Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Leonid Novak, Alexandre Zamulin

ALGEBRAIC SEMANTICS OF XML SCHEMA

Preprint
117

Novosibirsk 2004

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.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Леонид Новак, Александр Замулин

АЛГЕБРАИЧЕСКАЯ СЕМАНТИКА ЯЗЫКА XML SCHEMA

Препринт
117

Новосибирск 2004

Дано отображение основных компонентов языка описания данных XML Schema в компоненты модели данных XQuery 1.0 и XPath 2.0, описанной посредством многоосновных алгебр. Состояние базы данных представлено как алгебра, основами которой являются множества значений типов данных и множества узлов различных типов деревьев, а функциями — операции типов данных и анализаторы узлов. Результаты некоторых анализаторов, как, например, “родитель”, “дети” и “атрибуты”, определяют дерево документа с фиксированным порядком узлов. Результаты других анализаторов позволяют дифференцировать виды узлов и узнавать имена, типы и значения содержимого соответствующих компонентов исходного документа, т.е. обеспечивать примитивные средства для языка запросов. Благодаря предложенной семантике языка XML Schema исходный документ может быть легко отображен во внутреннее представление в базе данных в виде узлов с определенными на них анализаторами.

1. INTRODUCTION

In this paper, we present a formalization of some core ideas of XML Schema [1, 15] (which is nowadays a widely used standard of XML databases) by means of algebraic techniques. The benefits of a formal description are well known: it is both concise and precise [2]. This is not the first attempt to formalize an XML language. A detailed review of related work is given in Section 9. 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 [17] is designed to support the query language XQuery [9] and any other specification that references it. Since XML Schema is designed for defining databases that may be searched by XQuery (in fact, the type system of XQuery is based on XML Schema), 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 (for instance, Costello's tutorial [5] indicates 17 ways of element declaration and 20 ways of attribute declaration, which gives no way to describe the options in full in a research paper). 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 [6] 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

[1]This research is supported in part by Russian Foundation for Basic Research under Grant 04-01-00272.

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. A review of related work is presented in section 9, and concluding remarks are given in Section 10.

2. ELEMENT DECLARATIONS AND TYPE DEFINITIONS

In this section we present an abstract syntax of element declarations and type definitions in XML Schema. The 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).
$FM(T1, T2)$ — type of ordered sets (empty set included) of pairs of values of types $T1$ and $T2$ defining final mappings from $T1$ to $T2$.
$Union(T_1, ..., T_n)$ — type of the disjoint union of values of types $T_1, ..., T_n$.
Enumeration — enumeration type constructor.
$Pair(T1, T2)$ — type of pairs of values of types $T1$ and $T2$.
$Interleave(T1, T2)$ — type of two-item sets of values of types $T1$ and $T2$ (if a and b are values of respective types $T1$ and $T2$, then both ($a\&b$ and $b\&a$ are instances of this type).
$Tuple(T1, ..., Tn)$ — type of tuples of values of types $T1$, ..., Tn.

The presentation is supplied with examples written in the XML Schema language. We hope that the reader will easily map examples to the corresponding abstract syntax constructions.

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 *TypeName* or *SimpleTypeName* or *ComplexTypeName*.

ElementDeclaration =
    Tuple(*ElemName*, *Type*, *RepetitionFactor*, *NillIndicator*);
RepetitionFactor = *Pair*(*Minimum*, *Maximum*);
Minimum = *NaturalNumber*;
Maximum = *Union*(*NaturalNumber*, {"*unbounded*"});
NillIndicator = *boolean*;

The *RepetionFactor* 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.

Example 1:

```
<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>
```

Three element declarations are presented in the example. *RepetionFactor* is indicated 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.

$GroupDefinition = Tuple(Seq(LocalGroupDefinition),$
$CombinationFactor, RepetitionFactor);$
$LocalGroupDefinition = Union(ElementDeclaration, GroupDefinition);$
$CombinationFactor = Enumeration(\text{“}sequence\text{”}, \text{“}choice\text{”});$

A group definition consists of a sequence of local group definitions, which are either element declarations or group definitions. Thus, element declarations and group definitions may be intermixed in a group definition. The *CombinationFactor* indicates whether the group defines a sequence or choice. The element names in a sequence of local group definitions must be different. The 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.

Example 2:

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

Example 3:

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

Example 4:

```
<xsd:sequence minOccurs="0" maxOccurs="unbounded"
   <xsd:sequence minOccurs="0" maxOccurs="unbounded"
      <xsd:element name="work" type="xsd:string"/>
      <xsd:element name="eat" type="xsd:string"/>
   </xsd: sequence>
   <xsd:choice>
      <xsd:element name="work" type="xsd:string"/>
      <xsd:element name="play" type="xsd:string"/>
   </xsd:choice>
   <xsd:element name="sleep" type="xsd:string"/>
</xsd:sequence>
```

In Example 2 the group is defined as a sequence of elements and in Example 3 as a choice of elements. Example 4 presents nested group definitions.

