Modeling Web services, Part 1

XML Schema

Simon Johnston November 29, 2005

This article is the first in a three-part series looking at the use of UML modeling (in particular using IBM Rational Software Architect) to model the detailed design of standardized Web services. Where other articles--see Resources, [1]--have focused on the modeling of software services, this series describes the detailed modeling of Web services, the capture of specific technology decisions and artifacts corresponding to XML Schema, WSDL (Web Service Definition Language), and more. This article will focus on modeling and generating XML Schema.

Introduction

This series of articles describes how to model, in detail, Web service-related structures, and how you can use IBM® Rational® Software Architect (RSA) to realize these models in actual implementations. The series will show, with practical examples, how RSA can be used to capture the design of Web services in a simple and maintainable fashion, while capturing much of the explicit detail required in the final artifacts.

This series will look at the following topics:

- **Part One - Modeling and generating XML Schema.**
- Part Two - Modeling and generating WSDL.
- Part Three - Applying patterns to the generation of Web Service Artifacts.

You will start by looking at modeling XML Schema; first, because using only WSDL requires that messages be described with schema, but more importantly because message and indeed all data design should be an explicit and carefully managed activity rather than a side-effect of interface design. The data structures you will look at in this article will be reused in the next article as components in WSDL messages.

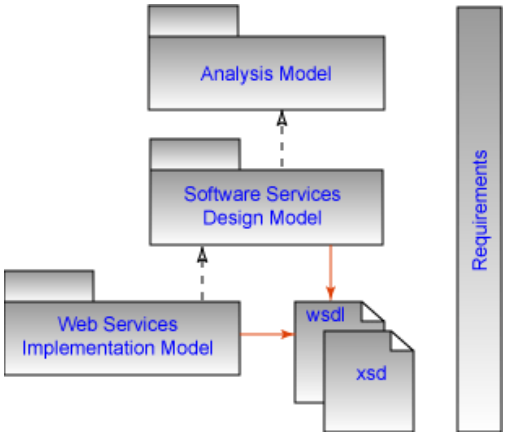
Why model Web services in detail?

One interesting question is whether modeling web services in such detail provides any value. After all, isn't a model intended to abstract away unnecessary detail so as to allow allow you to understand the system in a way that may not be clear at the implementation level? Well obviously the answer on one hand is yes, you want to provide a model of the underlying implementation

artifacts, yet this approach allows you to leverage the capabilities offered by visual languages for more easily expressing complex relationships and scaling up to present larger data sets.

This approach of modeling the detailed design elements fits well within an overall Model-Driven Development (MDD) approach, in which you can define multiple connected models presenting progressively refined views of the problem or solution. This approach is demonstrated by Figure 1. This particular configuration of models is specific to modeling services, and so does not use the terminology often associated with Object Management Group's (OMG) Model-Driven Architecture (MDA), including terms such as Computation Independent, Platform Independent, and Platform-Dependent Models.

Figure 1. An MDD configuration for Web service development



The elements of the configuration are described as follows:

- **Analysis model:** A highly abstract model containing elements representative of the solution, yet without any platform- or architecture-specific concerns
- **Software services design model:** A model -- derived from the analysis model -- for which the Service-Oriented Architectural (SOA) style has been applied, yet the actual implementation technology and platform are not introduced
- **Web service implementation model:** A model -- derived from the software services design model -- for which the particular details of a Web services implementation are added and refined to ensure that generated artifacts are precisely specified
- **XML or WSDL artifacts:** Actual deployable artifacts, including XML Schema and WSDL documents
- **Requirements:** Requirements that are used in the development of each model, and also used to configure the transformation(s) between models

You will notice that there are two ways of getting to the actual artifacts, either directly from the software services model or via the Web service model. In the first case a default transformation is performed from the more abstract software services model. If, however, more control over the generated artifacts is required, then the more detailed Web service model can be used to define the target more precisely.

