A compact XML schema syntax

A Compact XML Schema Syntax

Keywords: W3C XML Schema, XSCS, Java, DOM, Data Representation

Erik Wilde
Swiss Federal Institute of Technology
Zürich
Switzerland
net.dret@dret.net
http://dret.net/netdret/

Biography

Erik Wilde is currently working as senior researcher at the Computer Engineering and Networks Laboratory (TIK), which is part of the Department of Information Technology and Electrical Engineering of the Swiss Federal Institute of Technology (ETH) in Zürich.

Kilian Stillhard
Swiss Federal Institute of Technology
Zürich
Switzerland
kstillha@ee.ethz.ch

Biography

Kilian Stillhard studies electrical engineering at the Swiss Federal Institute of Technology (ETH) in Zürich. For his diploma thesis at the Computer Engineering and Networks Laboratory (TIK), he designed and implemented the XML Schema compact syntax.

Abstract

The new schema language defined by the W3C, XML Schema, is used in a number of applications, such as Web Services and XQuery, and will probably be used by an increasing number of users in the near future. Currently, XML Schema's data model, the "XML Schema Components", can only be represented in the rather verbose XML syntax defined in the XML Schema specification itself. We propose an alternative non-XML syntax, which is (1) much more compact than the XML syntax, (2) defined

by EBNF productions, (3) re-uses well-known syntactic concepts where appropriate, and (4) is easy to implement using standard parser-generating tools. Our approach is comparable to the approach of the RELAX NG schema language, which also supports two alternative syntaxes, an XML-based one, and a more compact non-XML one.

We believe that XML Schema could be made easier to use by supporting a compact syntax. Currently, complex schemas are very hard to read due to the large amount of XML markup, and the various tools and GUIs that are on the market differ widely and in all cases support only a subset of the features of XML Schema. We believe that there should be a compact syntax, optimized for human users, which makes it easy to read and write XML Schemas, and which supports the full feature set of XML Schema. Obviously, a non-XML syntax makes it necessary to introduce new tools. However, generating parsers from EBNF productions is rather simple and well-supported by standard tools (such as yacc and JavaCC), and the other direction (i.e., generating non-XML syntax) can be implemented by using XML tools.

Our *XML Schema Compact Syntax (XSCS)* is geared towards human users, by re-using language constructs known from other application areas, such as DTDs and programming languages, and making them available for XML Schema component representation. Examples for this re-use of syntactic constructs are DTD-style content models, number ranges ("[a,b]" or "(a,b]" as in standard mathematical notation), and qualifying attributes like "abstract" or "final" known from programming languages ("final abstract type { ... }"). We also believe that graphical representations of complex structures such as schemas are not always suitable because some people prefer textual representations, editing might be faster when using keyboard input instead of using click-and-point operations, and graphical representations (usually) hide some information.

We fully integrate the processing of our syntax into the existing pipeline of XML-based tools by creating a parser that generates SAX events or DOM trees from the compact syntax documents. This way, we can use the existing XML Schema validation engines and XML Schema error checking facilities already implemented in validation engines like the Xerces parser. In addition, we have a serialization module to generate compact syntax documents from XML Schema DOM trees.

Our overall goal is to improve XML Schema acceptance by providing a syntax that is easier to work with than the XML syntax, and tools to process this syntax.

Table of Contents

Introduction

XML Schema **[XMLSchema1]** **[XMLSchema2]** is very successful as an XML schema language, and since the W3C has decided to use XML Schema as the foundation for other technologies (most notably, XML Query **[XQuery]**), it is unlikely that XML Schema will be replaced by another schema language anytime soon. One property of XML Schema is that is currently uses XML as representation. The abstract model of XML Schema is based on so-called *components*, which are the logical constituents of XML Schema. The XML Schema specification is defined in terms of components and also defines an XML representation for these components. When working with non-trivial XML Schemas, it becomes apparent that the XML syntax of XML Schema is hard to read and hard to write. This is partly due to the fact that it is XML (which inherently results in verbosity due to the usage of full markup), but partly also due to the design of the XML syntax for XML Schema components.

There is an increasing number of software tools available which make working with XML Schema more easy, for example by providing graphical representations (often including editing functionality) for XML Schemas. However, for using these tools it is necessary to purchase the software, and different tools support different graphical representations. Drawing from the approach taken in **[RELAXNGCS]** (which defines a compact, non-XML syntax for **[RELAXNG]**), we therefore aimed at creating a compact syntax for XML, which should make it easy for humans to read and write XML Schemas. We see this as a complementary syntax for XML Schema, which is not a competitor for the XML syntax, but a supplement. In the same way as XML is optimized for interpretation and generation by computers, we aimed at designing a syntax which is optimized for human users of XML Schema.The often intimidating complexity of XML Schema syntax should be replaced with a more intuitive and more compact way to represent XML Schema components. We call this new syntax *XML Schema Compact Syntax (XSCS)*.

