Object-oriented Constraints for XML Schema

Suad Alagić[1], Philip A. Bernstein[2] and Ruchi Jairath[1]

[1] Department of Computer Science,
University of Southern Maine,
e-mail: alagic@usm.maine.edu, ruchi.jairath@maine.edu
[2] Microsoft Research
e-mail: philbe@microsoft.com

Abstract. This paper presents an object-oriented representation of the core structural and constraint-related features of XML Schema. The structural features are represented within the limitations of object-oriented type systems including particles (elements and groups) and type hierarchies (simple and complex types and type derivations). The applicability of the developed representation is demonstrated through a collection of complex object-oriented queries. The main novelty is that features of XML Schema that are not expressible in object-oriented type systems such as range constraints, keys and referential integrity, and type derivation by restriction are specified in an object-oriented assertion language Spec#. An assertion language overcomes major problems in the object-oriented/XML mismatch. It allows specification of schema integrity constraints and transactions that are required to preserve those constraints. Most importantly, Spec# technology comes with automatic static verification of code with respect to the specified constraints. This technology is applied in the paper to transaction verification.

1 Introduction

XML Schema Definition language (XSD for short) is a widely-used standard for specifying structural features of XML data [18]. 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 [9]. 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, groups, and simple and complex types. 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 [7, 8, 11, 12, 19, 20]. A more detailed account of these issues as they apply to specific approaches is given in [3]. Here we just mention some of these problems:

– 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.

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. The main problem lies in the complexity of XSD, its semantics, and its mismatch with features of object-oriented type systems.

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 [2, 3].
– We demonstrate the utility of our interfaces by giving a variety of typical object-oriented queries specified in LINQ [10].
– 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).
– We specify XML Schema constraints in an object-oriented assertion language Spec#.
– We show how object-oriented schemas and transactions could be specified using general object-oriented constraints.
– We show how transactions are verified using static automatic verification in Spec#.

Due to the limitations of object-oriented type systems, we can represent only some of the constraints of the source XML Schema structurally. 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.

While we can overcome some of the representation problems using a suitable structural representation, the key problem of the object-oriented/XML mismatch is that type systems cannot represent constraints expressible in XML Schema. This is why we use an object-oriented assertion language Spec# [13] which allows specification of range constraints, key and referential integrity constraints, and type derivation by restriction. In addition, more general application-oriented constraints not expressible in XML Schema or standard database technologies can now be specified and enforced using automatic verification that Spec# offers. Note that we always assume that the original XML Schema has been validated. For checking satisfaction of XML Schema constraints such as keys and referential integrity see [15]. In this research we consider object-oriented representation of those constraints. Automatic static verification of the object-oriented representation of XSD constraints is a major distinction with respect to our previous work [1, 4, 5] as well as with respect to other work [6, 16, 17].

The idea of static verification of transaction safety with respect to the database integrity constraints is not new [6, 16, 17] but it has never been implemented at a very practical level so that it can be used by typical database programmers. The first problem with object-oriented technology is that object-oriented schemas are not equipped with general integrity constraints, primarily because mainstream object-oriented languages do not have them. This problem gets resolved using an object-oriented assertion language such as JML, OCL or Spec#. Using an assertion language, schemas can now be specified with general database integrity constraints (invariants) and transactions can be specified in a declarative fashion with preconditions and postconditions.

However, the ability to verify statically that a transaction implemented in a mainstream object-oriented language satisfies those constraints has been out of reach. Our previous results [1, 4, 5] were based on a higher-order interactive verification system which is so sophisticated that it is unlikely to be used by database programmers. A pragmatic goal has been static automatic verification which hides completely the prover technology from users. With recent development in the Spec# technology this becomes possible. This is the main novelty of this paper.