Modeling XML schema

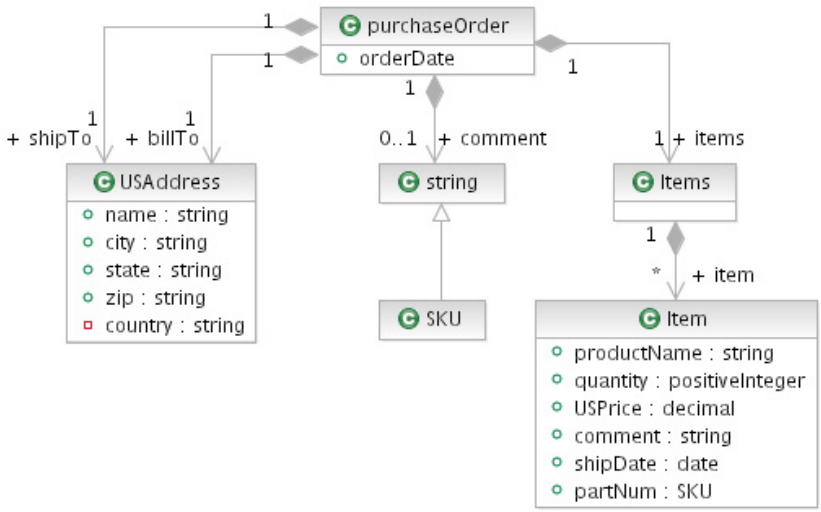
In modeling XML Schema--see Resources, [3,4]--it is important to realize that there are some relatively "obvious" mappings -- from the UML object-oriented view of the world to the XML Schema document-centric view -- that allow you to generate a schema automatically from a simple class model. Unfortunately, this is usually not a satisfying experience, as these default-generated schema elements rarely match the kinds of structures you would create by hand. So the model has to be created in such a way that the generated XML Schema can be tuned to create exactly what the designer had in mind.

The RSA UML to XML schema transformation

The examples shown in this article were developed using RSA--see Resources, [6]. Version 6.0.1 introduced a number of new features, including the UML to XML schema transformation described in the following sections. The UML to XSD transformation is one of a number of model-to-model and model-to-text transforms delivered with RSA and built using an open extensibility interface, so new transformations can be developed by you or by third parties.

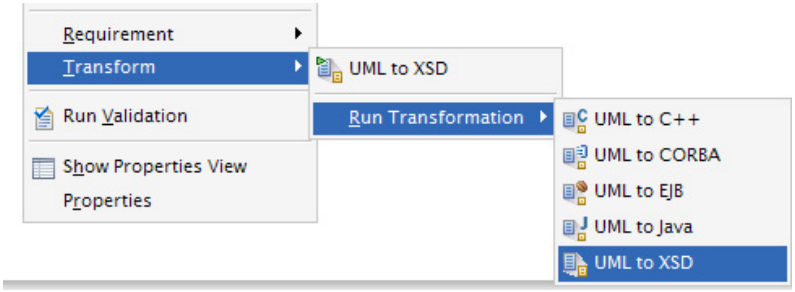
Without any further delay let's look at a simple example of a model for which you wish to generate an XML schema. To do so, you will use a form of the Purchase Order from the XML Schema Primer itself (Resources, [5]), as shown in Figure 2.

Figure 2. The initial Purchase Order model



To run the UML to XSD Transformation, right-click the package containing the purchase order elements above, either on the diagram surface or in the model explorer. Click **Transform > Run Transformation > UML to XSD**, as shown in Figure 3. When you select a transformation, you will be presented with a dialog where you may enter additional transformation options. The first time you run the transform, be sure to set the Eclipse project containing your models as the target of the transform.

Figure 3. The RSA Transformation menu



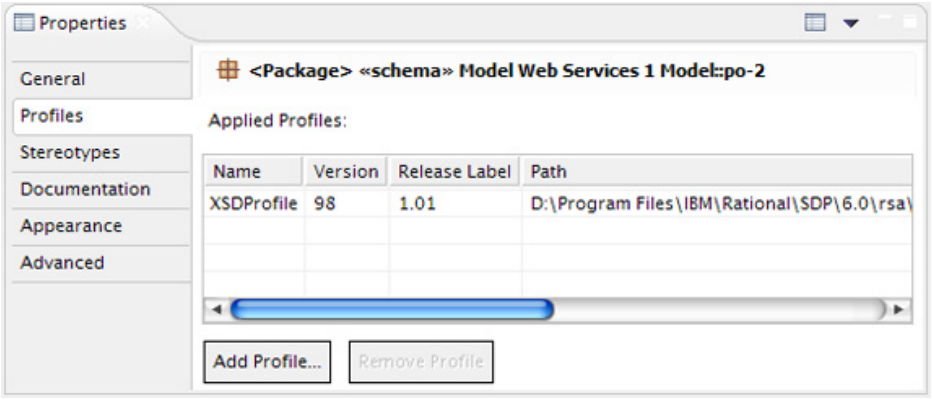
The generated schema for this model looks like that shown in Listing 1. Note that the transformation will generate a warning because the model does not specify a target namespace, a requirement for the schema.

