Schema-aware processing with XSLT 2.0

Reap the benefits of designing your XSLT stylesheets to be schema-aware

Mukul Gandhi

May 15, 2008
(First published April 29, 2008)

With the release of version 2.0, XSLT now allows you to design your stylesheets to be schema-aware. A schema-aware XSLT system offers many benefits, including the ability to validate input trees prior to the XSLT transformation to ensure that the XSLT stylesheet only processes valid input, as well as the ability to validate output trees to ensure that the XSLT transformation is producing the valid XML output. You are also able to specify data types for variables, for input parameters for user-defined functions and templates, and for return values from the functions. In this article, learn more about the concept of schema-aware facilities and follow some examples that illustrate the benefits.

With the publication of the W3C specification for the XSLT 2.0 language, one of the most important innovations was introduced into the XSLT language: the ability for the XSLT processor to utilize XML schemas for input and output documents, as well as for temporary trees and constructs that expect types to be specified, such as function and template parameters, and variables.

Frequently used acronyms

- W3C: World Wide Web Consortium
- XML: Extensible Markup Language
- XSLT: Extensible Stylesheet Language Transformations

Schema awareness is an optional feature for the XSLT processor to implement. An XSLT processor that doesn't implement schema-aware facilities is known as a basic XSLT processor, whereas one that does implement such facilities is known as a schema-aware XSLT processor.

This article assumes that you have knowledge of XML and the W3C XML Schema language, and preferably some knowledge of XSLT. To exploit the schema-aware facilities in the XSLT stylesheets effectively, you need to understand the syntax and semantics of XML Schema in detail. To try the examples in this article, you'll need an XSLT 2.0 processor that implements the schema-aware features of the XSLT 2.0 language. For the purpose of this article, I used an evaluation copy of the commercial product Saxon-SA. You can download a free 30-day evaluation license to try the features; see Related topics for details.
An overview of XML Schema

An XML document can either be stand-alone or designed to correspond to a schema. A stand-alone XML document merely contains nested tags with text and obeys only the well-formed constraints of XML. On the other hand, an XML document designed for an XML schema obeys the constraints of the schema. Nearly all modern applications that work with XML contain a well-defined XML schema. The XML schema assigns structure to the XML document and defines the data types of elements and attributes.

The W3C XML Schema language is much more enhanced than the previous XML validation language, Document Type Definition (DTD). Unlike XML schemas, and the much-enhanced data-typing facility in particular, DTD could not express complex XML validation constraints. The finer details of XML Schema language are beyond the scope of this article, but you can refer to Related topics to learn more.

Why write schema-aware stylesheets?

Schemas are typically available for well-known XML vocabularies and other large applications. As the stylesheet writer, however, you are able to maintain the schemas and the types yourself. By doing so, you can extract numerous benefits for the application architecture and the business problem the application is intended to solve.

You can put XML schemas to use in a schema-aware XSLT environment in three ways:

1. **Validate the input XML documents**: When you validate the input XML documents with an XML schema, the XSLT schema subsystem attaches type annotations to the nodes from the input document. This allows type-aware operations to be performed on the nodes in the XSLT stylesheet. Input validation can also ensure that the XSLT stylesheet doesn't process invalid input.

2. **Validate the output XML documents**: This is one of the biggest benefits of schema-aware XSLT stylesheet design from an overall application architecture point of view. By validating the output of XSLT transforms before handing over control of the XML stream to some other forward process, you can detect many errors early and avoid errors later in the processing chain.

3. **Import the element, attribute, and type information from a schema into the XSLT stylesheet**: Using the schema components in the stylesheet allows for enhanced type checking. For example, you can define data types of variables to be built-in or user-defined schema types. Similarly, you can define types for input and output parameters of XSLT functions and for XSLT template parameters.

You can put XML schemas to use during the compilation of the stylesheet or during the runtime—in other words, when the input XML document is transformed. The XSLT 2.0 specification says nothing about the compile-time usage of schemas, but computer language theory makes it well known that having extra type information during compile time allows the compiler to make compile-time optimizations—to generate efficient code, for example. It's also worth noting that you can write schemas inline in the XSLT stylesheet (I'll present an example for this later).
can be useful for small applications or to validate the temporary trees during the course of the transformation.

