A Mapping of XML Schema Types To C#

Wolfgang Gehring

University of Ulm, Faculty of Computer Science, D-89069 Ulm, Germany
wgehring@informatik.uni-ulm.de

Abstract. The following work proposes a mapping from XML Schema Definition Language (XSD) to C#. XSDs are the type system of XML. In XSD, many more types can be defined than in C# or a comparable programming language. C#'s class library provides many classes for creation and manipulation of XML objects; however, C# does not support the XSD type system (like native C# types). We thus propose a mapping which maps XSDs to C#. The mapping is injective such that all semantically meaningful information in the schema is maintained. C# properties are used to ensure restrictions on types that have no native counterpart in C#.

1 General Design Guidelines

In our goal to obtain a mapping which is as appropriate as possible, we tried to adhere to the following general design guidelines:

- Equivalent Schemas should be mapped to equivalent (or same) classes.
- If there exists an instance document corresponding to a given schema, then the mapping should produce a type view and vice versa. This implies that *every* top-level element becomes a type.
- For every document fragment that can be manipulated out of its context there should be a type. This implies that every particle is mapped to a *nested* type.
- All elements in the scope of a complex type become properties; we call this set of properties the *public interface* of the type. If there are several elements of the same name and same type within the same scope, these are mapped to properties that return an array.
- All semantically meaningful information about the schema should be maintained by the mapping (the mapping must be injective).
- The mapping has to be *consistent*, *intuitive* and *easy to understand* for the human user.

In the following sections, we will show how each of the main constructs of XML Schema can be mapped into C#. We will start with elements at top-level in various scenarios, then look at particles, and then at nested elements in different contexts. We will conclude with the mapping of complex elements and a quick look at substitution groups.

It is assumed that the reader is familiar with XML Schema and with C#.

2 Top-level elements

Top-level elements are always mapped into classes. The next few sections describe the different possible types and content models of top-level elements together with their corresponding mappings by means of simple examples.

2.1 Top-level element with predefined simple type

Top-level element with a predefined simple type (e.g. string, integer, boolean and such)

```
<schema ...>
    <xs:element name="Price" type="xs:integer"/>
</schema>
```

becomes

```
class Price {
    int _Content;
    public int Content {
        get{ return _Content;  }
        set{ _Content = value; }
    }
}
```

2.2 Top-level element with anonymous simple type

```
<schema ...>
<xs:element name="Price">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="0.0"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
</schema>
```

becomes

```
class Price {
    int _Content;
    public int Content {
        get{ return _Content; }
        set{
          if (value < 0)
            throw( new(ArgumentOutOfRangeException()) );
          else
```

```
            _Content = value;
        }
    }
}
```

The validation of the restriction on the type is hidden in the property's set-method.

2.3 Top-level element with with no type

Elements without explicit type get the default type "object":

```
<xs:schema ...>
    <xs:element name="Vehicle"/>
</xs:schema>
```

becomes

```
class Vehicle{
    object _Content;
    public object Content {
        get{ return _Content;  }
        set{ _Content = value; }
    }
    }
}
```

2.4 Top-level element with anonymous complex type

Example:

```
<xs:schema ...>
<xs:element name="Stock">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Symbol" type="xs:string"/>
            <xs:element name="Price" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

becomes

```
class Stock {
    struct sequence {
        string _Symbol;
        public string Symbol {
```

```
            get{ return _Symbol: }
            set{ _Symbol = value; }
        }
        string _Price;
        public string Price {
            get{ return _Price: }
            set{ _Price = value; }
        }
    }
    // Property for the content of the complex type
    sequence _Content;
    public sequence Content {
        get{ return _Content: }
        set{ _Content = value; }
    }
    // Properties for the public interface of the complex type
    [System.Xml.Serialization.XmlElementAttribute("Symbol")]
    public string Symbol {
        get{ return Content.Symbol: }
        set{ Content.Symbol = value; }
    }
    [System.Xml.Serialization.XmlElementAttribute("Price")]
    public string Price {
        get{ return Content.Price: }
        set{ Content.Price = value; }
    }
}
```

The translation of <sequence> will be treated in more detail in section 3. The general case looks like this:

```
<xs:schema ...>

<xs:element name="Name">
    <xs:complexType>
        ...
    </xs:complexType>
</xs:element>

