A Formal Model of XML Schema

Leonid Novak
Institute for System Programming
Russian Academy of Sciences
25, B. Kommunisticheskaia str.
Moscow 109104, Russia
novak@ispras.ru

Alexandre Zamulin [*]
A.P. Ershov Institute of Informatics Systems
Siberian Branch of Russian Academy of Sciences
Novosibirsk 630090, Russia
zam@iis.nsk.su

Abstract

The semantics of the core features of XML Schema in terms of XQuery 1.0 and XPath 2.0 data model algebraically defined is given. The database state is represented as a many sorted algebra whose sorts are sets of data type values and different kinds of nodes and whose operations are data type operations and node accessors. The values of some node accessors, such as "parent", "children" and "attributes", define a document tree with a definite order of nodes. The values of other node accessors help to make difference between kinds of nodes, learn the names, types and values associated with the corresponding document entities, etc., i.e., provide primitive facilities for a query language. As a result, a document can be easily mapped to its implementation in terms of nodes and accessors defined on them.

1 Introduction

In this paper, we present a formalization of some core ideas of XML Schema [2, 20] by means of algebraic techniques. The benefits of a formal description are well known: it is both concise and precise [3]. This is not the first attempt to formalize an XML language. A detailed review of related work is given in Section 10. It is sufficient to mention at the moment that in all previous work an XML document rather than an XML database is practically formalized. For this reason, one cannot easily map a document to its implementation in terms of nodes and accessors defined on them. Moreover, any operation of an XML algebra should be defined as a function on the underlining sets. Therefore an algebraic model of the XML database is needed for definition of such operations.

A data model [21] is designed to support the query language XQuery [9]. Since XML Schema is designed for defining databases that may be searched by XQuery, it is natural to use this model as semantics of XML Schema. For this purpose, we need to define formally the model and map syntactic constructs of XML Schema to the components of the model. As a result, we can get an abstract implementation of XML Schema, which may be helpful both in the concise description of XML Schema and the understanding of its implementation.

To save space, we define only the semantics of a representative part of XML Schema, simplifying many of its constructions. We consider only the most important document components: elements and attributes, other components such as comments, namespaces, and processing instructions can be easily added to the presented model without its redefinition.

It is assumed that the reader is familiar with XML and some document type definition language like DTD. The familiarity with XML Schema is desirable, but not mandatory.

The rest of the paper is organized as follows. The abstract syntax of element declarations and type definitions in XML Schema is presented in Section 2, and the abstract syntax of the document schema is given in Section 3. Basic types of XML Schema are listed in Section 4. Base classes of the data model are described in Section 5. The database itself is defined in Section 6. The document order is defined in Section 7. It is shown in Section 8 that an XML document can be converted into a database tree and vice versa. Some aspects of practical implementation of the data model are outlined in Section 9. A review of related work is presented in section 10, and concluding remarks are given in Section 11.

2 Element declarations and type definitions

In this section we present an abstract syntax of element declarations and type definitions in XML Schema. The

[*]The work of this author is supported in part by Russian Foundation for Basic Research under Grant 04-01-00272.

syntax is given in terms of syntactic types representing syntactic domains, and the following type constructors:

$Seq(T)$ — type of ordered sets of values of type T (empty set included).
$FM(T_1, T_2)$ — type of ordered sets (empty set included) of pairs of values of types T_1 and T_2 defining final mappings from T_1 to T_2.
$Union(T_1, ..., T_n)$ — type of the disjoint union of values of types $T_1, ..., T_n$.
$Enumeration$ — enumeration type constructor.
$Pair(T_1, T_2)$ — type of pairs of values of the indicated types.
$Interleave(T_1, T_2)$ — type of two-item sets of values of types T_1 and T_2 (if a and b are values of respective types T_1 and T_2, then both $a\&b$ and $b\&a$ are instances of this type).
$Tuple(T_1, ..., T_n)$ — type of tuples of values of the indicated types.

The presentation is supplied with examples written in the XML Schema language. We hope that the reader will easily find the correspondence between abstract syntax constructions and their XML representations.

There is a predefined syntactic type, $Name$, whose elements are used for denoting different document entities. Depending on the context where this type is used, we denote it either by $ElemName$ or $AttrName$ or $SimpleTypeName$ or $ComplexTypeName$.

$ElementDeclaration =$
$\quad Tuple(ElemName, Type, RepetitionFactor, NillIndicator);$
$RepetitionFactor = Pair(Minimum, Maximum);$
$Minimum = NatNumber;$
$Maximum = Union(NatNumber, \{\text{“}unbounded\text{”}\});$
$NillIndicator = Boolean;$

The $RepetitionFactor$ indicates here how many element information items with this $ElemName$ a document may have. The $NillIndicator$ indicates whether the element may have the nil value. $NatNumber$ and $Boolean$ respectively denote conventional natural numbers and boolean values.

```
<xsd:element name="annotation"
    type="xsd:string" nillable="true"/>
<xsd:element name="Book" type="Book-type"
    minOccurs="1" maxOccurs="unbounded"/>
<xsd:element name="A">
 <xsd:complexType>
       ...
 </xsd:complexType>
</xsd:element>
```

Example 1.

Three element declarations are given in Example 1. The $RepetitionFactor$ is indicated there by the pair $(minOccurs, maxOccurs)$. In the first and third element declarations the default value (1, 1) is used, in the second declaration the value is set explicitly. An anonymous complex type is used in the third declaration. $NillIndicator$ is set to $false$ by default in the second and third declarations. Thus only the first element may have the nil value.

