Document Engineering with Extensible Abstract Document Structures*

Anne Brüggemann-Klein
Institut für Informatik, Technische Universität München
Arcisstr. 21, 80290 München, Germany
E-mail: `brueggem@informatik.tu-muenchen.de`
WWW: `http://www11.informatik.tu-muenchen.de`

Derick Wood
Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong SAR
E-mail: `dwood@cs.ust.hk`
WWW: `http://www.cs.ust.hk/~dwood`

April 29, 2000

Abstract

Document engineering is the systematic development of document presentations, representations and document tools. We particularly address document engineering for the Extensible Markup Language (XML); both its Document Type Definitions (DTDs) and its DTD-conforming documents.

Our thesis is that document engineering should be firmly based on explicit, formal document models. This precept is well accepted and practiced in, for example, the database community but it is, apparently, less accepted in the XML community. We want to change this position.

As a first step in the demonstration of the value of explicit, formal document models we derive and discuss an explicit, formal model for XML DTDs and their

*The work of both authors was supported partially by a joint DAAD-HK grant. In addition, the work of the second author was supported under a grant from the Research Grants Council of Hong Kong.

DTD-conforming documents. This new model captures what can be considered to be the essence of an XML DTD and its DTD-conforming documents.

1 Introduction

Document engineering (that is, the systematic development of document presentations, representations and tools) should be based on document models not on document representations. This precept, old but still valid, was stated by Meyrowitz and van Dam [3], and by Furuta, Quint, and André [2] among others. The precept is well accepted and practiced in, for example, the database community but it is, apparently, less accepted in the XML community.

We propose an explicit, formal document model that is sufficiently rich to capture the essential aspects of XML DTDs and their DTD-conforming documents yet is also "small" in that it omits as much as possible. Since one can argue how much needs to be kept and how much must be kept, we demonstrate the model's viability by sketching how it can be used in the development of XML applications.

We are concerned about **structured documents**[1], a central concept in the world of information since structured documents enable:

1. Document applications that are central to information-processing needs such as:
   (a) Query processing.
   (b) Information extraction.
   (c) Document transformation.
   (d) Customized document collections.
   (e) Views into document databases.
2. Cross-media publishing and platform independence.

We call documents that follow a formal model "abstract" and we require document structures to be extensible so that they can be customized for specific applications. We suggest that the following five steps constitute a document-engineering methodology based on extensible abstract document structures:

1. Develop a document model.
2. Develop representations for the documents that follow the model.
3. State requirements of functionality and performance for operations on documents in the model, based on document applications.
4. Implement the model, which involves:

[1] We use the phrases "structured documents" and "document structures" synonymously.

   (a) The specification and implementation of document classes (in the sense of object orientation, not document-type definitions) for various groups of requirements of functionality and performance. The class specification should be based on the model, not on document representations.
   (b) The provision of interfaces to the document representations (parsing and unparsing, import and export).
5. Validate the implementation by developing document applications that are based on it.

Current approaches to document engineering are based on formats or representations rather than on models; the models are usually implicit and occasionally explicit, but in both cases they are informal. *Our thesis is that it is paramount to make a document model both explicit and formal.*

Some benefits of an explicit and formal document model are:

1. We can reason about the model and pose such questions as: What is needed in a document model for a particular type of application? How do various models compare? What are the benefits and drawbacks of one model over another?
2. It supplements document-language specification for the Web and also serves as the kernel of a concept definition from which a language representation can be derived as a secondary and perhaps even as an automatic process. This liberates us from discussions about syntax—conceptual issues can be discussed directly. It enables us to argue about concepts and ideas rather than about syntax.
3. It may and could be applied in current standardization efforts, such as XML Schema, to argue the pros and cons of the current, DTD-based model for XML documents and to communicate and reason about alternatives (such as tree automata and schemas).

