On why C#'s type system needs an extension

Wolfgang Gehring

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

Abstract. XML Schemas (XSD) are the type system for XML. Class libraries of object-oriented programming languages like C# provide many classes for creation and manipulation of XML objects; however, they do not support the XSD type system like native types. In XSD, many more types can be defined than in today's object-oriented programming languages. In this paper, we look at a few examples of why and where there are mismatches between the two worlds of XML and OO in order to motivate why the type system of C# and related OO-languages needs to be extended.

1 A mapping from XSD to C#

In our main goal to extend C#'s type system such that it could support XSD types natively, we first created a mapping which maps XML Schemas to C#. In this manner, we tried to find common structures in the various XSD constructs and to see how new C# types could possibly subsume certain XSD types and remedy occuring conflicts.

For the remainder of this document, it is assumed that the reader is familiar with XML Schema and with C#. In order to motivate why we postulate an extension to C#'s type system, we will look at two different exemplary cases: In the first one, we will see that the same XSD construct is mapped into different types in C#, depending on the context in which the construct appears in the original XML Schema. In the second one, we will see that the same C# type view can originate from different XML Schema constructs.

In obtaining an appropriate mapping, we tried to adhere to the following general design guidelines:

– Equivalent schemas should be mapped to equivalent (or same) classes.
– If there exists an XML 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 (which is allowed in XSD), these are mapped to properties that return an array.

– All semantically meaningful information about the schema should be maintained by the mapping (i.e. the mapping must be injective).
– The mapping has to be *consistent*, *intuitive* and *easy to understand* for the human user.

In the next section, we will give a general outline of our mapping and introduce some features that will not be discussed at length in the remainder of the paper. In section 3 we will then take a closer look at the mapping of nested elements. Section 4 will discuss the correspondence between complex type extensions, restrictions and substitution groups versus inheritance. For the interested reader and for better understanding, the complete mapping can be found at http://www.wgehring.de.

2 Introduction to the mapping

Top-level elements. Top-level elements are always mapped into classes. This is due to our requirement that the mapping should produce a type view for every schema that can have an instance document.

Attributes. Attributes in a complex type definition become fields of the class, independent of whether it is a nested or top-level complex type.

Particles. 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, all other nested elements get mapped into fields, as we will see in the next section.

Top-level complex types. 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
}
```

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

Complex types with mixed content, independent of whether they are nested or top-level, get in addition to their regular translation a new field, a string array, which can hold the mixed content.

Complex type extensions and restrictions will be treated in section 4.

3 Nested elements versus fields

Nested elements (i.e. elements inside a <sequence>, <choice>, or <all>-particle; the element is not a direct descendant of a <schema> node) come in five different kinds: they can be

– with a simple type
– with an element reference
– with a named complex type
– with an anonymous simple type
– with an anonymous complex type

All but the very last kind become fields (implemented as C# properties) of the struct that the particle is transformed into. 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 (keep in mind that the struct comes from the translation of the surrounding particle):

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

with appropriate get- and set-methods. 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. Also, we want to maintain all the information about the elements such that roundtripping back to XSD could be possible.

Element references are simply unfolded in the place of the reference; analogously for nested elements with a named complex type. Nested elements with an anonymous simple type imply that there is a restriction on the simple type:

the translation is as usual, except that the validation of the restriction is hidden inside the set-method of the property.

The only case out of the five that differs is, as mentioned, the case of a nested element with an anonymous complex type. These elements are transformed into classes exactly in the same manner as *top-level elements* with anonymous complex type. We made this choice because we thought it would best capture the concept that is behind the construct of having an anonymous complex type inside an element.

The general case looks like this:

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

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
```

We see that just by looking at the *nesting level* of an element, we cannot determine what it will be mapped into in C#. This seems to be an inherent mismatch between the XML Schema (type) world and the C# type world. Furthermore, as we have stated, attributes become fields of a class in our mapping as well. Thus, without an additional C#-custom attribute on the field, we could not tell whether the field was derived from an element or an attribute.

4 Extension, Restriction, and Substitution Groups versus Inheritance

There are three constructs in XSDs that are represented by inheritance in our mapping: Complex type extension, complex type restriction and substitution groups. It should be fairly straightforward to see with substitution groups: 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 abstract superclass.
Example:

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

```
    ....
</xs:element>
```

becomes

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

Similarly, in a complex type extension another complex type is enriched e.g. by an additional element. The correspondence to class inheritance where a base class can be enriched by additional fields should be fairly obvious.

In the case of a complex type restriction the analogy to inheritance may not be as easy to see. There are two cases: One, where in a complex type with simple content only the value space of a simple type is being restricted. We handle this by modifying the property's set-method accordingly. Thus, we get a derived subclass where a method is overwritten.

In the second case, we either have the restriction of complex content with the omission of one or more elements, or we have the restriction of simple content where the use of an attribute is being prohibited. This sounds like instead of having a more "specialized" class (which tells us that we would use a derived subclass in OO) we have a more general class that has less fields. However, in another sense we actually do have a more special class, because in the schema we are first told that we have a certain number of fields (and also which these are) and *then* we are told that we now leave off one ore more fields. While the class as such is more general speaking in terms of number of fields, it actually is more special in the sense that we are given the information that there used to be more fields, but we are now not allowed to use these any more. It is important that we keep this very information, and our mapping should reflect this as well.

The following program fragment shows how we use inheritance and a little trick to hide the omitted field(s) and still maintain the information about which field was being hidden:

```
<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(){} }
}
```

5 Conclusion and outlook

Although XSDs provide many more type constructs than are currently representable in C#, we can translate the majority of the XSD types into C# through an appropriate mapping. If we provide sufficient additional information for C# by means of custom attributes so that no schema information is lost, we could even achieve bijectivity and make roundtripping from C# back to XSD possible. However, looking at only two cases, we have already seen that currently existing C# types cannot represent the full range of possibilities that XSD offers; more examples can be easily found. This implies, of course, that in spite of the extensive XML class libraries that are provided in C#, we cannot have a fully satisfying treatment of XSDs and XML documents in C#. We thus strongly recommend an extension to the type system of C# to make a (native) treatment of additional types possible.

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/ .