$GroupDefinition =$
$\quad Tuple(Seq(LocalGroupDefinition),$
$\quad\quad CombinationFactor, RepetitionFactor);$
$LocalGroupDefinition = ElementDeclaration;$[1]
$CombinationFactor =$
$\quad Enumeration(\text{“}$`sequence`$\text{”}, \text{“}$`choice`$\text{”});$

A group definition consists of a sequence of element declarations. The $CombinationFactor$ indicates whether the group defines a sequence or choice. The element names in a sequence of local group definitions must be different. A group definition has the *empty content* if the sequence of local group definitions is empty. The $CombinationFactor$ and $RepetitionFactor$ do not make sense in this case.

A group as a sequence of elements is defined in Example 2 and as a choice of elements in Example 3.

```
<xsd:sequence>
 <xsd:element name="B" />
 <xsd:element name="C" />
</xsd:sequence>
```

Example 2.

```
<xsd:choice minOccurs="0" maxOccurs="unbounded">
 <xsd:element name="zero" type="xsd:unsignedByte"/>
 <xsd:element name="one" type="xsd:unsignedByte"/>
</xsd:choice>
```

Example 3.

$Type =$
$\quad Union(TypeName, AnonymousTypeDefinition);$

A type may be defined inline in an element declaration (third declaration in Example 1) or supplied with a name in a type definition (Example 7), which binds the type name to a type definition. Some type names are predefined, they denote primitive simple types.

$TypeName =$
$\quad Union(SimpleTypeName, ComplexTypeName)$

A simple type in an element declaration means the definition of zero or more tree leaves. A complex type in an

[1]In fact, a local group definition may also include another group definition, see [16] for its formal treatment.

element declaration means, as a rule, the definition of zero or more intermediate nodes of a tree. We consider in the sequel that all simple types are predefined and have a name.

$$AttributeDeclarations = FM(AttrName, SimpleTypeName);$$

$AttributeDeclarations$ introduce a number of attributes with different names. The type of an attribute is always a simple type. For simplicity, we do not indicate properties (REQUIRED, PROHIBITED, OPTIONAL) and default values.

Two attribute declarations are presented in Example 4.

```
<xsd:attribute name="InStock" type="xsd:boolean"/>
<xsd:attribute name="Reviewer"type="xsd:string"/>
```

Example 4.

$$\begin{aligned}
&AnonymousTypeDefinition = \\
&\quad Union(SimpleContentDefinition, \\
&\qquad\qquad ComplexContentDefinition); \\
&SimpleContentDefinition = \\
&\quad Pair(SimpleTypeName, AttributeDeclarations); \\
&ComplexContentDefinition = \\
&\quad Pair(MixedIndicator, ComplexTypeContent), \\
&ComplexTypeContent = \\
&\quad Union(LocalElementDeclarations, \\
&\qquad AttributeDeclarations, \\
&\qquad Pair(LocalElementDeclarations, \\
&\qquad\qquad AttributeDeclarations)); \\
&MixedIndicator = Boolean; \\
&LocalElementDeclarations = GroupDefinition;
\end{aligned}$$ [2]

A complex type may have either a simple content or a complex content. In the first case, a simple type is extended by attribute definitions. In the second case, the definition of a complex type typically consists of (local) element declarations or attribute declarations or both. If the $MixedIndicator$ in the $ComplexContentDefinition$ is set to $true$, then a document may contain text nodes in between the element nodes of the corresponding group.

```
<xsd:complexType>
 <xsd:simpleContent>
  <xsd:extension base="xsd:decimal">
   <xsd:attribute name="currency" type="xsd:string"/>
  </xsd:extension>
 </xsd:simpleContent>
</xsd:complexType>
```

Example 5.

The definition of a complex type with a simple content is presented in Figure 5. An element of this type may have a decimal value and an attribute.

```
<xsd:complexType mixed = "true">
 <xsd:sequence>
  <xsd:element name="Book" minOccurs = "0"
              maxOccurs="1000">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="Title"  type="xsd:string"/>
     <xsd:element name="Author" type="xsd:string"/>
     <xsd:element name="Date" type="xsd:string"/>
     <xsd:element name="ISBN" type="xsd:string"/>
     <xsd:element name="Publisher"
                 type="xsd:string"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
 <xsd:attribute name="InStock" type="xsd:boolean"/>
 <xsd:attribute name="Reviewer" type="xsd:string"/>
</xsd:complexType>
```

Example 6.

The definition of a complex type with complex content is presented in Figure 6. The mixed indicator of the outer type indicates that ”Book” elements can be interleaved by texts.

3 Document schema

In this model we permit only one element information item as a child of the document information item. This model is more restrictive than the one specified in [21] (where several element information items may be children of the document information item), but it strictly follows the model specified in [20] (see also Section 2.2.2 in [7]).

$$\begin{aligned}
&DocumentSchema = \\
&\quad Interleave(ComplexTypeDefinitionSet, \\
&\qquad\qquad GlobElementDeclaration); \\
&ComplexTypeDefinitionSet = \\
&\quad FM(ComplexTypeName, \\
&\qquad\qquad AnonymousTypeDefinition); \\
&GlobElementDeclaration = \\
&\qquad Tuple(ElemName, Type, NillIndicator);
\end{aligned}$$

Thus, a document schema defines a set of documents each having a root element with the same name. The schema may contain a number of complex type definitions preceding or following the $GlobElementDeclaration$ and introducing type names used within the $GlobElementDeclaration$ and $ComplexTypeDefinitionSet$[3]. For any type T used

[2]In fact, a local element declaration may also be a so called *all option definition*, see [16] for its formal treatment.

[3]In fact, the document schema may also contain a number of other element declarations and attribute declarations. However, attributes are always part of complex types and may be declared inline. Multiple global element declarations may also be considered as a kind of syntactic sugar permitting one either to combine several document schemas in one schema or save space by referencing an element declaration from within several complex types.

in a document schema with a complex type definition set ctd, the following requirement on type usage must be satisfied: $T \in dom(ctd)$[4] or $T \in SimpleTypeName$ or $T \in AnonymousTypeDefinition$.

