Specifying Hyperdocuments with Algebraic Methods

Volker Mattick

Technische Universität Dortmund
Fakultät für Informatik
volker.mattick@tu-dortmund.de

Abstract

Algebraic specification methods, well-known in the area of programming languages, are adapted to present a tailored framework for hyperdocuments and hyperdocument systems. In this framework, a hyperdocument is defined via its abstract syntax, which is a variable-free term of a suitable constructor-based signature. Both the representation in a markup language and the graphical presentation on the screen as well as further representations are elements of particular algebraic interpretations of the same signature. This technique allows the application of well-known methods from the field of compiler construction to the development of hyperdocument systems. Ideas for its implementation in the functional language Haskell are roughly drafted. It is shown how XML-based markup languages with schemas and stylesheets can be defined in terms of this framework and how this framework can be extended so that it can deal with partially specified documents, called semi documents. These semi documents can be automatically adapted to the users' needs, which e.g. is helpful to ensure accessibility.

1 Introduction

A *hyperdocument*[1] is a particular electronic document, which one does not have to read in a linear way. Parts of the document, referenced by *anchors*, are connected with other documents or parts of them via *hyperlinks*, which induce a hyper structure on the text. Hyperdocument engineering is a special discipline of software engineering [17]. Instead of general programming languages and assembly languages, markup and layout languages are used; instead of context-free grammars, there are schemas; instead of compilers, browsers transform source documents from a markup language into a layout language. The life cycle for hyperdocuments is much shorter than for most other kinds of software products [14], and in no other field of software engineering have programmers, designers and users become more closely related over the last decade than in hyperdocument engineering. The differences between producers and consumers blur so much that the term *prosumer* [13] is sometimes used.

Software engineering in general [27] and compiler construction in particular ([25], [26]) benefit from the rigorous use of algebraic specification techniques, with e.g. concepts like constructor-based signatures, syntax trees, morphisms, semantic interpretations, specifications and refinement [5], which originate mainly in the area of mathematical logic and universal algebra [10]. A clean separation between the level of modeling and the level of implementation is always recommended. With a precise and clear model, implementation becomes much easier and more reliable.

This work adapts general algebraic methods to define a tailored framework for hyperdocuments and hyperdocument systems and shows how hyperdocument engineering can benefit from this approach.

[1] The term hyperdocument is used here as a generalization of the terms *hypertext document* and *hypermedia document*.

2 Preliminaries

Research and industry have created many different approaches for modeling hyperdocuments, starting with Vannevar Bush's concepts in 1945, continuing with the Dexter model and the tower model in the nineties of the last century, and leading to the document object model and area model of the WWW era. The models can be categorized into two classes, the *information-centered models*, which focus on the structure of a hyperdocument, and the *screen-based models*, which focus on its presentation.

From a technical point of view, hyperdocuments are a particular kind of electronic documents. With the growing success of the WWW, markup languages have practically become the standard for the concrete description of the information-centered aspect of *hyperdocuments*. The screen-based aspects are usually described by *layout languages*. Both are subclasses of the context-free languages. A *hyperdocument system* is a tool that transforms a hyperdocument from one representation into another. Today, the most commonly used hyperdocument systems are browsers. They take a hyperdocument, coded in a markup language, and transform it into a visual presentation of this document.

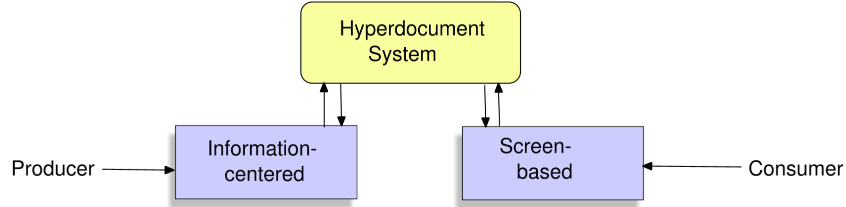

Figure 1: Hyperdocuments

An *(algebraic) specification* $SP = (\Sigma, \mathcal{A})$ consists of a (many-sorted) signature Σ, which captures the abstract syntax, and a set of Σ-algebras $\mathcal{A}$[2], which capture the allowed semantics. The Σ-algebras can either be given explicitly by writing them as tuples of carrier sets and functions, or implicitly by a set of Σ-Horn formulas.

A *(many-sorted) signature* $\Sigma = (S, F, R)$ consists of a set S of *sorts*, an $S^* \times S$-sorted set F of *function symbols*, and an s-sorted set R of relation symbols. If $R = \emptyset$, we write $\Sigma = (S, F)$ for short. Instead of $f \in F_{(e,s)}$, we write $f : e \to s \in F$. If $e = \epsilon$, then f is called a *constant*. Function symbols that only build up data are called *constructors* and are denoted by CO, other function symbols are called *defined functions* and are denoted by DF. If $F = CO$, that means all function symbols are constructors, then the signature is called a *constructor signature*; if $F = CO \cup DF$, it is called a *constructor-based signature*. If X is an s-sorted set of variables, the s-sorted set $T_\Sigma(X)$ of Σ *terms* is defined inductively, so that for all $s \in S$ holds $X_s \subseteq T_\Sigma(X)_s$, and for all $w \in S^*, s \in S, f : w \to s \in \Sigma$ and $t \in T_\Sigma(X)_w$ holds $f(t) \in T_\Sigma(X)_s$. The set of variable-free terms, called Σ *ground terms*, is denoted by T_Σ. The s-sorted set $At_\Sigma(X)$ of Σ *atoms* is defined inductively, so that for all $w \in S^+, r : w \in \Sigma$ and $t \in T_\Sigma(X)_w$ holds $r(t) \in At_\Sigma(X)$. The set of variable-free atoms, called Σ-*ground atoms*, is denoted by At_Σ.