The remainder of this paper is organized as follows:

In Section 2, we give a brief overview of the Extensible Markup Language (XML). In Section 3, we define the **GCSD model,** for **grammar-constrained structured documents,** that captures the essence of XML DTDs and their DTD-conforming documents. Basically, we re-engineer XML since, in contrast to our document-engineering methodology, we build the document model from document representations rather than the other way around. In Section 4.1, we sketch how the GCSD model can be used in the development of XML applications; in Section 4.2, we discuss related work; and we conclude with some suggested future work in Section 4.3.

2 The Extensible Markup Language (XML)

The Extensible Markup Language (XML) describes a class of data objects called **XML documents**. The XML specification [1] defines the syntax of XML documents and partially describes the behavior of computer programs that process them.

XML documents are made up of what are called **entities** (essentially macros) that denote either parsed or unparsed data. Parsed data consists of characters, some of which form **character data**, and some of which form **markup**. Markup encodes a description of the document's organization into entities (physical structure, storage layout) and its so-called **logical structure**.

The XML specification formally defines the syntax of XML documents using an extended context-free grammar and additional natural-language constraints.

The XML specification assumes that a software module called an **XML processor** is used to read XML documents and to provide access to their data. It further assumes that an XML processor is doing its work on behalf of another module, called the **application.** The XML specification describes the required behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.

The problems of the current XML approach are:

1. The document model that underlies the XML specification is only implicit, it is not explicit.
2. The XML specification focuses on syntax. An XML document is a specific syntactic representation of a structured document. Meta-syntactic sugar, such as the sequence of declarations and overwrite rules, and accidental surface properties such as character references and white space, muddy the clear picture of the "real" or "pure" structured documents underlying the syntactic surface.
3. An XML document simultaneously represents the logical and the physical structure. Database design, on the other hand, separates the conceptual (or logical) and physical layers.

3 Re-engineering XML

We instantiate our methodology of document engineering by re-engineering XML. We construct an explicit model of GCSDs that underlies the logical part of XML. In the GCSD model, we describe GCSDs as mathematical objects (an abstract data type). We model both document instances and document grammars. We construct the model from only elementary mathematical objects and operations such as strings, sets, set operations and functions.

Our goal in creating the model is to capture the essential ideas that we believe underlie the logical part of XML in a concise, unambiguous and understandable way without being hindered by XML syntax and aspects of physical structure. Our reasons are two-fold. First, we wish to come to terms with what an XML document "is", so that we can more easily communicate the underlying concepts when teaching XML or when exploring its usefulness in document-engineering projects. Second, since XML is the obvious choice of a document representation for Web projects, which are becoming much more of an engineering challenge, we need to re-engineer XML so that we can apply the model-first strategy and our document-engineering methodology.

We use XML to syntactically represent documents that conform to the GCSD model. In fact, we specify a subset of XML, the logical part of XML (XML-L), that retains only the syntactic parts of XML that are needed to syntactically represent GCSD-modeled documents. XML-L consists of only the logical parts of XML and it omits everything that is physical (in particular, XML entities).

Although we omit both the use and definition of entities in the GCSD model, we agree that they have an important part to play in practice. But, from the modeling viewpoint, we assume that all entities are already resolved.

Our aim is that the GCSD model should capture the essence of structured documents. A GCSD-modeled document is an abstract concept; its XML-L representation is one way to make it tangible. We present the GCSD model in the following subsections.

3.1 Preamble

The GCSD model uses a finite set *Character* of characters that we use for the content of a document and four pairwise disjoint infinite[2] sets *DocSort*, *AttSort*, *DocID* and *AttValue* of names that we use as a source of names for documents, attributes, document IDs and attribute values. From these five sets we build other sets using the standard set operations given in Table 1. We summarize the objects in the GCSD model and their definitions in Table 2.

Recall that XML defines both grammars and their associated structured documents. Each grammar defines or specifies the structure of its documents, the content of its documents and subdocuments, and the documents' attributes and their possible values. This simple, intuitive summary of XML is reflected in the GCSD model.

[2] Each GCSD in the model uses finite subsets of *DocSort*, *AttSort*, *DocID* and *AttValue*.

Table 1 Set operations and their notation.