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.books.org"
  xmlns="http://www.books.org"
  elementFormDefault="qualified">
  <xsd:complexType name="BookPublication">
   <xsd:sequence>
    <xsd:element name="Title"
                 type="xsd:string"/>
    <xsd:element name="Author"
                 type="xsd:string"/>
    <xsd:element name="Date"
                 type="xsd:string"/>
    <xsd:element name="ISBN"
                 type="xsd:string"/>
    <xsd:element name="Publisher"
                 type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="BookStore">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="Book"
                  type="BookPublication"
                  maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:schema>
```

Example 7.

One named and one anonymous data type are defined in Example 7.

4 Basic types

We consider that the data model contains all primitive types listed in [2]. An *atomic type* is a primitive type or a type derived by restriction from another atomic type [2]. A *simple type* is an atomic type or list type or union type or a type derived by restriction from another simple type.

Simple types create a type hierarchy resembling that of object-oriented languages. The type `xs:anyType` is at the top of the hierarchy (i.e., it is the base type of all types). The type `xs:anySimpleType` is a subtype of `xs:anyType` and is the base type of all simple types. The type `xdt:anyAtomicType` is a subtype of `xs:anySimpleType` and is the base type for all the primitive atomic types, and `xdt:untypedAtomic` is its subtype.

[4]Here and in the sequel, $dom(f)$ denotes the domain of a finite mapping f.

In this paper, we additionally use the type constructor $Seq(T)$ defining the set of all sequences (ordered sets) of elements of type T. Any sequence type possesses the following operations among others: the operation $|s|$ returns the length of the sequence s, the operation $s_1 + s_2$ attaches the sequence s_2 to the sequence s_1, and the operation $s[i]$ returns the i-th element of the sequence s.

5 Base classes

The data model defined in [21] has a flavor of an object-oriented model in the sense that its main building entities are unique *nodes* possessing the state that can be viewed by a number of *accessor* functions. There are several disjoint classes of nodes (elements, attributes, etc.) representing different document information items. All of these classes may be considered as subclasses of the base class $Node$. Therefore, the following class hierarchy may be designed:

$Node$: base class with the following accessors:
base-uri: Seq(anyURI) (empty or one-element sequence),
node-kind: string,
node-name: Seq(QName)
(empty or one-element sequence),
parent: Seq(Node) (empty or one-element sequence),
string-value: string,
typed-value: Seq(anyAtomicType)
(sequence of zero or more atomic values),
type: Seq(QName) (empty or one-element sequence),
children: Seq(Node) (sequence of zero or more nodes),
attributes: Seq(Node) (sequence of zero or more nodes),
nilled: Seq(boolean) (empty or one-element sequence).

$Document$: a subclass of the class $Node$ with three extra accessors not considered in this paper.

$Element$: a subclass of the class $Node$ without extra accessors.

$Attribute$: a subclass of the class $Node$ without extra accessors.

$Text$: a subclass of the class $Node$ without extra accessors.

Instances of these classes serve for representing document information items, element information items, attributes and texts, respectively.

6 Database

6.1 State algebra

Because of frequent insertion of new documents, updating existing documents and deleting obsolete documents, a database evolves through different database states. Each state can be formally represented as a many-sorted algebra called a *state algebra* in the sequel[5]. Each class C is supplied in a state algebra $\mathtt{A}$ with a set of object identifiers $\mathtt{A_C}$ in such a way that the sets of identifiers $\mathtt{A_{Document}}$, $\mathtt{A_{Element}}$, $\mathtt{A_{Attribute}}$, $\mathtt{A_{Text}}$, etc. are disjoint and the set $\mathtt{A_{Node}}$ is the union of the above sets. In the sequel, the node identifier is meant each time a node is mentioned (in the same way as an object identifier, or reference, represents an object in object-oriented languages and databases).

Each simple data type T is supplied in any state algebra $\mathtt{A}$ with a set of values $\mathtt{A_T}$ and a set of meaningful operations. The following accessor values are set in $\mathtt{A}$:

- for each $\mathtt{nd} \in \mathtt{A_{Document}}$: node-kind(nd) = "document", node-name(nd), parent(nd), type(nd), attributes(nd), and nilled(nd) are set to empty sequences;
- for each $\mathtt{nd} \in \mathtt{A_{Element}}$: node-kind(nd) = "element";
- for each $\mathtt{nd} \in \mathtt{A_{Attribute}}$: node-kind(nd) = "attribute", children(nd), attributes(nd), and nilled(nd) are set to empty sequences;
- for each $\mathtt{nd} \in \mathtt{A_{Text}}$: node-kind(nd) = "text", node-name(nd), children(nd), attributes(nd), and nilled(nd) are set to empty sequences.