A Σ-*algebra* is a tuple $\mathcal{A} = (A, OP)$, where for all $s \in N$ there exists exactly one non-empty carrier set $A_s \in A$, for all $(co : s) \in CO$ there exists exactly one element $co^A \in A_s$, and for all $(co : e \to s) \in CO$ there exists exactly one function $co^A : A_e \to A_s \in OP$. A function σ, which assigns an algebra to a given signature, is called a *semantic function*. A Σ-algebra $\mathcal{A}_{init}$ is said to be initial in the class of all Σ-algebras if there is for each Σ-algebra $\mathcal{A}$ exactly one

[2] If $\mathcal{A}$ is a singleton, the name of the set of algebras and the name of the algebra are identical.

Σ-*homomorphism init* : $\mathcal{A}_{init} \rightarrow \mathcal{A}$, called *initial homomorphism.* A Σ-homomorphism is an s-sorted mapping $h : \mathcal{A}_1 \longrightarrow \mathcal{A}_2$ between two Σ-algebras $\mathcal{A}_1$ and $\mathcal{A}_2$, so that for all function symbols $f : w \longrightarrow s \in \Sigma$ holds $h_s \circ f^{A_1} = f^{A_2} \circ h_w$ [10]. The initial algebra, if it exists, is unique up to isomorphism. So in Fig. 2, σ_{init} is the semantic function, which assigns the initial Σ-algebra $\mathcal{A}_{init}$ to the signature Σ, and each h_i is an initial morphism. Because σ_{init} is unique if it exists, and, according to the definition, there is exactly one h_i, it holds $\sigma_i = \sigma_{init} \circ h_i$, and so it is possible to define the semantics of a signature via the initial algebra. It can be proven that for each constructor signature, the set of all ground terms is initial in the class of all Σ-algebras [10].

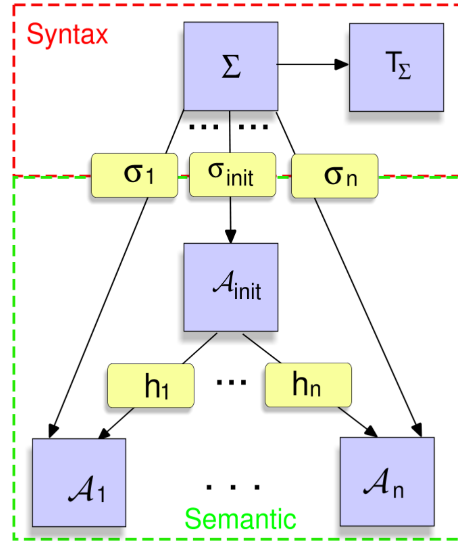

Figure 2: Syntax and semantic

Σ-*Horn formulas* [24] are of the form $f(t_1, \ldots, t_n) \equiv u \Leftarrow G$ or $r(t_1, \ldots, t_n) \Leftarrow G$, where $f \in DF$, $t_1, \ldots, t_n, u$ are constructor terms, G is a conjunction of Σ atoms, called *goal*, and $var(u) \subseteq var(t_1, \ldots, t_n, G)$, where $var : T_\Sigma \rightarrow \mathcal{P}(X)$ denotes the set of variables in a constructor term.

A specification can be built hierarchically. Given a specification $SP' = ((S', F'), (A', OP'))$, a set S of sort symbols disjoint from S', and a set of $(S \cup S')^* \times (S \cup S')$-sorted function symbols F disjoint from F', the specification $SP_{SP'} = (\Sigma_{SP'}, \mathcal{A}_{SP'})$ with $\Sigma_{SP'} = (S \cup S', F \cup F')$ and $\mathcal{A}_{SP'} = (A \cup A', OP \cup OP')$ is a $\Sigma_{SP'}$-algebra called the specification over basic specification SP'.

A *compiler* translates source code, written in a concrete syntax of a source language, usually described by a context-free grammar, into a semantically equivalent artifact of a target language. The first part of a compiler is called *front-end* or *parser*, denoted by $parse_{L(G)}$. It analyses the source code, singles out erroneous words regarding the given language $L(G)$, and assigns to each valid word $w \in L(G)$ a tree-structured representation t, called *(abstract) syntax tree*. The second part is called *back-end* or *evaluator*, denoted by $eval^{L'}$. It assigns to each syntax tree t, possibly enriched with additional attributes, artifacts of the target language L'. For particular classes of context-free grammars, it can be shown that $parse_{L(G)}$ can be automatically generated from the grammar G. In these cases, it suffices to define a compiler by a tuple $(G, eval^{L'})$.

A *context-free grammar* is a tuple $G = (N, T, P, S)$, where N is a finite set of *non-terminal symbols*, T is a finite set of *terminal symbols*, P is a finite set of *production rules*, $S \in N$ is the *start symbol*, and all production rules are of the form $X \longrightarrow w$, where $X \in N$ and $w \in (N \cup T)^*$. It is called a *(linear) tree grammar* if all production rules have the form $X \longrightarrow tw$, where $X \in N$, $t \in T$ and $w \in N^*$, and for all production rules $X \to tw$ and $X' \to tw'$, $X, X' \in N, w, w' \in N^*, t \in T$ holds $X = X'$.

