An Object-oriented Core for XML Schema

Suad Alagić and Philip A. Bernstein

Microsoft Research
Redmond, USA
alagic@usm.maine.edu
philbe@microsoft.com

ABSTRACT

This paper presents object-oriented interfaces that capture the essence of the structural complexity of XML Schema. We develop two such interfaces: a lightweight object-oriented interface that hides some of the complexity of XML Schema by simplifying the particle and type hierarchies, and a more complete but more complex interface that contains explicit specification of XSD groups. We define a meta level that can store the full details of XSD schemas, such as content models, type derivations for simple and complex types, and identity constraints, which is available to application programmers via reflection. The applicability of the developed interfaces is demonstrated through a collection of complex object-oriented queries.

1. INTRODUCTION

1.1 The Problem

XML Schema Definition language (XSD for short) is a widely-used standard for specifying structural features of XML data [20]. In addition, XSD allows specification of constraints that XML data is required to satisfy. But producing an object-oriented schema that reflects correctly the source XSD schema and adheres to the type systems of mainstream object-oriented languages presents a major challenge. Such an object-oriented interface is required by database designers, users writing queries and transactions, and application programmers in general when processing XML data that conforms to XSD.

There are two broad types of features in XSD: structural and constraint-based. The structural features are represented by the features of the type system. This includes elements, attributes, and various grouping mechanisms such as sequence, choice and all groups, and complex types (which group elements and attributes).

Typical XSD constraints are range constraints that specify the minimum and maximum number of repeated occurrences, rules for type derivation by restriction that restrict the set of valid instances of a type, and identity constraints that define keys and referential integrity. Unfortunately, object-oriented type systems have severe limitations in representing these XSD constraints.

Most of the existing object-oriented interfaces to XSD exhibit a number of problems due to the mismatch of XML and object-oriented type systems. Some of those problems are:

- Not distinguishing between elements and attributes in the object-oriented representation or not representing attributes at all.
- Not being able to represent repetition of elements and attributes with identical names (tags).
- Failing to represent correctly the particle structure of XSD (with elements and groups) and the range of occurrences constraints in particular.
- Confusing the particle hierarchy (with elements and groups) and the type hierarchy (with simple and complex types and type derivations) of XSD.
- Not distinguishing different types of XSD groups in the object-oriented representation (sequence versus choice) or not representing groups at all.
- In the object-oriented representation, not distinguishing the two type derivation techniques in XSD: by restriction and by extension.
- Failing to represent accurately XSD type derivation by restriction, facets and range constraints in particular.
- Having no representation of the XSD identity constraints (keys and referential integrity) and thus no way of enforcing them.

Object-oriented database designers, database users, and application programmers are seldom willing to suffer through the complexities and peculiarities of XML Schema. Hence there exists a real practical need to offer an object-oriented view of the XML Schema. The existing interfaces typically present a view that is very remote from the XSD source and hence there is little that the type system can do to enforce the structural rules of XSD when the corresponding object instances are manipulated.

The key question is whether it is possible to develop an object-oriented interface that captures the core XSD structural features while avoiding at least some of the above problems. Such an interface has not been published so far. The main reason lies in the complexity of XSD, its semantics, and its mismatch with features of object-oriented type systems.

1.2 Contributions

Our research contributions are as follows:

- We isolate the structural core of XSD which contains the essential structural features of XSD and abstracts away a variety of other XSD features (Section 2).

- We specify a light-weight object-oriented interface to schemas expressed in the XSD core that is structurally simple, matches closely the structure of the XML instances, hides much of the complexity of XSD, and simplifies programming (Section 2) and queries (Section 4).
- We specify a more elaborate object-oriented interface to schemas expressed in the XSD core for more sophisticated users who require a deeper understanding of more complex XSD structures (Section 3).
- We demonstrate the utility of our interfaces by giving a variety of typical object-oriented queries (Section 4).
- We specify the object-oriented meta-level which consists of a full representation of features of the XSD core including particle structures (elements and groups), types and type derivations, content models and identity constraints (Section 5).
- We compare our approach to related work (Section 6).

Unlike most other object-oriented interfaces to XSD, we reflect the presence of at least some of the semantic constraints in the target object-oriented schemas. However, due to the limitations of object-oriented type systems, we can do this only structurally. We summarize some examples which are described in detail in the paper: In the generated object-oriented schemas, range constraints are present as minOccurs and maxOccurs methods that return the bounds. The distinction in the semantics of different types of groups is represented by different interfaces whose default implementation is required to support different semantics. Type derivation by restriction is represented using not only inheritance, but also a hierarchy of interfaces representing different types of facets and overriding minOccurs and maxOccurs methods. Full details of the XSD schema are represented at the meta level. The XSD identity constraints are represented at the meta level by a hierarchy of interfaces representing different types of identity constraints. The same applies to the content models and the type derivation hierarchies.

