Mapping XSD to OO Schemas

Suad Alagić and Philip A. Bernstein
Microsoft Research
One Microsoft Way
Redmond, WA, USA
alagic@cs.usm.maine.edu, philbe@microsoft.com

ABSTRACT

This paper presents algorithms that make it possible to process XML data that conforms to XML Schema (XSD) in a mainstream object-oriented programming language. These algorithms are based on our previously developed object-oriented view of the core of XSD. The novelty of this view is that it is intellectually manageable for object-oriented programmers while still capturing the complexity of the core structural properties of XSD. This paper develops two mappings based on this view. The first one is specified by a set of rules that map a source XSD schema into its object-oriented schema. The second one maps XML instances that conform to an XSD schema to their representation as objects. In addition to mapping elements and attributes, these mappings reflect correctly the particle structures including different types of groups, and type derivation by restriction and extension. The structural properties of identity constraints are also mapped correctly. Formally defined mappings or algorithms of this sort have not been available so far, and existing industrial tools typically do not handle the level of complexity of XSD that our mappings do.

1. INTRODUCTION

1.1 The Problem

XML Schema (XSD for short) is a standard for specifying structural features of XML data [14]. In addition, XSD allows specification of constraints that XML data is required to satisfy. Application programmers are faced with the problem of processing data that conforms to XSD in a general-purpose object-oriented programming language. For this to be possible, an object-oriented interface to XML data must be available to application programmers.

To enable this scenario we need a schema mapping that translates each XSD schema X into a corresponding object-oriented schema O. The schema mapping from X to O creates the object-oriented interface for application programmers. We also need an instance mapping between instances of X (i.e., XML documents) and instances of O (i.e., sets of objects). The instance mapping is used to translate XML documents into objects that can be manipulated by applications, and to translate objects that are created or modified by applications back into XML documents.

Developing such translations poses nontrivial problems due to the mismatch of the core XSD features and the features that are expressible in type systems underlying mainstream object-oriented languages. All object-oriented interfaces to XML suffer the implications of this mismatch [7].

One implication is that XML data conforming to an XSD schema can be manipulated in its object-oriented representation to produce data that, when translated back into XML, no longer conforms to the XSD schema. A related implication is that two distinct XSD schemas may be represented by the same object-oriented interface, making it impossible for the application programmer to predict the exact nature of the XML documents that her program will produce by looking only at the object-oriented interface.

The starting point is a user's XSD schema. An off-the-shelf XSD schema compiler is used in our approach to translate the user's schema into an object-oriented representation, such as .NET's XML Schema Object Model (SOM) [12]. The schema mapping rules translate the user's XSD schema into object-oriented interfaces. These interfaces comprise the user's programming model. They are a combination of predefined interfaces that are based only on XSD itself and user-schema-specific interfaces that are generated from a user's XSD schema. A program can use these interfaces to access pieces of an XML document that conforms to the XSD schema. Enabling this access requires that there be a mapping that translates an XML document into objects whose classes implement the generated XSD interfaces.

1.2 Motivating example

To motivate some of the detailed problems that need to be solved by such a system, let us consider how to map an example XSD schema into object-oriented (OO) interfaces.

Consider the complex type `DictionaryType` defined in the XSD schema above. The structure of this type is defined as a sequence group where the number of elements in the sequence ranges from zero to an arbitrary and unspecified natural number. An OO representation of this type will obviously be based on a parametric type of sequence or list. However, as soon as we specify a type `SmallDictionaryType` that is derived by restriction from `DictionaryType`, we encounter a nontrivial problem.

```
<xsd:schema id="XMLDictionarySchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd: complexType name="DictionaryType"/>
  <xsd: sequence>
   <xsd:element name="item" type="ItemType"
             minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="ItemType">
     <xsd:sequence>
       <xsd:element name="typeOfEntity"
                    type="EntityType" />
     </xsd:sequence>
     <xsd:attribute name="key" type="xsd:string" />
 </xsd:complexType>
 <xsd:element name="dictionary"
              type="DictionaryType">
       <xsdkey name="searchKey">
         <xsd:selector xpath="item"/>
         <xsd:field xpath="@key"/>
       </xsdkey>
 </xsd:element>
 <xsd:complexType name="SmallDictionaryType"/>
   <xsd:restriction base= "DictionaryType"/>
     <xsd:sequence>
       <xsd:element name="item" type="ItemType"
                  minOccurs="1" maxOccurs="1000"/>
     </xsd:sequence>
   </xsd:restriction>
 </xsd:complexType>
 <xsd:complexType  name="AddressType"/>
   <xsd:extension base="ItemType"/>
     <xsd:sequence>
       <xsd:element name="firstName"
                    type="xsd:string"/>
       <xsd:element name="lastName"
                    type="xsd:string"/>
       <xsd:choice>
         <xsd:element name="POboxAddress"
                      type="xsd:string"/>
         <xsd:element name="streetAddress"
                      type="streetAddressType"/>
       </xsd:choice>
     </xsd:sequence>
    </xsd:extension>
 </xsd:complexType>
</xsd:schema>
```

XSD type definitions are represented in the OO schema by inheritance. But in this case, the structural specification of `DictionaryType` and `SmallDictionaryType` is identical. What is different is that the range-of-occurrences constraint has been strengthened. OO type systems cannot represent this constraint and hence cannot represent type derivation by restriction in XSD. Well-known OO interfaces to XSD typically lack any suitable representation of this type of construct and of type derivation by restriction as defined in XSD.