A *grammar over a basic specification* is a tuple $G(SP') = (N \cup S', T \cup A', P, S)$, where N is a finite set of non-terminal symbols disjoint from S', T is a finite set of terminal symbols disjoint from A', P and S are as previously defined, and $SP' = ((S', F'), (A', OP'))$. In a context-free grammar over a basic specification, all rules have the form $X \longrightarrow \epsilon$ or $X \longrightarrow w_1X_1 \ldots w_nX_nw_{n+1}$ with $X, X_i \in N$ and $w_i \in (N \cup S' \cup T \cup A')^*$ for $1 \leq i \leq n$. In a tree grammar over a basic specification, all rules have the form $X \longrightarrow tw$, with $X \in N$, $t \in T \cup A'$ and $w \in (N \cup S')^*$ for $1 \leq i \leq n$.

Context-free grammars are usually described in a metasyntax named Backus-Naur Form (BNF cf. [16]), or in Extended Backus-Naur Form (EBNF cf. [9]). Also, the Augmented Backus-Naur Form (ABNF cf. [1]) is commonly used for internet-technical specifications. Each of the enhanced forms can be converted into the basic BNF.

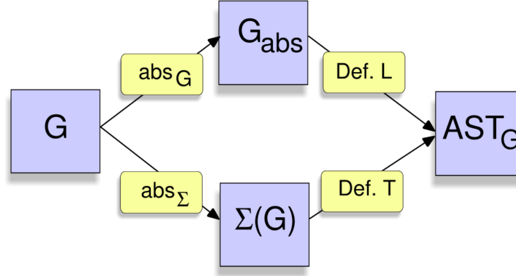

Figure 3: Two ways to define the abstract syntax

The *abstract syntax* for each context-free grammar G, denoted by $AST(G)$, can be constructed in two ways (Fig. 3). The first way, originating from logic and more common in the area of compiler construction and verification, is by a signature $\Sigma(G) = (N, CO)$, where N is now seen as a finite set of *sort symbols*, and CO is a set of *constructors* with $CO = \{c_p : X_1 \times \ldots \times X_n \longrightarrow X \mid \exists : p = (X \longrightarrow w_1X_1w_2 \ldots w_nX_nw_{n+1}) \in P, w_i \in T^*, 1 \leq i \leq n+1, X_j \in N, 1 \leq j \leq n\}$. The set of all ground terms $T_{\Sigma(G)}$ is the abstract syntax of $L(G)$. Because $T_{\Sigma(G)}$ results from $\Sigma(G)$, and $\Sigma(G)$ can be uniquely constructed from G, we use the term G-algebra instead of $\Sigma(G)$-algebra and T_G instead of $T_{\Sigma(G)}$ in the following. The second way, originating from reasoning about formal languages itself and mainly found in the area of complexity and efficiency analysis, is by an *abstract grammar* G_{abs}, which is a (linear) tree grammar. The language $L(G_{abs})$ is the abstract syntax of $L(G)$. It can be shown that T_G is equal to $L(G_{abs})$.

The *language* of a context-free grammar $G = (N, T, P, S)$, denoted by $L(G)$, can be constructed in two ways (Fig. 4). The first is via derivation from the start symbol in G, $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$. The second is via a homomorphism $eval^{Word}$, which maps each syntax tree to a string of T^* in the following way. Each constructor $co_{Xw_0X_1w_1\ldots X_nw_n} \in \Sigma(G)$, that is, the constructor resulting from the production rule $X \longrightarrow w_0X_1w_1 \ldots X_nw_n \in P$ with $w_0, \ldots, w_n \in Z^*$ and $X_1, \ldots, X_n \in N$, is interpreted as $co^{Word(G)}_{Xw_0X_1w_1\ldots X_nw_n} : T^* \times \ldots \times T^* \to T^*$

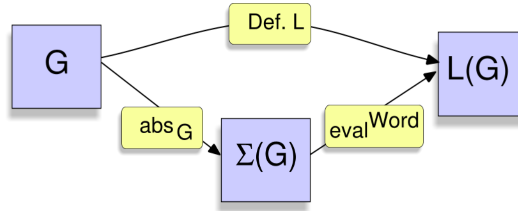

Figure 4: Two ways to define L(G)

with $(v_1, \ldots, v_n) \mapsto w_0 v_1 w_1 \ldots v_n w_n$ for all $v_1, \ldots, v_n \in T^*$. Let $L(T_G) = eval^{Word(G)}(T_G)$, then the resulting tuple $(T^*, L(T_G))$ is a G-algebra, named *word algebra*, and $L(G)$ is the subset of $L(T_G)$ that only contains strings resulting from syntax trees of sort S. Fig. 5 gives an overview. Areas of concrete syntax are highlighted in the figures by a red background, and areas of abstract syntax by a green background[3].