</xs:schema>
```

and is translated into

```
class Name {
    // translation of the content:
    struct {
        ...
```

```
    }
    // Property for the content of the complex type
    ...
    // Properties for the public interface of the complex type
```

2.5 Top-level element with named complex type

```
<xs:schema ...>
<import book.xsd/>
    <xs:element name="Fiction" type="book"/>
    <xs:element name="NonFiction" type="book"/>
</xs:schema>
```

becomes

```
class Fiction {
    book _Content;
    public book Content {
        get{ return _Content; }
        set{ _Content = value; }
    }
}
        // Properties for the public interface of book
        .....
} class NonFiction {
    book _Content;
    public book Content {
        get{ return _Content; }
        set{ _Content = value; }
    }
}
```

3 All, Choice, Sequence

The <all>, <choice>, or <sequence>-particles become structs (at all nesting levels). If P := (all | choice | sequence), then P gets mapped into

```
struct P {
    // recursive translation of all the particles inside P

    // properties for every simple element in the scope of P
} P content {get; set;}

 // properties for every element and for every particle in the scope of P
```

Simple element here means: every element except those with anonymous complex type, because those get mapped into classes (see section 2.4 and section 7), all other nested elements get mapped into fields.

For every propery we put [] to denote an array type iff minOccurs and maxOccurs are not both equal to 1. The reason is obvious for maxOccurs > 1; if minOccurs = 0, we want to have an array because in the case that the occurence is actually = 0, we can have a null reference even for value types. Note that if P is a <choice>, we want to put the [] on *every* element/property, because the choice implies minOccurs = 0 on every element inside the choice.

– Example 1. General case of a sequence (with the default value 1 for minOccurs and maxOccurs)

```
<xs:element name="Book">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Author" type="xs:string"/>
            <xs:element name="Title"  type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

becomes

```
class Book {
    struct sequence {
        string _Author;  // in the remainder of the document we
                         // will omit these for clarity
        public string Author {
            get{ return _Author; }
            set{ _Author = value; }
        }
        string _Title;
        public string Title {
            get{ return _Title; }
            set{ _Title = value; }
        }
    }
    // Property for the content of the complex type
    sequence _Content;
    sequence Content {
        get{ return _Content; }
        set{ _Content = value; }
    }
    // Properties for the public interface of the complex type
    [System.Xml.Serialization.XmlElementAttribute("Author")]
    public string Author {
```

```
        get{ return Content.Author; }
        set{ _Content.Author = value; }
    }
    [System.Xml.Serialization.XmlElementAttribute("Title")]
    public string Title {
        get{ return Content.Title; }
        set{ _Content.Title = value; }
    }
}
```

(See section 4 for explanation of the custom attributes on the elements.)

– Example 2. Sequence with minOccurs and maxOccurs on the sequence

```
<xs:element name="Book">
    <xs:complexType>
        <xs:sequence minOccurs="1" maxOccurs="10">
            <xs:element name="Author" type="xs:string"/>
            <xs:element name="Title" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

becomes

```
class Book {
  struct sequence {
    string Author {get; set;}
    string Title {get; set;}
  }
  sequence[] content {get; set;}

  [System.Xml.Serialization.XmlElementAttribute("Author")]
  string[] Author {get;}
  [System.Xml.Serialization.XmlElementAttribute("Title")]
  string[] Title {get;}
}
```

The difference between this example and example 1 is that the struct sequence[] as well as the (element-)properties outside of the struct are now of array type. The (element-)properties inside the struct become an array type iff not both minOccurs and maxOccurs on the corresponding element are equal to 1. Also note that when we have an array, we only put get;, not set; on the properties outside of the struct. This is because the elements inside a sequende can only appear together, so in this example Author and Title can only be paired. It is the Content-property's task to ensure that this constraint is not violated.

– Example 3. <Choice>

```
<xs:element name="Temperature">
    <xs:complexType>
        <xs:choice>
            <xs:element name="Celsius" type="xs:integer" />
            <xs:element name="Fahrenheit" type="xs:integer" />
        </xs:choice>
    </xs:complexType>
</xs:element>
```

becomes

```
class Temperature {

  struct choice {
    int[] Celsius {get;}
    int[] Fahrenheit {get;}
  }
  choice content {get; set;};

  [System.Xml.Serialization.XmlElementAttribute("Celsius")]
  int[] Celsius {get;}
  [System.Xml.Serialization.XmlElementAttribute("Fahrenheit")]
  int[] Fahrenheit {get;}
}
```

As mentioned above, because of the implied minOccurs = 0 every member of the choice becomes an array type. The choice content itself becomes an array type if not both minOccurs and maxOccurs are equal to 1 on the choice itself.

<all> is a special case. The elements inside the <all> can appear in an arbitrary order in the instance document; however, it is important *which* order was chosen.

4 Nested element with simple type

Nested elements (inside a <sequence>, <choice>, or <all>-particle) become properties of the struct that the particle is transformed into (for mapping of particles see previous section), with the exception of nested elements with *anonymous complex type* (see section 7). Thus, e.g.