2. XML Schema Light

2.1 The Basics of XSD Light

In this section, we define the lightweight object-oriented interface to the XSD core, which we call XSD Light. It has two type hierarchies just like in XML Schema.

The particle hierarchy contains a direct specification of the actual XML instances, which are documents. An XML document is a single element, which is the basic case of the XSD notion of a particle. In general, a particle consists of a sequence of other particles, which may be elements or more general particles. The range of occurrences in a sequence is determined by invoking methods `minOccurs` and `maxOccurs`, but this range cannot be enforced by the type system.

```
interface XMLParticle
{ int minOccurs();
  int maxOccurs();
  XMLSequence<XMLParticle> particles();
}
```

An element is a particular case of a particle. An element has a name (i.e., a tag) and a value. The value of an element may be simple or complex. The default values of minOccurs and maxOccurs are both equal to 1. The types of values of elements are structured into a separate hierarchy. If an element has a value of a complex type, that type contains the specification of the complex element structure.

```
interface XMLElement: XMLParticle
{ XMLName name();
  XMLanyType value();
}
```

`XMLSequence` is an immutable parametric type that extends the type IEnumerable which represents an immutable sequence as in C#.

```
interface XMLSequence<T>:  IEnumerable<T>
```

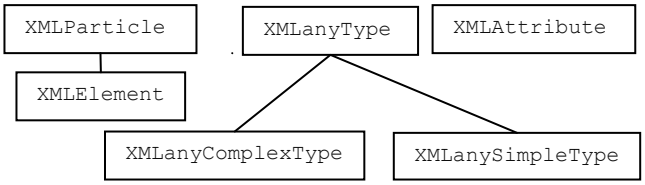

Figure 1 Lightweight XSD Representation

In the lightweight representation in Figure 1, the notion of an XML group is hidden from the users. A group is represented as a sequence of particles. This representation fits XSD specifications of sequence-groups and choice-groups. In the case of an all-group the particles are elements (i.e., not more general particles) and the ordering is irrelevant.

Types of values of elements are structured into the type hierarchy specified below. The root of this type hierarchy is `XMLanyType`:

```
interface XMLanyType {...}
```

An XML type may be simple or complex, hence the two immediate subtypes of `XMLanyType` are `XMLanySimpleType` and `XMLanyComplexType`.

```
interface XMLanySimpleType: XMLanyType {...}
```

Specific simple types are derived from `XMLanySimpleType`, for example:

```
interface XMLString: XMLanySimpleType {...}
```

A value of an XML complex type in general consists of a set of attributes and a content model, where the latter is represented in this interface by its particle structure:

```
interface XMLanyComplexType:  XMLanyType {
   XMLSequence<XMLAttribute> attributes();
   XMLParticle particle();
}
```

Although the ordering of attributes is irrelevant in XSD, the sequence representation for a set of attributes is used above to allow access by methods like those in the interface `IEnumerable`.

So if the value of an element is complex, the type of the element's value will be derived directly or indirectly from `XMLanyComplexType`. Hence the element is in general equipped with a set of attributes and a particle structure, which consists of other particles. This representation of the structure of a complex element corresponds to its structure in XSD, except that the XSD specification of the particle structure is more elaborate

and includes groups. But in the actual element instance, groups appear as sequences of particles.

An attribute has a name (its tag) and a value. The value of an attribute is required to be simple, hence the following specification of an attribute type:

```
interface XMLAttribute  {
   XMLName name();
   XMLanySimpleType value();
}
```

2.2 Sample representation

This XML example consists of a single element whose name is addressBook and whose type is AddressList. AddressList is a complex type whose particle structure is specified as a sequence group. The repeated elements in the sequence are of another complex type AddressType. The particle structure of AddressType is a sequence-group consisting of three element types. The types of these elements are simple.

```
<xsd: element name ="addressBook" type ="AddressList "/>

<xsd: complexType name = "AddressList" >
 <xsd:sequence >
      <xsd: element name = "address" type = "AddressType"
                minOccurs= "0"  maxOccurs = "unbounded " />
 </xsd:sequence>
</xsd: complexType>

<xsd: complexType name ="AddressType">
  <xsd:sequence >
    <xsd: element name = "name"  type = "xsd: string"  />
    <xsd: element  name = "zip"  type ="xsd: int" />
    <xsd: element  name = "street" type = "xsd: string" />
  </xsd:sequence>
</xsd:complexType>
```

In the full, most elaborate representation of the above XML fragment, each element type is specified separately as a type derived from the type `XMLElement`. The result type of the method `value` is overridden covariantly to a specific XML type. Types `XMLString` and `XMLInt` are types derived from `XMLanySimpleType`.

```
interface NameElement: XMLElement {
   new XMLString value();
}
```

Interfaces `ZipElement` and `StreetElement` follow the same pattern.