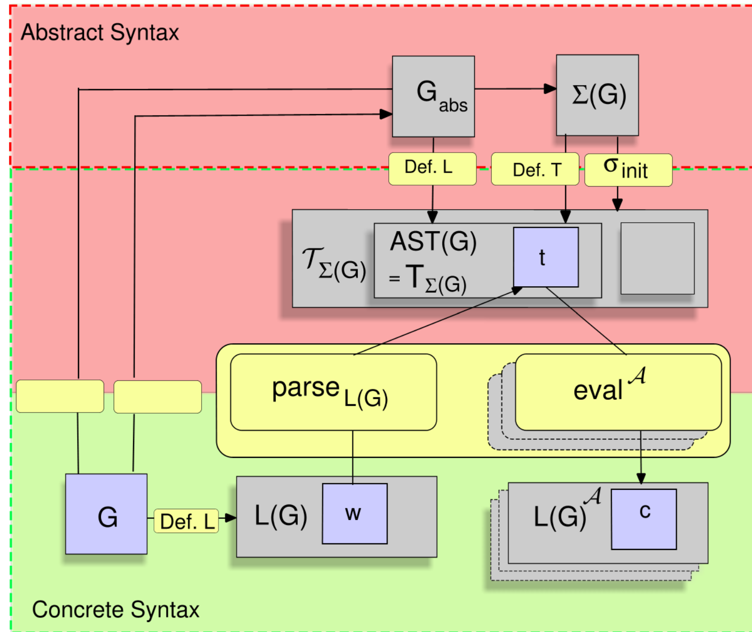

Figure 5: Concrete and abstract syntax

3 Algebraic Framework for Hyperdocument Systems

This new framework for hyperdocuments and hyperdocument systems is restricted to hyperdocuments that can be described with *markup languages*, amalgamates central aspects of existing hyperdocument models, and strips off a lot of model-specific ballast and proprietary notation.

[3]The notation stems from SeeMe [12] and is mainly used in knowledge management. Blue-filled rectangles represent *entities*, and yellow-filled rectangles with rounded corners represent *functions*, which are called *activities* there.

Markup languages can be characterized via particular context-free grammars, so-called XML grammars. Let $G(SP') = ((N \cup \tilde{N}) \cup (AT \cup \tilde{AT}) \cup S', T \cup A', P, S)$ be a grammar over a basic specification. $G(SP')$ is an *XML grammar*[4] if all production rules p are either of the form $X \longrightarrow \langle \mathtt{t}\ \tilde{Y} \rangle (\ \tilde{X}_1 \mid \ldots \mid \tilde{X}_n) \langle /\mathtt{t} \rangle$ for $X \in N$ or $X \rightarrow u = "v"$ with $u \in String$[5], $v \in A'$ for $X \in AT$ or $\tilde{X} \longrightarrow X\tilde{X} \mid \epsilon$ for $X \in \tilde{N} \cup \tilde{AT}$. Converting the rule for $X \in N$ from the shorter EBNF into a basic BNF notation results in n rules, where each rule $p_i, 1 \leq i \leq n$, has the form $X \longrightarrow \langle \mathtt{t}\ \tilde{Y} \rangle\ \tilde{X}_i\ \langle /\mathtt{t} \rangle$. With the previously shown abstraction mechanism, this leads to a constructor $c_{p_i} : \tilde{Y} \times \tilde{X}_i \rightarrow X$. As the terminal symbol t provides us with a unique identifier, we name the constructor t_{X_i} instead of c_{p_i}. Each rule p of the form $Y \longrightarrow u = "v"$ with $u \in String, v \in S'$ leads to a constructor $c_p : String \times S' \rightarrow Y$, which we name av_Y instead of c_p. Therefore, an abstract syntax and a validating parser, which reads a correct document in a particular markup language into its abstract representation, can be generated automatically, and a hyperdocument can be defined via a variable-free term of a constructor-based signature. This is the first part of the hyperdocument system depicted on the left-hand side of Fig. 6.

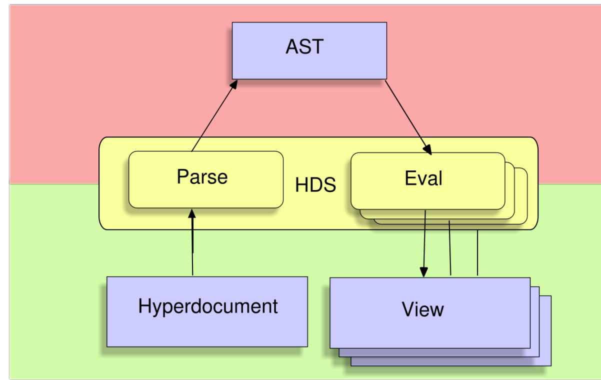

Figure 6: Algebraic framework for hyperdocument systems

We now have different possibilities to interpret or evaluate the syntax tree into a *view*, depicted on the right-hand side of Fig. 6. A view in this context is every non-textual representation, most often a graphically rendered output on a screen. A language that describes views is called *layout language*, and the graphical representation is created by a *rendering engine*[6]. We can identify at least three typical kinds of views. First, the *document view* on the screen, well known from each browser. Second, the *document tree view* as defined by the W3C and often used for theoretical examination of hyper structures (cf. e.g. [15]). A *document tree* is an unranked, sibling-ordered, labeled tree that has two different kinds of leaves, called *content nodes* and *attribute nodes*. And third, the *site map view* that focuses on the links and relations between documents or parts of documents. Additionally, output for special devices, such as screen readers or Braille terminals, can be simply seen as another algebraic interpretation. Therefore, browsers can also be classified by different algebras.

A real-world hyperdocument usually does not consist of a single word of a markup language, but is a bundle consisting of a *schema*, written in a *schema language* S, a *document description*, written in a *markup language* $ML(S)$ that depends on the schema, and a *stylesheet*, written in

[4]This is an attributed extension of the XML grammars introduced by [4] or the similar balanced grammars of [6].