Consider now an actual dictionary specified as an XSD element. This element type will also be represented as an object type (a class or an interface). One problem of existing OO interfaces to XML is not distinguishing between two type hierarchies in XSD as in [10]. One hierarchy represents the actual instances, starting with elements. But an element has a name (i.e., a tag) and a value. The value of an element may be simple or complex, and hence belongs to a type that is defined in a different hierarchy of XSD types, all of which are derived from the root type `anyType`.

Yet another subtlety in this example is that a dictionary element is equipped with a key constraint, a typical database constraint that captures the essential semantics of a dictionary. Well-known OO interfaces to XML do not consider representation of this constraint. In fact, key constraints are not representable in OO type systems.

Now consider `ItemType`, which is the type of dictionary elements. Its complex structure is specified as a sequence group. In addition, this type is equipped with an attribute key. In the most straightforward representation of `ItemType` its corresponding object type will have properties `typeOfEntity` and `key`. This seems to be a preferred OO user view of `ItemType`[10]. However, it comes with nontrivial problems.

The first problem is the lack of distinction between elements and attributes. The second is that in XML two elements or an element and an attribute may have the same name. In those situations the straightforward representation does not work because the names of elements and attributes cannot be the property names in the corresponding object type, as they must be unique.

Consider now a specific item type `AddressType` of a dictionary. This type will be specified in XSD as derived by extension from the type `ItemType`. This type derivation has a fairly accurate representation by inheritance. The extension is specified as a sequence group with one subtlety. The third component of the sequence is specified as a choice-group, so an XML instance has either `POboxAddress` or `streetAddress` but not both.

XSD choice represents a major problem for OO interfaces to XML. Specifying a fixed number of subtypes of a type is contrary to the core features of the OO model. Because of the lack of a suitable representation for choice, some OO interfaces use the same representation for choice and sequence groups. This representation has nontrivial implications because these two types of groups have different semantics. In fact, widely known OO interfaces to XML do not have a suitable representation of XSD groups and its three subtypes (i.e., sequence, choice, and all groups).

There are many more problems in mapping XSD schemas to OO schemas [7]. The above analysis outlines the most important structural problems as we see them and the mismatch between the two models. Many of these problems can be resolved by an OO constraint language, i.e., a much more expressive formal system than an OO type system underlying the mainstream OO languages. But OO constraint languages are not supported for the mainstream OO languages, and their underlying technology is significantly more complex and still under development. We refer to some approaches like these in Section 5.

The challenge that we face in this paper is to produce an OO representation of the core of XSD within the limitations of OO type systems. The existing interfaces fail to represent accurately the core structural features of XSD, which can compromise data integrity. In addition, OO programmers do not have an OO view of XSD that communicates the

core structural and semantic features of XSD.

To see the implications of structural misrepresentation of an XSD schema in its corresponding OO schema, assume that we have a database that conforms to an XSD schema. Suppose that application programs will be developed in an OO language and will be based on the OO representation of XSD schema. Manipulating OO representations of XSD instances will now lead to object structures that do not reflect the structure of XSD instances. Installing the updated XSD instances presents a huge challenge because of this structural mismatch. This is why our goal is to produce an OO representation of XSD that is as structurally accurate as possible, so that manipulating OO instances does not violate structural and semantic constraints of XSD.

1.3 Contributions and outline

In our previous work we isolated the structural core of XSD, which contains the essential structural features of XSD and abstracts away a variety of other XSD features [1]. However, that work gives an intuitive description with examples, but no algorithms, formal syntax or formal semantics. This paper complements [1] with a more rigorous technical development. The main research contributions of this paper are as follows:

- We specify the syntax for the XSD core (in Section 2).
- We specify the rules for mapping an XSD core schema to its corresponding OO schema (Section 3).
- We specify an algorithm for mapping instances that conform to a source XSD core schema to their OO counterparts (Section 4).

Formally defined mappings or algorithms of this sort have not been available so far, and existing industrial tools typically do not handle the level of complexity of XML Schema that our mappings do.

Related work is described in Section 5, and Section 6 is the conclusion.

2. XML SCHEMA CORE

2.1 Syntactic specification

In this section we define a subset of XSD, which we call the XSD core [1]. It is comprised of features that we regard as essential to XSD and is the focus of our mapping from XSD schemas to OO interfaces.

In the XSD core, attributes and elements are specified as (Name, Type) pairs. Name stands for the tag and Type for the type of the associated value. The type of an attribute is required to be simple, and the type of an element may be either simple or complex.

$Attribute ::= Name\ simpleType$
$Element ::= Name\ Type$

The key notion of the XSD core is that of a *Particle*, which is a term followed by the range of occurrences. A term is either an *Element* or a *Group*. So a particle is a sequence of repeated terms where the number of occurrences of the term is between *minOccurs* and *maxOccurs*.

$Particle := Term\ Range$
$Term := Element \mid Group$
$Range := [minOccurs][maxOccurs]$

There are three types of groups in the core: sequence-groups, choice-groups and all-groups. A sequence-group is specified as a sequence of particles. The same applies to a choice-group, but its semantics is different. In a sequence-group all particles must be present, while in the choice-group exactly one of them must be present in a document fragment that conforms to the group definition. Particles of an all-group are of a particular type: they are elements. So an all-group is specified as a sequence of elements.