The type `AddressType` is derived from the type `XMLanyComplexType`. Since there are no attributes in this example, specification of the type `AddressType` contains only its particle structure. The prefix `new` is a C# peculiarity which would not occur in Java.

```
interface AddressType: XMLanyComplexType {
   new Address particle();
}
```

The particle `Address` is a type derived from the type `XMLParticle`. This particle is a sequence of elements, hence the result type of the method `particles` is overridden covariantly to be an `XMLSequence` of `XMLELement`, not just of `XMLParticle`. This overriding of the result type departs from the rules of mainstream object-oriented languages. The reason is that "B is a subtype of A" does not imply "C<B> is a subtype of C<A> for a parametric class C<T>". However, in spite of that, the covariant overriding given above is in fact type safe because `XMLsequence` is an immutable type. `XMLSequence` does not have any mutator or binary methods, just like `IEnumerable`.

Overriding covariantly the result type of an inherited method is a very frequent situation in developing an object-oriented interface to XSD. This feature fits the recent changes in the type systems of mainstream object-oriented languages except in the case discussed above where the result type is an instantiated parametric type whose actual parameter is overriden covariantly. Although this is type safe in the views that we are presenting, we could regard this situation as another instance of the mismatch of what XSD naturally requires and what the mainstream object-oriented languages will allow.

In this example the elements of this particle are also specified. There is an obvious condition that the `XMLsequence` of particles representing the result of the method `particles` consists exactly of the specified elements (i.e., `name`, `zip`, and `street`), but this condition cannot be specified in a type system alone.

```
interface Address: XMLParticle
{ new XMLSequence<XMLElement> particles();
  NameElement   name();
  ZipElement    zip();
  StreetElement street();
}
```

This dual representation of the members of `Address`—as a sequence of elements and as specifically-typed members— is novel as far as we know. It reconciles the more abstract representation of the members in the interface `XMLParticle` with the more specialized and easier-to-use specifically-typed members. The cost of this reconciliation is that it places a requirement on the implementation to keep these two representations in sync.

`AddressListType` is a complex type, hence it is derived from `XMLanyComplexType`. There are no attributes belonging to this type, hence only its particle structure is specified.

```
interface AddressListType: XMLanyComplexType {
   new AddressList  particle();
}
```

`AddressList` is a particle which consists of a sequence of Address particles. `minOccurs` and `maxOccurs` must be overridden accordingly, so their values are specific to the type `AddressList`.

```
interface AddressList:  XMLParticle {
   new int minOccurs();
   new int maxOccurs();
   new XMLSequence<Address>  particles();
}
```

It is now possible to justify a shorthand representation for `Address` that makes programming easier. It is based on knowing that `Address` is a sequence of elements. This list of elements is available as the result of the overridden method `particles`. In addition we can specify the element names and their types as members of the type `Address`. These members do not appear to be of type `XMLElement`, which avoids one level of indirection in using the method `value` to get the element value. The value can now be accessed directly using the corresponding member of

the interface `Address`; for example, `name` is specified to be of type `XMLString`, not of type `NameElement`.

```
interface Address: XMLParticle {
  new XMLSequence<XMLElement> particles();
  XMLString  name();
  XMLInt     zip();
  XMLString  street();
}
```

2.3 Type derivations

An example of simple type derivation by restriction is given below. StateType is derived from its base type string by specifying an enumeration of values of the base type that belong to the derived type.

```
<xsd: simpleType  name = "StateType"  >
  <xsd: restriction base = "xsd:string" >
    <xsd:enumeration value = "Alabama" />
    <xsd:enumeration value = "Alaska"  />
    <!- other enumeration values ->
 </xsd: restriction>
</xsd: simpleType >
```

In the object-oriented representation, type derivations are represented using inheritance:

```
interface StateType:  XMLString {
  // enumeration of values as constants
}
```