Operation	Notation
Cross product	$\square \times \square$
Free monoid; set of sequences	$\square^*$
Set of functions	$\mathcal{F}(\square \longrightarrow \square)$
Set of partially-defined functions with finite domain	$\mathcal{F}_{\text{fin}}(\square \longrightarrow \square)$
Disjoint union	$\square \mathbin{\dot\cup} \square$
Set of subsets; power set	$\mathcal{P}(\square)$
Set of finite subsets	$\mathcal{P}_{\text{fin}}(\square)$
Finite set	$\{\square, \ldots, \square\}$
Set of regular languages over a given alphabet	$\mathrm{Reg}(\square)$
Set of restricted-regular languages over given character and symbol sets	$\mathrm{RReg}(\square, \square)$

Table 2 The objects in the GCSD model

(1)	$\mathit{GCSDocument} \subseteq \mathit{DocInstance} \times \mathit{DocGrammar}$.
(2)	$\mathit{DocInstance} = \mathit{DocSort} \times \mathit{Attributes} \times \mathit{Content}$.
(3)	$\mathit{Attributes} = \mathcal{F}_{\text{fin}}(\mathit{AttSort} \longrightarrow \mathit{AttValue} \mathbin{\dot\cup} \mathit{Character}^* \mathbin{\dot\cup} \mathit{DocID})$.
(4)	$\mathit{Content} = (\mathit{DocInstance} \mathbin{\dot\cup} \mathit{Character})^*$.
(5)	$\mathit{DocGrammar} = \mathit{ContentRules} \times \mathit{AttRules}$.
(6)	$\mathit{ContentRules} = \mathcal{F}_{\text{fin}}(\mathit{DocSort} \longrightarrow \mathit{DocDomain})$.
(7)	$\mathit{DocDomain} = \mathrm{Reg}(\mathit{DocSort}) \mathbin{\dot\cup} \mathrm{RReg}(\mathit{Character}, \mathit{DocSort})$.
(8)	$\mathit{AttRules} = \mathcal{F}_{\text{fin}}(\mathit{DocSort} \times \mathit{AttSort} \longrightarrow \mathit{AttDomain})$.
(9)	$\mathit{AttDomain} = \mathcal{P}_{\text{fin}}(\mathit{AttValue}) \mathbin{\dot\cup} \{\texttt{String}, \texttt{ID}, \texttt{IDREF}\}$.

3.2 The GCSD model

3.2.1 Mixed content

XML has the concept of mixed content that we capture as follows. Given a finite set C of characters and a finite set S of symbols that is disjoint from C, a restricted-regular set over C and S is a set of strings $(C \cup F)^*$, where F is a subset of S. Hence, a restricted-regular set over C and S is a set of strings whose symbols are taken from C and from a subset F of S. We denote the restricted-regular sets over C and S by $\mathrm{RReg}(C, S)$.

3.2.2 Grammars and structured documents

We define a set *DocInstance* of structured documents and a set *DocGrammar* of context-free grammars that constrain structured documents. The set *GCSDocument* consists of all pairs (di, dg) in $\mathit{DocInstance} \times \mathit{DocGrammar}$ such that the structured document di conforms to the grammar dg. Clearly,

$$\mathit{GCSDocument} \subseteq \mathit{DocumentInstance} \times \mathit{DocGrammar}. \tag{1}$$

A structured document in *DocInstance* has a document-type name from *DocSort*, a number of attributes and some content; hence

$$\mathit{DocInstance} = \mathit{DocSort} \times \mathit{Attributes} \times \mathit{Content}. \tag{2}$$

3.2.3 Attributes and content

The attribute specification for a structured document assigns values to the attribute names that are associated with the document. Hence,

$$\mathit{Attributes} = \mathcal{F}_{\mathrm{fin}}(\mathit{AttSort} \longrightarrow \mathit{AttValue} \,\dot{\cup}\, \mathit{Character}^* \,\dot{\cup}\, \mathit{DocID}). \tag{3}$$

Now, the content of a structured document is a sequence of characters and, recursively, lower-level structured documents:

$$\mathit{Content} = (\mathit{DocInstance} \,\dot{\cup}\, \mathit{Character})^*. \tag{4}$$

The recursive definition of document instances terminates, at its lowest levels, with content that is consists of only characters.