Spec# has limitations in expressiveness dictated by the requirement for automatic static verification. We show that the range of Spec# features is surprisingly suitable for specification for XML Schema and other typical database integrity constraints. This specifically applies to existential and universal quantification required for key and referential constraints and Spec# comprehensions (sum, max, min, count etc.). The Spec# type system includes non-null object types and hence eliminates statically a very frequent error in application programs (and transactions) of trying to dereference a null pointer. Spec# allows explicit representation of aggregation of a complex object in terms of its components and constraints (invariants) that apply to a complex object and its components. Typically, a transaction does not maintain the required integrity constraints until its completion (commit). Spec# has a mechanism that allows specification of this situation so that it is properly handled by the verifier.

This paper is organized as follows. In sections 2, 3 and 5 we discuss the structural representation of the core of XML Schema within the limitations of object-oriented type systems. Sections 4, 6 and 7 deal with XML Schema constraints, their representation in Spec#, and their static automatic verification as it applies to transactions.

2 Object-oriented core of XML Schema

An XML document is a single element, which is the basic case of the XSD notion of a particle. The particle hierarchy contains a direct specification of the actual XML instances, which are documents. 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. The default values of `minOccurs` and `maxOccurs` are both equal to 1.

```
interface XMLParticle
{ int minOccurs();
  int maxOccurs();
}
```

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 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();
}
```

Types of values of elements are structured into the type hierarchy specified below. The root of this type hierarchy is `XMLanyType`. An XML type may be simple or complex, hence the two immediate subtypes of `XMLanyType` are `XMLanySimpleType` and `XMLanyComplexType`.

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

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();
}
```

XMLSequence is a parametric type whose implementation is C# IList. 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();
}
```

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. So we have:

```
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. This is a situation that appears often and violates the typing rules for parametric types. This problem is circumvented somewhat below using the `new` feature of C# which amounts to hiding rather than overriding.

```
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 a satisfactory manner in an object-oriented type system alone [9]. It requires an assertion language. The underlying classes implementing the above interfaces have to correctly implement this semantics.

3 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 [10]. The queries given below reflect complex group structure. `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: XMLElement {
  new JobOffers value();
}
```

`JobOffers` is a complex type whose particle is of type `JobSequence`. `JobSequence` is a sequence-group.

```
interface  JobOffers: XMLanyComplexType {
  new JobSequence particle();
}
interface JobSequence: XMLSequenceGroup {
   XMLString JobID();
   XMLSequence<JobGroup>  jobOffers();
   // set minOcurs and maxOccurs
}
```

`XMLString` and `XMLInt` are simple types derived from `XMLanySimpleType` and represented by C# `string` and `int` types respectively. `JobGroup` is a sequence-group whose particles are two elements: `Name` and `SSN`. The representation given below is based on [2].

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

An example of a LINQ query is given below:

```
static AllJobOffers J;
static JobSequence offers = J.value().particle();
IEnumerable<JobGroup> ProgrammingJobs =
              from x in offers.jobOffers()
              where offers.JobID() == "Programmer"
              select x;
```

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

```
class AnOffer: XMLElement
{ AnOffer(XMLString name, XMLint salary){// . . .};
}
```

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

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

4 Constraints in XML Schema

In this section we illustrate a variety of constraint related features of XML Schema that are not expressible in object-oriented type systems and hence require a strictly more powerful paradigm offered by object-oriented assertions languages. A range constraint is illustrated in the example below where the number of occurrences of a job offer is at least 1 and at most 100:

```
<xsd:complexType name ="JobOfferType" >
 <xsd:sequence>
      <xsd:element  name="JobID" type ="xsd:string" />
      <xsd:element  name="candidateName" type ="xsd:string" />
      <xsd:element  name ="SSN" type ="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType  name = "JobOffersType">
  <xsd:sequence>
    <xsd:element  name = "jobOffer"   type = "JobOfferType"
            minOccurs  = "1"  maxOccurs = "100" />
  </xsd:sequence>
</xsd:complexType>
```