The XSD type derivation by extension allows extending the set of attributes and extending the particle structure of a complex type. There are no attributes in the example below. The particle structure of the base type AddressType is extended by specifying particles to be appended to the base particle structure.

```
<xsd:complexType name = "ExtendedAddressType" >
  <xsd: extension base = "AddressType" >
    <xsd: sequence >
      <xsd:element name= "city" type = "xsd:string" />
      <xsd:element name ="state"  type= "StateType" />
    </xsd:sequence>
  </xsd:extension>
</xsd:complexType>
```

In the object-oriented representation, `ExtendedAddress` is a particle that extends the particle `Address`. `ExtendedAddressType` is a complex type that extends the complex type `AddressType`.

```
interface ExtendedAddressType: AddressType {
new ExtendedAddress particle();
}

interface ExtendedAddress: Address {
 cityElement  city();
 stateElement  state(); }
```

3. Programming with XSD groups

A more accurate and more complex programming model than XSD Light is obtained by recognizing that sequences of particles are specified in XSD not only by range constraints, but also by three types of groups (see Figure 2). More precisely, a particle amounts to a sequence of terms. A term is either an element or a group. Since a range constraint may be associated with any type of a term, in a slightly simplified view, elements and groups are viewed as particles, which have range constraints.

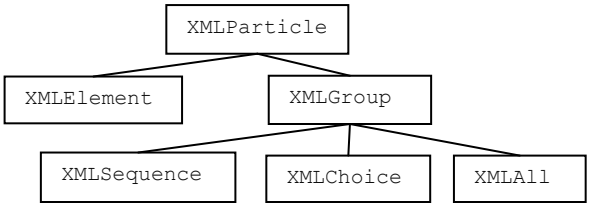

Figure 2 Representing XSD Groups

So we have:

```
interface XMLElement: XMLParticle { ... }
interface  XMLGroup: XMLParticle {
   XMLSequence<XMLParticle> particles();
}
```

There are three types of groups in XSD. Each of them is specified as a sequence of particles. For an all-group these particles must be elements. Hence the result of the method `particles` is covariantly overridden in the all-group as previously explained.

```
interface XMLSequenceGroup: XMLGroup {
}

interface XMLChoiceGroup: XMLGroup {
}

interface XMLAllGroup: XMLGroup {
  new XMLSequence<XMLElement> particles();
}
```

The semantics of sequence-group and choice-group are very different in XSD. An instance of a sequence-group is a sequence of particle instances. An instance of a choice-group contains just one of the particles specified in the choice-group. Specification of this semantic difference cannot be expressed in an object-oriented type system alone. The underlying classes implementing the above interfaces have to correctly implement this semantics.

The following is an example representation of choice that appears in the syntactic specification of expressions. An expression is defined below in XSD as a choice-group, i.e., it is either a constant or has the form of an additive expression:

```
<xsd:complexType name = "Exp" >
  <xsd:choice>
    <xsd:element name = "const" type = "xsd:int"/>
    <xsd:element name = "add" type= "Add" />
  <xsd:choice>
</xsd:complexType>
```

An Add expression is binary so it is specified as a sequence-group with left and right components of type expression.

```
<xsd: complexType name="Add" >
   <xsd:sequence >
     <xsd:element name= "left"   type ="Exp" / >
     <xsd: element name = "right"  type = "Exp" />
  </xsd:sequence>
</xsd:complexType>
```

In the object-oriented representation Add is defined as a complex type whose particle structure is a sequence-group. Exp is a complex type whose particle structure is a choice-group:

```
interface Add: XMLanyComplexType {
  new AddSequenceGroup particle()
}
interface  Exp: XMLanyComplexType {
  new ExpChoiceGroup particle()
}
```

`ExpChoiceGroup` is thus derived from `XMLChoiceGroup` and `AddSequenceGroup` from `XMLSequenceGroup`:

```
interface ExpChoiceGroup: XMLChoiceGroup {
  IntElement  const();
  AddElement  add();
}

interface AddSequenceGroup: XMLSequenceGroup {
  ExpElement  left();
  ExpElement  right();
}
```

In the shorthand representation we would have to override the method `particles` in both groups to obtain the following:

```
interface ExpChoiceGroup: XMLChoiceGroup {
  new XMLSequence<XMLElement> particles();
  XMLInt  const();
  Add add();
}

interface AddSequenceGroup: XMLSequenceGroup {
new XMLSequence<XMLElement> particles();
  Exp  left();
  Exp right();
}
```

If groups are present in the object-oriented representation, the model becomes more expressive but at the same time more complex. This is illustrated in Section 4 by queries with respect to an object-oriented model which contains groups.