[5]*String* is assumed to be a sort in each basic specification.

[6]How rendering engines work in detail is beyond the scope of this work.

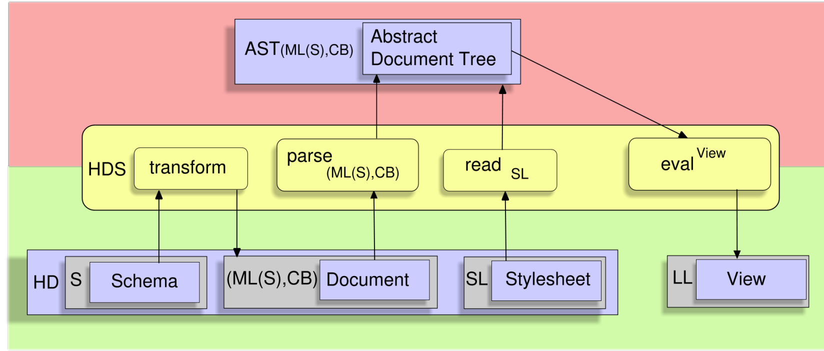

Figure 7: Real-world hyperdocuments

a *stylesheet language* SL, depicted by the entity at bottom left in Fig. 7.

The most popular schema languages are the document-based XML Schema (c.f. [30]), the Document Type Definition (DTD), and the pattern-based RelaxNG (c.f. [23]), where each DTD language definition can easily be transformed into an XML Schema language definition.

Document-based means that a schema S is in principle a tree grammar, coded in a meta syntax similar to EBNF notation. This tree grammar does not directly describe the markup language we want to define, but is the abstract syntax for this markup language. The markup language itself is a particular interpretation, named the $ML(S)$*-algebra*, of the abstract syntax defined by the schema. Moreover, the schema defines additional attributes and some constraints, e.g. for the number of times an element can occur. For the simple special cases that it can occur either zero or one times, exactly once or infinitely often, we can find a context-free representation, but characterizing a range in between is not possible, at least not with reasonable effort. Also, the fact that an attribute is required and that it has a default value cannot be captured by a context-free grammar. So we need an additional constraint base that does not influence the syntax of the markup language but influences the parser. The parser must reject badly-formed documents, but also documents that do not fulfill the additional constraints given by the schema.

Pattern-based means that the set of document trees, not that of syntax trees, is characterized via allowed patterns of the paths of the document trees. In the framework presented, pattern-based approaches are harder to handle. The reason is that there is no canonical way to build an abstract syntax for the markup language for the algebra of documents trees. So usually one has to find a morphism between the document tree algebra and the markup language algebra. Each document-based schema can be translated into a pattern-based one ([18]), and a subset of RelaxNG, called BonXai, [18], can be translated into a document-based form.

As a running example, we use MiniGPX, a shortened version of the GPS Exchange Format for exchanging geodata[7]. Though this is not a typical markup language for hyperdocuments, it has a comparatively short and understandable schema definition[8], and nearly all features can be demonstrated with this language.

All elements and types of the schema are represented in the abstract syntax as constructors. Attributes that belong to complex types are represented as constructors of the corresponding attribute sort. Constraints, such as `minOccurs` or `maxOccurs`, have the default value 1. This

[7] http://www.topografix.com/gpx.asp
[8] http://www.topografix.com/gpx/1/1/gpx.xsd

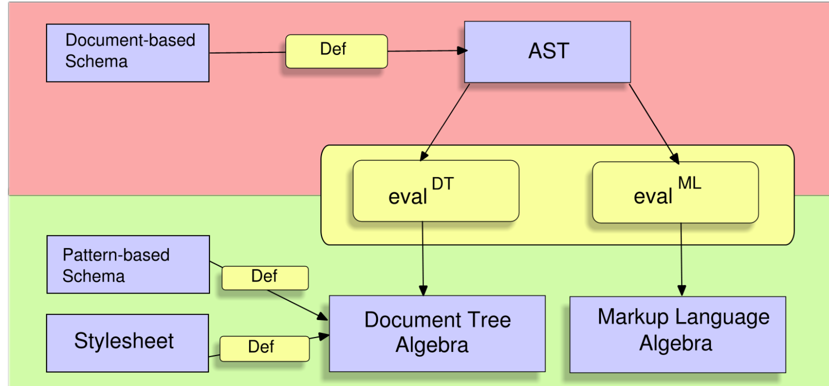

Figure 8: Schema and stylesheet languages

means that e.g. `metadata` should appear either exactly once or not at all. Basic types, such as `xsd:string`, are assumed to have a predefined interpretation.

```
<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="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>
```

This complex type from the schema S_{GPX} given above can be represented by a signature $\Sigma(S_{GPX}) = (N, OP)$ with

$$
\begin{aligned}
OP = \{\; & \text{root_} :: \texttt{gpxType_av} \times \texttt{gpxType} \rightarrow \texttt{gpx} \\
& \text{gpxType_} :: \texttt{metadataAlt} \times \texttt{wptList} \times \texttt{trkList} \rightarrow \texttt{gpxType} \\
& \text{gpx_MiniGPX} :: \rightarrow \texttt{gpxType_av} \\
& \text{gpx_creator} :: \texttt{xsd:string} \rightarrow \texttt{gpxType_av} \\
& _ : _ :: \texttt{gpxType_av} \times \texttt{gpxType_avList} \rightarrow \texttt{gpxType_avList} \\
& [] :: \rightarrow \texttt{gpxType_avList} \ldots \}
\end{aligned}
$$

To show that the proposed framework works reasonably well, we have prototypically implemented some core features in the functional programming language Haskell[9] [11]. Both markup languages and functional programming languages are declarative, but programming languages are better suited for structuring problems and building abstractions than markup languages. The goal of structural markup, where documents are specified in terms of their logical features rather than of particular rendering procedures, is similar to the ideals of functional programming, where computations are specified in mathematical rather than machine-oriented terms. Hyperdocuments can be seen as trees, and functional languages usually offer extensive facilities

[9] `http://www.haskell.org`

for representing and manipulating trees. Moreover, if a typed functional language is used, the type system can provide additional structure and integrity.