$Group := Sequence \mid Choice \mid All$
$Sequence ::= Particle\{Particle\}$
$Choice ::= Particle\{Particle\}$
$All ::= Element\{Element\}$

A type is either simple or complex.

$Type ::= [Name]\ simpleOrComplexType$
$simpleOrComplexType ::= simpleType \mid complexType$

A simple type either is a built-in type or is derived from another simple type (its base) by simple type restriction.

$simpleType ::= builtInType \mid$
$\qquad simpleType\ simpleTypeRestriction$

A simple type restriction is specified by a sequence of facets.

$simpleTypeRestriction ::= facet\{facet\}$

Facets include direct enumeration, specification of ranges of values, and specification of patterns of regular expressions. All of these facets specify the values belonging to the restricted type.

$facet ::= enumeration \mid range \mid regExpression$

A complex type is derived from its base type (denoted by Type below) by a complex type derivation:

$complexType ::= Type\ complexTypeDerivation$

There are three kinds of complex type derivation:

$complexTypeDerivation ::= simpleTypeExtension \mid$
$complexTypeExtension \mid complexTypeRestriction$

Simple type extension applies to complex types with simple content. Since the content is simple, the only allowed extension is adding attributes. Hence, this form of type derivation by extension is specified by a sequence of additional attributes.

simpleTypeExtension ::= {*Attribute*}

Complex type extension includes both extending the set of attributes and extending the particle structure of the base type. The extended particle structure is specified by a group. This group is obtained by forming a sequence-group of the base type particle of the complex type and appending additional particles specified in the complex type derivation.

complexTypeExtension ::= {*Attribute*} *Group*

Complex type restriction allows restriction of the base type by a set of facets, making changes in the set of attributes of the base type, and restricting the constraints in the particle structure of the base type. The particle structure may omit optional elements. Otherwise it remains the same, hence it is repeated, but the constraints will be different. An exception is omitting optional elements. The particle structure obtained this way is specified as a group.

complexTypeRestriction ::= {*Facet*}{*Attribute*} *Group*

There are three types of identity constraints in XSD: uniqueness, key and referential integrity (foreign key) constraints. An identity constraint consists of a specification of the key fields along with the scope to which the constraint applies. This scope is specified by an *XPath* expression.

identityConstraint ::= *Name field* {*field*} *path*

In addition, a referential integrity constraint contains specification of the key constraint to which it refers.

Specification of a schema includes its name and sets of global elements, types, attributes, groups and identity constraints:

Schema ::= *Name Element*{*Element*}{*Type*}{*Attribute*}
{*Group*}{*identityConstraint*}

Both sets and sequences in this syntactic specification are represented as sequences.

2.2 Core interfaces

The library of predefined interfaces includes the two type hierarchies presented by the diagrams in figures 1 and 2. Figure 1 represents the particles as defines in XSD. Since the range constraint may be associated with any type of a term, in a slightly simplified representation that follows SOM [12], elements and groups are viewed as subtypes of the particle type. Specific types of groups are defined as subtypes of the group type. The range constraints are specified by methods `minOccurs` and `maxOccurs` of the particle interface.

All types are derived from XSD `anyType` as shown in Figure 2. We specify two subtypes of `anyType` that stand for simple and complex types. Specific simple and complex types will be derived from those.

The above two hierarchies are related. Since a complex type will in general be equipped by a set of attributes and a particle structure, it will in general refer to the types specified in the particle hierarchy.

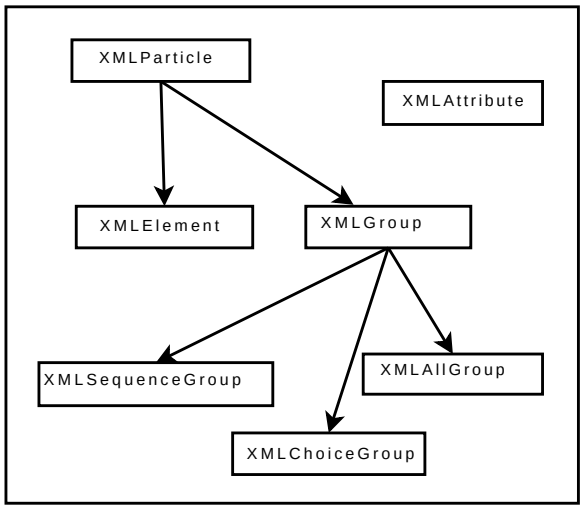

Figure 1: XSD particle hierarchy

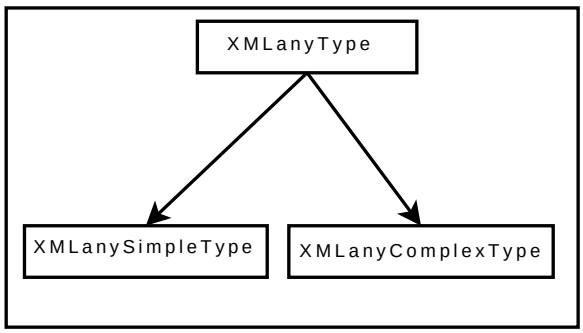

Figure 2: XSD type hierarchy

The third type hierarchy represents XSD identity constraints, shown in Figure 3. In this paper we do not consider the implications of using a constraint language such as JML or Spec#, so that the representation of constraints is necessarily structural.