AllJobOffers is an element whose type is a complex type JobOffers:

<xsd: element name="AllJobOffers " type= "JobOffers" />

The particle structure of the type JobOffers is a sequence group. The first particle of this sequence-group is an element JobId. The second particle is a sequence-group which consists of two elements: Name and SSN. This latter sequence-group is repeated an unbounded but finite number of times, including zero times.

```
<xsd: complexType name = "JobOffers" >
  <xsd: sequence >
    <xsd: element name = "JobId" type = "xsd:string" />
    <xsd:sequence minOccurs="0" maxOccurs="unbounded"/ >
        <xsd:element name = "Name" type = "xsd:string"/>
        <xsd:element name = "SSN" type = "xsd:int"/>
   </xsd:sequence>
 </xsd:sequence>
</xsd:complexType>
```

In the object-oriented representation AllJobOffers is an element whose value has the type JobOffers.

```
interface AllJobOffers: XMLEement {
  new JobOffers value();
}
```

`JobOffers` is a complex type whose particle is of type `JobSequence`:

```
interface  JobOffers: XMLanyComplexType {
  new JobSequence particle();
}
```

`JobSequence` is a sequence-group.

```
interface JobSequence: XMLSequenceGroup {
   XMLString JobId();
   XMLSequence<JobGroup>  jobOffers();
   // set minOcurs and maxOccurs
}
```

`JobGroup` is a sequence-group whose particles are two elements: `Name` and `SSN`. A shorthand representation is used below:

```
interface JobGroup: XMLSequenceGroup {
  new XMLsequence<XMLElement> particles();
  XMLString Name();
  XMLInt SSN(); }
```

4. Object-oriented queries

In this section we illustrate the usage and suitability of the presented object-oriented interfaces to XSD by presenting a collection of object-oriented queries in the Language-Integrated Query (LINQ) feature of .NET [16].

The shorthand representation makes writing queries simpler. An example query over the address list is given below:

```
AddressList  addressBook =  ...  ;
IEnumerable<Address> JohnDoeAddresses =
   (from x in addressBook.particles()
    where x.name() = "JohnDoe"
    select x)
```

Without the shorthand notation, the where-clause would have to be: `where x.name().value() = "JohnDoe".`

The queries given below reflect complex group structure:

```
AllJobOffers J = ...
IEnumerable<JobGroup> ProgrammingJobs =
 from y in J.value().particle().jobOffers()
 where y.JobId = "Programmer"
 select y);
```

To construct intances of a new type, the corresponding class must be defined first. Given a class

```
class AnOffer: XMLElement {
   AnOffer(XMLString name, int salary);
}
```

the query given below now makes use of the constructor in the above class for producing the output sequence of objects:

```
AllJobOffers J = ...;
JobSequence G = J.value().particle();
AnOffer ProgrammerOffer =
 (from j in G.jobOffers()
where j.JobId()= "Programmer"
select new AnOffer(j.JobId(), 100000);
```

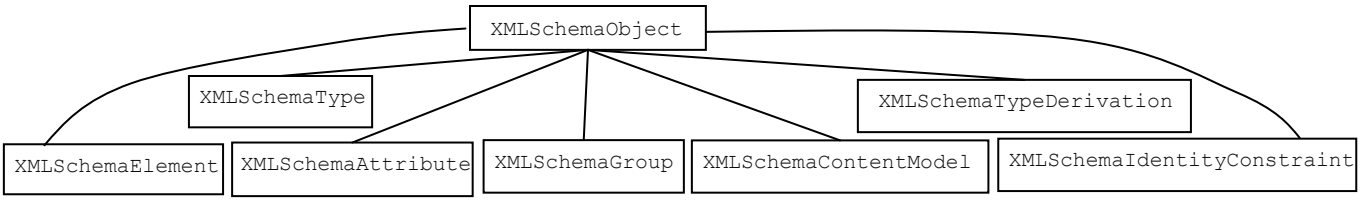

Figure 3 XSD Schema Objects

To illustrate construction of objects of more complex structure, the implementing classes are defined first:

```
class  AllJobOffersClass: AllJobOffers  {
AllJobOffersClass(XMLString name,
                  JobOffers          offers)
}
class JobOffersClass: JobOffers {
   JobOffersClass(JobSequence jobs);
}

class JobGroupClass: JobGroup {
 JobGroupClass(XMLString Name,XMLInt SSN);
}

class JobSequenceClass: JobSequence {
 JobSequenceClass(XMLString jobId,
        XMLSequence <JobGroup> offers)
}

class XMLSequenceClass<T>: XMLSequence <T>{
}
```

