

Making XML Schema Easier to Read and Write

Erik Wilde and Kilian Stillhard, ETH Zürich (Swiss Federal Institute of Technology)

Abstract

XML Schema is a rather complex schema language, partly because of its inherent complexity, and partly because of its XML syntax. In an effort to reduce the syntactic verbosity and complexity of XML Schema, we designed the *XML Schema Compact Syntax (XSCS)*, a non-XML syntax for XML Schema. XSCS is designed for human users, and transformations from and to XML Schema XML syntax are implemented using Java-based tools.

1 Introduction

While XML Schema [1, 3] will certainly play an important role in the future of XML, there is an ongoing controversy about it, which is partly based on XML Schema itself, and partly based on its verbose syntax. At least the second issue can be resolved by designing a more compact syntax for XML Schema. The reason for the verbosity of XML Schema's current syntax is partly because it uses XML, and partly because the design of the XML in some places is less than ideal, a notorious example being complex type derivation.

We present the *XML Schema Compact Syntax (XSCS)*, which has been inspired by the RELAX NG compact syntax [2], which also is based on the perception that XML syntax might not always be ideal for human users. XSCS is designed for human users, and does not change anything of XML Schema's data model, it simply is a more compact representation of XML Schema components. Consequently, XSCS is of no interest to users of other interfaces which have been designed for XML Schema (such as graphical schema editors), and may be regarded as simply another interface (a character-based) for XML Schema.

2 Syntax Design

XSCS is designed for human users. It is defined by an EBNF grammar, which also serves as the foundation for XSCS parsing (see Section 3). During the design of XSCS, the following features were of particular importance:

- *Compactness*: The syntax should be less verbose than XML syntax and use short notations for frequently used language features. The compactness should not compromise the legibility of the syntax and should reuse established notational conventions whenever possible.
- *DTD-style content models*: Content models should be expressed using well-known DTD syntax, including the '&' connector¹. Local elements may be declared within or outside of the content model.

¹Which disappeared during the transition from SGML to XML DTDs, but has been reintroduced by XML Schema in the form of the `all` model group.

- *Easier mixed content*: Mixed content in complex type derivation should be less confusing than in XML Schema XML syntax (where it may be specified on the `complexType` and/or `complexContent` elements).
- *Easier complex type derivation*: Complex type derivation should contain less redundant information, in particular whether the base type is simple or complex.
- *Easier facet definition*: Facet notation should be much easier, and associated facets (such as interval bounds) should be combined to create a more intuitive syntax.

Based on these principles, we designed a syntax that is best illustrated by looking at some examples (in this case taken from the schema for schemas):

```
<xs:simpleType name="short">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="-32768"/>
    <xs:maxInclusive value="32767"/>
  </xs:restriction>
</xs:simpleType>
```

This is a named simple type definition that restricts another named simple type. The restriction uses two facets to define a closed interval.

```
simpleType short { xs:int { [-32768,32767] } }
```

The base type of the type derivation precedes the facets contained in curly brackets. The interval facets are joined in a mathematical notation that is easily recognizable as a closed interval (open intervals use parentheses rather than square brackets).

```
<xs:complexType name="extensionType">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:group ref="xs:typeDefParticle"
          minOccurs="0"/>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:attribute name="base" type="xs:QName"
        use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

This is the type definition that is used for the XML Schema `extension` element for complex type extension definitions (and as base type for the `simpleExtensionType`, which is derived from it by restriction and used for simple content

extensions). It extends the base type `xs:annotated` by references to two model groups in its content model and an additional attribute.

