Safe, IDE-Friendly, Extensible, XML Schema

Andrew Cooke*

August 2007

Abstract

I show how to make schema for XML documents that are safe, IDE-friendly, and extensible. This is particularly used for defining Domain Specific Languages for Java configuration via Spring 2.0[1], but the approach is general.

There is nothing particularly new here, but I couldn't find suitable documentation when I needed this information — I hope this helps others in the future.

Contents

*andrew@acooke.org

1 Introduction

I will show how to define a schema which is safe, extensible, IDE-friendly, and future-proof.

Safe The schema should *not* allow the user to construct an XML document that will cause problems when it is processed. To some extent this is a tautology — if a schema validates then it *is* correct — but we need to be careful to avoid issues with global elements. This is explained further in section 3.3.

Extensible It should be possible for a third party to write an extension schema that adds additional features to the original in a controlled way. A document written using both schema should include extra elements (in a new namespace) at certain pre-determined points. There should be no need to update the original schema to allow this.

IDE Friendly When an XML document is being written, an editor or IDE should be able to use the schema to provide focused, useful prompts. This lets an inexperienced user construct a valid document by choosing items from a list provided by the IDE.

2 Structure and Implementation

XML schema include a variety of extension mechanisms. Here I show just one approach (the one that seems most useful in practice), which is to separate the *structure* of the schema from any particular *implementation*.

2.1 The Example

The simplest way to show how structure and implementation can be separated is by giving an example. The schema and XML documents below let someone describe a list of transports. A transport is a vehicle that carries a load.

The initial schema (section 2.2) describes how lists of transports are formed, but doesn't give any vehicles or loads. Other schema extend this initial structure (section 2.3) to define a horse and cart, a truck, and various loads. Finally (section 2.4) I will put everything together in a valid document that combines elements from all these schema.

2.2 Defining Structure

Abstract element types can be used to define structure. Below is a schema that defines a structure containing abstract vehicle and load elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.example.com/example/transport"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.com/example/transport"
            elementFormDefault="qualified">
    <xsd:element name="all-transports">
```

```
        <xsd:complexType>
            <xsd:sequence maxOccurs="unbounded">
                <xsd:element name="transport">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element ref="abstract-vehicle"/>
                            <xsd:element ref="abstract-load"
                                         maxOccurs="unbounded"/>
                        </xsd:sequence>
                        <xsd:attribute name="name"/>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="abstract-vehicle" abstract="true"
                 type="abstractVehicleType"/>
    <xsd:complexType name="abstractVehicleType"/>
    <xsd:element name="abstract-load" abstract="true"
                 type="abstractLoadType"/>
    <xsd:complexType name="abstractLoadType"/>
</xsd:schema>
```

In this schema some elements are abstract (`abstract="true"`). This means that they cannot be used directly in an XML document. However, while they cannot be used directly, they define a structure which other schema can implement.

2.3 Defining Implementation

To provide an implementation — a set of usable (non–abstract) elements that follow the same structure as the schema in section 2.2 — we need to do two things:

1. Extend the structural type.
2. Provide a substitution group.

Below I provide two vehicles (in separate namespaces) and some loads (in a single namespace). First, the transports.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.example.com/example/cart"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:transport="http://www.example.com/example/transport"
            targetNamespace="http://www.example.com/example/cart"
            elementFormDefault="qualified">
    <xsd:import namespace="http://www.example.com/example/transport"/>
    <xsd:element name="horse-and-cart" type="horseAndCartType"
                 substitutionGroup="transport:abstract-vehicle"/>
    <xsd:complexType name="horseAndCartType">
        <xsd:complexContent>
            <xsd:extension base="transport:abstractVehicleType">
                <xsd:attribute name="horseName" default="Dobbin"/>
            </xsd:extension>
        </xsd:complexContent>
```

```
    </xsd:complexType>
</xsd:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.example.com/example/truck"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:transport="http://www.example.com/example/transport"
            targetNamespace="http://www.example.com/example/truck"
            elementFormDefault="qualified">
    <xsd:import namespace="http://www.example.com/example/transport"/>
    <xsd:element name="truck" type="truckType"
                 substitutionGroup="transport:abstract-vehicle"/>
    <xsd:complexType name="truckType">
        <xsd:complexContent>
            <xsd:extension base="transport:abstractVehicleType">
                <xsd:choice>
                    <xsd:element name="petrol"/>
                    <xsd:element name="diesel"/>
                </xsd:choice>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

Note how types extend types defined in the transport schema while the associated elements use `substitutionGroup` to identify the transport element they will replace. So, for example, `truck` has `substitutionGroup="transport:abstract-vehicle"` and `truckType` extends `transport:abstractVehicleType`.

Also, the `truckType` defines additional elements (`petrol/diesel`), while `horse-and-cart` has an extra attribute (`horseName`). Implementations can add both elements and attributes to, but cannot remove content from, the types they extend. In this case the new element is not abstract (although it could be, if we wanted to add further extensibility).

Second, here are the loads:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.example.com/example/loads"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:transport="http://www.example.com/example/transport"
            targetNamespace="http://www.example.com/example/loads"
            elementFormDefault="qualified">
    <xsd:import namespace="http://www.example.com/example/transport"/>
    <xsd:element name="potatoes" type="perishableGoodsType"
                 substitutionGroup="transport:abstract-load"/>
    <xsd:element name="carrots" type="perishableGoodsType"
                 substitutionGroup="transport:abstract-load"/>
    <xsd:element name="rocks" type="simpleGoodsType"
                 substitutionGroup="transport:abstract-load"/>
    <xsd:complexType name="simpleGoodsType">
        <xsd:complexContent>
            <xsd:extension base="transport:abstractLoadType"/>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="perishableGoodsType">
```

```
        <xsd:complexContent>
            <xsd:extension base="simpleGoodsType">
                <xsd:attribute name="expiry" type="xsd:date"
                               use="required"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

Here, although `carrots` has `substitutionGroup="transport:abstract-load"`, it extends the type `simpleGoodsType` and not `transport:abstractLoadType`. This is acceptable because `simpleGoodsType` itself extends `transport:abstractLoadType`, which is sufficient for `perishableGoodsType` to be consistent with `transport:abstractLoadType`.

2.4 Using the Schema

All the schema above are used in the following document:

```
<?xml version="1.0" encoding="UTF-8"?>
<all-transports xmlns="http://www.example.com/example/transport"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:cart="http://www.example.com/example/cart"
                xmlns:truck="http://www.example.com/example/truck"
                xmlns:loads="http://www.example.com/example/loads">
    <transport name="a truck with rocks">
        <truck:truck>
            <truck:petrol/>
        </truck:truck>
        <loads:rocks/>
    </transport>
    <transport>
        <cart:horse-and-cart horseName="Ned"/>
        <loads:carrots expiry="2007-03-10"/>
        <loads:potatoes expiry="2007-02-23"/>
    </transport>
</all-transports>
```

2.5 Summary

In the sections above I have:

1. Defined an initial schema that gives the *structure* for a list of transports.
2. Provided different *implementations* of the elements in the initial structure.
3. Written an XML document that conforms to the schema, combining the implementations in a way that agrees with the initial structure.

It is important to understand the balance between what is fixed and what can be changed. Given the initial namespace and root element (`all-transports`) the final document must contain a list of transports, and each transport must contain a vehicle and one or more loads. But exactly what vehicles or loads are used depends on the additional schema.

This is the kind of controlled flexibility we need to make extensible but safe systems.

In the next sections I will cover some extra details and make the (rather obvious) connection between this approach and object–oriented programming.

3 Further Details

3.1 Future–Proofing

In the example above additional schema can extend the original structure by adding different vehicles and loads, but it is not possible to add a completely new element at the same level as `transport`.

Allowing extension inside the root element is a powerful (but possibly dangerous) tool — it allows other schema to add completely new structures. This can be very useful for applications which must be "future–proof".

There are two ways to allow arbitrary extension of this kind. We can use the `<xsd:any/>` element with the `##other` namespace, or we can introduce an additional abstract element that has an empty type. I prefer an empty abstract element since the `<xsd:any/>` exacerbates the problems discussed in sections 3.2 and 3.3.

If we choose the abstract element approach and then, in the future, decide that we do need to directly include a namespace, we can always add an implementation that itself introduces the `<xsd:any/>` element (often it is sufficient for the extension to introduce a specific namespaces, rather than `##other`, which is a good thing). This is shown in the example below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.example.com/example/extensible"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.com/example/extensible"
            elementFormDefault="qualified">
    <xsd:element name="root">
        <xsd:complexType>
            <xsd:sequence maxOccurs="unbounded">
                <xsd:element ref="abstract-extension"/>
                <!-- other elements here -->
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="abstract-extension" abstract="true"
                 type="abstractExtensionType"/>
    <xsd:complexType name="abstractExtensionType"/>
</xsd:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.example.com/example/extension"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:base="http://www.example.com/example/extensible"
            targetNamespace="http://www.example.com/example/extension"
            elementFormDefault="qualified">
```

```
    <xsd:import namespace="http://www.example.com/example/extensible"/>
    <xsd:element name="add-namespace"
                 substitutionGroup="base:abstract-extension">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="base:abstractExtensionType">
                    <xsd:sequence>
                        <xsd:any namespace="http://..."/>
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

3.2 Unambiguous Schema

There's an obscure restriction on schema that was inherited from SGML, called the Unique Particle Attribution Constraint (as far as I can tell, this requires that schema be context–free). Any particular element in the document must be associated unambiguously with an element in the schema.

For simple schema this is not likely to be a issue, but it can easily become a problem once namespaces are imported with the `<xsd:any/>` element.

The simplest problems (that the current namespace is re–included) are avoided by using the `##other` namespace rather than `##any`, but when working with several namespaces this is often not sufficient — explicitly name schema instead.

Many XML processing systems do not check for the Unique Particle Attribution Constraint by default. This helps protect end–users from accidental errors but is a nuisance when trying to develop a "correct" system. To help detect problems as early as possible use a unit test. For example, in Java:

```
public void testUnambiguous(Source schema) throws Exception
{
    SchemaFactory schemaFactory =
        SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
    schemaFactory.setFeature(
        "http://apache.org/xml/features/validation/schema-full-checking",
        true);
    Schema schema = schemaFactory.newSchema(schema);
}
```

3.3 Global Elements

When a schema is included via `<xsd:any/>` any global (ie not from a nested set of definitions) element can be added to the document. This is normal behaviour and would not be a problem except that implementation elements (section 2.3) are also defined globally (the elements in which they might be nested are defined in a separate schema).

The unfortunate consequence of this is that, despite all our care in restricting elements to extend particular abstract elements, a valid document can contain inappropriate structure. For example, consider using Spring beans with the transport schema defined above (Spring allows `##other` elements at the top level):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:transport="http://www.example.com/example/transport"
       xmlns:cart="http://www.example.com/example/cart"
       xmlns:truck="http://www.example.com/example/truck"
       xmlns:loads="http://www.example.com/example/loads">
    <!-- correct use -->
    <transport:all-transports>
        <transport:transport>
            <cart:horse-and-cart/>
            <loads:rocks/>
        </transport:transport>
    </transport:all-transports>
    <!-- incorrect use, but validates -->
    <loads:carrots expiry="2005-12-02"/>
</beans>
```

We have an unfortunate and unexpected pile of carrots!

I have not found a way to completely avoid this problem, but the following approaches help reduce the impact:

- Avoid using `<xsd:any/>` whenever possible.
- Provide a well-documented root element. Global elements are not a problem "inside" the root element (eg. `all-transports`).
- Manage problematic external schema by embedding them. For example, see section 4.

4 Spring

Spring 2.0[1] allows applications to define schema and associated Java code that, together, allow complex system configurations to be specified in XML and assembled from Java classes. I discovered the ideas documented here while working on the Spring 2.0 support for Mule[2].

4.1 Java, OO

The separation of structure and implementation using abstract elements, described in section 2, is very similar to the subclass polymorphism present in Java and other Object–Oriented languages. This is an advantage for two reasons:

1. Developers can draw upon existing design skills and patterns when writing complex schema.
2. Extensible Spring configurations can closely match the implementation's class structure, reducing "impedance mismatch" between the configuration and the underlying system.

4.2 Automatic Extensibility

Spring provides a mechanism[3] (via NamespaceHandler) for dynamically resolving code to handle schema. This allows libraries to be added in a modular manner, providing schema and Java code, extending an existing system without modifying the core.

4.3 Embedding Spring

To avoid the problems described in section 3.3 I suggest embedding Spring inside your application's schema. This can be achieved by defining an element that introduces the Spring namespace (the same technique illustrated in section 3.1). Spring XML handling code can be called from within your own BeanDefinitionParserDelegate by extracting the DOM sub–tree associated with your own `<beans>` element, creating a new DOM Document, and delegating to the DefaultBeanDefinitionDocumentReader.

This allows users to start documents with you own, restricted, root element, but access Spring features when necessary (sharing the same ApplicationContext).

To guarantee that users avoid using Spring's `<beans>` element as document root, runtime code in the appropriate BeanDefinitionParsers can check that elements have the correct parent (our own restricted root element) and, if not, provide the user with a specific, helpful error message.

5 Acknowledgements

The Spring implementation details include lessons learnt from working with Ross Mason's code; Daniel Feist provided useful background information on schema and the Unique Particle problem. This paper was written in my own time but documents lessons learnt while working for MuleSource[4].

References

[1] Spring Application Framework
http://www.springframework.org/

[2] Mule
http://www.mulesource.org/

[3] Extensible XML Authoring
http://www.springframework.org/docs/reference/extensible-xml.html

[4] MuleSource
http://www.mulesource.com/