A state algebra A sets values of the other accessors. The following variables are used in the definition of the state algebra:

$el, el_1, el_2, ...$ — element name,
$at, at_1, at_2, ...$ — attribute name,
eld — element declaration,
$elds$ — sequence of element declarations,
$leds$ — local element declarations,
ctd — set of complex type definitions,
$atds$ — attribute declarations,
$gd, gd_1, gd_2, etc.$ — group definition,
gds — sequence of group definitions,
$T, T_1, T_2, ...$ — data type,
cf — combination factor,
rf — repetition factor,
$min_1, min_2, ...$ — minimum number of occurrences of an element or group,
$max_1, max_2, ...$ — maximum number of occurrences of an element or group,
mid — mixed content indicator,
nid — nil indicator.

The state algebra extensively uses trees of nodes and sequences of trees of nodes. A parent node in such a tree is either a document node or an element node. The children of a particular parent node are those nodes that are indicated by the accessors $children$ and/or $attributes$. We can formally define such a tree as follows.

- a node nd is a tree with the root nd;
- if s is a tree with root nd and $\mathtt{s_1, ..., s_n}$ are trees with roots $\mathtt{nd_1, ..., nd_n}$ such that $\mathtt{children(nd) = (nd_1, ..., nd_n)}$, $\mathtt{parent(nd_1) = nd, ..., parent(nd_n) = nd}$, then $\langle \mathtt{s, (s_1, ..., s_n)} \rangle$ is a tree with the root nd;
- if s is a tree with root nd and $\mathtt{nd_1, ..., nd_n}$ are nodes such that $\mathtt{attributes(nd) = (nd_1, ..., nd_n)}$, $\mathtt{parent(nd_1) = nd, ..., parent(nd_n) = nd}$, then $\langle \mathtt{s, (nd_1, ..., nd_n)} \rangle$ is a tree with the root nd;

The set of these trees constitute the set of values of the data type $Tree$. The function
$root : Tree \rightarrow Node$
applied to a tree yields its root node, and the function
$roots : Seq(Tree) \rightarrow Seq(Node)$
applied to a sequence of trees yields the sequence of their root nodes.

6.2 Document tree

A document schema $S = (eld, ctd)$ or $S = (ctd, eld)$, where $eld = (el, T)$ is an element declaration and ctd a set of complex type definitions, is mapped in a state algebra A to zero or more trees of nodes. Denote such a tree by s. It must satisfy the following requirements:

1. $\mathtt{nd = root(s)} \in \mathtt{A_{Document}}$, string-value(nd) = string-value(children(nd)), and base-uri(nd) $\in \mathtt{A_{anyUri}}$ if the *base-uri* property exists for this document, otherwise base-uri(nd) = (). Thus, the string value of the document node is the string value of its single child.

2. A node end $\in$ s is associated with the element declaration $eld = (el, T)$ so that:

3. end $\in \mathtt{A_{Element}}$, parent(end) = nd, children(nd) = (end) (i.e., a document node has only one child, an element node; it is the node with name "BookStore" in a tree associated with the Example 7 schema); and

4. node-name(end) = el, base-uri(end) = base-uri(parent(end)), type(end) = T if T is a type name and type(end) = "xs : anyType" if T is an anonymous type definition, and string-value(end) is computed according to the algorithms described in [21], Section 6.2.2.

5. If $nid = false$ (i.e., the element may not have the nil value), then nilled(end) = $false$, and

5.1. If T is a simple type then:

5.1.1. There is in s a node tnd $\in \mathtt{A_{Text}}$ such that parent(tnd) = end, base-uri(tnd) = base-uri(end), type(tnd) = "xdt : untypedAtomic", string-value(tnd) $\in \mathtt{A_{String}}$, and children(end) = (tnd).

[5]Algebra components are written in this paper in true type while the names defined in the document schema are written in italics.

For instance, a text node is associated with each of the element nodes with names `Title`, `Author`, `Date`, `ISBN` and `Publisher` in a tree associated with the Example 8 schema.

5.2. If T is a complex type with simple content $(T_1, atds)$, where $atds = (at_1, T_1), ..., (at_u, T_u)$ (attributes are declared), then items 5.1.1 and 5.3.1 hold. For instance, a text node and attribute node will be associated with an element declared with the type presented in Example 5.

5.3. If T is a complex type with complex content $(mid, leds, atds)$ or $(mid, atds)$, where $atds = (at_1, T_1), ..., (at_u, T_u)$ (attributes are declared), then

5.3.1. $\mathtt{s}$ contains a sequence of nodes $\mathtt{as} = (\mathtt{and_1}, ..., \mathtt{and_u})$ such that $\mathtt{attributes(end)} = \mathtt{as}$ (the sequence consists of two nodes for the attribute declarations of Example 6) and, having an automorphism σ on $\{1, ..., u\}$ (we need it because the sequence of nodes may be different from the sequence of the corresponding attribute declarations), it holds for each $\mathtt{and_j} \in \mathtt{as}$: $\mathtt{and_j} \in \mathtt{A_{Attribute}}$, $\mathtt{parent(and_j)} = \mathtt{end}$, $\mathtt{base\text{-}uri(and_j)} = \mathtt{base\text{-}uri(end)}$, $\mathtt{node\text{-}name(and_j)} = at_{\sigma(j)}$, $\mathtt{type(and_j)} = T_{\sigma(j)}$, $\mathtt{string\text{-}value(and_j)} \in \mathtt{A_{String}}$.

5.4. If T is a complex type with complex content $(mid, leds, atds)$ or $(mid, leds)$ (subelements are declared), then:

5.4.1. If $leds$ is empty (i.e., the type has the empty content), then

5.4.1.1. If $mid = true$ (mixed type definition), then

- $\mathtt{children(end)} = ()$ or
- $\mathtt{children(end)} = (\mathtt{tnd})$ where $\mathtt{tnd}$ is a text node ($\mathtt{tnd} \in \mathtt{A_{Text}}$) with the following accessor values: $\mathtt{parent(tnd)} = \mathtt{end}$, $\mathtt{base\text{-}uri(tnd)} = \mathtt{base\text{-}uri(end)}$, $\mathtt{type(tnd)} =$ "$\mathtt{xdt:untypedAtomic}$", and $\mathtt{string\text{-}value(tnd)} \in \mathtt{A_{String}}$.

Thus, only a text node may be attached to an element node if it has no element child. For instance, an element node corresponding to the element declared with the type presented in Example 7 may have only one text node as child if there are no subordinated `"Book"` elements.

5.4.1.2. If $mid = false$ (no text node is allowed), then $\mathtt{children(end)} = ()$.

5.4.2. If $leds$ is not empty, then there is in $\mathtt{s}$ a sequence of trees $\mathtt{ss}$ such that, for each $\mathtt{rnd} \in \mathtt{roots(ss)}$, it holds: $\mathtt{parent(rnd)} = \mathtt{end}$ and $\mathtt{rnd} \in \mathtt{A_{Element}}$. For instance, a sequence of trees may be associated with a `BookStore` element node (roots of these trees are children of the `BookStore` node) and a sequence of trees may be associated with a `Book` element node (roots of these trees are children of the `Book` node) in a tree associated with the Example 7 schema.

5.4.2.1. If $mid = false$ (no intermediate text nodes are allowed), then $\mathtt{children(end)} = \mathtt{roots(ss)}$.

5.4.2.2 If $mid = true$ (mixed type definition), then

- there is in $\mathtt{s}$ a sequence of text nodes $\mathtt{ts}$, such that, for each $\mathtt{tnd} \in \mathtt{ts}$, it holds: $\mathtt{tnd} \in \mathtt{A_{Text}}$, $\mathtt{parent(tnd)} = \mathtt{end}$, $\mathtt{base\text{-}uri(tnd)} = \mathtt{base\text{-}uri(end)}$, $\mathtt{type(tnd)} =$ "$\mathtt{xdt:untypedAtomic}$", and $\mathtt{string\text{-}value(tnd)} \in \mathtt{A_{String}}$,
- $\mathtt{children(end)} = \mathtt{sss}$, where the sequence of nodes $\mathtt{sss}$ involves all the nodes of the sequences $\mathtt{roots(ss)}$ and $\mathtt{ts}$ in such a way that for any $i \in \{1, ..., |sss| - 1\}$ there do not exist nodes $\mathtt{sss_i}$ and $\mathtt{sss_{i+1}}$ such that $\mathtt{sss_i} \in \mathtt{A_{Text}}$ and $\mathtt{sss_{i+1}} \in \mathtt{A_{Text}}$ (there are no adjacent text nodes). Thus, `Book` nodes of Example 7 may be interleaved with text nodes (note that the children nodes of a `Book` node may not).

5.4.2.3. If $leds$ is a GroupDefinition $(gds, cf, (m, n))$, then $\mathtt{ss}$ consists of k ($m \leq k \leq n$) subsequences of trees $\mathtt{ss_1}$, ..., $\mathtt{ss_k}$ (multiple occurrences of complex type values)[6] and it holds for a subsequence $\mathtt{ss_j}$, $j = 1, ..., k$: if $gds = (eld_1, ..., eld_u)$, where eld_q is an element declaration $(el_q, T_q, (min_q, max_q))$, then

- if $cf =$ "$\mathtt{sequence}$", then $\mathtt{ss_j}$ consists of u subsequences (one for each element definition)[7] of trees $\mathtt{ss^j_q}$, $q = 1, ..., u$, so that $\mathtt{ss^j_q}$ is a sequence of from min_q to max_q trees such that (if $\mathtt{ss^j_q}$ is not empty) each $\mathtt{end} \in \mathtt{roots(ss^j_q)}$ satisfies the requirements starting from item 4, assuming that $el = el_q$ and $T = T_q$;
- if $cf =$ "$\mathtt{union}$", then $\mathtt{ss_j}$ is associated with an eld_q, $q \in \{1, ..., u\}$ (for instance, $\mathtt{ss_j}$ is associated either with the declaration of the element `zero` or with the declaration of the element `one` in Example 3), so that $\mathtt{ss_j}$ is a sequence of from min_q to max_q trees (exactly one tree for any element declaration in in Example 3) such that (if $\mathtt{ss_j}$ is not empty) each $\mathtt{end} \in \mathtt{roots(ss_j)}$ satisfies the requirements starting from item 4, assuming that $el = el_q$ and $T = T_q$;

6. If $nid = true$ (i.e., the element may have the nil value), then:

6.1. If T is a simple type, then either $\mathtt{children(end)} =$ () and $\mathtt{nilled(end)} = \mathtt{true}$ or $\mathtt{nilled(end)} = \mathtt{false}$ and item 5.1.1 holds.

6.2. If T is a complex type with simple content $(T_1, atds)$, where $atds = (at_1, T_1), ..., (at_u, T_u)$, then either $\mathtt{children(end)} =$ () and $\mathtt{nilled(end)} = \mathtt{true}$ and item 5.3.1 holds or $\mathtt{nilled(end)} = \mathtt{false}$ and items 5.1.1 and 5.3.1 hold.

[6]For instance, an $\mathtt{ss}$ associated with the group definition presented in Example 3 may be empty or consist of any number of such subsequences.

[7]For instance, each $\mathtt{ss_j}$ that is part of an $\mathtt{ss}$ associated with the group definition presented in Example 2 consists of two such subsequences.

6.3. If T is a complex type with complex content, then either children(end) = () and nilled(end) = true and item 5.3 holds or nilled(end) = false and items 5.3 and 5.4 hold.

7. There are no other nodes in s.

7 Document order