The following examples illustrate the three usages of schemas in stylesheets, as described previously.

**Validate the input XML documents**

The first example demonstrates how you can utilize input document validation in XSLT stylesheets. Listing 1 shows an XML document, named po.xml, that represents a purchase order.

**Listing 1. po.xml**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder orderid="10010"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="po1.xsd">
    <orderFrom>XYZ Ltd.</orderFrom>
    <shipAddress>
        <name>XYZ Ltd.</name>
        <address>123, Wisconsin Street</address>
        <city>London</city>
        <country>United Kingdom</country>
    </shipAddress>
    <billAddress>
        <name>XYZ Ltd.</name>
        <address>123, Wisconsin Street</address>
        <city>London</city>
        <country>United Kingdom</country>
    </billAddress>
    <item id="100" type="book">
        <title>Water for Elephants</title>
        <note>Author(s): Sara Gruen</note>
        <quantity>1</quantity>
        <price>18.34</price>
    </item>
    <item id="101" type="book">
        <title>Glass Castle: A Memoir</title>
        <note>Author(s): Jeannette Walls and Julia Gibson</note>
        <quantity>1</quantity>
        <price>23.09</price>
    </item>
    <item id="200">
        <title>5 Amp Electric plug</title>
        <quantity>5</quantity>
        <price>10.10</price>
    </item>
</PurchaseOrder>
```

Listing 2 shows the XML schema (named po1.xsd) for this document.

**Listing 2. po1.xsd**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="PurchaseOrder">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="orderFrom" type="xs:string"/>
                <xs:element name="shipAddress">
                    <xs:complexType>
                        <xs:sequence/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="billAddress">
                    <xs:complexType>
                        <xs:sequence/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="item" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```
Nothing is complicated about this schema. You need to be aware of the XML schema syntax to understand this example.

Now, write a simple XSLT 2.0 stylesheet that utilizes the schema in Listing 2 and works on the XML in Listing 1. Listing 3 shows the code for the stylesheet, which is named printitems1.xsl.

**Listing 3. printitems1.xsl**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/1999/XMLSchema"
version="2.0">

<xs:element name="name" type="xs:string"/>
<xs:element name="address" type="xs:string"/>
<xs:element name="city" type="xs:string"/>
<xs:element name="country" type="xs:string"/>
</xs:complexType>
</xs:element>

<xs:element name="billAddress">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="address" type="xs:string"/>
<xs:element name="city" type="xs:string"/>
<xs:element name="country" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:element>

<xs:element name="item" maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>
<xs:element name="title" type="xs:string"/>
<xs:element name="note" type="xs:string" minOccurs="0"/>
<xs:element name="quantity" type="xs:positiveInteger"/>
<xs:element name="price" type="xs:decimal"/>
</xs:sequence>
<xs:attribute name="id" type="xs:string" use="required"/>
<xs:attribute name="type" type="xs:string" use="optional"/>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="orderid" type="xs:string" use="required"/>
</xs:schema>
```
Using the Saxon-SA product, invoke the XSLT process as follows:

```java
com.saxonica.Transform po.xml printitems1.xsl
```

This produces the following output:

```
Source document is not a purchase order
Processing terminated by xsl:message at line 16 in printitems1.xsl
```

In this case, the second template in Listing 3 is invoked, because the input document was not validated.

Now invoke the XSLT transformation as follows:

```java
com.saxonica.Transform -val:strict po.xml printitems1.xsl
```

This produces the following output:

```
100: Water for Elephants
101: Glass Castle: A Memoir
200: 5 Amp Electric plug
```

In this case, the first template in Listing 3 is invoked, because the input document was validated with the corresponding schema.

This stylesheet illustrates the idea that you can execute useful processing in the stylesheet only if the input document is validated with the desired schema. If the XML document is not validated, then the stylesheet won’t do anything useful, as illustrated by the first output.

**Validate the output XML documents**

The next example demonstrates how you can request the validation of output trees prior to serialization. Listing 4 shows a stylesheet named printitems2.xsl.