A query constructing a complex object representing all programming jobs would now have the following form:

```
XMLSequence<JobGroup> Source = ...;
AllJobOffers programmingJobs =
new AllJobOffersClass("JobOffers",
   new JobOffersClass(
     new JobSequenceClass ("Programmer",
     (XMLSequence<JobGroup>)
      (from g in Source
       where g.JobId()="Programmer"
       select g))))
```

5. Meta level for XSD core

The above queries show that for many applications the presented object-oriented interfaces are adequate. But more sophisticated applications require a full representation of the application's XSD schema. For example, transactions should be written in such a way that they respect the identity constraints (keys and referential integrity) specified in the XSD schema [4]. Accurate representation of the subtleties of the XSD content models and different rules for type derivations in XSD are required when mapping, extending or integrating schemas [1]. The meta (schema) level (see Figure 3) that we present serves these purposes. We tried to make the meta level as complete and accurate a representation of an XSD source schema as is possible within the framework of object-oriented type systems.

Like in SOM [14] there exists an abstraction XMLSchemaObject so that all other schema object types are derived from it.

A content model consists of a specification of a type and its type derivation:

```
interface XMLSchemaContentModel:
                                XMLSchemaObject
{XMLSchemaType content();
 XMLSchemaTypeDerivation typeOfDerivation();
}
```

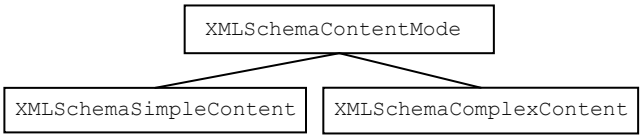

Figure 4 XSD Content Models

A content model may be simple or complex (see Figure 4). If it is simple, the underlying type is simple and so is its type derivation:

```
interface XMLSchemaSimpleContent:
                      XMLSchemaContentModel {
 new XMLSchemaSimpleType content();
 new XMLSchemaSimpleTypeDerivation
                          typeOfDerivation();
}
```

In the above interface the result types of both methods are overridden covariantly.

If a content model is complex, its underlying type may be either simple or complex. This is why the result type of the method content remains XMLSchemaType. If the underlying type is simple, the content model still may contain attributes. But if the content model is complex, the type derivation will be one of complex type derivations, as reflected in the result type of the method typeOfDerivation:

```
interface XMLSchemaComplexContent:
                     XMLSchemaContentModel {
  XMLSchemaComplexTypeDerivation
                          typeOfDerivation();
}
```

The interfaces that follow represent XSD type derivation rules (see Figure 5). Every type derivation has a base type:

```
interface XMLSchemaTypeDerivation:
                            XMLSchemaObject {
  XMLSchemaType base();
 }
```

If the type derivation is simple, the base type must be simple:

```
interface XMLSchemaSimpleTypeDerivation:
                  XMLSchemaTypeDerivation {
  new XMLSchemaSimpleType base();
}
```

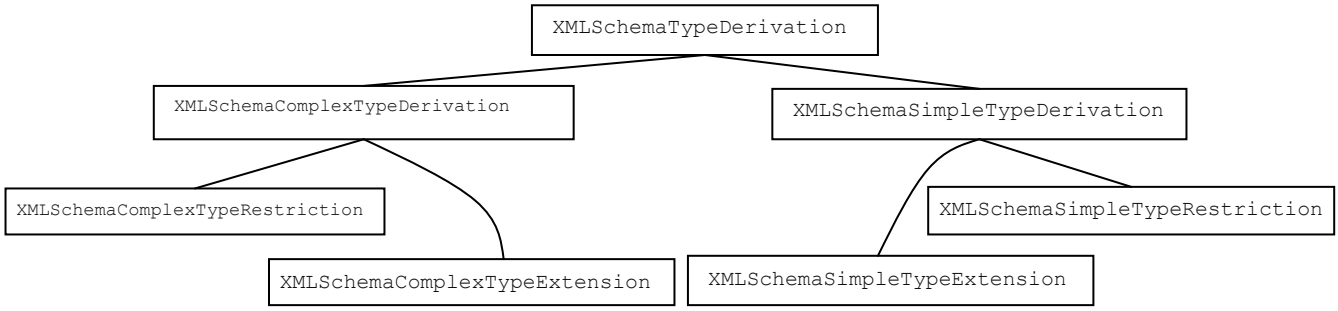

Figure 5 XSD Type Derivation

There are two types of simple type derivation. Simple type derivation by restriction requires specification of a set of constraining facets:

```
interface XMLSimpleTypeRestriction:
            XMLSchema SimpleTypeDerivation {
  XMLSchemaSet<XMLFacet> facets();
}
```