The ordering of nodes in the tree s defines the document order, which is used in some operations of XQuery [9] and other XML query languages. As in XQuery, the notation $\mathtt{nd}_1 << \mathtt{nd}_2$ means in this paper that the node $\mathtt{nd}_1$ occurs in s before the node $\mathtt{nd}_2$ and the notation $\mathtt{tree}(\mathtt{nd}_1) << \mathtt{tree}(\mathtt{nd}_2)$ means that any node in the tree with the root node $\mathtt{nd}_1$ occurs in s before any node in the tree with the root node $\mathtt{nd}_2$. The relation $<<$ is a total order. Recall that the root node in s is the document node nd. The tree s is ordered as follows:

- let $\mathtt{children}(\mathtt{nd}) = (\mathtt{end})$, then $\mathtt{nd} << \mathtt{end}$;
- for any element node $\mathtt{end} \in \mathtt{s}$, let $\mathtt{attributes}(\mathtt{end}) = (\mathtt{and}_1, ..., \mathtt{and}_\mathtt{k})$ and $\mathtt{children}(\mathtt{end}) = (\mathtt{end}_1, ..., \mathtt{end}_\mathtt{m})$, then $\mathtt{end} << \mathtt{and}_1$, $\mathtt{and}_\mathtt{i} << \mathtt{and}_{\mathtt{i}+1}$, $i = 1, ..., k\text{-}1$, $\mathtt{and}_\mathtt{k} << \mathtt{end}_1$, and $\mathtt{tree}(\mathtt{end}_\mathtt{j}) << \mathtt{tree}(\mathtt{end}_{\mathtt{j}+1})$, $j = 1, ..., m\text{-}1$.

8 XML document vs. document tree

In this section, we address the issue of expressive power and correctness of the data model presented in the paper. In order to do this, we formulate the proposition of the existence of a mapping between XML-documents and document trees that preserves the document validity and content. We respectively write *S-document* and *S-tree* for an XML-document and document tree valid with respect to the document schema S.

First, we introduce an equivalence relation on the set of XML-documents that is based on the document content - *content equality* denoted by $=_c$. The relation is an important basis for formalization of one of the basic notions of the paper, the XML-document. Second we state and prove the following theorem:

Theorem. For any document schema S, there is a function f that maps a set of S-documents to a set of S-trees and a function g that serializes an S-tree to an S-document such that $g(f(X)) =_c X$.

The proof of the theorem can be found in [16].

9 Data model physical representation

In this section, we consider the issue of the physical representation of the data model presented above. It was developed as part of the storage system of the Sedna DBMS [10]. The following features of the Sedna storage system are presented below: descriptive schema, xml node descriptors, block structure, principles of node distribution between blocks, and Sedna numbering scheme [17] foundations.

9.1 Descriptive schema

A descriptive schema is a concise and accurate summary of the structure of an XML document. Formally, the descriptive schema is a tree. Every path (an ordered sequence of nodes, such that any two contiguous nodes in the sequence are in the parent-child relationship) of the document has exactly one path in the descriptive schema, and, vice versa, every path of the descriptive schema is a path of the document.

Let E be the Cartesian product of the set of values of type Qname and the set of node-type identifiers, with respective projection functions $name$ and $type$: $E = Pair(name : Qname, type : \{\mathtt{element}, \mathtt{attribute}, \mathtt{text}, \mathtt{document}\})$. The descriptive schema $\mathtt{X}'$ (also known as DataGuide [13]) of a document tree X is then defined as follows:

- $\mathtt{X}'$ is a tree over the nodes (*schema nodes* in the sequel) of type E.
- Let $\mathtt{e} = (\mathtt{root}(\mathtt{X}), \mathtt{nd}_1, ..., \mathtt{nd}_\mathtt{k})$ be a sequence of nodes from X, such that $\mathtt{parent}(\mathtt{nd}_\mathtt{i}) = \mathtt{nd}_{\mathtt{i}-1}$, $\mathtt{parent}(\mathtt{nd}_0) = \mathtt{root}(\mathtt{X})$. Then, there exists one and only one sequence $\mathtt{s} = (\mathtt{v}_0, .., \mathtt{v}_\mathtt{k})$ of the schema nodes of $\mathtt{X}'$ such that: $\mathtt{parent}(\mathtt{v}_\mathtt{i}) = \mathtt{v}_{\mathtt{i}-1}$, $\mathtt{node\text{-}name}(\mathtt{nd}_\mathtt{i}) = \mathtt{v}_\mathtt{i}.\mathtt{name}$, $\mathtt{type}(\mathtt{nd}_\mathtt{i}) = \mathtt{v}_\mathtt{i}.\mathtt{type}$, $\mathtt{node\text{-}name}(\mathtt{root}(\mathtt{X})) = \mathtt{v}_0.\mathtt{name}$, $\mathtt{type}(\mathtt{root}(\mathtt{X})) = \mathtt{s}_0.\mathtt{type}$.

Consider a simple XML document and its descriptive schema in Example 8.

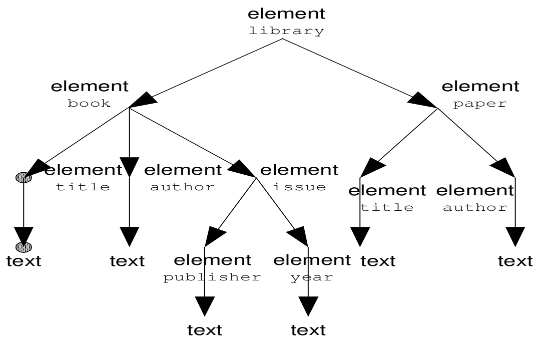
The principles of the schema tree organization allow us to create a surjective mapping f from the set of the nodes in X to the set of schema nodes in $\mathtt{X}'$.

9.2 Data blocks and node descriptors

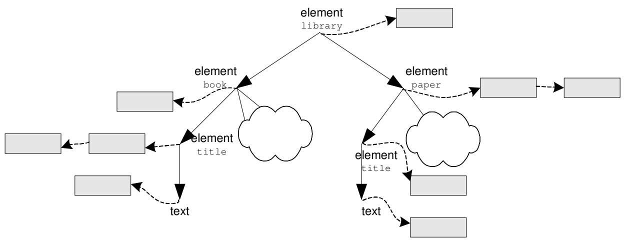
The descriptive schema serves as an entry point to the node storage. Every schema node x has a list of blocks attached, which store node descriptors (physical representations of node instances) associated with the schema node (see Example 9).

Data blocks belonging to one schema node are linked by pointers creating a bidirectional list. Node descriptors in the list are partially ordered according to document order.