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

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 8), which binds the type name to a type definition. Some type names are predefined, they denote primitive simple types (for instance, the type "xsd:string" in the above examples).

$TypeName = Union(SimpleTypeName,\ ComplexTypeName)$

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

AllOptionDefinition =
FM(*ElemName*, *Tuple*(*Type*, *OptionFactor*, *NillIndicator*));
OptionFactor = {0, 1};

This is the declaration of a special group containing the declared elements in any order. This group may not consist of nested groups. An element of the group is optional in a document if the value of the *OptionFactor* is 0, and it must be present if the value is 1. The declaration has the *empty content* if the final mapping is empty.

Example 5:

```
<xsd:all>
   <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:all>
```

In the above example the option factor has the default value 1 (each element must be present in a document).

AttributeDeclarations = *FM*(*AttrName*, *SimpleType*);

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.

Example 6:

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

AnonymousTypeDefinition =
Union(*SimpleContentDefinition*, *ComplexContentDefinition*),
SimpleContentDefinition = *Pair*(*SimpleTypeName*, *AttributeDeclarations*),
ComplexContentDefinition = *Pair*(*MixedIndicator*, *ComplexTypeContent*),
ComplexTypeContent =
Union(*LocalElementDeclarations*, *AttributeDeclarations*,
Pair(*LocalElementDeclarations*, *AttributeDeclarations*));

MixedIndicator = *Boolean*;
LocalElementDeclarations = *Union*(*AllOptionDefinition*, *GroupDefinition*);

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 element nodes of the corresponding group.

Example 7:

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

A complex type with a simple content is defined. An element of this type may have a decimal value and an attribute.

Example 8:

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

A complex type with complex content is defined. The mixed indicator of the outer type indicates that "Book" elements can be interleaved by texts (note that the elements declared in the inner complex type may not).

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 [17] (where several element information items may be children of the document information item), but it strictly follows the model specified in [4] or [15] (see also Section 2.2.2 in [7]).

DocumentSchema =
    Interleave(*ComplexTypeDefinitionSet*, *GlobElementDeclaration*);
ComplexTypeDefinitionSet =
    FM(*ComplexTypeName*, *AnonymousTypeDefinition*);
GlobElementDeclaration = *Tuple*(*ElemName*, *Type*, *NillIndicator*);

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*[1]. For any type T used in the document schema with the complex type definition set *ctd*, the following requirement on type usage must be satisfied:

$T \in$ *SimpleTypeName* or $T \in$ *AnonymousTypeDefinition* or $T \in dom(ctd)$[2]

Example 9:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.books.org"
      xmlns="http://www.books.org"
      elementFormDefault="qualified">
```

[1] 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.

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