Listing 1. Generated purchase order schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="purchaseOrder">
    <xsd:sequence>
      <xsd:element name="orderDate"/>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element minOccurs="0" name="comment" type="string"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="item" type="Item"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="string"/>
      <xsd:element name="city" type="string"/>
      <xsd:element name="state" type="string"/>
      <xsd:element name="zip" type="string"/>
      <xsd:element name="country" type="string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Item">
    <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity" type="xsd:positiveInteger"/>
      <xsd:element name="USPrice" type="xsd:decimal"/>
      <xsd:element name="comment" type="string"/>
      <xsd:element name="shipDate" type="xsd:date"/>
      <xsd:element name="partNum" type="SKU"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="SKU">
    <xsd:complexContent>
      <xsd:extension base="string">
        <xsd:sequence/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

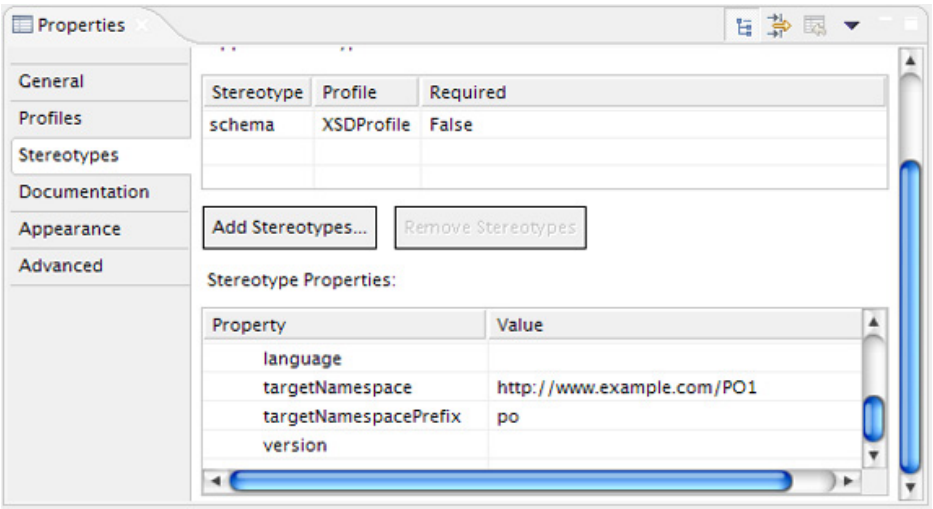
So let's look at some of the issues with the generated schema by comparing it with the hand-crafted version in the primer. First, you need to add the target namespace, as identified by the transform itself. To do this you have to apply a UML profile to your model, a profile that allows you to fine-tune the generation by providing hints to the transformation using UML stereotypes to capture these additional properties. To apply the profile and stereotype, select the package containing your schema elements, then open the Properties window and the **Profiles** tab, as shown in Figure 4.

Figure 4. Applying the profile to your model



Click **Add Profile** and select the XSDProfile from the list. This will make the transformation stereotypes available, so select the same package and open the **Stereotypes** tab in the Properties window. Once you have added the «schema» stereotype, scroll down through the stereotype properties and add the required values to the targetNamespace and targetNamespacePrefix properties, as shown in Figure 5.

Figure 5. The «schema» Stereotype Properties



These actions will allow the transformation to generate a more complete XML <schema/> element. But before you rerun the transformation, add one more item. Open the **Documentation** tab in the Properties window and enter the text `Purchase order schema for Example.com. Copyright 2000`

`Example.com. All rights reserved.`. Now, rerun the transformation and check out the generated schema -- just the top of the file for now -- as shown in Listing 2.

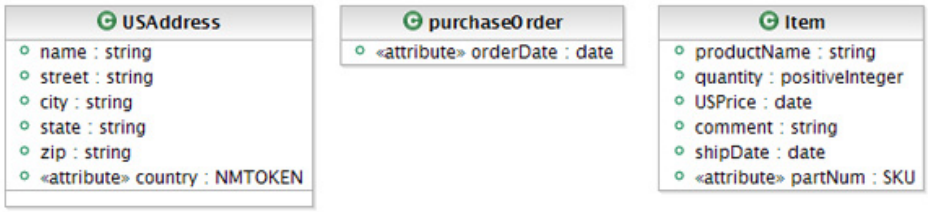
Listing 2. Regenerated schema

```
<xsd:schema xmlns:po="http://www.example.com/PO1"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.com/PO1">
  <xsd:annotation>
    <xsd:documentation>Purchase order schema for Example.com.
    Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>
```

As you can see, you now have not only specified your target namespace, but also created a namespace value `PO` to qualify the schema and included the documentation annotation from the example.