Simple type extension allows only additional attributes:

```
interface XMLSchemaSimpleTypeExtension:
            XMLSchemaSimpleTypeDerivation {
  XMLSchemaSet<XMLSchemaAttribute>
                              attributes();
}
```

In a complex type derivation the base type is complex, hence the result type of the method `base` is overridden covariantly. In a complex type derivation additional attributes may be added and the new particle structure is specified:

```
interface XMLSchemaComplexTypeDerivation:
                  XMLSchemaTypeDerivation {
  new XMLSchemaComplexType  base();
  XMLSchemaSet<XMLSchemaAttribute>
                             attributes();
  XMLSchemaParticle  particle();
}
```

A complex type extension amounts to extending the particle structure of the base type. The new particle structure is a sequence group, the first component of which is the base particle, and the rest are particles that are appended.

```
interface XMLSchemaComplexTypeExtension:
           XMLSchemaComplexTypeDerivation {
 new XMLSchemaSequenceGroup particle();
}
```

In a complex type restriction changes may be made to the attributes and the particle structure of the base is restricted by restricting the ranges of occurrences or omitting optional elements:

```
interface XMLSchemaComplexTypeRestriction:
           XMLSchemaComplexTypeDerivation {
//restricted attributes and particle structure
}
```

XSD allows specification of typical database integrity constraints such as uniqueness, keys and referential integrity. In XSD these constraints are called identity constraints, modeled by an XSD schema interface `XMLSchemaIdentityConstraint` given below.

An identity constraint has a name, a selector that specifies the XML structure for which the constraint holds, and a sequence of fields whose values will have the desired property. The selector is specified by a simple XPath expression. These expressions will be instances of the type XMLPath:

```
interface XMLSchemaIdentityConstraint:
                              XMLSchemaObject {
   XMLString name();
   XMLSchemaSequence<XMLString> fields();
   XMLPath selector();
}
```

The uniqueness and key constraints require nothing else in their specification. A referential integrity constraint requires an additional reference to a key which is given by the key name:

```
interface XMLSchemaKeyRef:
              XMLSchemaIdentityConstraint {
   XMLString referTo();
}
```

6. Related work

One of the first object-oriented models of XML was DOM. Although it is a part of W3C activities, DOM preceded XSD and hence it is very limited in its support of XSD. A tool that works with DOM and its Java version JDOM is JAXP [7] which is a Java API for XML processing.

LINQ to XML is an object-oriented interface to XML data that is based on the assumption that an XML schema is not available [11]. This approach requires extensive type casting and hence dynamic type checking of both imperative code and LINQ queries (which LINQ to XML also supports).

LINQ to XSD [12] has a variety of techniques for representing some structural features of XSD such as sequence groups, type derivation by inheritance, etc. However, it does not represent the notion of a particle with range constraints, and it does not distinguish between the type and the particle hierarchies in XSD. Type derivation by restriction and the identity constraints are not represented either because they are based on constraints.

Paper [15] attempts to present the essence of XSD but is not object-oriented. This approach does not represent particle structures with general range constraints, type derivation by restriction in general, or identity constraints.

The .NET Schema Object Model (SOM) is the most accurate and object-oriented representation of XSD that we know of [14]. Given an XSD schema SOM produces its object-oriented representation which we use in our approach. However, the complexity of SOM is prohibitive for typical application

programmers. Lack of parametric polymorphism in SOM creates undesirable representation problems which we do not have.

Data Contracts [10] is a system based on SOM, but it has nontrivial limitations as to what kind of XSD schema features it can handle. For example, it cannot handle attributes and it can handle only certain object types whose structure is such that this system can map them to XSD types.

XML Data Binder [19] also maps XSD schemas into a collection of classes that could be in Java, C#, C++, and VB, and generates code for those classes for access and update methods. However, it will not handle representation of general particle structures and the XML Schema type hierarchy with type derivations by restriction and extension. As in most other approaches, XML Data Binder has no way of representing range of occurrences constraints of a general form or the identity constraints.

XML Beans [18] seems to have a more elaborate and more accurate representation of XML Schema in comparison with XML Data Binder. For example, this applies to representation of XML Schema groups and XML Schema types. XML Beans also has a structural representation of the identity constraints, similar to ours. However, XML Beans will have the same problems as XML Data Binder in representing the range constraints or type derivation by restriction in general.