**Listing 4. printitems2.xsl**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    version="2.0">
  <xsl:output method="xml" indent="yes" />  
  <xsl:import-schema>  
    <xs:schema>  
      <xs:element name="items">  
        <xs:complexType>  
          <xs:sequence>  
            <xs:element name="item" maxOccurs="unbounded">  
              <xs:complexType>  
                <xs:sequence>  
                  <xs:element name="title" type="xs:string"/>  
                  <xs:element name="note" type="xs:string" minOccurs="0"/>  
                  <xs:element name="quantity" type="xs:positiveInteger"/>  
                  <xs:element name="price" type="xs:decimal"/>  
                </xs:sequence>  
              </xs:complexType>  
            </xs:element>  
          </xs:sequence>  
        </xs:complexType>  
      </xs:element>  
    </xs:schema>  
  </xsl:import-schema>
</xsl:stylesheet>
```
Note that the `xsl:validation="strict"` option on the `<items>` tag causes the `<items>` element to be validated as it gets generated from the transformation.

Invoke the XSLT process as follows (the input XML remains same):

```java
java com.saxonica.Transform po.xml printitems2.xsl
```

The following output is produced:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="200">
  <item xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <title>5 Amp Electric plug</title>
    <quantity>5</quantity>
    <price>10.10</price>
  </item>
</items>
```

This is the intended output.

Suppose you want to modify a portion of the stylesheet, as follows:

```xml
<xsl:template match="/PurchaseOrder">
  <itemsTag xsl:validation="strict">
    <xsl:copy-of select="item[price < 15]" />
  </itemsTag>
</xsl:template>
```

Note that you have modified the root element name from `<items>` to `<itemsTag>`. Running the same command line as shown previously produces the following output:

```
Error on line 32 of file:/E:/xml/sa-xslt/printitems2.xsl:
XTEI512: There is no global element declaration for itemsTag, so strict validation will fail
Failed to compile stylesheet. 1 error detected.
```
This error occurs during transformation, because the validation of the output tree with the inline schema did not succeed. As demonstrated, you cannot produce an invalid output from the XSLT transformation.

Follow another interesting example in **Listing 5** for the validation of the output XML tree. This example illustrates how you can request validation of different parts of output tree with different schemas.

**Listing 5. outputval.xsl**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">
  <xsl:output method="xml" indent="yes" />

  <!-- import 1st schema -->
  <xsl:import-schema>
    <xs:schema>
      <xs:element name="x">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="y" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </xsl:import-schema>

  <!-- import 2nd schema -->
  <xsl:import-schema>
    <xs:schema>
      <xs:element name="p">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="q" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </xsl:import-schema>

  <xsl:template match="/">
    <xsl:variable name="temp1">
      <x>
        <y/>
      </x>
    </xsl:variable>
    <xsl:variable name="temp2">
      <p>
        <q/>
      </p>
    </xsl:variable>
    <result>
      <xsl:copy-of select="$temp1" validation="strict" />
      <xsl:copy-of select="$temp2" validation="strict" />
    </result>
  </xsl:template>
</xsl:stylesheet>
```
Now invoke the XSLT transformation as follows:

```
java com.saxonica.Transform outputval.xsl outputval.xsl
```

Note that in this command line, you use the stylesheet itself as the input XML. Here the stylesheet acts as a dummy input XML.

The above command line produces the following output:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <x>
</x>
  <y/>
</result>
```

This is the intended output. Nothing wrong happened here, because validation of tree fragments (shown with bold in Listing 5) succeeded without problems with the two inline schemas.

Now change the root template (xsl:template match="/" as below:

```
<xsl:template match="/"
  <xsl:variable name="temp1">
    <x>
      <something/>
    </x>
  </xsl:variable>
  <xsl:variable name="temp2">
    <p>
      <q/>
    </p>
  </xsl:variable>
  <result>
    <xsl:copy-of select="$temp1" validation="strict" />
    <xsl:copy-of select="$temp2" validation="strict" />
  </result>
</xsl:template>
```

You have introduced a junk tag, <something/> which is invalid as per any of the two inline schemas (from Listing 5).

Now run the stylesheet of Listing 5, with the root template changed as above (the command line remaining the same).