Type derivation by restriction is illustrated below by a short list of job offers. The type `ShortListedOffers` is derived by restriction from the type `JobOffersType` by restricting the range of occurrences constraint in the type `JobOffersType`.

```
<xsd:complexType  name = "ShortListedOffers"
 <xsd:restriction base =  "JobOffersType"
   <xsd:sequence>
     <xsd:element name = "jobOffer"   type = "JobOfferType"
            minOccurs = "1"  maxOccurs = "10" />
   </xsd:sequence>
 </xsd:restriction>
</xsd:complexType>
```

The XML Schema style of specification of a key constraint is given below. This key is specified on the field `JobID`. The scope to which this key applies is specified by a selector which is a simplified XPath expression.

```
<xsd:complexType  name="JobType" >
 <xsd:sequence>
    <xsd:element  name ="JobID"  type="xsd:string"  />
    <xsd:element  name ="JobTitle" type ="xsd:string" />
    <xsd:element  name = "salary" type ="xsd:float" />
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="JobsType" >
<xsd:sequence>
 <xsd:element name="job"  type = "JobType"
           minOccurs = "1"   maxOccurs ="1000" />
</xsd:sequence>
<xsd:key name = "JobIDkey" >
        <xsd:selector  xpath=" ./job" >
        <xsd:field xpath  ="JobID" >
</xsd:key>
</xsd:complexType>
```

The example below illustrates a key and referential integrity constraint. The key constraint specifies that `JobID` is a key in the sequence of job offers. The referential integrity constraint refers to the key constraint in the sequence of jobs. It requires that a job offer refers to an existing job in the sequence of jobs.

```
<xsd:complexType  name = "JobOffersType">
  <xsd:sequence>
    <xsd:element name = "jobOffer"   type = "JobOfferType"
                minOccurs = "1"  maxOccurs = "100" />
 </xsd:sequence>
<xsd:key name="candidateKey" >
   <xsd:selector xpath = "./jobOffer"  />
   <xsd:field  xpath="JobID" />
</xsd:key>
<xsd:keyref  name = "JobRef" >
 <xsd:selector  xpath="JobsType/job" />
 <xsd:field xpath="JobID" />
</xsd:keyref>
</xsd:complexType>
```