```
<xs:element name="name" type="xs:string"/>
```

becomes

```
 string name;
```

inside the struct, but there is also a property added *outside* of the struct:

```
[System.Xml.Serialization.XmlElementAttribute("name")]
string name{get; set;}
```

The custom attribute holds the name of the element. It could be omitted in this case, but it is important in cases where there are name conflicts. These can occur because in XML it is possible to have a sequence with the same element twice, but in a C# class two fields of the same name are prohibited. In such a case we can resolve the name conflict by introducing an array that holds the clashing elements. Thus, e.g.

```
<xs:element name="name" type="xs:string"/>
<xs:element name="name" type="xs:string"/>
```

becomes

```
[System.Xml.Serialization.XmlElementAttribute("name")]
string name0;
[System.Xml.Serialization.XmlElementAttribute("name")]
string name1;
string[] name {get;}
```

inside the struct and

```
[System.Xml.Serialization.XmlElementAttribute("name")]
string[] name {get;}
```

outside of the struct.

5 Nested element with element reference

Element references are unfolded in the place of the reference.
Example:

```
 <xs:element ref="author"/>
```

becomes

```
author author0;
string author {get; set;}
```

inside the struct and

```
[System.Xml.Serialization.XmlElementAttribute("author")]
string name {get; set;}
```

outside of the struct, if we assume that the original ”author”-element definition looks like this:

```
<xs:element name="author" type="xs:string"/>
```

6 Nested element with named complex type

Example:

```
<xs:element name="name" type="personType"/>
```

Analogously to the previous examples, this becomes

```
personType name0;
personType name {get; set;}
```

inside the struct and

```
[System.Xml.Serialization.XmlElementAttribute("name")]
personType name {get; set;}
```

outside of the struct.

7 Nested element with anonymous complex type

Nested elements with anonymous complex type are transformed into classes exactly in the same manner as *top-level elements* with anonymous complex type (see section 2.4).

8 Nested element with anonymous simple type

Example:

```
<xs:element name="quote">
    <xs:simpleType>
        <xs:restriction base="xs:decimal">
            <xs:minInclusive value="0.0"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

becomes

```
decimal quote;
```

inside the struct and

```
[System.Xml.Serialization.XmlElementAttribute("quote")]
decimal quote {get; set;}
```

outside of the struct. Like in the case of top-level elements with an anonymous simple type, the validation of the restriction is hidden inside the set-method.

9 Top-level complex type

Top-level (i.e. named) complex types are children of the schema element and become, like elements, classes:

```
<xs:schema ...>
    <xs:complexType name="book">
        ....  <!-- content of the complex Type -->
    </xs:complexType>
</xs:schema>
```

becomes

```
class book {
    // recursive translation of the content of the complex type
}
```

10 Nested complex type

Nested complex types are unnamed (anonymous) children of element declarations. What construct they are mapped into depends on their content.

11 Complex type with attribute

Attributes in a complex type definition become fields of the class, independent of whether it is a nested or a top-level complex type.
Example (of an attribute in a top-level complex type):

```
<xs:schema ...>
    <xs:complexType name="book">
        <xs:attribute name="year" type="xs:integer"/>
    </xs:complexType>
</xs:schema>
```

becomes

```
class book {
  [XmlAttributeAttribute("year")]
  int year {get; set;}
}
```

Example (of an attribute in a nested complex type):

```
<xs:schema ...>
    <xs:element name="book">
        <xs:complexType>
            ....
            <xs:attribute name="year" type="xs:integer"/>
```

```
        </xs:complexType>
    </xs:element>
</xs:schema>
```

becomes

```
class book {
    // recursive translation of the content of the complex type
    [XmlAttributeAttribute("year")]
    int year {get; set;}
}
```

The field corresponding to the attribute gets an [XmlAttributeAttribute] custom attribute with its name in order to specify that this field was derived from an attribute rather than from an element (e.g. important when the original schema has to be reconstructed).