The output produced by the transformation now is:

```
Validation error on line 47 of file:/E:/xml/sa-xslt/outputval.xsl:
XTTE1510: In content of element <x>: The content model does not allow element <something>
to appear here. Expected: y (See http://www.w3.org/TR/xmlschema-1/#cvc-complex-type clause 2.4)
Transformation failed: Run-time errors were reported
```
Since you introduced a validation error in the generated markup, the transformation did not succeed. This example illustrates that XSLT 2.0 is very flexible, where you want the validation in the output tree to occur.

**Import type information from a schema**

Now look at another example, which uses the schema-defined user types as function parameters. This is a powerful concept and illustrates that you can extend the type system of XSLT in an unlimited way. **Listing 6** shows the schema, named po2.xsd.

**Listing 6. po2.xsd**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="PurchaseOrder" type="POType" />
  <xs:complexType name="POType">
    <xs:sequence>
      <xs:element name="orderFrom" type="xs:string"/>
      <xs:element name="shipAddress">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="billAddress">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
          <xs:attribute name="id" type="xs:string" use="required"/>
          <xs:attribute name="type" type="xs:string" use="optional"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

This schema is not much different than po1.xsd in **Listing 2**. The only difference is that it names the `POType` type explicitly rather than using it anonymously in the schema. You will use this type name in the function parameter.
Now try to run the ordersummary.xsl stylesheet, which uses the po2.xsd schema in Listing 6. This stylesheet, in Listing 7, displays an order summary (as XHTML) for the purchase order represented by the sample XML.

**Listing 7. ordersummary.xsl**

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                   xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:my="http://localhost/myfunctions"
                   exclude-result-prefixes="xs my"
                   version="2.0">
  <xsl:output method="xhtml" />
  <xsl:import-schema schema-location="po2.xsd" />
  <xsl:import-schema namespace="http://www.w3.org/1999/xhtml"
                       schema-location="http://www.w3.org/2002/08/xhtml/xhtml1-transitional.xsd"/>
  <xsl:template match="/PurchaseOrder">
    <html xmlns="http://www.w3.org/1999/xhtml" xsl:validation="strict">
      <head>
        <title>Order Summary</title>
      </head>
      <body>
        <h2>Order from: <xsl:value-of select="orderFrom" /></h2>
        <table>
          <tr>
            <td>Item type</td>
            <td>Total for the item type</td>
          </tr>
          <xsl:for-each-group select="item" group-by="if (@type) then @type
          else 'uncategorized item'">
            <tr>
              <td><xsl:value-of select="current-grouping-key()" /></td>
              <td><xsl:value-of select="my:categoryTotal(..,
              current-grouping-key())" /></td>
            </tr>
          </xsl:for-each-group>
        </table>
        Total amount for the order: <xsl:value-of select="my:orderTotal(.)" />
      </body>
    </html>
  </xsl:template>

  <!-- function to find order amount for a particular category of items -->
  <xsl:function name="my:categoryTotal" as="xs:decimal">
    <xsl:param name="po" as="element(*, POType)" />
    <xsl:param name="category" as="xs:string" />
    <xsl:choose>
      <xsl:when test="not($category = 'uncategorized item')">
        <xsl:sequence select="sum($po/item[@type = $category]/price)" />
      </xsl:when>
      <xsl:otherwise>
        <xsl:sequence select="sum($po/item[not(@type)]/price)" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:function>

  <!-- function to find total order amount -->
  <xsl:function name="my:orderTotal" as="xs:decimal">
```

Schema-aware processing with XSLT 2.0
Now invoke the XSLT transformation as follows:

```
java com.saxonica.Transform po.xml ordersummary.xsl
```

This produces the following output:

```
Error on line 32 of file:/E:/xml/sa-xslt/ordersummary.xsl:
XPTY0004: Required item type of first argument of my:categoryTotal() is element(*, POType); supplied value has item type element(PurchaseOrder, xs:anyType)
In template at line 14 in file:/E:/xml/sa-xslt/ordersummary.xsl
```

Try to understand what this error means and how to resolve it. Because you didn't validate the input XML document, the proper type annotations didn't get attached to the XML nodes. As a consequence, the element nodes had the `xs:anyType` type. The error occurred because the `my:categoryTotal` function expected the parameter value with `POType` type.

Run the transformation as follows:

```
java com.saxonica.Transform -val:strict po.xml ordersummary.xsl
```

Note that you add the `-val:strict` option on the command line. This time, you get the correct output, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Order Summary</title>
  </head>
  <body>
    <h2>Order from: XYZ Ltd.</h2>
    <table>
      <tr>
        <td colspan="1" rowspan="1">Item type</td>
        <td colspan="1" rowspan="1">Total for the item type</td>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">book</td>
        <td colspan="1" rowspan="1">41.43</td>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">uncategorized item</td>
        <td colspan="1" rowspan="1">10.1</td>
      </tr>
    </table>
    Total amount for the order: 51.53
  </body>
</html>
```

Consider these interesting points in this example.
• First, because the input document was validated, the function got the arguments with the correct type.
• Second, adding `<html xmlns="http://www.w3.org/1999/xhtml" xsl:validation="strict">` in the stylesheet caused the XHTML output to be validated against the XHTML schema (whose location is specified by the instruction `schema-location="http://www.w3.org/2002/08/xhtml/xhtml1-transitional.xsd"`).
• Finally, try introducing some errors in the output XHTML syntax, and you'll see that the validation with the XHTML schema will fail. Note that the XSLT processor fetches the schema from the Web, so the machine should be connected to the Internet.

Summary

This article demonstrates the capabilities of a schema-aware XSLT system. Making XSLT stylesheets schema-aware produces the following benefits:

• You can perform type-aware operations on the nodes by validating the input trees and attaching the type annotations to the XML nodes. This also ensures that invalid input is not processed by the stylesheet.
• You can validate output trees with a particular schema, thereby making sure that you don't produce invalid output from the XSLT transformation.
• You can assign types to XSLT variables, function/template parameters, and return values. This provides enhanced static typing, which is beneficial during the compilation phase of the stylesheet.
• Enhanced compile-time type checking reduces the likelihood of errors popping up in later phases. The sooner that you detect the errors, the less amount of time you require to fix them.
• Having user-defined schema types available in the stylesheet makes the type system of XSLT infinitely extensible. As a result, the stylesheet comes closer to solving the business problem.
Related topics

- **What kind of language is XSLT?** by Michael Kay (developerWorks, April 2005): Read this introduction to the XSLT language and learn where the language comes from, what it's good at, and why you should use it.
- **XSL Transformations (XSLT) Version 2.0**: In the W3C specification, learn about the syntax and semantics of XSLT 2.0.
- **XML Path Language (XPath) 2.0**: Read the W3C specification that defines the XPath 2.0 language.
- **XQuery 1.0 and XPath 2.0 Data Model**: Read the W3C specification that defines the XQuery 1.0 and XPath 2.0 Data Model.
- **XML Schema Part 0: Primer Second Edition**: Learn how to create schemas in this W3C primer document.
- **XML Schema Part 1: Structures Second Edition**: This W3C document sets out the structural part of the XML Schema definition language. Read this W3C specification to learn more about it.
- **XML Schema Part 2: Datatypes Second Edition**: Read the W3C document that defines facilities for defining datatypes to be used in XML schemas.
- **XSLT 2.0 Programmer's Reference** (Michael Kay, Wrox, 2004): Read this book as a good source to learn about the XSLT 2.0 language.
- **XPath 2.0 Programmer's Reference** (Michael Kay, Wrox, 2004): In this book, find a good explanation of the XPath 2.0 language.
- **XSLT 2.0 and XPath 2.0 Programmer's Reference** (Michael Kay, Wrox, 2008): Read this latest book about XSLT 2.0 and XPath 2.0. The book supersedes the previous two books by Michael Kay.
- **Saxonica: XSLT and XQuery Processing**: Download a free 30-day evaluation license of the Saxon-SA schema-aware XSLT 2.0 processor. All the examples discussed in this article are tested with Saxon-SA.
- **AltovaXML**: Download the free AltovaXML 2008, which includes the XSLT 1.0/2.0 engine, the XQuery engine, and the XML validator.
- **IBM certification**: Find out how you can become an IBM-Certified Developer.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **IBM trial software**: Build your next development project with trial software available for download directly from developerWorks.
- **developerWorks podcasts**: Listen to interesting interviews and discussions for software developers.