```
    <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>
```

One named and one anonymous data type are defined in the example.

4. BASIC TYPES

We consider that the data model contains all primitive types listed in [1]. These are *string, boolean, decimal, float, double, duration, dateTime, time, data, gYearMonth, gYear, gYearDay, gDay, gMonth, hexBinary, based64Binary, anyURI*, and *QName*. An *atomic type* is a primitive type or a type derived by restriction from another atomic type [1]. 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:

- `xs:anyType` is at the top of the hierarchy (i.e., it is the base type of all types);
- `xs:anySimpleType` is a subtype of `xs:anyType` and the base type of all simple types;
- `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 (unfortunately, the model specification [17] does not indicate which `xdt:anyAtomicType` values belong to this subtype).

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 the others: $|s|$ returns the length of the sequence s, $s_1 + s_2$ attaches the sequence s_2 to the sequence s_1, and $s[i]$ returns the i-th element of the sequence s.

5. BASE CLASSES

The data model defined in [17] 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)[3],
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.

[3]Because of complex rules of computing this value for different kinds of nodes, we do not consider this accessor in the sequel.

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. Each class C is supplied in a state algebra $\mathtt{A}$ with a set of node identifiers $\mathtt{A}_{\mathtt{C}}$ in such a way that the sets of identifiers $\mathtt{A}_{\mathtt{Document}}$, $\mathtt{A}_{\mathtt{Element}}$, $\mathtt{A}_{\mathtt{Attribute}}$, $\mathtt{A}_{\mathtt{Text}}$, etc. are disjoint and the set $\mathtt{A}_{\mathtt{Node}}$ 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, see [11] for a formal definition of an object-oriented model).

Each simple data type T is supplied in $\mathtt{A}$ with a set of values $\mathtt{A}_{\mathtt{T}}$ and a set of meaningful operations. One of these operations denoted by the type name and called *constructor* converts a string value into an atomic value of this type, i.e., the operation has the following signature:

$$T : String \rightarrow T.$$

The following node accessor values are set in any state algebra $\mathtt{A}$:

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

A state algebra $\mathtt{A}$ sets values of the other accessors. The following variables are used in the definition of the state algebra:

$el, el_1, el_2, ...$ — element names,
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 types,
rf — repetition factor,
cf — combination 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 $\mathtt{nd}$ is a tree with the root $\mathtt{nd}$;
- if $\mathtt{s}$ is a tree with root $\mathtt{nd}$ and $\mathtt{s}_1, ..., \mathtt{s}_n$ are trees with roots $\mathtt{nd}_1, ..., \mathtt{nd}_n$ such that $\mathtt{children(nd)} = (\mathtt{nd}_1, ..., \mathtt{nd}_n)$, $\mathtt{parent}(\mathtt{nd}_1) = \mathtt{nd}$, ..., $\mathtt{parent}(\mathtt{nd}_n) = \mathtt{nd}$, then $\langle \mathtt{s}, (\mathtt{s}_1, ..., \mathtt{s}_n) \rangle$ is a tree with the root $\mathtt{nd}$;
- if $\mathtt{s}$ is a tree with root $\mathtt{nd}$ and $\mathtt{nd}_1, ..., \mathtt{nd}_n$ are nodes such that

  $\mathtt{attributes(nd)} = (\mathtt{nd}_1, ..., \mathtt{nd}_n)$,
  $\mathtt{parent}(\mathtt{nd}_1) = \mathtt{nd}, ..., \mathtt{parent}(\mathtt{nd}_n) = \mathtt{nd}$,

  then $\langle \mathtt{s}, (\mathtt{nd}_1, ..., \mathtt{nd}_n) \rangle$ is a tree with the root $\mathtt{nd}$;

The set of these trees constitutes the set of values of the data type $Tree$. The function

$root : Tree \to Node$

applied to a tree yields its root node and the function

$roots : Seq(Tree) \to Seq(Node)$

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 $\mathtt{A}$ to zero or more trees of nodes. Denote such a tree by $\mathtt{s}$. It must satisfy the following requirements:

1. `nd = root(s)` $\in A_{Document}$,
`string-value(nd) = string-value(children(nd))`, and
`base-uri(nd)` $\in$ $\mathtt{A}_{\mathtt{anyUri}}$ 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` $\in$ $\mathtt{A}_{\mathtt{Element}}$, `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 9 schema); and

4. `node-name(end)` = el, `base-uri(end)` = `base-uri(parent(end))`, `type(end)` = T if T is a type name, `type(end)` = "`xs:anyType`" if T is an anonymous type definition, and `string-value(end)` and `typed-value(end)` are computed according to the algorithms described in [17], 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` $\in$ $\mathtt{A}_{\mathtt{Text}}$ such that `parent(tnd)` = `end`, `base-uri(tnd)` = `base-uri(end)`, `type(tnd)` = "`xdt:untypedAtomic`", `string-value(tnd)` $\in$ $\mathtt{A}_{\mathtt{String}}$, and `children(end)` = `(tnd)`.