To make the gap between the concept and the implementation as large as necessary and as small as possible, a signature is implemented as a polymorph data type[10]. Let $\Sigma = (S, OP)$ be a signature with $S = \{s_1, \ldots, s_n\}$ and $OP = \{f_{11} : w_{11} \to s_1, \ldots, f_{1n_1} : w_{1n_1} \to s_1, \ldots, f_{k1} : w_{k1} \to s_k, \ldots, f_{kn_k} : w_{1n_k} \to s_k\}$. Each sort is now implemented as a type variable, and each function symbol is implemented as an attribute. Let $(f : \epsilon \to s)' =_{def}$ `f::S` and $(f : s_1 \ldots s_n \to s)' =_{def}$ `f::s_1 -> ... -> s_n -> s`. We get the following data type:

`data SIG s_1 ... s_k = SIG` $\{(f_{11} :: w_{11} \to s_1)', \ldots, (f_{1n_1} :: w_{1n_1} \to s_1)', \ldots,$
$(f_{k1} :: w_{k1} \to s_k)', \ldots, (f_{kn_k} :: w_{kn_k} \to s_k)' \}$

The signature $\Sigma(S_{GPX})$ is implemented by the following data type:

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

A framework is called *schema-aware* if the data binding and the processing, manipulating or generation of documents depend on a given schema. E.g. the special purpose functional languages XSLT[11] and FXT [3], implemented in SML [22], transform arbitrary document trees independently of the schema, and so they need no implementation of XML types. The *Web Authoring System Haskell (WASH)* [28] represents the XHTML schema as a Haskell data type. *HaXml* [21] gives a translation of DTDs to Haskell types [31], and *UUXML* [2] gives a type-preserving XML Schema/Haskell data binding. As far as we know, all schema-aware approaches for hyperdocuments embed XML values by finding a suitable translation between XML types, described by schemas and types of the programming language. Our implementation first translates schemas into constructor signatures and then uses a translation between constructor signatures and Haskell types.

Each algebra is then implemented by simply instantiating the type variables with the concrete types of the carrier sets, and each attribute is instantiated by the corresponding function of the given algebra. So the algebra that interprets a document as an abstract syntax tree of $AST_{ML(S)}$ looks like this in the Haskell notation:

```
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] [Trk]
data GpxType_av = Gpx_MiniGPX | Gpx_creator XSD_string
```

A second algebra, which interprets a document as an object of the Scalable Vector Graphics (SVG)[12], can look like this:

[10] This implementation technique is proposed by [26] in the context of compiler construction and is still under development.

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

[12] `http://www.w3.org/Graphics/SVG/`

```
svgALG = GpxSIG gpx_ gpxType_ gpxType_av_ ... =
        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 trk = "<g>" ++ (conc wpt) ++
                                          (conc trk) ++ "</g>"
              gpx_MiniGPX = "MiniGPX"
              gpx_creator c = c
```

The *stylesheet* is the formatter for a hyperdocument. It defines path expressions to locate a particular place in the document tree, and actions that modify the selected part of the tree. In practice, a stylesheet is usually described by CSS (c.f. [29]) or XSLT[13]. CSS stylesheets can only modify values of attributes, but not the document tree itself. XSLT is a tree-transformation language that can also change the structure of the tree. Because the hyperdocument is represented by a syntax tree $doc \in AST_{ML(S)}$, the stylesheet must be interpreted by paths on doc, and the action by tree transformations at the located place. After executing the actions, we get a modified syntax tree, which is then interpreted by a layout language, depicted by the entity at bottom right in Fig. 7.

The parser that can be generated from the grammar can be realized in Haskell in a monadic style. It not only parses the document itself, but is parameterized, so that it can be used to compile a source document directly into a target interpretation.

```
parseGpx :: GpxSIG gpx gpxType gpxType_av metadata metadataType wpt wptType
                   wptType_av trk trkType name time bounds boundsType_av ele
                   sym number 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 (parseTrk alg)
                      return (gpxType_ alg result1 result2 result3)

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

[13] http://www.w3.org/TR/xslt

```
                                     isChar '='
                                     result <- parseDQ
                                     return (gpx_creator alg result)
```

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

```
<gpx version="MiniGPX" creator="JOSM GPX export">
  <metadata>
    <bounds minlat="51.4813" minlon="7.3855"
            maxlat="51.5019" maxlon="7.4255"/>
  </metadata>
  <trk>
    <name>H-Bahn</name>
    <trkseg>
      <trkpt lat="51.4921" lon="7.4166">
        <time>2008-03-28T17:02:06Z</time>
      </trkpt>
      <trkpt lat="51.4922" lon="7.4167">
        <time>2008-03-31T22:29:55Z</time>
      </trkpt>
      <trkpt lat="51.4922" lon="7.4168">
        <time>2008-11-27T12:47:33Z</time>
      </trkpt>
    </trkseg>
  </trk>
</gpx>
```

then the result is a syntax tree from $AST_{ML(S)}$.