```
<library>
  <book>
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
  </book>
  . . .
  <book>
    <title>An Introduction to Database
           Systems</title>
    <author>Date</author>
    <issue>
      <publisher>Addison-Wesley</publisher>
      <year>2004</year>
    </issue>
  </book>
  <paper>
    <title>A Relational Model for
           Large Shared Data Banks</title>
    <author>Codd</author>
  </paper>
  . . .
  <paper>
    <title>The Complexity of
           Relational Query Languages</title>
    <author>Codd</author>
  </paper>
</library>
```

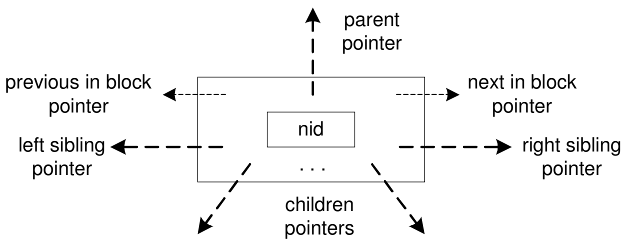


Example 8.



Example 9.

It means that every node descriptor in the block i precedes every node descriptor in the block j in document order, if $i < j$ (i.e., if the block i precedes the block j in the list). However, node descriptors in the same block are not ordered in document order. This decision has been made to simplify updates [10]. To reconstruct the order of node descriptors, we have implemented special short pointers used to link node descriptors from the same block. The general structure of a node descriptor is shown in Example 10.



Example 10.

A node descriptor contains a `parent` pointer, `left sibling` pointer, `right sibling` pointer (whose meanings are straightforward), `nid` field (a numbering label described below), and some other fields. The `next in block` and `previous in block` pointers connect nodes in the same block with the goal of reconstructing the document order as mentioned above. They are short and take only 2 bytes each. Every node that can have children (i.e. an element or document node) has a variable number of pointers to children, in addition. To save space, we store in a node only a fixed set of pointers to the *first children by schema* rather than pointers to all children (this decision has been made to speed up the Xpath execution [10]). Consider the element `library` in Example 8. It has several child `book` elements (two of them are explicited) and several child `paper` elements (two of them are explicited as well), but the descriptive schema element `library` has only two children. So, pointers only to two children will be placed in a node descriptor for the `library` element. They are the pointers to the first child `book` element and the first child `paper` element. Besides this, each block contains a header including a pointer to the corresponding schema node. It is easy to show that the data stored in the node descriptor together with the data stored in the corresponding schema node are sufficient to produce the result of any accessor.

Every decision concerning the content of the node descriptor together with the description of "text-enabled" nodes and the kind of pointers we use are discussed in [10].

9.3 Numbering scheme

Each node descriptor contains a special identifier, a *numbering label*. The set of the numbering labels of the nodes from a tree $\mathtt{X}$ forms the *numbering scheme of* $\mathtt{X}$ [1, 5, 12, 22]. The numbering label encodes information about the relative position of the node in the document. Practically, the main purpose of the numbering scheme is to provide a mechanism to quickly determine the structural relations between a pair of nodes such as *ancestor-descendant* relationship or *document order* relationship. The numbering scheme approach used in Sedna is based on the Dewey ordering [19] with some enhancements serving to prevent the growing of numbering labels after updates of the XML-document.

In a formal setting, let Ω be a finite alphabet with linear ordered symbols, and $\Omega_{\mathtt{min}}$ and $\Omega_{\mathtt{max}}$ be the minimal and maximal symbols of the alphabet. Each numbering label is a finite non-empty sequence of symbols from Ω. Assume $\mathtt{x}$ and $\mathtt{y}$ are two nodes with respective numbering labels $(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k})$ and $(\mathtt{y}_1, ..., \mathtt{y}_\mathtt{n})$. The relationships between these nodes are checked in the following way:

- $\mathtt{x}$ *is less then* $\mathtt{y}$ *in document order if one of the following holds*:
  - $\exists\, i < min(k, n)\, \forall\, j < i :\ \mathtt{x}_\mathtt{j} = \mathtt{y}_\mathtt{j}$ *and* $\mathtt{x}_\mathtt{i} < \mathtt{y}_\mathtt{i}$ *or*
  - $k < n$ *and* $\forall\, i \leq k :\ \mathtt{x}_\mathtt{i} = \mathtt{y}_\mathtt{i}$;
- $\mathtt{x}$ *equals* $\mathtt{y}$ *in document order if* $k = n$ *and* $\forall\, i \leq k : \mathtt{x}_\mathtt{i} = \mathtt{y}_\mathtt{i}$;
- $\mathtt{x}$ *is the parent of* $\mathtt{y}$ *if* $k < n$ *and* $\forall\, i \leq k$: $\mathtt{x}_\mathtt{i} = \mathtt{y}_\mathtt{i}, \mathtt{y}_{\mathtt{k+1}} \leq \Omega_{\mathtt{max}}$.

Other relationships easily outcome from the presented ones. The following proposition is proved in [17]:

Proposition 1. It is possible to generate the numbering scheme for any XML-document and to keep its properties after the updates (insertion or removal of the nodes) of the document.

10 Related work

There are very few papers devoted to formal foundation of XML Schema or another document definition language. More popular subjects are, to our knowledge, validation of a document against a schema [14, 15] and development of an algebra for an XML query language [8, 11].

The paper [3] is a work that directly concerns the problem of formal semantics of XML Schema. Like our paper, it formalizes some core idea of XML Schema. Model Schema Language (MSL) is designed for this purpose. It is described with an inference rule notation originally developed by logicians. These inference rules show in what cases a document validates against a document schema. Thus, the main difference between this paper and our paper is in the fact that this paper does not suggest any internal model of the document schema. As a result, such important aspects as node identity constraints and mappings from XML Schema syntax into internal model components are not touched in the paper. The authors have mentioned that they had begun to work on these topics, but we have not managed to find a paper presenting such a work.