```
complexType extensionType extends xs:annotated {
  ( @xs:typeDefParticle?, @xs:attrDecls );
  required attribute base { xs:QName };
};
```

In XSCS, the complex content specification disappeared, because the content of the type derivation (in this case a complex type, indicated by the content model) already makes it clear whether the type derivation will result in simple or complex content. The group references are specified by the special '@' character, and the rest of the content model uses standard DTD syntax. The attribute specification uses a keyword to identify the attribute as `required`, and otherwise simply declares the attribute's name and type, using the standard XSCS syntax with curly brackets.

These examples illustrate that XSCS is much less verbose than the XML syntax. It also introduces some useful abbreviations and avoids some of the irritating aspects of the XML syntax. Because XSCS is purely text-based rather than using XML markup, we need special software to generate and interpret it. However, using state-of-the-art software tools, this can be done rather easily, as shown in the following section.

3 Implementation

The implementation of XSCS is based on the EBNF grammar of the language, which is used as input to the JavaCC parser generator. We then use additional code to generate the XML syntax using a standard DOM interface. Unfortunately, there is no standard API for XML schema components. Apache's Xerces XML parser implements a proprietary component API, but it is read-only and thus cannot be used for constructing XML Schema components. The result of our XSCS parser is a DOM tree which can either be serialized as XML Schema XML syntax, or passed to a Schema processor. Consequently, errors may occur during the interpretation of XSCS (where syntax errors may be detected), or during the interpretation of the generated DOM tree as XML Schema components (where Schema component constraint violations may be detected).

Generating XSCS from existing schemas is also possible. In this case, we use a standard XML parser to read the XML syntax, and then use a DOM-based program to generate XSCS. Operating on the DOM tree rather than the (more abstract) component model allows us to retain the structure of the source, which in most cases is important. Since we have avoided any syntax constructs that would make it necessary to follow the component structure when generating XSCS, a DOM API is sufficient and it is not necessary to use a schema component API.

4 Results

As an example, we took the schemas defined by XML Schema itself, and converted them to compact form. We stripped the schemas from annotations, comments, and the internal DTD subset, and based the size comparison on non-whitespace characters only.

| Schema | Lines | Characters |
|----------------|-------|------------|
| structures.xsd | 58% | 58% |
| datatypes.xsd | 71% | 67% |

It can be seen that the size reduction is substantial, and that in all cases the size reduction was more than half. Typically, simple type definitions can be compressed better than complex types, because facets are the part of the syntax which are compressed most. Apart from the reduction in size, XSCS is easier to read for humans because it removes a number of redundant syntactic elements.

5 Limitations

Our current version of the syntax has limitations regarding annotations (special elements in the XML Schema XML syntax) and XML comments. XML Schema annotations may contain arbitrarily complex XML and are hard to handle. We currently simply take the string-value of XML Schema annotations, which is a considerable simplification. Furthermore, XML Schema annotations may appear in a number of places where there is no corresponding place in the compact syntax, in which case the annotation is ignored. XML comments are always ignored, because there is no clear association with XML Schema components.

Namespace handling is based on some simplifications. Namespace declarations may only appear on the top level of XSCS (and are only accepted if they appear on the top level of the XML), which is a reasonable way to use namespaces, but not the only possibility. Namespace handling could be improved by also dealing with namespace declarations further down the XML tree, and even more through accommodating namespace redeclarations by introducing artificial global prefixes. However, we believe that these perfectly legal namespace uses should not be endorsed, and therefore so far our implementation simply rejects them.

6 Conclusions

It was our goal to define a syntax for XML Schema which makes it much easier for humans to read and write XML Schemas. XSCS largely reduces the size of XML Schema documents and makes them easier to read, at the cost of some minor limitations that do not compromise the features of XML Schema. A more complete description of our work can be found in another publication about XSCS [4].

References

- [1] PAUL V. BIRON and ASHOK MALHOTRA. XML Schema Part 2: Datatypes. World Wide Web Consortium, Recommendation REC-xmlschema-2-20010502, May 2001.
- [2] JAMES CLARK. RELAX NG Compact Syntax. Organization for the Advancement of Structured Information Standards, Committee Specification, November 2002.
- [3] HENRY S. THOMPSON, DAVID BEECH, MURRAY MALONEY, and NOAH MENDELSON. XML Schema Part 1: Structures. World Wide Web Consortium, Recommendation REC-xmlschema-1-20010502, May 2001.
- [4] ERIK WILDE and KILIAN STILLHARD. A Compact XML Schema Syntax. In *Proceedings of XML Europe 2003*, London, UK, May 2003.