3.2.4 Document grammars

A document grammar has two kinds of rules: Ones that constrain the content and ones that constrain the attributes of structured documents:

$$\mathit{DocGrammar} = \mathit{ContentRules} \times \mathit{AttRules}. \tag{5}$$

This equation implies that the grammar has a rule for finitely many document names in *DocSort*.

3.2.5 Content

The rules in *ContentRule* that constrain the content of a structured document, assign document domains to document names. Document domains, in turn, are either regular or restricted-regular sets over the alphabet *DocSort*. Hence,

$$\mathit{ContentRules} = \mathcal{F}_{\text{fin}}(\mathit{DocSort} \longrightarrow \mathit{DocDomain}), \tag{6}$$

where

$$\mathit{DocDomain} = \mathrm{Reg}(\mathit{DocSort}) \,\dot{\cup}\, \mathrm{RReg}(\mathit{Character}, \mathit{DocSort}). \tag{7}$$

3.2.6 Attributes

Analogously, rules in *AttRule*, which constrain the attributes of a structured document, assign attribute domains to some combinations of document names and attribute names. An attribute domain is either a finite subset of *AttValue* or one of the symbols `STRING`, `ID` or `IDREF`. Hence,

$$\mathit{AttRules} = \mathcal{F}_{\text{fin}}(\mathit{DocSort} \times \mathit{AttSort} \longrightarrow \mathit{AttDomain}), \tag{8}$$

where

$$\mathit{AttDomain} = \mathcal{P}_{\text{fin}}(\mathit{AttValue}) \,\dot{\cup}\, \{\mathtt{String}, \mathtt{ID}, \mathtt{IDREF}\}. \tag{9}$$

Values in the attribute domain `String` are any character strings in $\mathit{Character}^*$; values in the attribute domain `ID` and `IDREF` are any document IDs in *DocID*.

3.3 Access operators

We next define a number of operators that access parts of GCSDs, structured documents and grammars. We adopt the following two notational conventions:

1. For an operator X that operates on a set Y, the application of X to an element y in Y is denoted by $y.X$.
2. If a set X is defined as the cross product of sets $Y_1 \times \cdots \times Y_n$, then each Y_i, $1 \leq i \leq n$, denotes an operator on X; namely, the projection to the ith component. Hence, $(y_1, \ldots, y_n).Y_i = y_i$. We use this convention only if the $Y_1,\ldots,Y_n$ are all different sets so that the operators Y_i are defined unambiguously.

Following these conventions, the operators *DocInstance* and *DocGrammar* operate on *GCSDocument*; for any grammar-constrained structured document $gcsd = (di, dg)$, the equations

$$gcsd.\mathit{DocInstance} = di$$

and

$$gcsd.DocGrammar = dg$$

hold.

For structured documents, the convention gives us operators *DocSort*, *Attributes* and *DocContent*. We need to define three more access operators on structured documents, namely *AttSorts*, *SubDocs* and *ContentString*. For any structured document *di* in *DocInstances*: first, $di.AttSorts$ is the (finite) domain of $di.Attributes$; second, the set of *di*'s subdocuments $di.SubDocs$ forms a set of structured documents, which contains *di* itself and any subdocuments of any structured documents in the string $di.DocContent$; third, $di.ContentString$ is formed from the string $di.DocContent$ by leaving any character as it is and replacing any structured document by its *DocSort*.

For grammars, the convention gives us access operators *ContentRules* and *AttRules*.

Finally, we define two more access operators on structured documents. The operators *IDs* and *IDREFs* collect all values of ID attributes and IDREF attributes, respectively. Both operators need a grammar as additional input, since the information about attribute domains, whether an attribute is an ID attribute, an IDREF attribute or something different is not specified in the document proper, it is only specified in the grammar.

