstract syntax trees can be used to store xml-based hyperdocuments unambiguously, a great advantage to document trees. For the core topics a Haskell implementation shows the usability of the model.

References

1. Jean Berstel and Luc Boasson. XML Grammars. In *Mathematical Foundations of Computer Science*, pages 182–191, 2000. `http://www-igm.univ-mlv.fr/~berstel/Articles/XMLgrammars.ps`.
2. Manfred Broy, Martin Wirsing, and Peter Pepper. On the Algebraic Definition of Programming Languages. *ACM Trans. Program. Lang. Syst.*, 9(1):54–99, 1987. `http://doi.acm.org/10.1145/9758.10501`.
3. Paul de Bra. Design Issues in Adaptive Web-Site Development. In *Prroceedings of the 2nd Workshop on Adaptive Systems and User Modeling on the WWW*, volume 99-07 of *TUE Computing Science Report*, pages 29–39, 1999. `wwwis.win.tue.nl/~debra/asum99/debra/debra.html`.
4. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation. 3 Edition.* Addison-Wesley, 2007.
5. Wim Martens. *Static Analysis of XML Transformation and Schema Languages.* University of Antwerp, 2006. Ph.D. Thesis, `http://lrb.cs.uni-dortmund.de/~martens/data/phdthesis.pdf`.
6. Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Interet Technol.*, 5(4):660–704, 2005.

7. RELAX NG specification. Technical report, Organization for the Advancement of Structured Information Standards [OASIS], December 2001. http://relaxng.org/spec-20011203.html.
8. Cascading Style Sheets, level 2 revision 1. CSS 2.1 Specification. Technical report, World Wide Web Consortium, October 2006. W3C Working Draft 06 November 2006, www.w3.org/TR/CSS21.
9. XML Schema Definition Language (XSDL) 1.1 Part 1: Structures. Technical report, World Wide Web Consortium, August 2007. W3C Recommendation, http://www.w3.org/TR/2007/WD-xmlschema11-1-20070830.