3. MAPPING SCHEMAS

This section presents an algorithm for mapping XSD schemas to OO interfaces. The algorithm assumes that the source XSD schema is valid. Its representation could be of any form, as long as its XSD schema components are available via correctly typed expressions such as:

XMLElement(elementName, typeName),
XMLAtttribute(attributeName,typeName),
XMLsimpleType(baseTypeName, typeName, facets)

and the like. For example, the first expression above indicates that the source XSD schema contains an element type whose name is *elementName* and the type of its value is *typeName*. For concreteness, we use the representation of the source XSD schema that would be generated by SOM [12].

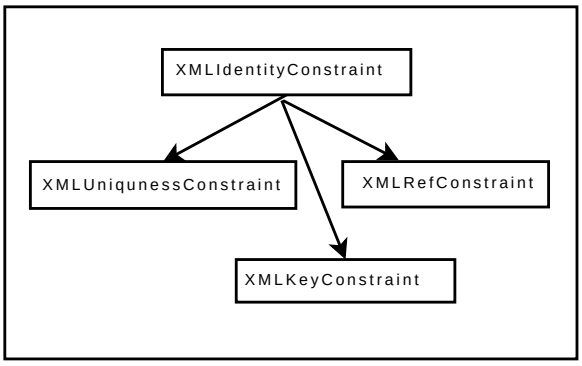

Figure 3: XSD constraints hierarchy

The source XSD schema contains global elements, attributes, types, groups, etc. The algorithm is a set of mapping rules, each of which specifies how to map one of these source constructs to its OO representation. For each rule, we specify the typing assumptions under which the rule applies. The typing assumptions follow from our assumption that the XSD schema is valid. They are specified as typing rules with respect to a typing environment.

The typing environment, denoted by $\mathcal{T}$, includes facts about the types in an XSD schema. In particular, it includes a mapping from names to types and sub-typing relationships. The fact that an identifier id has a type typeName in the environment $\mathcal{T}$ is expressed as

$\mathcal{T} \vdash id : typeName.$

Since the inheritance relationships are identified with subtyping in mainstream object-oriented languages, we will use the subtyping symbol $<:$ in the typing rules. So if *typeName* is the name of an XML type, the typing environment will allow the following deduction, which says that the XML type *typeName* is a subtype of *XMLanyType* in $\mathcal{T}$:

$\mathcal{T} \vdash typeName <: XMLanyType$

The typing environment is initialized with the core interfaces such as *XMLElement*, *XMLAttribute*, *XMLParticle*, *XMLGroup*, *XMLSequenceGroup*, *XMLChoiceGroup*, *XMLAllGroup*,*XMLanySimpleType*, *XMLanyComplexType*, etc.

3.1 Mapping elements and attributes

First consider mapping element types and attribute types from the source XSD into OO interfaces. If *typeName* stands for an object representation of an XML type, and *elementName* is a valid name, then the expression *XMLElement(elementName, typeName)* is well typed and its type is *XMLElement*:

$$\frac{\mathcal{T} \vdash typeName <: XMLanyType,\ \mathcal{T} \vdash elementName : NameType}{\mathcal{T} \vdash XMLElement(elementName, typeName) : XMLElement}$$

The above conditions summarize the typing assumptions about an element type in the source XSD schema. The conditions are the consequence of the fact that the source XSD schema has been validated. If e is a well typed expression *XMLElement(elementName,typeName)*, then its object-oriented image is map(e):

$$\frac{e = XMLElement(elementName, typeName)}{map(e) = interface\ elementName : XMLElement\ \{\ NameType\ name();\ typeName\ value()\}}$$

The typing and mapping rule for the attribute types follows the same pattern except the value of an attribute must be of a simple type so that we would have the following in the corresponding typing rule:

$\mathcal{T} \vdash typeName <: XMLanySimpleType$

3.2 Mapping types

If the source XSD schema contains a specification of an XML simple type whose name is *typeName*, then this type will in general be derived by restriction from its base type which is also simple. The set of constraining facets must also be specified in the source schema. Hence the information about a simple type in the source schema is summarized in an expression of the form *XMLanySimpleType(baseTypeName, typeName,facets)*.

The conditions under which *XMLanySimpleType(baseTypeName,typeName,facets)* is a well typed expression of type *XMLanySimpleType* that are based on the assumed validation are as follows:

$$\frac{\mathcal{T} \vdash baseTypeName <: XMLanySimpleType,\ \mathcal{T} \vdash typeName : NameType,\ \mathcal{T} \vdash facets : XMLSet < XMLFacet >}{\mathcal{T} \vdash XMLanySimpleType(baseTypeName, typeName, facets) : XMLanySimpleType}$$

If T is a well-typed expression

XMLanySimpleType(baseTypeName, typeName, facets)

then its object-oriented image $map(T)$ is:

$$\frac{T = XMLanySimpleType(baseTypeName, typeName, facets)}{map(T) = interface\ typeName :\ baseTypeName\ \{\ XMLSet < XMLFacet >\ facets()\}}$$

A complex XML type is derived from some other type, its base type. In the simplest case the base type is simple and the complex type is obtained by adding attributes. So in this case the information coming from the XSD source will be given by an expression *XMLanyComplexType(baseTypeName, typeName, attributes)*. The typing constraints for *XMLanyComplexType(baseTypeName, typeName, attributes)* that follow from the fact that the source has been validated are:

$$\frac{\mathcal{T} \vdash baseTypeName <: XMLanySimpleType,\ \mathcal{T} \vdash typeName : NameType,\ \mathcal{T} \vdash attributes : XMLSet < XMLAttribute >}{\mathcal{T} \vdash XMLanyComplexType(baseTypeName, typeName, attributes) : XMLanyComplexType}$$