12 Complex type with mixed content

Complex types with mixed content, independent of whether they are nested or top-level, are translated in their usual manner; in addition, they get a new field, a string array, which can hold the mixed content.
Example. The following schema

```
<xs:complexType name="Letter" mixed="true">
    <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Title" type="xs:string"/>
        <xs:element name="Publisher" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

could have an XML instance document like this:

```
<Letter> Dear Mr.<Name>John Writealot</Name>, we are pleased to
tell you that your book <Title>XML Powerhouse</Title> has been
published at <Publisher>Independent Press</Publisher>. </Letter>
```

This means that in a complex type with mixed content there can be text items between every element, as well as before the first and after the last element. For these text items we therefore provide a string array as additional property of the class that is generated from the complex type:

```
class Letter {
    struct sequence {
        // usual translation of the content of the complex type
        // additionally:
        string[] _Textitems;
```

```
        public string[] Textitems {
            get{ return _Textitems; }
            set{ _Textitems = value; }
        }
    }

    // Property for the content of the complex type as usual
    // Properties for the public interface of the complex type as usual
    // additionally:
    public string[] Textitem {
        get{ return Content.Textitems; }
        set{ Content.Textitems = value; }
    }
```

13 Complex type extension

The extension of a complex type corresponds to class inheritance.

```
<xs:complexType name="Person">
    <xs:sequence>
        <xs:element name="name"   type="xs:string" />
        <xs:element name="height" type="xs:integer"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="Employee">
    <xs:complexContent>
        <xs:extension base="Person">
            <xs:sequence>
                <xs:element name="salary" type="xs:decimal"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

becomes

```
class Person {
  struct sequence {
    string name {get; set;}
    int height {get; set;}
  }
  sequence content {get; set;}
  string name {get; set;}
  int height {get; set;}
}
```

```
class Employee : Person {
  struct sequence {
     decimal salary {get; set;}
  }
  sequence content {get; set;}
  decimal salary {get; set;}
}
```

14 Complex type restriction

In the case of restriction of simple content where only the value space of a simple type is being restricted, the validation of that new constraint is hidden in the property's set-method.

The restriction of complex content with the omission of a field, and also the restriction of simple content where the use of an attribute is being prohibited, however, both correspond to class inheritance. We use a little trick to hide the omitted field(s).
Example:

```
<xs:complexType name="Person">
    <xs:sequence>
        <xs:element name="name" type="xs:string" minOccurs="0"/>
        <xs:element name="weight" type="xs:integer"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="Anonymous">
    <xs:complexContent>
        <xs:restriction base="Person">
            <xs:sequence>
                <xs:element name="weight" type="xs:integer"/>
            </xs:sequence>
        </xs:restriction>
    </xs:complexContent>
</xs:complexType>
```

becomes

```
class Person {
    struct sequence {
        string[] name {get;}
        int weight {get; set;}
    }
    sequence content {get; set;}
    string[] name {get;}
```

```
    int weight {get; set;}
}

 // Hide Person::name
class Anonymous : Person {
    new public class name { private name(){} }
}
```

15 Substitution groups

Substitution groups are another mechanism to define inheritance hierarchies. The element marked as abstract is the head of the substitution group and becomes an abstract superclass. The elements which belong to its substitution group are descendants of the superclass.
Example:

```
<xs:element name="Vehicle" abstract="true"/>
<xs:element name="Car" substitutionGroup="Vehicle">
    ....
</xs:element>
```

becomes

```
abstract class Vehicle {}
class Car: Vehicle {
    ....
}
```

16 Groups and attribute groups

Groups and attribute groups are unfolded in the place where they are referenced and translated according to their content.

17 Conclusion

In this paper, we have shown a mapping from XML Schema Definition Language to C#. To check the validity of our work, we implemented a compiler that uses this mapping and translates XSD to C# (for most cases). Future work includes examining roundtripping, i.e. the way from C# types back to XML Schema.

References

1. Skonnard, A, Gudgin, M.: Essential XML Quick Reference (2001)
2. Gunnerson, E.: A Programmer's Introduction to C# (2000)
3. W3C Recommendation, XML Schema Part 0: Primer. See also Parts 1 and 2. http://www.w3.org/TR/xmlschema-0/ .