Design Guidelines

One very simple approach would have been to simply replace the verbose XML syntax with its start and end tags with a more compact representation, for example using a keyword/paranthesis-based syntax. However, our goals also included to improve some of the less fortunate syntax constructs of XML Schema, in particular, the design guidelines were as follows:

- *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 `&` connector (Which disappeared during the transition from SGML to XML DTDs, but has been reintroduced by XML Schema in the form of the `all` model group). Local elements may be declared within or outside of the content model.
- *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.
- *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 design guidelines, we designed a syntax using a grammar-based approach, using *Extended Backus-Naur Form (EBNF)* **[ISO14977]** for the grammar.

Syntax Design

A complete definition and description of the XSCS syntax is available in a technical report **[TIKrep166]** . The notation used for the grammar is EBNF, a format that is commonly used to specify grammars, and that is also supported by the majority of parser-generation tools. However, different tools accept different classes of grammars, which not only influences the languages that can be processed with these tools, but also influence the way these tools work. We evaluated three parser generators, *SableCC*, *JavaCC*, and *CUP*. While SableCC and CUP support LALR(1) grammars, JavaCC supports LL(n) grammars. Due to the internal design, LALR(1) parsers work differently than LL(n) parsers regarding the code that can be executed during the parsing process. Part of our syntax design consequently was influenced by the choice of tools and their support for implementing an XSCS parser. Our implementation is built on top of JavaCC (see **Implementation**), and consequently XSCS is LL(n).

XSCS Examples

XSCS syntax has been designed to be as easy readable and understandable as possible. In order to demonstrate some of the features of XSCS, the following three examples show XML Schema XML syntax and the corresponding XSCS syntax. The first two examples are taken from the XML Schema for XML Schema, while the last example is purely hypothetical.

One of the goals of XSCS has been to make fact definitions more compact, and to combine associated facets syntactically. The following example shows a simple type definition in XML Schema XML syntax:

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

This definition defines a simple type named `short` as a derivation by restriction of the type `xs:int`. The derivation specifies two facets, which define the upper and lower bounds of the new type. The bounds are specified using a closed interval, where the bounds are part of the interval. The XSCS syntax for this type definition has the following form:

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

The `simpleType` keyword is followed by the type's name and the type derivation enclosed in braces. Since the name following the opening brace is not `list` or `union`, it is implicitly clear that the type definition uses derivation by restriction (`list` or `union` names as base types for derivation by restriction must be escaped with a leading backslash). The type derivation then contains a sequence of facets. In this case, XSCS combines the two facets into a single notation, defining the number range, using the well-known mathematical notation of a closed interval.

In the following example, a complex type is defined, using more complex features of XML Schema:

```
<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 XML Schema XML code defines a complex type with the name `extensionType`, which is defined using type extension. The type extension extends the base type `xs:annotated` with a sequence of particles (in this case, the particles are groups instead of elements), and an attribute. The XSCS code for this type definition has the following form:

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

The `complexType` keyword is followed by the type's name and the type derivation method indicated by the keyword `extends` and the name of the extended type. The type extension is enclosed in braces, first specifying the sequence (using DTD-style notation), which contains groups rather than elements. This is indicated by the leading `@` character of the name, which indicates the names to be group names. The attribute definition starts with the keywords `required attribute`, then specifies the attribute name, and then is followed by a anonymous simple type definition enclosed in braces, in this case a reference to a built-in simple type.

In the following example, a hypothetical element `page` is defined, containing only local element definitions.

```
<element name="page">
 <complexType>
  <sequence>
   <element name="head" type="string"/>
   <element name="section" type="string" maxOccurs="unbounded"/>
   <element name="foot" type="string"/>
  </sequence>
 </complexType>
</element>
```

The XML Schema XML code defines the element `page` by using an anonymous complex type type declaration. The type is defined by a sequence of three elements, which are locally declared. XSCS allows two different notations for this kind of definition, shown in the following code fragments:

```
element page {
 ( head { string }, section { string }+, foot { string } ) }
```

In this case, the element definitions are directly embedded in the DTD-style content model notation of XSCS. This notation is very similar to the XML syntax, because it embeds the type definitions into the content model. However, in particular for more complex content models or type definitions, this makes it rather hard to read the content model itself. Consequently, XSCS supports another style of specifying this type:

```
element page {
 ( head, section+, foot)
 element head { string }
```

```
element section { string }
element foot { string } }
```

In this second case, the DTD-style content model only contains the element names, while the actual element definitions are outside of the content model. It is important to notice that the elements still are local elements (because they are defined inside of the complex type), so this style of writing complex types in XSCS should not be confused with using global elements (where the elements are globally defined in the XML Schema and referenced from the complex type definition where they appear).

Implementation

Since we did not want to implement our own XML schema processor, the question was how to integrate the XSCS syntax processing with an existing XML Schema processor. We decided to use Apache's Xerces XML processor, because it is open source and also because it provides the most advanced XML Schema implementation that we could find. Xerces processes XML Schemas by first parsing them into a DOM tree, and then using this DOM tree to generate the schema components (using an internal representation). We decided to plug XSCS into this architecture by providing

- a component for parsing XSCS syntax and then transforming it into a DOM tree of XML Schema XML syntax,
- and a component for generating XSCS syntax from a DOM tree of XML Schema XML syntax.

This way, we were able to minimize our implementation efforts, and to maximize the amount of existing Xerces code we could use. The disadvantage is that we are not able to perform schema component constraint checks directly on XSCS syntax, because we always have to take the intermediate step of converting XSCS into a DOM tree.

The XSCS parser consists of two components, the generated parser class, and a class that generates the DOM representation in XML Schema syntax. When converting from XSCS to XML, a DOM tree of the schema is first generated and then written to a file using a standard DOM serializer module. From XML to XSCS, the process starts by parsing the XML file using a standard DOM parser, and then handing over the generated DOM tree to the XSCS serializer component. All coding and tests have been conducted using the Xerces parser library, however other DOM implementations could be used as well. However, some code changes would be necessary, because the current DOM interface cannot boot-strap itself (i.e, there is no implementation independent way of creating a *DOMImplementation* object which is needed to create new documents).

A detailed description of the implementation is given in **[Stillhard]** .

Future Work

The following list contains some areas where XSCS could be improved, or where additional work is necessary to find out whether improvements would be possible or necessary:

- Currently, XSCS schemas have to be converted to the XML syntax (in the form of a DOM tree) before they can be used for validation. For a better integration of XSCS into the XML processing tool-chain, the XSCS parser could be directly integrated into a schema-validating XML processor. The DOM based implementation has been chosen because of the easy integration within the Xerces parser. To validate XML documents, Xerces uses a DOM parser to read the associated schema, and then traverses this document to create an internal representation similar to the XML Schema Components data model. Therefore, only the code that loads schemas had to be modified to allow validation against XSCS schemas.
- Error reporting currently is done only rudimentary. XSCS syntax errors (i.e., code violating the XSCS syntax) is detected by the XSCS parser. However, semantic errors (i.e., code using correct XSCS syntax, but specifying things that violate XML Schema's component constraints) are not detected in our software. We rely on Xerces' error checking, which is less than perfect. Also, even if Xerces detects and report an error, the error message refers to the generated DOM tree rather than the XSCS code, which makes it somewhat hard to find errors in the XSCS code.
- Furthermore, the XSCS syntax itself could be extended, particularly in the field of annotation handling, in order to achieve full compatibility (enabling lossless roundtripping between the syntaxes) with the XML syntax.
- Some of XSCS's syntactic variations (for example, the option whether local element declarations in content models are specified embedded in the content model or outside of it) are currently generated by the XSCS serializer according to built-in rules. The serializer could provide options for controlling the rules, allowing users to choose the flavor XSCS's syntactic variations that they prefer.

Conclusions

XSCS provides a compact syntax for XML Schema. After a short learning period, XML Schemas in XSCS are significantly easier to read than they are in the XML syntax. We see XSCS as an improved character-based interface for XML Schema. It is not intended to compete with or replace sophisticated graphical representations of XML Schema as they are provided by XML Schema editing software. Instead, XSCS's focus is to provide a lowest common denominator that is easy to read and write for human users. Our XSCS implementation still is in a prototypical stage, but using the grammar and well-known programming tools it is relatively

easy to implement XSCS.

Bibliography

[ISO14977]

International Organization for Standardization. *Information Technology — Syntactic Metalanguage — Extended BNF*. ISO/IEC 14977, 1996.

[RELAXNG]

James Clark. *RELAX NG Specification*. Organization for the Advancement of Structured Information Standards, Committee Specification, December 2001.

[RELAXNGCS]

James Clark. *RELAX NG Compact Syntax*. Organization for the Advancement of Structured Information Standards, Committee Specification, November 2002.

[Stillhard]

Kilian Stillhard. *A Compact Syntax for XML Schema*. Master's thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland, March 2003.

[TIKrep166]

Kilian Stillhard and **Erik Wilde**. *XML Schema Compact Syntax (XSCS) Version 1.0*. Technical Report TIK-Report No. 166, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland, March 2003.

[XMLSchema1]

Henry S. Thompson, **David Beech**, **Murray Maloney**, and **Noah Mendelsohn**. *XML Schema Part 1: Structures*. World Wide Web Consortium, Recommendation REC-xmlschema-1-20010502, May 2001.

[XMLSchema2]

Paul V. Biron and **Ashok Malhotra**. *XML Schema Part 2: Datatypes*. World Wide Web Consortium, Recommendation REC-xmlschema-2-20010502, May 2001.

[XQuery]

Scott Boag, **Don Chamberlin**, **Mary F. Fernández**, **Daniela Florescu**, **Jonathan Robie**, and **Jérôme Siméon**. *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, Working Draft WD-xquery-20021115, November 2002.