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 9 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 7.

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. `s` contains a sequence of leaf nodes `as` = $(\mathtt{and}_1, ..., \mathtt{and}_u)$ such that `attributes(end)` = `as` (the sequence consists of two nodes for the attribute declarations of Example 8) 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 $\mathtt{and}_j \in$ `as`:

- $\mathtt{and}_j \in \mathtt{A}_{\mathtt{Attribute}}$, `parent(`$\mathtt{and}_j$`)` = `end`,
  `base-uri(`$\mathtt{and}_j$`)` = `base-uri(end)`, `node-name(`$\mathtt{and}_j$`)` = $at_{\sigma(j)}$,

$\texttt{type}(\texttt{and}_j) = T_{\sigma(j)}$, $\texttt{string-value}(\texttt{and}_j) \in \texttt{A}_{\texttt{String}}$,
$\texttt{typed-value}(\texttt{and}_j) = T_{\sigma(j)}(\texttt{string-value}(\texttt{and}_j))$.

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

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

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 8 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

$$\texttt{children}(\texttt{end}) = ().$$

5.4.2. If $leds$ is not empty, then there is in $\texttt{s}$ a sequence of trees $\texttt{ss}$ such that, for each $\texttt{rnd} \in \texttt{roots}(\texttt{ss})$, it holds:

$$\texttt{parent}(\texttt{rnd}) = \texttt{end} \text{ and } \texttt{rnd} \in \texttt{A}_{\texttt{Element}}.$$

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 9 schema.

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

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

- there is in $\texttt{s}$ a sequence of text nodes $\texttt{ts}$, such that, for each $\texttt{tnd} \in \texttt{ts}$, it holds:
  $\texttt{tnd} \in \texttt{A}_{\texttt{Text}}$,
  $\texttt{parent}(\texttt{tnd}) = \texttt{end}$,
  $\texttt{base-uri}(\texttt{tnd}) = \texttt{base-uri}(\texttt{end})$,
  $\texttt{type}(\texttt{tnd}) =$ "$xdt : untypedAtomic$",
  $\texttt{string-value}(\texttt{tnd}) \in \texttt{A}_{\texttt{String}}$, and
  $\texttt{typed-value}(\texttt{tnd}) = xdt : untypedAtomic(\texttt{string-value}(\texttt{tnd}))$,
- $\texttt{children}(\texttt{end}) = \texttt{sss}$, where the sequence of nodes $\texttt{sss}$ involves all

the nodes of the sequences $\texttt{roots}(\texttt{ss})$ and $\texttt{ts}$ in such a way that for any $i \in \{1, ..., |sss| - 1\}$ there do not exist nodes $\texttt{sss}_\texttt{i}$ and $\texttt{sss}_{\texttt{i+1}}$ such that $\texttt{sss}_\texttt{i} \in \texttt{A}_\texttt{Text}$ and $\texttt{sss}_{\texttt{i+1}} \in \texttt{A}_\texttt{Text}$ (there are no adjacent text nodes). Thus, ''Book'' nodes of Example 8 may be interleaved with text nodes (note that the children nodes of a ''Book'' node may not).

5.4.2.3. If *leds* is an AllOptionDefinition(el_1, T_1, min_1), ..., (el_u, T_u, min_u), then $\texttt{ss}$ is a sequence of q trees $(1 \leq q \leq u)$ so that zero or one tree is associated with the element declaration (el_j, T_j, min_j), $j = 1, ..., u$, if $min_j = 0$ and exactly one tree if $min_j = 1$, and each $\texttt{end}_\texttt{i} \in \texttt{roots}(\texttt{ss})$ satisfies the requirements starting from item 4, assuming that $\texttt{end} = \texttt{end}_\texttt{i}$, $el = el_{\sigma(i)}$ and $T = T_{\sigma(i)}$, where $\sigma : \{1, ..., q\} \rightarrow \{1, ..., u\}$ and $i = 1, ..., q$. For instance, an $\texttt{ss}$ associated with the group definition of Example 5 is a sequence of five trees whose root nodes are element nodes with the declared names sequenced in any order.

5.4.2.4. If *leds* is a GroupDefinition $(gds, cf, (m, n))$, then $\texttt{ss}$ consists of k $(m \leq k \leq n)$ subsequences of trees $\texttt{ss}_\texttt{1}$, ..., $\texttt{ss}_\texttt{k}$ (multiple occurrences of complex type values)[4] and it holds for a subsequence $\texttt{ss}_\texttt{j}$, $j = 1, ..., k$:

- if $gds = (gds_1, ..., gds_u)$ and $cf =$ "*sequence*", then $\texttt{ss}_\texttt{j}$ consists of u subsequences (one for each group definition)[5] of trees $\texttt{ss}^\texttt{j}_\texttt{q}$, $q = 1, ..., u$, and