For any structured document *di* and any grammar *dg*, the set $di.IDs(dg)$ contains $sd.Attributes(as)$, for any subdocument *sd* of *di* in $di.SubDocs$ and for any attribute sort *as* in $di.AttSorts$, such that $dg.AttRules(sd, as) = \texttt{ID}$. Analogously, the set $di.REFIDs(dg)$ contains $sd.Attributes(as)$ for any subdocument *sd* of *di* in $di.SubDocs$ and any attribute sort *as* in $sd.AttSorts$ such that $dg.AttRules(sd, as) = \texttt{REFID}$.

Table 3 summarizes the access operators we have just defined.

3.4 GCSDs

To complete the definition of the GCSD model, we set down the conditions under which a structured document and a grammar together form a grammar-constrained structured document. In other words, we define exactly what are the elements of *GCSDocument*.

A pair (di, dg) of a structured document *di* and a grammar *dg* is in *GCSDocument* if and only if it satisfies the following four conditions:

1. Each subdocument *sd* of *di* in $di.SubDocs$ has a content rule in *dg* and satisfies it; that is, $dg.ContentRules(sd.DocSort)$ is defined and contains $sd.ContentString$.
2. Each subdocument *sd* of *di* in $di.SubDocs$ satisfies its attribute rule; that is, first, that the attribute sorts for which values are defined in $sd.Attributes$ are exactly the attribute sorts for which the attribute rule $dg.AttRules(sd, \Box)$ is defined and,

Table 3 A summary of the access operators for grammars, structured documents and grammar-constrained structured documents.

Operand	Operator	Codomain
GCSDocument	*DocumentInstance*	*DocumentInstance*
	DocGrammar	*DocGrammar*
DocInstance	*DocSort*	*DocSort*
	Attributes	*Attributes*
	DocContent	*DocContent*
	AttSorts	$\mathcal{P}(AttSort)$
	SubDocs	$\mathcal{P}_{\text{fin}}(DocInstance)$
	ContentString	$(Character \dot{\cup} DocSort)^*$
	$IDs(DocGrammar)$	$\mathcal{P}_{\text{fin}}(DocID)$
	$REFIDs(DocGrammar)$	$\mathcal{P}_{\text{fin}}(DocID)$
DocGrammar	*ContentRules*	*ContentRules*
	AttRules	*AttRules*

second, that the value of the attribute in the subdocument is compatible to the domain in the grammar rule. Formally, two conditions must be satisfied:

(a) An attribute sort as in $AttSort$ is in $sd.AttSorts$ if and only if $dg.AttRules(sd, as)$ is defined.

(b) For any as in $sd.AttSorts$, $sd.Attributes(as)$ is in $dg.AttRules(sd.as)$.

3. ID attribute values must be unique within di; that is, for any subdocument sd of di, if $dg.AttRules(sd, as) = \texttt{ID}$, then $sd.Attributes(as)$ is not in $sd'.IDs$ for any subdocument sd' of sd other than sd.
4. There are no dangling IDREFs; that is, $di.IDREFs \subseteq di.IDs$.

We illustrate the relationship between the GCSD model and XML by example before discussing it systematically in Section 3.6.

3.5 An XML example

We give an example XML DTD for poems and, as an XML-document instance, a poem by Frost.

3.5.1 An XML DTD for poems

```
<?XML VERSION="1.0"?>
<!DOCTYPE poem [
```

```
<!ENTITY % text "(#PCDATA | emph)*">
<!ELEMENT poem   (title,author?,verse+)>
<!ELEMENT verse  (line+)>
<!ELEMENT title  %text;>
<!ELEMENT author %text;>
<!ELEMENT line   %text;>
<!ELEMENT emph   %text;>
<!ATTLIST poem   editor CDATA #REQUIRED
                 date   CDATA #REQUIRED>
<!ATTLIST poem   status (prelim | checked) "prelim">
]>
```