If T is an expression

XMLanyComplexType(baseTypeName, typeName, attributes),

then its object-oriented image $map(T)$ is:

$$\frac{T = XMLanyComplexType(baseTypeName, typeName, attributes)}{map(T) = interface\ typeName :\ baseTypeName\ \{ XMLSet < XMLAttribute > attributes()\}}$$

In a more complex specification of an XML complex type, the base type is complex, and the type derivation includes a set of attributes and a new particle structure obtained either by extending the particle of the base type or restricting its range constraints. So the information about a complex XML type coming from the source XSD schema is assumed to have the form of an expression *XMLanyComplexType(baseTypeName, typeName, attributes, particleType)*. The typing constraints for this expression that follow from its validation are:

$$\frac{\mathcal{T} \vdash baseTypeName <: XMLanyComplexType,\ \mathcal{T} \vdash typeName : NameType,\ \mathcal{T} \vdash attributes : XMLSet < XMLAttribute >,\ \mathcal{T} \vdash particleType <: XMLParticle}{\mathcal{T} \vdash XMLanyComplexType(baseTypeName, typeName, attributes, particleType) : XMLanyComplexType}$$

If T is a well typed expression
XMLanyComplexType(baseTypeName, typeName,attributes, facets,particleType), then its object-oriented image map(T) is:

$$\frac{T = XMLanyComplexType(baseTypeName, typeName, attributes, particleType)}{map(T) = interface\ typeName :\ baseTypeName\{ XMLSet < XMLAttribute > attributes();\ particleType\ particle()\}}$$

If the type is derived by restriction, a set of facets is also specified. So the information about a complex XML type coming from the source XSD schema is assumed to have the form of an expression *XMLanyComplexType(baseTypeName, typeName, attributes, facets, particleType)*. The typing constraints for this expression that follow from its validation are:

$$\frac{\mathcal{T} \vdash baseTypeName <: XMLanyComplexType,\ \mathcal{T} \vdash typeName : NameType,\ \mathcal{T} \vdash attributes : XMLSet < XMLAttribute >,\ \mathcal{T} \vdash facets : XMLSet < XMLFacet >,\ \mathcal{T} \vdash particleType <: XMLParticle}{\mathcal{T} \vdash XMLanyComplexType(baseTypeName, typeName, attributes, facets, particleType) : XMLanyComplexType}$$

If T is a well typed expression

XMLanyComplexType(baseTypeName, typeName, attributes, facets, particleType), then its object-oriented image $map(T)$ is:

$$\frac{T = XMLanyComplexType(baseTypeName, typeName, attributes, particleType)}{map(T) = interface\ typeName :\ baseTypeName\ \{ XMLSet < XMLAttribute > attributes();\ XMLSet < XMLFacet > facets();\ particleType\ particle()\}}$$

3.3 Mapping groups

If the source XSD schema contains a group, the first piece of information that is available is the type of the group. In addition, a group specifies a sequence of particles, which in the case of an all-group are elements. In the rules for groups we assume that a group has a name (as global groups do) so that information from the source for a sequence-group has the form of an expression *XMLSequenceGroup(groupName, particles)*. The typing constraints for an expression *XMLSequenceGroup(groupName, particles)* that follow from the assumed validation are:

$$\frac{\mathcal{T} \vdash groupName : NameType,\ \mathcal{T} \vdash particles : XMLSequence < XMLParticle >}{\mathcal{T} \vdash XMLSequenceGroup(groupName, particles) : XMLSequenceGroup}$$

If g is a well typed expression

XMLSequenceGroup(groupName, particles)

then its object-oriented image $map(g)$ is:

$$\frac{g = XMLSequenceGroup(groupName, particles)}{map(g) = interface\ groupName : XMLSequenceGroup\ \{ XMLSequence < XMLParticle >\ particles()\}}$$

A choice-group is also specified in the source XSD schema as a sequence of particles. *XMLChoiceGroup(groupName, particles)* is a well typed expression of type *XMLChoiceGroup* under the following conditions:

$$\frac{\mathcal{T} \vdash groupName : NameType,\ \mathcal{T} \vdash particles : XMLSequence < XMLParticle >}{\mathcal{T} \vdash XMLChoiceGroup(groupName, particles) : XMLChoiceGroup}$$

If g is a well typed expression *XMLChoiceGroup(groupName, particles)* then its object-oriented image $map(g)$ is:

$$\frac{g = XMLChoiceGroup(groupName, particles)}{map(g) = interface\ groupName : XMLChoiceGroup\ \{ XMLSequence < XMLParticle >\ particles()\}}$$

The only difference in the specification of an all-group is that in its sequence of particles, the particles must be elements.

3.4 Mapping identity constraints

The above developed mapping framework allows specification of mapping rules for the XSD identity constraints, a feature missing in just about all other approaches. The approach presented in this paper cannot express the semantics of the identity constraints, but it makes it possible to map their structural specification.