  - if gds_q is an element declaration $(el_q, T_q, (min_q, max_q))$, then $\texttt{ss}^\texttt{j}_\texttt{q}$ is a sequence of v $(min_q \leq v \leq max_q)$ trees such that (if $\texttt{ss}^\texttt{j}_\texttt{q}$ is not empty) each $\texttt{end} \in \texttt{roots}(\texttt{ss}^\texttt{j}_\texttt{q})$ satisfies the requirements starting from item 4, assuming that $el = el_q$ and $T = T_q$ (for instance, $\texttt{ss}^\texttt{j}_\texttt{q}$ is a sequence consisting of one tree for the declaration of the element with the name $\texttt{sleep}$ in the group definition presented in Example 4);
  - if gds_q is a group definition (for instance, the first and second inner group definitions in Example 4), then $\texttt{ss}^\texttt{j}_\texttt{q}$ satisfies the requirements starting from item 5.4.2.4, assuming that $leds = gds_q$ and $\texttt{ss} = \texttt{ss}^\texttt{j}_\texttt{q}$.
- if $gds = (gds_1, ..., gds_u)$ and $cf =$ "*union*", then $\texttt{ss}_\texttt{j}$ is associated with a gds_q, $q \in \{1, ..., u\}$ (for instance, $\texttt{ss}_\texttt{j}$ is associated either with

[4] For instance, an $\texttt{ss}$ associated with the group definition presented in Example 4 may be empty or consist of any number of such subsequences. The same refers to the first inner group definition of this example. The second inner group definition may result only in an $\texttt{ss}$ consisting of one subsequence of trees.

[5] For instance, each $\texttt{ss}_\texttt{j}$ that is part of an $\texttt{ss}$ associated with the group definition presented in Example 4 consists of three such subsequences.

the declaration of the element `work` or with the declaration of the element `play` in the second inner group definition in Example 4), and

- if gds_q is an element declaration $(el_q, T_q, (min_q, max_q))$, then $\mathtt{ss_j}$ is a sequence of v ($min_q \leq v \leq max_q$) trees (exactly one tree for any element declaration in the second inner group definition in Example 4) such that (if $\mathtt{ss_j}$ is not empty) each $\mathtt{end} \in \mathtt{roots}(\mathtt{ss_j})$ satisfies the requirements starting from item 4, assuming that $el = el_q$ and $T = T_q$;
- if gds_q is a group definition, then $\mathtt{ss_j}$ satisfies the requirements starting from item 5.4.2.4, assuming that $leds = gds_q$ and $\mathtt{ss} = \mathtt{ss_j}$.

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

6.1. If T is a simple type, then
either $\mathtt{children}(\mathtt{end}) = ()$ and $\mathtt{nilled}(\mathtt{end}) = true$
or $\mathtt{nilled}(\mathtt{end}) = false$ 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 $\mathtt{children}(\mathtt{end}) = ()$ and $\mathtt{nilled}(\mathtt{end}) = true$ and item 5.3.1 holds
or $\mathtt{nilled}(\mathtt{end}) = false$ and items 5.1.1 and 5.3.1 hold.

6.3. If T is a complex type with complex content, then
either $\mathtt{children}(\mathtt{end}) = ()$ and $\mathtt{nilled}(\mathtt{end}) = true$ and item 5.3 holds
or $\mathtt{nilled}(\mathtt{end}) = false$ 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 $\mathtt{attributes}(\mathtt{end}) = (\mathtt{and_1}, ..., \mathtt{and_k})$ and $\mathtt{children}(\mathtt{end}) = (\mathtt{end_1}, ..., \mathtt{end_m})$, then

$\texttt{end} << \texttt{and}_1$, $\texttt{and}_i << \texttt{and}_{i+1}$, *i = 1, ..., k-1*, $\texttt{and}_k << \texttt{end}_1$, and $\texttt{tree}(\texttt{end}_j) << \texttt{tree}(\texttt{end}_{j+1})$, *j = 1, ..., m-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. We use an XML document's information set [4] rather than the XML document itself because of the set-theoretic nature of the information set ("infoset" in the sequel), convenient for proofs. The following infoset terms are used in this section:

- *Information item.* The notion is similar to the notion of node in the Xquery data model: document information item (i.i.) corresponds to the document node, element i.i. to the element node, and attribute i.i. to the attribute node. There is a difference between the text node and character i.i. Unlike the text node, the character information item represents a stand-alone character in the document while the text node represents an ordered sequence of one or more characters.
- *Item properties.* We use a number of item properties described in the XML infoset data model (written in bold and enclosed into brackets):
  - *Character code*: ISO 10646 code of the character.
  - *Normalized value*: the value of the attribute after its preliminary processing as described in Section 3.3.3 of [6].
  - *Local name*: local part of the attribute or element name (without the namespace prefix)
  - *Children*: sequence of the child information items.
  - *Element content whitespace*: a boolean indicating whether the character is the white space appearing within the element content.
  - *Attributes*: set of the element's attributes.