3.5.2 A poem-document instance

```
<poem editor="X.YZ" date='01/01/2000'>
<title>The Road Not Taken</title>
<author>Robert Frost</author>
<verse>
  <line>Two roads diverged in a yellow wood,</line>
  <line>And sorry I could not travel both</line>
  <line>And be one traveler, long I stood</line>
  <line>And looked down one as far as I could</line>
  <line>To where it bent in the undergrowth;</line>
  <line>Then took the other, as just as fair,</line>
  <line>And having perhaps the better claim,</line>
  <line>Because it was grassy and wanted wear;</line>
  <line>Though as for that, the passing there</line>
  <line>Had worn them really about the same,</line>
  <line>And both that morning equally lay</line>
  <line>In leaves no step had trodden black.</line>
  <line>Oh, I kept the first for another day!</line>
  <line>Yet knowing how way leads to way,</line>
  <line>I doubted if I should ever come back.</line>
  <line>I shall be telling this with a sigh</line>
  <line>Somewhere ages and ages hence:</line>
  <line>Two roads diverged in a wood, and I-</line>
  <line>I took the one less traveled by,</line>
  <line>And that has made all the difference.</line>
</verse>
</poem>
```

3.5.3 The corresponding poem GCSD

We now define a GCSD in *GCSDocument* that models the XML DTD and the DTD-conforming document instance.

Let *Character* contain all printable ASCII characters (which include the blank character), let the symbols poem, verse, title, author, line and emph be in *DocSort*, let the symbols editor, date, status be in *AttSort* and let the symbols prelim and $\mathsf{checked}$ be in *AttValue*.

We define for the 20 lines of the XML document instance 20 structured documents $\mathit{line\_1}, \ldots, \mathit{line\_20}$ of the form $(\mathsf{line}, \emptyset, \texttt{"...")}$, where $\emptyset$ denotes the function in *Attributes* with empty domain and $\texttt{"..."}$ contains the line's text as a catenation of characters. For example,

$$\mathit{line\_19} = (\mathsf{line}, \emptyset, \texttt{"I took the one less traveled by,"}).$$

Further structured documents in *DocInstance* are

$$\mathit{verse} = (\mathsf{verse}, \emptyset, \mathit{line\_1} \cdot \cdots \cdot \mathit{line\_20}),$$

$$\mathit{title} = (\mathsf{title}, \emptyset, \texttt{"The Road Not Taken"}),$$

$$\mathit{author} = (\mathsf{author}, \emptyset, \texttt{"Robert Frost"})$$

and

$$\mathit{poem} = (\mathsf{author}, \mathit{edAtts}, \mathit{title} \cdot \mathit{author} \cdot \mathit{verse}),$$

where *edAtts* is the function

$$\mathsf{editor} \longrightarrow \texttt{"X.YZ"},\ \mathsf{date} \longrightarrow \texttt{"01/01/2000"},\ \mathsf{status} \longrightarrow \mathsf{prelim}.$$

The XML DTD for poems is modeled as the tuple

$$\mathit{poemGrammar} = (\mathit{poemContentRules}, \mathit{poemAttRules}),$$

where *poemContentRules* is the function

$$\mathsf{poem} \longrightarrow \mathrm{L}(\mathsf{title}(\mathsf{author} \mid \epsilon)\mathsf{verseverse}^*),$$

$$\mathsf{verse} \longrightarrow \mathrm{L}(\mathsf{lineline}^*),$$

$$\mathsf{title}, \mathsf{author}, \mathsf{line}, \mathsf{emph} \longrightarrow (\mathit{Character}\, \dot{\cup} \{\mathsf{emph}\})^*$$

and *poemAttRules* is the function

$$(\mathsf{poem}, \mathsf{editor}) \longrightarrow \texttt{String},\ (\mathsf{poem}, \mathsf{date}) \longrightarrow \texttt{String},\ (\mathsf{poem}, \mathsf{status}) \longrightarrow \{\mathsf{prelim}, \mathsf{checked}\}.$$

Finally, the complete XML document with DTD and instance is modeled by the tuple $(\mathit{poem}, \mathit{poemGrammar})$.