5 Meta level

The meta (schema) level contains 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();
}
```

A content model may be simple or complex. 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. Unlike Java, for some reason C# does not allow this type safe change of the signatures of inherited methods and hence requires hiding.

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. Every type derivation has a base type. If the type derivation is simple, the base type must be simple:

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

There are two types of simple type derivation. Simple type derivation by restriction requires specification of a set of constraining facets. This structural representation is augmented in our approach using assertions as explained in section 6. Simple type extension allows only additional attributes:

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

In a complex type derivation the base type is complex, hence the result type of the method base should be overridden covariantly, but we are forced to use the previously described C# technique. 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. The structural representation below is augmented by assertions as explained in section 6.

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

XSD allows structural 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. We use assertions to specify these constraints as explained in section 6. 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. A referential integrity constraint requires an additional reference to a key which is given by the key name.

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

6 Object-oriented constraints

The fundamental requirement for automatic static verification dictates some limitations on expressiveness of Spec# constraints. In this section we show that these limitations fit precisely the XML Schema constraints. On the other hand, Spec# allows specification of application oriented constraints that are not XML Schema constraints.

A frequent problem in object-oriented programs is an attempt to dereference a null reference. If this happens in a database transaction, the transaction may fail at run-time with nontrivial consequences. The Spec# type system allows specification of non-null object types. Static checking will indicate situations in which an attempt is made to access an object via a possibly null reference. Examples presented in this paper include a non-null reference to a job to be inserted in the sequence of jobs, and a non-null reference to a job identifier when updating or deleting a job. Further examples of the non-null constraint are references to a sequence of jobs and a sequence of job offers which are required to be non-null.

A method in Spec# is in general equipped with a precondition expressed by the `requires` clause, and a postcondition specified by the `ensures` clause. A postcondition in general refers both to the object state before method execution, denoted by the keyword `old`, and the object state after method execution. A class is in general equipped with an invariant which specifies valid object states outside of method execution. These assertions allow usage of universal and existential quantifiers as in first-order predicate calculus, as well as combinators typical for database languages such as min, max, sum, count, avg etc.

Spec# constraints limit universal and existential quantification to variables ranging over finite integer intervals. Such intervals are in XML Schema determined by `minOccurs` and `maxOccurs` and thus it is possible to specify Spec# constraints that apply to finite sequences of particles. This is precisely what is needed for specification of keys and referential integrity in XML Schema. The limitation that quantifiers are restricted to integer variables ranging over finite intervals was a design decision to sacrifice expressiveness in order to allow automatic static verification. As explained above, this limitation is no problem in the application considered in this paper.

To make the job of the verifier possible, Spec# requires specification of the frame conditions for methods that change the object state, such as database updates. This is done by the `modifies` clause, which specifies those objects and their components that are subject to change. The frame assumption is that these are the only objects that will be affected by the change, and the other objects remain the same. An attempt to assign to the latter objects will be a static error.

One of the features that makes Spec# suitable for database applications is explicit support for the aggregation abstraction. A complex object is represented by its root object called the owner along with references to the immediate components of the owner specified as `[Rep]` fields. This way a complex object is defined as a logical unit that includes all of its components, direct and indirect. Object invariants may now be specified in such a way that they refer both to the owner object and to its components defined by the `[Rep]` fields.

Yet another feature that makes Spec# suitable for database transactions is an explicit mechanism for allowing methods to temporarily violate object invariants. This typically happens in database transactions where the integrity constraints are violated during transaction execution and then the constraints are reinstated when the transaction is completed and enforced at commit time.

For example, the structure of the job deletion transaction presented in this paper has the following form:

```
expose(schema){
delete job;
delete offers that refer to the deleted job;}
```

After the first action of job deletion the referential integrity constrains are temporarily violated to be reinstated after the second action of deletion of the related job offers. The purpose of the `expose` block is precisely to indicate that the schema object invariant may be violated in this block. Otherwise, the verifier will indicate violation of the schema invariant. In the `expose` block the object is assumed to be in a mutable state and hence violation of the object invariant is allowed. Outside of the `expose` block every assignment that violates the invariant will be a static error.

The code in this section and in section 7 is for presentation purposes and differs somewhat from the actual Spec# code. A very simple example is the range-of-salary constraint given in the class `JobType`.

```
public class JobType : XMLSequenceGroupClass {
//constructor
//definition of properties JobID, jobTitle and salary
 invariant salary >0 && salary < 500000;
}
```

The salary range constraint can be strengthened in the subtype `WellPaidJobType` in accordance with the rules of behavioral subtyping. Type derivation by restriction in XML Schema is based on a similar idea.

```
public class WellPaidJobType: JobType {// constructor
 invariant salary >= 100000;
}
```

Two typical XML Schema constraints are given in the class `JobSequence`. One of them specifies the range of occurrences of jobs in the sequence of jobs in terms of properties `minOccurs` and `maxOccurs`. `XMLSequenceGroupClass` implements properties `minOccurs` and `maxOccurs`. Here we override them so that they will denote the actual range of occurrences. The other constraint specifies that the property `JobID` is a key for the sequence of jobs. This key constraint requires universal quantification that is expressible in Spec# with the already explained limitations, but then it is statically verifiable. A distinctive feature of the Spec# type system and its verification technology is non-null types illustrated by the type `List<JobType>!`.