Since an infoset contains character information items rather than character strings, it is convenient to use in the sequel a notion similar to a character string introduced in [17].

Def. 1 (Ordered set of maximum adjacent character items). An ordered set of character information items is called an *Ordered set of maximum adjacent character items* iff it satisfies the following constraints:

- all of the information items in the set have the same parent;
- the set consists of adjacent character information items with **[element content whitespace]** property set to "false" such that there is no other information item between them (in the document order);
- no other such set exists that contains any of the same character information items and is larger.

Def. 2 (Content equality of character information items). Two character information items, a and b, are said to be contently equal iff

$a_{[\textbf{character code}]} = b_{[\textbf{character code}]}$.

Def 3. (Content equality of ordered sets of character information items). Two ordered sets of character information items s_1 and s_2 are said to be contently equal iff $|s_1| = |s_2|$ and $s_1[i] =_c s_2[i]$, where $1 \leq i \leq |s_1|$.

Def. 4 (Content equality of attribute information items). Two attribute information items, a and b, are said to be contently equal iff

$a_{[\textbf{normalized value}]} = b_{[\textbf{normalized value}]}$ and $a_{[\textbf{local name}]} = b_{[\textbf{local name}]}$

(we disregard namespaces in the paper and therefore omit the requirement of prefix and namespace equality).

The next notion serves for replacing individual character children by ordered sets of maximum adjacent character information items as children. The term "stripped" borrowed from XQuery [9] implies here that something should be ignored.

Def. 5 (children_stripped). If a is an element or document information item, then $a_{[\textbf{children\_stripped}]}$ is the item's property, which is a sequence of information items produced by replacing in $a_{[\textbf{children}]}$ individual character information items by ordered sets of maximum adjacent character information items.

Def. 6 (Content equality of element information items). Two element information items, a and b, are said to be contently equal iff

- $a_{[\textbf{local name}]} = b_{[\textbf{local name}]}$ (once again, we disregard namespaces),
- for each $at_1 \in a_{[\textbf{attributes}]}$ there exists an $at_2 \in b_{[\textbf{attributes}]}$ such that $at_1 =_c at_2$, and vice versa.
- $|a_{[\textbf{children\_stripped}]}| = |b_{[\textbf{children\_stripped}]}|$ and, for each $0 \leq i < |a_{[\textbf{children\_stripped}]}|$,

both $a_{[\textbf{children\_stripped}]}[i]$ and $b_{[\textbf{children\_stripped}]}[i]$
are either element i.i. or ordered sets of character information items
and $a_{[\textbf{children\_stripped}]}[i] =_c b_{[\textbf{children\_stripped}]}[i]$.

Example. The following three XML elements are contently equal:
$< a\ b$=“1” c=“2”$> < b/ > < /a >$
$< a\ c$=“2” b =“1”$> < b/ > < /a >$
$< a\ b$=“1” c = “2”$>< b/ >< /a >$

Def. 7 (Content equality of document information items). Two document information items, a and b, with

$a_{[\textbf{children\_stripped}]} = (e_1)$ and $b_{[\textbf{children\_stripped}]} = (e_2)$
are said to be contently equal iff $e_1 =_c e_2$.

Finally, we define the content equivalence relation on the set of XML-documents.

Def. 8 (Content equality of XML-documents). Two XML-documents are said to be contently equal iff document information items from the XML information sets representing the documents are contently equal.

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 $g(f(X)) =_c X$.

Proof.
1. *Mapping definitions.* We begin with the definition of the function f. In order to do this, we use an auxiliary function f' mapping the information items of an XML document's infoset to the nodes of a corresponding document tree. Thus, let X be an S-document, X' its infoset, and s a document tree such that:

- if x is an ordered set of maximum adjacent character items in X', then $f'(x)$ is a text node in s such that
  string-value$(f'(x))[i] = x[i]_{[\textbf{character code}]}$
  (note that a String value in XML is a sequence of character codes [1]);
- if x is an attribute information item in X', then $f'(x)$ is an attribute node in s such that
  string-value$(f'(x)) = x_{[\textbf{normalized value}]}$ and
  node-name$(f'(x)) = x_{[\textbf{local name}]}$;

- if x is an element information item in X' with

  $x_{[\textbf{children\_stripped}]} = c_0, ..., c_n$ and $x_{[\textbf{attributes}]} = a_0, ..., a_k$,

  then $f'(x)$ is an element node in s such that