The typing information about an identity constraint coming from a validated specification of such a constraint in the source XML schema is summarized in the rule given below:

$$\frac{\mathcal{T} \vdash name : NameType,\ \mathcal{T} \vdash fields : XMLSequence < XMLString >,\ \mathcal{T} \vdash path : XMLPath}{\mathcal{T} \vdash XMLIdentityConstraint(Name, fields, path) : XMLIdentityConstraint}$$

The corresponding mapping rule that maps a constraint c into its corresponding object-oriented interface is:

$$\frac{c = XMLIdentityConstraint(Name, fields, path)}{map(c) = interface\ Name : XMLIdentityConstraint\ \{ XMLSequence < String >\ fields();\ XMLpath\ path()\}}$$

The above rules apply to the uniqueness and key constraints. Referential key constraint is trivially more complex as it contains specification of a key constraint to which is refers.

4. MAPPING INSTANCES

The mapping rules for schemas and documents are clearly independent of the underlying implementation platform. But the XSD core may be viewed as an abstraction on top of SOM [11] which is our implementation platform. Given an XSD schema, SOM will process it and make its OO representation available. This is why in the algorithm for mapping instances we assume that the source schema has been mapped to the target OO schema according to the rules in Section 3. We also assume that the source XML instances have been validated with respect to the source XSD schema. So the presented algorithm implements the map from XML instances to their corresponding OO instances according to figure 4.

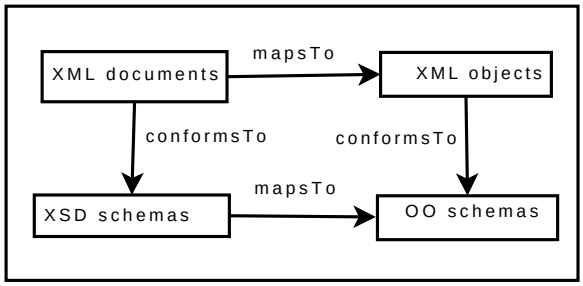

Figure 4: Mapping schemas and objects

Since the source schema has been mapped to the OO schema, the algorithm will consult the OO schema for the schema information required to correctly map the source XML instances to the corresponding objects. The information in the core interfaces in Section 2 is available both at the schema level and in the programming language interface. The distinction between the two levels will be indicated by the prefix `Schema` for the interfaces at the schema level.

A given object instance will in general represent a particle object. A whole document will be represented as an element object. The complete structural representation is available using reflection. From a particle object one can get all the information required to generate a valid XSD particle recursively from a sequence of sub-particles. The recursion terminates when a particle of type element whose type is simple is reached. For a particle object that represents an element, the name and the type will be available from the element object. If the type of an element is derived from XML complex type, the actual complex particle structure of the element will be discoverable from the corresponding object type information. The object type representing an XML complex type contains a specification of the underlying particle structure which is how the algorithm gets invoked recursively.

The source XML instance is assumed to be available via a collection of methods of the class `Input`. The source XML document consists of a single element. Hence invocation of the method `mainDocument` will create a single element object. The tag of the element will be retrieved from the input, and then the `createElementObject` method will be invoked with the element tag as its argument.

```
XMLElement mainDocument() {
  string tag =Input.getTag();
  return(createElementObject(tag))
}
```

The tag of the input element is used to access the schema information. The type of the element value is looked up in the schema. If the type is simple, the value of the element is taken from the input. If the type is complex, then an object of the type `XMLanyComplexType` will be created as the element's value. This is accomplished by invoking the method `createComplexValueObject` which parses the complex structure of the input element.

```
XMLElement createElementObject(string tag) {
    Schema.XMLanyType type = Schema.lookUpType(tag);
    if type  <:  Schema.XMLanySimpleType then
    XMLanySimpleType value =
              (XMLanySimpleType)Input.getValue();
    else  XMLanyComplexType value =
createComplexValueObject(
                    (Schema.XMLanyComplexType)type);
    return newInstance(getClass("XMLElementClass"),
                           []Object{tag,value});
}
```

Mapping an attribute instance follows a similar logic, but it is much simpler because the value of an attribute is of a simple type. The tag of the input attribute is taken from the input and used to look up the type of the value which must be simple. The value of the attribute is then taken from the input and a new attribute instance is created. There are some obvious typing details that are omitted here as well as in reading the value of an element from the input. The appropriate procedure for a specific simple type should be used and the result type cast to the specific simple type.

```
XMLAtttribute createAttributeObject(string tag)
{Schema.XMLanySimpleType type =
                            Schema.lookUpType(tag);
  XMLanySimpleType value =
              (XMLanySimpleType)Input.getValue();
  return newInstance(getClass("XMLAttributeClass"),
                         []Object{tag, value});
}
```

Classes such as `XMLElementClass` and `XMLAttributeClass` are not available in the user interface. Even in the above cases specific element and attribute classes will be used which would in fact happen if reflection is used. We do not show it since it would make the algorithm less readable.