3.5.4 Comments on the example

The example demonstrates the various ways in which the model abstracts from the syntactic sugar of XML, namely:

1. White space in the DTD.
2. Order of element declarations.
3. Order and grouping of attribute declarations, conflict rules for multiply-defined attributes.
4. Physical structure (entities).
5. Specification of content models.
6. White space between markup tags in document content.
7. Character representation (character references) and encoding.
8. Delimiters for attribute values.

3.6 GCSD model and design decisions

In Table 4, we indicate how the GCSD model relates to XML concepts.

The full version of this paper will discuss design decisions regarding:

1. Whitespace in element content.
2. Defaults for attributes.
3. Normalized attribute values.
4. Other attribute types (ENTITY).
5. Unambiguous content models.
6. EMPTY and ANY content declarations.
7. Several IDs per element.
8. Name of document type declaration.

It will also define an XML-conforming representation for the model.

4 Concluding remarks

We close by briefly mentioning a number of issues.

Table 4 The relationship between GCSD and XML.

GCSD	XML ($[< \mathit{rulenumber} >]$)
GCSDocument	*document* [1]
DocInstance	*element* [39]
DocSort	element type, named in the element's tags (corresponds to *Name* in $[40, 42, 44, 45, 52]$)
Attributes	*Attribute* [41]
Content	*content* [43]
DocGrammar	*doctypedecl* [28]
ContentRules	sequence of *elementdecl* [45]
DocDomain	*contentspec* [46]
Reg(*DocSort*)	*children* [47]
RReg(*Character*, *DocSort*)	*Mixed* [51]
AttSort	name of an attribute, corresponds to *Name* in attribute definition [53]
AttDomain	*AttType* [54]
$\mathcal{P}_f fin(\mathit{AttValue})$	*EnumeratedType* [57]
`ID`, `IDREF`	instances of *TokenizedType* [56]
`String`	*StringType* [55]
Character	*CharData* [14]

4.1 Engineering XML applications

1. State requirements of functionality and performance for operations on documents in the model, based on document applications.
2. Implement the model, which involves:
   (a) The specification and implementation of document classes (in the sense of object orientation, not document-type definitions) for various groups of requirements of functionality and performance. The class specification should be based on the model of grammar-constrained structured documents, not the XML-L format.
   (b) The provision of an interface to the persistent-representation format (parsing and unparsing, import and export).
   (c) The development of a model of an XML processor that builds the abstract document structure. Our XML processor follows the well-established scheme of programming-language parsers; large parts of it can be generated by tools from specifications, so they do not have to be manually programmed. To achieve this goal, we reformulate the language rules of XML so that we can express

XML processing in terms of the standard phases of compiler construction. In particular, our XML processor separates lexical analysis (tokenization) from syntactic analysis (parsing). Our model also deals with the novel (compared to programming languages) requirement of XML that the data are given as a collection of storage units and that the physical organization of these units is discoverd only during XML processing.

3. Validate the implementation by developing document applications that are based on it.

4.2 Related work

1. The GCSD model borrows heavily from the abstract-data-type (ADT) or object-oriented approach to programming. But, its novelty is that it provides a model for both a grammar and the document instances of the given grammar.
2. No re-engineering of XML has been done. Existing models do not cover grammars, attributes and secondary structures.
3. Models (or better representations) of structured documents for specific applications (databases) have been suggested.
4. The normalized ESIS structure proposed by James Clark is, perhaps, the closest approach to modeling XML documents.

4.3 Future work

1. Re-engineer the full XML, including the physical part (entities).
2. Re-engineer other Web standards, for example XML Schema or XSLT.

References

[1] T. Bray, J. P. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210/, February 1998.

[2] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing—Origination, Dissemination and Design*, 1(1):19–44, April 1988.

[3] N. Meyrowitz and A. van Dam. Interactive editing systems (Part I and II). *Computing Surveys*, 14(3):321–415, September 1982.