```
public class JobSequence : XMLSequenceGroupClass {
// constructor
 [Rep]  [ElementsRep] List<JobType>! jobs;
 [Pure] public List<JobType> jobList{get{ return jobs;}}
 [Pure] public new int minOccurs{get{return 0;}}
 [Pure] public new int maxOccurs{get{return (jobs.Count-1);}}
invariant minOccurs >=0 && maxOccurs < 1000 ;
invariant jobs.Count <= maxOccurs - minOccurs +1;
invariant forall {int i in (minOccurs..maxOccurs),
                  int j in (minOccurs..maxOccurs);
                 jobs[i].JobID.Equals(jobs[j].JobID) ==>
                                    jobs[i].Equals(jobs[j])};
}
```

In the above specification we see two cases of the aggregation abstraction as supported by the Spec# ownership model. The attribute `[Rep]` indicates that a list of jobs is a representation of a job sequence so that an object of type `JobSequence` is the owner of this list. Moreover, the attribute `[ElementsRep]` indicates that list elements are components of the list object which is their owner. These elements are then peers according to the Spec# ownership model. This has implications on invariants that can now be defined to apply to entire complex objects, i.e., including their components determined by the `[Rep]` and `[ElementsRep]` fields. These are called ownership-based invariants.

A sequence of job offers has a similar representation with some additional complexity. It has the range constraint for the number of offers and a key constraint on `JobID`. In addition, it has a referential integrity constraint which specifies that a job offer must refer to an existing job in the given list of jobs out of which the offers are constructed. This referential integrity constraint requires both universal and existential quantification expressible and verifiable as shown in the last invariant of the class `OfferSequence`.

```
public class OfferType : XMLSequenceGroupClass {
// constructor
// definition of properties  JobID, candidate name and SSN
}
public class OfferSequence:  XMLSequenceGroupClass {
  // constructor
 [Rep][ElementsRep] List<OfferType>! offers;
 [Rep] JobSequence! jobseq;
 [Pure] public List<OfferType>! joffers {get{return offers;}}
 [Pure] public new int minOccurs{get{return 0;}}
 [Pure] public new int maxOccurs{get{return (offers.Count-1);}}

invariant minOccurs >=0 && maxOccurs < 100;
invariant offers.Count <= maxOccurs - minOccurs +1;
invariant forall {int i in (minOccurs..maxOccurs),
                  int j in (minOccurs.. maxOccurs);
           offers[i].JobID.Equals(offers[j].JobID) ==>
                             offers[i].Equals(offers[j])};
invariant forall {int i in (minOccurs..maxOccurs);
          exists {int j in (jobseq.minOccurs..jobseq.maxOccurs);
               jobseq.jobList[j].JobID.Equals(offers[i].JobID)}};
}
```

Representation of type derivation by restriction as defined in XML Schema is illustrated in the class `ShortListed` in which the range of occurrences of job offers is narrowed with respect to the range of occurrences in the base class `OfferSequence`. This pattern fits precisely the discipline of behavioral subtyping as implemented in Spec#.

```
public class ShortListed: OfferSequence {
invariant minOccurs >=1 && maxOccurs <= 10;
}
```

7 Transaction verification

Our final contribution is integration of the technologies presented in this paper into an implemented model of automatic static verification of object-oriented transactions with respect to the object-oriented representation of XSD schemas equipped with constraints. To our knowledge this is the first time this was possible for a full-fledged mainstream object-oriented language and object-oriented schemas and transactions extended with very general constraints. The components of this model are more sophisticated features of the type system such as bounded parametric polymorphism available and statically verifiable in C#, representation of XML Schema constraints, pre and post conditions for transactions in Spec#, and their automatic static verification.

In our approach, the class `Transaction` is bounded parametric, where the bound type is the type of schema to which a specific transaction type is bound.

```
interface Schema {//. . .}
class Transaction<T> where T: Schema {
Transaction(T! schema) {. . .}
T schema(){get{return schema;}}
//. . .
}
```

With the examples developed in the previous section the simplest way of specifying a schema of job offers is an aggregation of a sequence of jobs and a sequence of job offers.