Inference rules are also used in defining the semantics of another popular XML schema language, RELAX NG [4]. The way of defining the semantics in this work resembles that of [3] in the sense that the semantics of a schema consists of the specification of what XML documents are valid with respect to that schema. Like the work [3], this work has the same shortcomings and the same differences with our work.

Formal semantics of values, types, and named typing in XML Schema are defined in [18]. We have not touched these problems, considering that they are successfully solved in that paper.

The representation of an XML document as a data tree is also described in [11]. However, the work is not related with both XML Schema and XQuery 1.0 Data Model. For this reason, the tree consists only of element nodes, the node does not possess an identifier, the majority of node accessors are not defined, etc. In contrast to this work, our document tree is much closer to the tree informally specified in [21].

11 Conclusion

We have presented the semantics of the core features of XML Schema in terms of XQuery 1.0 and XPath 2.0 data model algebraically defined. The database state is represented as a many sorted algebra whose sorts are sets of data type values and different kinds of nodes and whose operations are data type operations and node accessors. The values of some node accessors, such as `parent`, `children` and `attributes`, define a document tree with a definite order of nodes. The values of other node accessors help to make difference between kinds of nodes, learn the names, types and values associated with the corresponding document entities, etc., i.e., provide primitive facilities for a query language. As a result, a document can be easily mapped to its implementation in terms of nodes and accessors defined on them. The main theorem of the paper proves this.

It is worth to note that, with this kind of semantics, XQuery 1.0 and XPath 2.0 data model may be considered as an abstract implementation of XML Schema. Hence, XML Schema and XQuery 1.0 and XPath 2.0 data model become tightly related, which may serve as a significant help for the XML Schema implementor.

Finally, the presented semantics may help in defining a simple semantics of a data manipulation language like XQuery. We intend to proceed with this work.

References

[1] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. *Proc. ICDE 2003*, pp. 705-707, 2003.

[2] P. V. Biron and A. Malhotra, eds. *XML Schema Part 2: Datatypes Second Edition*. W3C Working Draft, 28 October 2004, http://www.w3.org/TR/xmlschema-2/.

[3] Allen Brown, Matthew Fuchs, Jonathon Robie, Philip Wadler. MSL: A model for W3C XML Schema. *Proc. 10th Int'l World Wide Web Conf.*, pp. 191-200, Hong Kong, May 2001.

[4] James Clarke and Murata Makoto. RELAX NG specification. Oasis, December 2001, http://www.relaxng.org/spec-20011203.html/.

[5] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. *Proc. 21st PODS Symposium*, pp. 271–281, 2002.

[6] Roger L. Costello. XML Schemas Reference Manual. http://www.xfront.com/xml-schema.html

[7] David C. Fallside, ed. *XML Schema Part 0: Primer Second Edition*. W3C Working Draft, 28 October 2004, http://www.w3.org/TR/xmlschema-0/.

[8] Mary Fernandez, Jérôme Siméon, and Philip Wadler. An Algebra for XML Query. *FST TCS*, Delhi, December 2000, pp. 11-45.

[9] Daniela Florescu, Jonathan Robie, Jérôme Siméon, et. al., eds. *XQuery 1.0: An XML Query Language*. W3C Working Draft, 29 October 2004, http://www.w3.org/TR/xquery.

[10] Andrey Fomichev, Maxim Grinev, Sergey Kuznetsov. *Descriptive Schema Driven XML Storage*, Technical Report, ISP RAS, 2004. http://www.ispras.ru/ grinev/mypapers/sedna-storage.ps

[11] H. V. Jagodish, L. V. S. Lakshmanan, D. Srivastatva, and K. Thompson. Tax: A Tree Algebra for XML. *Proc. Intl. Workshop on databases and Programming Languages*, Marino, Italy, Sept., 2001.

[12] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *Proceedings of the 27th VLDB Conference*, pp. 361-370, 2001.

[13] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3), pp. 54-66, Sept. 1997.

[14] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. *Extreme Markup Languages*, Montreal, Canada, 2001.

[15] L. Novak and S. Kuznetsov. Canonical Forms of XML Schemas. *Programming and Computer Software*, No. 5, pp. 65-80, 2003.

[16] L. Novak and A. Zamulin. Algebraic Semantics of XML Schema. *Preprint No. 117, Institute of Informatics Systems of the Siberian Branch of the Russian Academy of Sciences*, 2004; http://www.iis.nsk.su/persons/zamulin/zam-preprint117.ps.

[17] L. Novak *Sedna labeling scheme for dynamic XML documents*, Technical Report, ISP RAS, 2005 (to be published).

[18] Jérôme Siméon and Philip Wadler. The Essence of XML. *POPL'03*, January 15-17, 2003, New Orlean, Loisiana, USA.

[19] Igor Tatarinov, Stratis Viglas, e. a. Storing and Querying Ordered XML Using a Relational Database System, *Proc. SIGMOD Conference*, pp. 204-215, 2002.

[20] H. S. Thompson, D. Beech, M. Maloney, and N.Mendelsohn (Eds). *XML Schema Part 1: Structures Second Edition*, W3C Working Draft, 28 October 2004, http://www.w3.org/TR/xmlschema-1/.

[21] *XQuery 1.0 and XPath 2.0 Data Model*, W3C Working Draft, 29 October 2004. http://www.w3.org/TR/xpath-datamodel/

[22] Xiaodong Wu, Mong-Li Lee, Wynne Hsu: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. *ICDE 2004*, pp. 66-78.