The method `createComplexValueObject` has a complex schema type as its argument. The newly created instance will have two components: a set of attributes (which we represent as a sequence) and a particle structure. These two components are created invoking the methods `createAttributesObject` and `createParticleObject`, and then the object representing complex element value will be created.

```
XMLanyComplexType  createComplexValueObject(
                  Schema.XMLanyComplexType type){
 XMLSequence<XMLAttribute> attributes =
       createAttributesObject(type.attributes());
 XMLParticle particle =
           createParticleObject(type.particle());
 return newInstance(getClass("XMLanyComplexType"),
                 []Object{attributes, particle})
}
```

Creating an object that represents a set of attributes in the input instance that belongs to a value of a complex type requires getting successive tags invoking the method `createAttributeObject` for each tag. The created attribute objects are appended to a sequence of attribute objects.

```
XMLSequence<XMLAttribute> createAttributesObject(
   XMLSet<Schema.XMLAttribute> attributes) {
  XMLSequence<XMLAttribute> attributes =
              new XMLSequence<XMLAttribute>();
  for (Schema.XMLAttribute a in attributes) {
    string tag = Input.getTag();
    XMLAttribute attribute =
                   createAttributeObject(tag);
    attributes.append(attribute); }
  return attributes;
  }
```

The second component of a complex value object is a particle object that conforms to the `XMLParticle` type. This object is constructed invoking the method `createParticleObject` which takes an argument of type `Schema.XMLParticle` so that it will have the source schema specification of the particle that is coming up in the input. An object of type `XMLParticle` will have the range specified by `minOccurs` and `maxOccurs` according to the schema information, and a sequence of particles coming up in the input whose number of occurrences will be determined by `minOccurs` and `maxOccurs`. Since each one of those particles appears as a sequence of particles, its particle sequence must be correctly recognized in the input and its sequence of particle objects constructed. This is possible only by looking at the schema information about the type of the particle under consideration

```
XMLParticle  createParticleObject
               (Schema.XMLParticle particle){
 XMLSequence<XMLParticle> particles =
                new XMLSequence<XMLParticle>;
 for (int i=1; i < particle.minOccurs; i=i+1){
    particles.Append(
       getParticleSequence(particle.particles(i)))};
 for (int  i = particle.minOccurs;
            < particle.maxOccurs; i=i+1) {
  particles.Append(
       getParticleSequence(particle.particles(i)))};
  return newInstance(getClass("XMLParticleClass"),
           []Object{particle.minOcccurs,
                   particle.maxOccurs,particles}) }
```

The method `getParticleSequence` tests the type of the particle as specified in the source XSD schema and returns a particle sequence that corresponds to each particular type of a particle.

```
XMLSequence<XMLParticle>
  getParticleSequence (Schema.XMLParticle particle){
 if particle instanceOf Schema.XMLSequenceGroup
 then return sequenceGroupParticles(
              (Schema.XMLSequenceGroup)particle)
 else if particle instanceOf Schema.XMLChoiceGroup
      then return choiceGroupParticles(
                  (Schema.XMLChoiceGroup)particle)
      else return allGroupElements(
                  (Schema.XMLAllGroup)particle)
}
```

Consider a sequence of particles appearing in the input that corresponds to a sequence group. The argument of the method `sequenceGroupParticles` is a sequence group in the

source XSD schema and hence it contains a specification of a sequence of particles. For each particle in the sequence, the type of particle is tested to see whether it is an element. If so, the element tag is read from the input, an element object is constructed and appended to the output particle sequence. If the type of the ith particle as specified in the source schema is not an element, then the method `createParticleObject` is invoked recursively.

```
XMLSequence<XMLParticle> sequenceGroupParticles(
                    Schema.XMLSequenceGroup seqGr)
{ XMLSequence<XMLParticle> particleSeq =
                  new XMLSequence<XMLParticle>;
  for( int i = 1;
       i <  seqGr.particles().high(); i=i+1)
  { ithParticle = seqGr.particles()(i);
    if ithParticle instanceOf Schema.XMLElement
    then {string tag = Input.getTag();
          XMLElement newElement =
                    createElementObject(tag);
          particleSeq.append(newElement) }
    else {XMLParticle newParticle =
          createParticleObject(ithParticle);
         particleSeq.append(newParticle) };
  return particleSeq
}
```

If the type of a particle is a choice group, the schema will still contain a specification of a sequence of particles, but only one of them will appear in the input. Which one is determined by the first tag that appears in the input particle. This is why we need a method `getFirstElementTag`.

The result of the `getChoiceGroupParticle` method is a sequence of particles because of type compatibility, but it will contain a single particle. The sequence of particles in the schema representation of the choice group is accessed and for each one of them the input tag is compared with the first element tag of the ith particle. When those are equal, the ith particle description in the schema will be taken as the valid description of the input particle.

```
XMLSequence<XMLParticle> getChoiceGroupParticle(
         Schema.XMLSchemaChoiceGroup choiceGr) {
XMLSequence<XMLParticle> particleSeq =
                  new XMLSequence<XMLParticle>;
string tag = Input.getTag();
 for (int  i=1;
      i < choiceGr.particles().high(); i=i+1)
 {ithParticle = choiceGr.particles()(i);
  if tag = getFirstElementTag(ithParticle) then
    {particleSeq.append(ithParticle);
     return(particleSeq)}
}
```

Constructing a particle sequence of an all group follows the above logic with one simplification. We know that a sequence of particles in the input should be interpreted as a sequence of elements of an all group.

5. RELATED WORK

One of the first OO models of XML was DOM [4]. Although it is a part of W3C activities, DOM is very limited in its support of XSD. It contains interfaces such as Element and Attribute which are subtypes of the interface Node. The DOM model has a variety of other XSD-specific features. However, it is far from capturing the structural complexity of XSD.