Now, you need to make a second change, because not all of the UML properties shown in Figure 1 really become XML elements, as shown in Listing 1. You want some of these UML properties to be generated as XML attributes, so again you use the profile to tell the transformation what to do; in this case, you will need to select the properties and add the stereotype «attribute» as shown in Figure 6. Before you finish with your attributes, also change the type of the property USAddress::country to the XML Schema type NMTOKEN, and specify a default value of `us`. To set the correct type, select the Country property, open the property's **General** tab, and click **Select type**. In the resulting dialog, scroll down to the XSDDataTypes model and choose the required type.

Figure 6. Specifying XML attributes in the model



But, now you have an additional issue: the class SKU is generated as a complex type, which you cannot use as the type for an attribute (if you go back to the example schema you can see that SKU is a simple type, with a specified pattern). So, apply the stereotype «simpleType» to the SKU class, and set the pattern stereotype property to `\d{3}-[A-Z]{2}`. Also, ensure that the SKU class inherits from the string class in the XSDDataTypes model, and not from a local string class. This completes the attributes and simple type definition, as shown in the fragments presented in Listing 3.

Listing 3. Code fragments illustrating changes

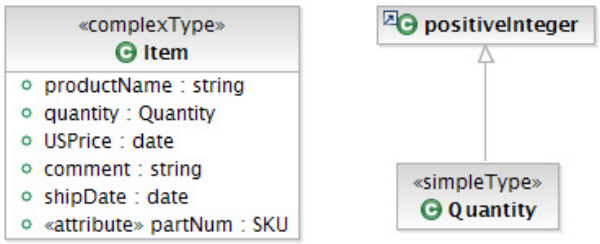
```
...
<xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute default="US" name="country" type="xsd:NMTOKEN"/>
  </xsd:complexType>
...
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
  </xsd:simpleType>
```

As you can see, you are getting much closer to the hand-crafted example purchase order schema in the primer, except for two more issues:

- Anonymous types in the Item's complex type
- The need for global element declarations

Looking at the anonymous types first, you need to add the stereotype «complexType» to the model element Item. This may seem redundant, as the default transformation creates a complex type for this model type, but take a closer look at the properties of the stereotype. For the purpose of this example, just set the value of the anonymous property to `True`. You also need to do something similar for the type of the Quantity property: you need to create an anonymous simple type. To do this, first create a new class and name it Quantity (in this case). Next, create a generalization to the positiveInteger class in the XSDDataTypes model. Now, in the stereotype properties for Quantity, set `anonymous=True` and `maxExclusive=100`, and then change the type of Item::quantity to the new Quantity class as shown in Figure 7.

Figure 7. Specifying the Quantity anonymous simple type



These changes result in a more complete definition for the Item complex type, shown in Listing 4.