An analysis of the mismatch between XML and object-oriented languages is presented in [9]. LINQ to XSD in fact follows some of the representation options from [9]. The main difference between our work and [9] is that we represent explicitly and accurately the structural core of XSD, its particle (elements and groups) and type hierarchies. In addition, at the meta level we represent content models, type derivations, and the identity constraints which are missing in most other approaches (SOM is an exception).

In a separate paper [1] we present the formal rules for mapping XSD to OO Schemas and the algorithm for mapping instances accordingly. Neither is available in any of the published work that we are familiar with.

The only work we know of that goes beyond the limitations of type systems is [2][3][4]. This research is based on object-oriented constraint languages such as the Java Modeling Language [8] or Spec# [13]. It is thus able to represent XSD constraint-related features such as general range constraints for particles, type derivation by restriction, semantics of different types of groups (sequence versus choice), and identity constraints (keys and referential integrity).

7. Conclusions

As a rule, object-oriented application programmers have very limited understanding of what XML Schema is all about. The reason is the complexity of XSD and its mismatch with object-oriented languages. Our contribution is the design of an object-oriented interface to the structural core of XSD which has not been available so far. The presented collection of interfaces constitutes a library which database designers, object-oriented application programmers, and users writing queries can understand and use in developing their applications that manage data that conforms to XSD.

The XSD core comes with a collection of formal rules based on type systems for mapping source XSD schemas into object-oriented schemas along with an algorithm for mapping XML instances to their object-oriented representation, described in [1]. The XSD core is also a basis for a more general model based on object-oriented assertion languages that allow representation of XSD constraints and more general schema integrity constraints that transactions are required to obey, described in [4].

8. References

[1] S. Alagic and P.A. Bernstein, Mapping XSD to OO schemas, Proc. of ICOODB, LNCS, 2009.

[2] S. Alagic, M. Royer, and D. Briggs, Verification theories for XML Schema, Proceedings of BNCOD, LNCS 4042, pp. 262-265, 2006.

[3] S. Alagic, M. Royer, and D. Briggs, Program verification techniques for XML Schema-based technologies, Proc. of the ICSOFT Conference, Vol. 2, pp. 86 - 93, 2006.

[4] S. Alagic, M. Royer, and D. Briggs, Verification technology for Object-oriented/XML Transactions, Proc. of ICOODB, LNCS, 2009.

[5] G. Bierman, E. Meijer, and W. Schulte, The essence of data access in C/omega, Microsoft Research, 2004.

[6] Document Object Model (DOM), http://www.w3.org/TR/REC-DOM-Level-1/.

[7] All about JXAP, http://www.ibm.com/developerworks/java/library/x-jaxp/.

[8] Java Modeling Language, http://www.eecs.ucf.edu/~leavens/JML/.

[9] R. Lammel and E. Meijer, Revealing the X/O impedance mismatch, Microsoft Corp., 2007, http://homepages.cwi.nl/~ralf/xo-impedance-mismatch/paper.pdf.

[10] Microsoft Corp., Using Data Contracts, http://msdn.microsoft.com/en-us/library/ms733127.aspx.

[11] Microsoft Corp., LINQ to XML, http://msdn.microsoft.com/en-us/library/bb387098.aspx.

[12] Microsoft Corp., LINQ to XSD Preview Alpha 0.2 Refresh, http://www.microsoft.com/downloads/details.aspx?FamilyID=A45F58CD-FCFC-439E-B735-8182775560AF&displaylang=en.

[13] Microsoft Corp., Spec#, http://research.microsoft.com/specsharp/.

[14] Microsoft Corp., "XML Schema Object Model (SOM)," http://msdn2.microsoft.com/en-us/library/bs8hh90b(vs.71).aspx

[15] J. Simeon and P. Wadler, The Essence of XML, Proc. of POPL 2003, ACM, pp. 1-13, 2003.

[16] C# , Version 3.0 specification, Microsoft.

[17] LINQ to XSD, Microsoft, 2007, http://blogs.msdn.com/xmlteam/archive/2006/11/27/typed-xml-programmer-welcome-to-LINQ.aspx.

[18] XMLBeans, http://xmlbeans.apache.org.

[19] XML Data Binder, http://www.liquid-technologies.com/XmlStudio/Xml-Data-Binder.aspx.

[20] W3C: XML Schema 1.1, http://www.w3.org/XML/Schema.