```
Root_ [Gpx_MiniGPX,Gpx_creator "JOSM GPX export"]
  (GpxType_
    (Just (Metadata_ (MetadataType_
       Nothing
       Nothing
       (Just (Bounds_mt [Bounds_minlat 51.4813,Bounds_minlon 7.3855,
                         Bounds_maxlat 51.5019,Bounds_maxlon 7.4255])))))
    []
    []
    [Trk_ (TrkType_ (Just (Name_ "H-Bahn"))
      [Trkseg_ (TrksegType_ [
       Trkpt_ [WptType_lat 51.4921,WptType_lon 7.4166]
          (WptType_ Nothing (Just (Time_ "2008-03-28T17:02:06Z"))
                    Nothing Nothing),
       Trkpt_ [WptType_lat 51.4922,WptType_lon 7.4167]
          (WptType_ Nothing (Just (Time_ "2008-03-31T22:29:55Z"))
                    Nothing Nothing),
       Trkpt_ [WptType_lat 51.4922,WptType_lon 7.4168]
          (WptType_ Nothing (Just (Time_ "2008-11-27T12:47:33Z"))
                    Nothing Nothing)])])]))
```

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

```
<svg>
 <g>
  <line fill="none" stroke="#000000" stroke-width="2" x1="51.4921"
   x2="51.4922" y1="7.4166" y2="7.4167"/>
  <line fill="none" stroke="#000000" stroke-width="2" x1="51.4922"
   x2="51.4922" y1="7.4167" y2="7.4168"/>
 </g>
</svg>
```

4 Hyperdocument Engineering

Hyperdocument engineering can benefit from this framework in multiple ways, e.g. in the areas of *adaptable hyperdocuments*, cf. [8], and universal design. There, it must be possible to design documents that can be to a great extent tailored to the reader's needs and wishes. So it would be a great help for the developer to have a mechanism for specifying a set of documents for which the specified aspects are fixed, and the rest is open to a user's adaption. This can be done by using *syntax trees with variables* as an abstract representation of so-called *semi documents* [19]. Given a markup language $ML(S)$ and assuming that the interpretation of $AST_{ML(S)}$ is known, a syntax tree with variables describes a subset of $AST_{ML(S)}$, whereas a syntax tree without variables describes a single element of it. Of course, in the end, the hyperdocument system must deliver only a single element and not a set of elements to the rendering engine. So, in addition, we need a user profile that characterizes the documents the user allows. In contrast to developers, users more often think in terms of views. To describe profiles, we use a particular profile description language P that can specify views in the way schemas describe documents (cf. the bottom right part of Fig. 9). Now the *adaptable hyperdocument system* must search for an assignment of the variables with which the interpretation of the corresponding variable-free syntax tree fits the constraints of the user profile. This can also be a first step towards a hyperdocument description language that specifies documents in a screen-oriented setting, which is useful because producers and consumers are increasingly the same persons.

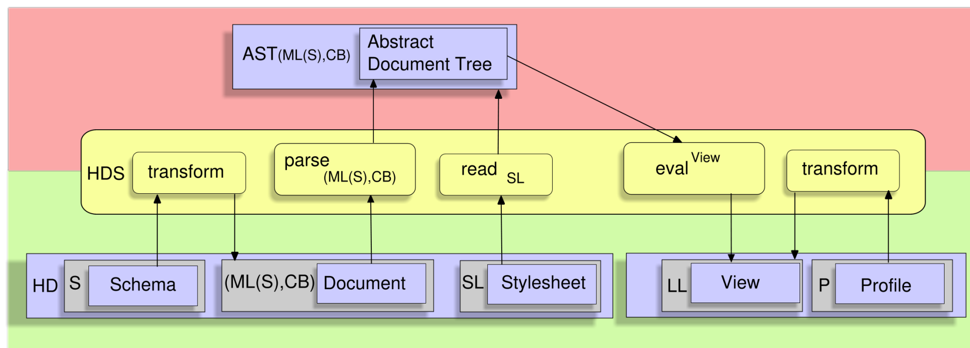

Figure 9: Modifications of the framework

Instead of obtaining the necessary information from a user profile, it is also possible to collect design goals in a requirements engineering phase. We then enrich the signature $\Sigma = (S, F)$ with

suitable relations that we need to express the requirements. With $\Sigma = (F, S, R)$, it is possible to capture the requirements by axioms AX, and the specification $SP = (\Sigma, AX)$ specifies a set of allowed algebras. With this technique, it is possible to semi-automatically generate documents out of semi documents.

Because the markup language is an algebra of the abstract syntax, it is possible to use syntax trees or a language that specifies syntax trees as an executable specification language for hyperdocuments (cf. [20]). As in programming languages, it is easier to test constraints and requirements on the abstract level than on the level of the concrete representation. The concrete hyperdocument is then just an interpretation with the $ML(S)$ algebra.

5 Conclusion and Future Work

The algebraic approach presented gives a precise and clear model for understanding XML-based hyperdocuments and hyperdocument processing. It shows new techniques for hyperdocument systems, especially browsers, resulting from research in compiler construction. Known techniques and results from this area can be adapted. The abstract syntax trees are used to store XML-based hyperdocuments unambiguously, a great advantage over document trees. It is demonstrated how real-world hyperdocuments, described by a schema, a document description, and a stylesheet, can be handled by our approach. Benefits for hyperdocument engineering are only sketched, but not elaborated to their full potential. For example, it has not yet been examined how these techniques can improve hyperdocument editors, where the source documents are specified via a layout language, and the document structure in a markup language is the target. The functionality of the core elements of our framework is proven by prototypical Haskell implementations, and it shows the usability of the model, but it is far from being a software system that can be used in real-world hyperdocument engineering. To exhaust the full potential of the framework, the term-rewriting language Maude [7] could be an interesting alternative to Haskell.