  $node\text{-}name(f'(x)) = x_{[\textbf{local name}]}$,
  $children(f'(x)) = (f'(c_o), ..., f'(c_n))$,
  $attributes(f'(x)) = f'(a_0), ..., f'(a_k)$,
  $parent(f'(c_i)) = f'(x)$, and $parent(f'(a_i)) = f'(x)$;
- if x is a document information item in X' and c a child of x, then $f'(x)$ is a document node in s such that $children(f'(x)) = (f'(c))$ and $parent(f'(c)) = f'(x)$.

Note that the function f' preserves all the properties of information items. In particular, it is *relationship preserving*, i.e., if two information items a and b are in the parent-child relationship (i.e., $b_{[\textbf{parent}]} = a$), then corresponding nodes are in the same relationship. This means that if X is an S-document and X' its infoset, then s is an S-tree, and we define $f(X) = s$.

Before introducing the function g that serializes an S-tree to an S-document, we first define an auxiliary function txt mapping the information items of an XML document's infoset, X', to an XML document text:

- if x is an ordered set of maximum adjacent character items in X', then $txt(x)[i] = x[i]_{[\textbf{character code}]}$;
- if x is an attribute information item in X', then

  $txt(x) =$
  $\quad$ " " + $x_{[\textbf{local name}]}$ + "= &*quot*" + $x_{[\textbf{normalized value}]}$ + " &*quot*",

  where &*quot* is the escape symbol for double quotes;
- if x is an element information item in X' with

  $x_{[\textbf{children\_stripped}]} = c_0, ..., c_n$ and $x_{[\textbf{attributes}]} = a_0, .., a_k$,

  then

  $txt(x) =$ "<" + $x_{[\textbf{local name}]}$ + $txt(a_0) + ... + txt(a_k)$ + ">" +
  $txt(c_0) + ... + txt(c_n)$ + "< /" + $x_{[\textbf{local name}]}$ + ">";
- if x is a document information item in X' and c the child of x, then $txt(x) =$ "<?*xml version* = &*quot*1.0&*quot*? >" + $txt(c)$ (remember that a document information item has exactly one child, an element node).

Then we define the function h mapping the nodes of a document tree s to the information items of an XML document's infoset:

- if x is a text node in s, then $h(x)$ is an ordered set of maximum

adjacent character items such that

$h(x)[i]_{[\textbf{character code}]} = \textit{string-value}(x)[i]$;

- if x is an attribute node in s, then $h(x)$ is an attribute information item such that

  $h(x)_{[\textbf{normalized value}]} = \textit{string-value}(x)$ and
  $h(x)_{[\textbf{local name}]} = \textit{node-name(x)}$;

- if x is an element node in s, then $h(x)$ is an element i.i. such that
  - $h(x)_{[\textbf{local name}]} = \textit{node-name(x)}$,
  - $h(x)_{[\textbf{children\_stripped}]} = e_0, .., e_n$, where $n = |\textit{children}(x)|$ and $e_i = h(\textit{children}(x)[i])$,
  - $|h(x)_{[\textbf{attributes}]}| = |\textit{attributes}(x)|$ and for each $a \in \textit{attributes}(x)$ there is a $b \in h(x)_{[\textbf{attributes}]}$ such that $b =_c h(a)$;
- if x is a document node in s such that $\textit{children}(x) = (c)$, then $h(x)$ is a document i.i. such that $h(x)_{[\textbf{children}]} = (h(c))$.

Thus, if s is an S-tree with the document node $nd = \textit{root}(s)$, then we define $g(s) = \textit{txt}(h(nd))$. In this way both functions, f and g, are defined.

2. *Content equality.* Let X be an S-document, X' its infoset with the document information item a, $s = f(X)$ the corresponding S-tree, $Y = g(s)$ an S-document produced by serialization of s, Y' an infoset with the document information item $b = h(\textit{root}(s))$. Since $Y = \textit{txt}(b)$, then, according to the definition of the function $\textit{txt}$, Y' is the infoset of Y. We want to prove that $X =_c Y$. According to Def. 8, this holds iff the document information items, a and b, in X' and Y' are contently equal. Note that $b = h(f'(a))$. According to Def. 7, the a and b are contently equal iff their children, say ela and elb, are contently equal.

Note that, according to items 4 in the definitions of functions f' and h, $elb = h(f'(ela))$ and therefore we want to prove that $ela =_c h(f'(ela))$. The ela and elb are contently equal iff the three conditions listed in Def. 6 hold. Indeed, the first condition holds since

$$ela_{[\textbf{local name}]} = \textit{element-name}(f'(ela)) = h(f'(ela))_{[\textbf{local name}]} = elb_{[\textbf{local name}]}.$$

In the same way we can prove that the second condition of Def. 6 holds. The third condition is proved by induction on the structure of the **[children_stripped]** property of an element information item e. Induction base: $e_{[\textbf{children\_stripped}]} = ()$, i.e., the element has no children. In this case we have:

$e_{[\mathbf{children\_stripped}]} = () \Rightarrow children(f'(e)) = () \Rightarrow$