```
class JobSchema: Schema {
 [Rep] public JobSequence! jobsSeq;
 [Rep] public OfferSequence! offerSeq;
 JobSchema(JobSequence! jobs, OfferSequence! offers){
          this.jobsSeq = jobs;
          this.offerSeq= offers;  }
}
class JobTransaction: Transaction<JobSchema> {//...}
```

A transaction that creates (inserts) a new job and maintains the schema integrity constraints is `JobInsert` given below. The precondition of this transaction includes a constraint on the admissible range of salaries and a condition which guarantees that the insertion would not violate the key constraint. Yet another constraint requires that the argument is in fact a non-null pointer to a job object. The postcondition guarantees that the insertion is actually performed.

```
class JobInsert: JobTransaction {
// constructor
 void addJob(JobType! job)
 modifies schema.jobsSeq;
 requires forall {int i in
  (schema.jobsSeq.minOccurs..schema.jobsSeq.maxOccurs)
   !schema.jobsSeq.jobList[i].JobID.Equals(job.JobID)};
 requires job.salary >0 && job.salary < 500000;
 ensures exists {int j in
  (schema.jobsSeq.minOccurs..schema.jobsSeq.maxOccurs);
    (schema.jobsSeq.jobList[j].Equals(job))};
 {expose(schema.jobsSeq)
   {schema.jobsSeq.jobList.Add(job);}}
}
```

A salary update transaction given below takes a non-null `jobId` pointer and requires that this `jobId` actually appears in the list of jobs. The postcondition guarantees the salary update is performed correctly.

```
class SalaryUpdate: JobTransaction {
// constructor
 void updateSalaries(string! jobId)
 modifies schema.jobsSeq;
 requires exists {int j in
  (schema.jobsSeq.minOccurs..schema.jobsSeq.maxOccurs);
     schema.jobsSeq.jobList[j].JobID.Equals(jobId)};
 ensures forall {int j in
  (schema.jobsSeq.minOccurs..schema.jobsSeq.maxOccurs);
      schema.jobsSeq.jobList[j].JobID.Equals(jobId)==>
          schema.jobsSeq.jobList[j].salary >= 100000 };
 {expose(schema.jobsSeq)
  {foreach (JobType! job in schema.jobsSeq.jobList){
      if ((job.JobID.Equals(jobId)) && (job.salary < 100000))
         {job.salary= 100000;}}}  }
}
```

The most complex example of a transaction is `JobDeletion`. This transaction deletes a job with an existing given `jobId`. This requirement is expressed in the precondition. There are two postconditions. The first one guarantees that there is no job with the given `jobId` in the list of jobs, i.e., the job has been deleted. The other postcondition guarantees that the referential integrity is maintained, i.e., this `jobId` does not appear in the list of offers either.

```
class JobDeletion: JobTransaction {
// constructor
 void deleteJobs(string! jobId)
 modifies schema.jobsSeq, schema.offerSeq;
 requires exists {int j in
  (schema.jobsSeq.minOccurs..schema.jobsSeq.maxOccurs);
                schema.jobsSeq.jobList[j].JobID.Equals(jobId)};
 ensures forall {int j in
```

```
  (schema.jobsSeq.minOccurs..schema.jobsSeq.maxOccurs);
       !schema.jobsSeq.jobList[j].JobID.Equals(jobId)};
 ensures forall {int j in
(schema.offerSeq.minOccurs..schema.offerSeq.maxOccurs);
    !(schema.jobsSeq.jobList[j].JobID.Equals(jobId))};