References

[1] Augmented BNF for Syntax Specifications: ABNF. RFC 5234, `ftp://ftp.rfc-editor.org/in-notes/std/std68.txt`, January 2008.

[2] Frank Atanassow and Johan Jeuring. Customizing an XML-Haskell data binding with type isomorphism inference in Generic Haskell. *Sci. Comput. Program.*, 65(2):72–107, 2007.

[3] Alexandru Berlea and Helmut Seidl. fxt - a transformation language for xml documents. *Journal of Computing and Information Technology*, 10:2002, 2001.

[4] 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/2000XMLMFCS.pdf`.

[5] 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`.

[6] Anne Brüggemann-Klein and Derick Wood. Balanced Context-Free Grammars, Hedge Grammar and Pushdown Caterpillar Automata. In *Proceedings of Extreme Markup Languages*, 2004. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.8957&rep=rep1&type=pdf`.

[7] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187 – 243, 2002.

[8] 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`.

[9] Information technology - Syntactic metalanguage - Extended BNF, December 1996. ISO/IEC 14977.

[10] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, 1985.

[11] Haskell 98: A Non-strict, Purely Functional Language. `www.haskell.org/definition/haskell98-report.pdf` (August 8 2000), February 1999.

[12] Thomas Herrmann, Marcel Hoffmann, and Kai-Uwe Loser. Modellieren mit SeeMe. Alternativen wider die Trockenlegung feuchter Informationslandschaften. In Desel, Pohl, and Schürr, editors, *Modellierung '99. Proceedings of Modellierung 1999*, pages 59–74. Teubner-Verlag, 1999.

[13] Isa Jahnke and Volker Mattick. Shift from Teaching to Learning with Web 2.0. *Proceedings of the ACM-IFIP IEEIII 2008, Informatics Education Europe III Conference*, pages 105–114, November 2008.

[14] Isa Jahnke, Volker Mattick, and Thomas Herrmann. inpud: Software Entwicklung und Community-Kultivierung: ein integrativer Ansatz. *i-com - Themenheft Communities, Zeitschrift für interaktive und kooperative Medien*, pages 14–21, February 2005.

[15] Nils Klarlund, Thomas Schwentick, and Dan Suciu. Xml: model, schemas, types, logics, and queries. In *In Logics for Emerging Applications of Databases*, pages 1–41. Springer, 2003.

[16] Donald E. Knuth. Backus normal form vs. Backus Naur form. *Commun. ACM*, 7(12):735–736, 1964.

[17] David Lowe and Wendy Hall. *Hypermedia & the Web. An engineering approach.* Wiley, 1999.

[18] Wim Martens and Volker Mattick (PG 530). Pattern Based Schema Languages, Final report. Technical report, Fakultät für Informatik, TU Dortmund, October 2009.

[19] Volker Mattick. Design for All as a Challenge for Hypermedia Engineering. *Journal of Universal Computer Science*, 8(10):881–891, 2002. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.6282`.

[20] Volker Mattick. HDDL: Syntax und Semantik von Hyperdokumenten algebraisch exakt und verständlich modellieren. pages 147–152. Universität zu Lübeck, October 2007. `http://www.isp.uni-luebeck.de/kps07/files/papers/mattick.pdf`.

[21] David Mertz. Transcending the limits of DOM, sax, and xslt: The haxml functional programming model for xml. *IBM developerWorks (XML Matters column)*, October 2001. `http://www-106.ibm.com/developerworks/library/x-matters14.html`.

[22] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised.* The MIT Press, May 1997. `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0262631814`.

[23] RELAX NG specification. Technical report, Organization for the Advancement of Structured Information Standards [OASIS], December 2001. `http://relaxng.org/spec-20011203.html`.

[24] Peter Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.

[25] Peter Padawitz. Algebraic compilers and their implementation in Haskell. Technical report, IFIP WG1.3 meeting, Sierra Nevada, January 2008. `http://fldit-www.cs.uni-dortmund.de/~peter/AlgComp.pdf`.

[26] Peter Padawitz. Folienskript übersetzerbau. `http://fldit-www.cs.uni-dortmund.de/~peter/`

`CbauFolien.pdf`, February 2010.

[27] Donald Sannella and Andrzej Tarlecki. Essential Concepts of Algebraic Specification and Program Development. *Formal Asp. Comput.*, 9(3):229–269, 1997. `http://dx.doi.org/10.1007/BF01211084`.

[28] Peter Thiemann. A typed representation for html and xml documents in Haskell. *J. Funct. Program.*, 12(5):435–468, 2002.

[29] 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, `http://www.w3.org/TR/CSS21`.

[30] 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`.

[31] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *ICFP '99*, pages 148–159. ACM Press, 1999.