$h(f'(e))_{[\mathbf{children\_stripped}]} = ()$.

Induction step: $e_{[\mathbf{children\_stripped}]} = (e_1, ..., e_k)$. In this case,
$children(f'(e)) = (f'(e_1), ..., f'(e_k))$ and
$h(f'(e))_{[\mathbf{children\_stripped}]} = (h(f'(e_1)), ..., h(f'(e_k)))$.
Consider each e_i, $i = 1, ..., k$. If e_i is an ordered set of maximum adjacent character information items, then by definition $e_i =_c h(f'(e_i))$. If e_i is an element i.i., then we assume that $e_i =_c h(f'(e_i))$. From this, it follows that the third condition holds and $e =_c h(f'(e))$. Since this holds for an arbitrary element e, this also holds for the element ela.

Thus, $g(f(X)) =_c X$ because the document information items from their infosets are contently equal. There are interesting corollaries of the theorem.

Corollary 1. If $X =_c Y$, then $g(f(X)) =_c g(f(Y))$.

Proof: $g(f(X)) =_c X =_c Y =_c g(f(Y))$.

Using definitions similar to the Def. 3 — Def. 7, we can introduce the notion of *content equality of document trees*. Then we can write the following corollary as a natural consequence of the previous one.

Corollary 2. There is an injective mapping fs of the factor set of the set of S-documents by the content equivalence relation to the factor set of S-trees by the content equivalence relation.

9. 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 [12, 13] and development of an algebra for an XML query language [8, 10].

The paper [2] 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 [3]. The way of defining the semantics in this work resembles that of [2] 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 [2], 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 [14]. 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 [10]. 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 [17].

10. 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. XML Schema Part 2: Datatypes. W3C Recommendation / Ed. by P. V. Biron, A. Malhotra. — May 2001. — http://www.w3.org/TR/xmlschema-2/.
2. Brown A., Fuchs M., Robie J., WadlerPh. MSL: A model for W3C XML Schema // Proc. 10th Int'l World Wide Web Conf.. — Hong Kong, May 2001. — P. 191–200.
3. Clarke J., Makoto M. RELAX NG specification. — Oasis, December 2001. — http://www.relaxng.org/spec-20011203.html/.
4. XML Information Set / Ed. by J. Cowan, R. Tobin. — World Wide Web Consortium, 24 Oct 2001. — http://www.w3.org/TR/xml-infoset.
5. Costello R. L. XML Schemas Reference Manual. — http://www.xfront.com/xml-schema.html
6. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation 04 February 2004. — http://www.w3.org/TR/2004/REC-xml-20040204
7. XML Schema Part 0: Primer. W3C Recommendation 2 May 2001 / Ed. by D. C. Fallside. — http://www.w3.org/TR/xmlschema-0/.
8. Fernandez M., Siméon J., Wadler Ph. An Algebra for XML Query // FST TCS. — Delhi, December 2000. — P. 11–45.
9. XQuery 1.0: An XML Query Language. W3C Recommendation 14 Nov 2003 / Ed. by D. Florescu, J. Robie, J. Siméon, et. al. — http://www.w3.org/TR/xquery.
10. Jagodish H. V., Lakshmanan L. V. S., Srivastatva D., Thompson K. Tax: A Tree Algebra for XML. — Proc. Intl. Workshop on databases and Programming Languages. — Marino, Italy, Sept., 2001.
11. Lellahi K., Zamulin A.V. An object-oriented database as a dynamic system with implicit state // Advances in Databases and Information Systems. — Proc. of the 5th East European Conf. (ADBIS 2001). — Vilnus, Lithuania, September 2001. — P. 239–252. — (Lect. Notes Comput.Sci.; Vol. 2151).
12. Murata M., Lee D., Mani M. Taxonomy of XML Schema Languages using Formal Language Theory // Extreme Markup Languages, Montreal, Canada, 2001.
13. Novak L., Kuznetsov S. Canonical Forms of XML Schemas // Programming and Computer Software. — 2003. — No. 5. — P. 65–80.
14. Siméon J., Wadler Ph. The Essence of XML // POPL'03. — January 15-17, 2003, New Orlean, Loisiana, USA.
15. XML Schema Part 1: Structures. W3C Recommendation May 2001 / Ed. by H. S. Thompson, D. Beech, M. Maloney, N.Mendelsohn. — http://www.w3.org/TR/xmlschema-1/.
16. XML Query Data Model, Working Draft, Feb 2001. — http://www.w3.org/TR/2001/WD-query-datamodel-20010215/
17. XQuery 1.0 and XPath 2.0 Data Model, Working Draft 12 November 2003. — http://www.w3.org/TR/xpath-datamodel/

Леонид Новак, Александр Замулин

АЛГЕБРАИЧЕСКАЯ СЕМАНТИКА ЯЗЫКА XML SCHEMA

Препринт
117

Рукопись поступила в редакцию 13.10.2004
Рецензент П. Г. Емельянов
Редактор Н. А. Черемных

Подписано в печать 15.11.2004
Формат бумаги 60×84 1/16 Объем 1,5 уч.-изд.л., 1,6 п.л.
Тираж 60 экз.

ЗАО РИЦ "Прайс-курьер" 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6,
тел. (383-2) 30-72-02