LINQ to XML is an OO interface to XML data that is based on the assumption that an XML schema is not available [9]. LINQ to XML has a fixed collection of classes such as XElement, XAttribute, XNode, XContainer, etc. An input XML document is parsed and viewed through the methods available in these classes. This approach requires extensive type casting and hence dynamic type checking. LINQ to XML supports LINQ queries, but the above typing issues apply to queries just as well.

LINQ to XSD takes a different approach in which specific classes are specified for specific element types that appear in the source XSD schema [10]. It has a variety of techniques for representing some structural features of XSD such as sequence groups, type derivation by inheritance etc. However, the representation model, as appealing as it may be, is too simple to represent XSD accurately. In particular, LINQ to XSD does not distinguish between elements and attributes, has nontrivial problems when the names of elements are repeated, does not represent the notion of a particle with range constraints, does not represent identity constraints, and cannot represent type derivation by restriction because this form of type derivation in XSD is based on constraints.

Paper [13] presents a view of the essence of XSD but it is not object-oriented. This model is limited to well-established and well-understood constructs in type systems. However, some of those constructs are actually not available in mainstream OO languages. Since this approach is based on what is expressible in type systems, it cannot 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 OO representation of XSD that we know of [12]. SOM is in fact our underlying implementation platform. Given an XSD schema SOM produces its OO representation which we use in our approach. However, the complexity of SOM is prohibitive for typical application programmers. This is why we develop an OO interface that represents a correct abstraction over XSD, but is intellectually manageable. We also use some more recent features of type systems of mainstream OO languages such as parametric polymorphism, which SOM does not have. Lack of such typing features in SOM creates undesirable representation problems for SOM which we do not have.

Data Contracts in .NET is the only system we know of that supports both schema level and instance level mappings in both directions: from XSD to OO and the other way around [4, 8]. This system relies on SOM. Data Contracts has nontrivial limitations as to what kind of XSD schema features it can handle. For example, it cannot handle attributes. In the other direction, Data Contracts can handle only certain object types whose structure is such that this system can map them to XSD types.

An analysis of the mismatch between XML and OO languages is presented in [7]. LINQ to XSD in fact follows some of the representation options from [7]. The main difference between our work [1] and [7] is that we represent explicitly and accurately the structural core of XSD, its particle (elements and groups) and type hierarchies. In addition, we represent accurately the complex structure of content models, type derivations, and the identity constraints which are missing in all other approaches except in SOM.

The only work we know of that goes beyond the limitations of type systems is [2, 3]. This research is based on OO constraint languages such as the Java Modeling Language [6] or Spec# [11]. It is thus able to represent all the 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). This approach is also equipped with a prover to verify constraint-related features and transaction safety with respect to the schema integrity constraints. The overall technology is considerably more sophisticated and more complex than the technology based on type systems, mostly because of the prover which requires sophisticated users.

6. CONCLUSIONS

Our first contribution is to show that in spite of the complexity of XML Schema, it is actually possible to define its structural core and specify it formally in terms of the syntactic and typing rules commonly used for mainstream object-oriented programming languages. This makes it possible to present to object-oriented programmers a well-defined core of XSD that is intellectually manageable and a solid basis for complex object-oriented applications that process XML data. This is important because most object-oriented programmers have a limited understanding of XML Schema and are not willing to get involved in deciphering its complexity.

Our second contribution is in the algorithm for mapping XSD schemas to object-oriented schemas. This algorithm is specified through a collection of rules that includes the typing assumptions under which the rules apply. Our rules for mapping XSD schemas into object-oriented schemas are the first such rules ever specified in an explicit and formal manner. One novelty in these rules is that they have two important properties: (i) they are lossless for the XSD core; and (ii) they produce object-oriented interfaces that conform to the rules of object-oriented type systems of mainstream object-oriented languages.

The rules are lossless in the sense that the mapping from the XSD-core structures of an XSD schema to object-oriented types preserves the core structural features which include particle structures (elements and different types of groups) and the type hierarchy based on type derivations by extension and by restriction. The structural specifications of range constraints and identity constraints are also preserved. We conjecture that the mapping can be proved to be lossless in a precise mathematical sense—a subject for future work.

Our third contribution is the first algorithm for mapping XML instances conforming to a given source schema to their object-oriented representation. The existing object-oriented interfaces to XML have underlying algorithms that are neither visible nor published.

The framework for the mapping rules allows for the first time mapping identity constraints of the source XSD schema into their object-oriented representation. Although this representation is necessarily structural, it is critical to avoid losing the integrity constraints of the source schema, as they are in just about all other approaches. The implications on data integrity are obvious and nontrivial.

7. REFERENCES

[1] S. Alagic and P. Bernstein, An object-oriented core of XML Schema, Microsoft Research Technical Report MSR-TR-2008-182, 2008.

[2] S. Alagic, M. Royer, and D. Briggs, Verification theories for XML Schema, Proc. 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 ICSOFT Conf., Vol. 2, pp. 86 - 93, 2006.

[4] Data Contracts, http://msdn2.microsoft.com/en-us/library/ms123402.aspx.

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

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

[7] R. Lammel and E. Meijer, Revealing the X/O impedance mismatch, Datatype-Generic Programming, Springer, LNCS 4719, 2007, pp. 285-367.

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

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

[10] Microsoft Corp., LINQ to XSD Alpha 0.2, 2008, http://blogs.msdn.com/xmlteam/archive/2006/11/27/typed-xml-programmer-welcome-to-LINQ.aspx

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

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

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

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