 {expose(schema)
   {foreach (JobType! job in schemaJob.jobsSeq.jobList)
   if (job.JobID.Equals(jobId))
       schema.jobsSeq.jobList.Remove(job);
   foreach (OfferType! job in schema.offerSeq.joffers)
   if (job.JobID.Equals(jobId))
        schema.offerSeq.joffers.Remove(job); }}
 }
```

As of this writing, the Spec# implementation is a prototype with problems that one can naturally expect from software that is still not a product. But even where static verification does not succeed, the Spec# compiler generates code that enforces the constraints at run-time, which is the prevailing technique in current object-oriented assertion languages such as JML and Eiffel.

8 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 initial 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.

More importantly, we argue that the only way to resolve major issues in the object-oriented/XML mismatch is to make use of an object-oriented assertion language that allows specification of constraints-based features of XML Schema. This approach has an additional major advantage: it allows specification of more general constraints in object-oriented schemas that reflect constraints in the application environment and are not expressible in common database technologies.

Most importantly, the assertion language that we used in this paper comes with automatic verification not available in other object-oriented assertion languages. This means that for the first time we can specify object-oriented schemas and transactions equipped with general constraints and carry out static automatic verification of transactions with respect to the specified constraints. The implications on data integrity, efficiency and reliability of transactions are obvious and non-trivial.

However, the presented technology has its limitations. Spec# is a promising development, but at the moment it is an industrial prototype and not a product. Its current limitations in expressiveness dictated by the requirement for automatic static verification did not present a problem in our quite complex application. But the error messages and the current tutorial [13] need improvement, which often makes it hard to drive programs through verification. The ownership model is complex, which many users may find hard to fully understand and apply correctly. All of this implies that Spec# run-time checks for assertions that have not been statically verified is at the moment an important feature of this technology. But the technology itself is clearly still under development.

References

1. S. Alagić, M. Royer, and D. Briggs, Verification technology for object-oriented/XML transactions, Proceedings of ICOODB 2009, pp. 23-40, *LNCS 5936*.
2. S. Alagić and P. Bernstein, An object-oriented core for XML Schema, Microsoft Research Technical Report MSR-TR-2008-182, December 2008, http://research.microsoft.com/apps/pubs/default.aspx?id=76533.
3. S. Alagić and P. Bernstein, Mapping XSD to OO schemas, Proceedings of ICOODB 2009, *LNCS 5936*.
4. S. Alagić, M. Royer, and D. Briggs, Verification theories for XML Schema, Proc. of BNCOD, *LNCS 4042*, pp. 262-265, 2006.
5. S. Alagić and J. Logan, Consistency of Java transactions, Proceedings of DBPL 2003, *LNCS 2921*, pp. 71-89, Springer, 2004.
6. V. Benzanken and X. Schaefer, Static integrity constraint management in object-oriented database programming languages via predicate transformers, *LNCS 1241*, pp. 60-84, 1997.
7. Data Contracts, http://msdn2.microsoft.com/en-us/library/ms123402.aspx.
8. Document Object Model (DOM), http://www.w3.org/TR/REC-DOM-Level-1/.
9. R. Lammel and E. Meijer, Revealing the X/O impedance mismatch, Datatype-Generic Programming, Springer, *LNCS 4719*, 2007, pp. 285-367.
10. Language Integrated Query, Microsoft Corporation, http://msdn.microsoft.com/en-us/vbasic/aa904594.aspx.
11. Microsoft Corp., LINQ to XML, http://msdn.microsoft.com/en-us/library/bb387098.aspx.
12. 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.
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. Md. S. Shariar and J. Liu, Checking satisfaction of XML referential integrity constraints, AMT 2009, *LNCS 5820*, pp. 148-159, Springer, 2009.
16. T. Sheard and D. Stemple, Automatic verification of database transaction safety, *ACM Transactions on Database Systems 14*, pp. 322-368, 1989.
17. D. Spelt and S. Even, A theorem prover-based analysis tool for object-oriented databases, *LNCS 1579*, pp 375 - 389, Springer, 1999.
18. W3C: XML Schema 1.1, http://www.w3.org/XML/Schema.
19. XML Data Binder, http://www.liquid-technologies.com/XmlStudio/Xml-Data-Binder.aspx.
20. XMLBeans, http://xmlbeans.apache.org.