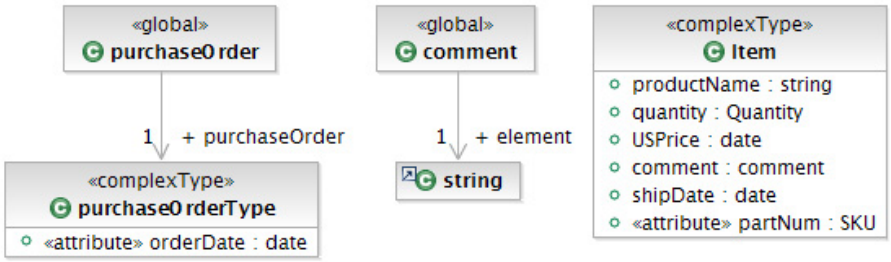
Listing 4. The Item complex type definition

```
<xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
```

```
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice"  type="xsd:decimal"/>
          <xsd:element ref="comment"   minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

The final issue is that the purchase order itself is defined in the primer as a global element declaration, with a global complex type. In addition, the comment element is also globally defined as a string type. To define global element declarations in the model is a little more effort than the "markings" you added to the model above. These changes are shown in Figure 8.

Figure 8. Global Element Declarations



To achieve this, you had to do the following to the model:

- Rename purchaseOrder to purchaseOrderType
- Add the stereotype «complexType» to purchaseOrderType (not absolutely necessary, but it makes the meaning of the model more obvious)
- Create a new class called *purchaseOrder*, and add the stereotype «global»
- Add a property to purchaseOrder, also called *purchaseOrder*, with the stereotype «element» and typed as purchaseOrderType
- Create a new class called *comment*, and add the stereotype «global»
- Add a property to comment, also called *comment*, with the stereotype «element» and typed as XSDDataTypes::string
- Change the type of Item::comment to the newly created comment class
- Change the multiplicity of Item::comment from 1 to 0..1

The result of all of these changes is shown in Listing 5: the generated schema, which while it may be generated in a different textual order to the original, is identical in meaning to the primer example.

Listing 5. The generated schema is now identical in meaning to the primer example

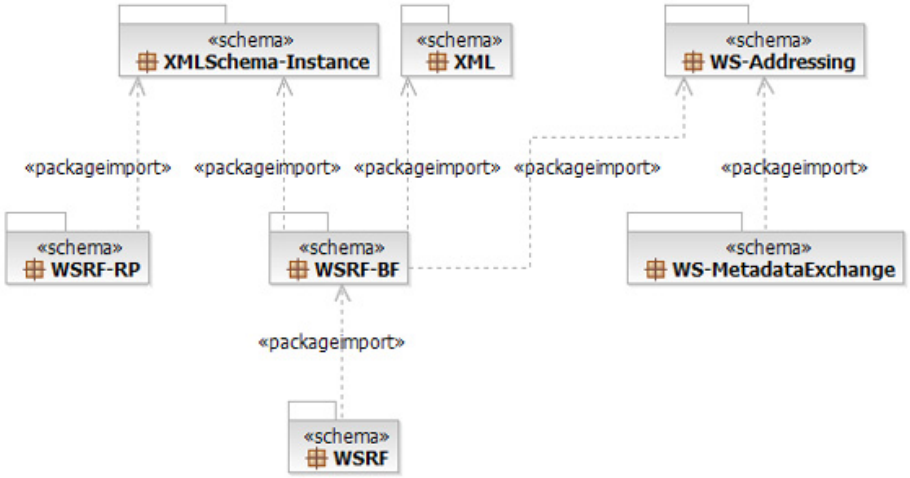
```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:PO="http://www.example.com/PO1"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.com/PO1">
  <xsd:annotation>
    <xsd:documentation>Purchase order schema for Example.com.
    Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute default="US" name="country" type="xsd:NMTOKEN"/>
  </xsd:complexType>
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="purchaseOrder" type="PO:purchaseOrderType"/>
  <xsd:element name="element" type="xsd:string"/>
  <xsd:complexType name="purchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="PO:USAddress"/>
      <xsd:element name="billTo" type="PO:USAddress"/>
      <xsd:element name="items" type="PO:Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>
  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="item">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice" type="xsd:date"/>
            <xsd:element minOccurs="0" ref="PO:element"/>
            <xsd:element name="shipDate" type="xsd:date"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="PO:SKU"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

A more complete example

As a part of a project within IBM Software dealing with Rational products, we have developed a library model that has representations of many of the Web Services family of specifications

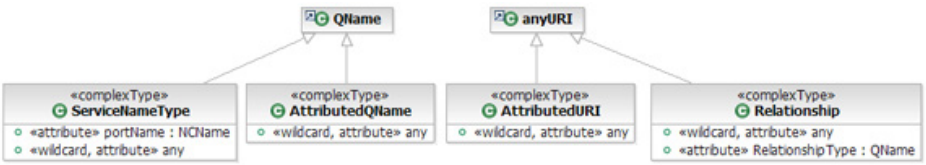
(commonly known as WS-*). Figure 9 shows the set of standards modeled so far, and also shows the use of UML «packageimport» relationships to denote the dependencies between the schema. Specifically, these UML relationships will generate XML Schema import statements, allowing us to model domain schema easily and naturally as a set of related component schema.

Figure 9. WS-* schema, and dependencies



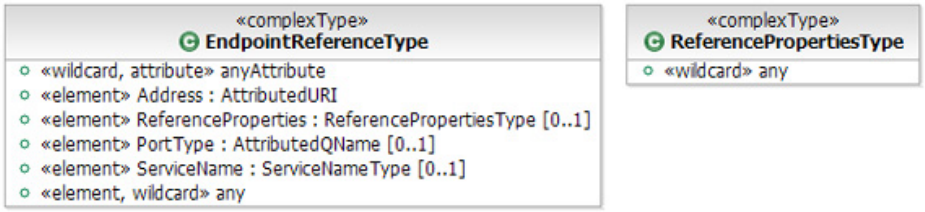
If you focus on the WS-Addressing schema as representative of the structures found in most of these schema, you can see that it has a number of complex types derived from simple types (we saw simple type derivation in the purchase order example above). Two of these complex types also add additional attributes: ServiceNameType::portName and Relationship::RelationshipType. More interestingly, though, all four of these new complex types also allow for wildcard attributes, as shown in Figure 10. This is an XML schema capability to allow for extension in the future by letting authors add additional attributes to these standard types. To accomplish this, use the combination of «wildcard» and «attribute» stereotypes (the name of the attribute is now immaterial, though by convention we either use *any* or *anyAttribute*).

Figure 10. complexTypes derived from XSD types



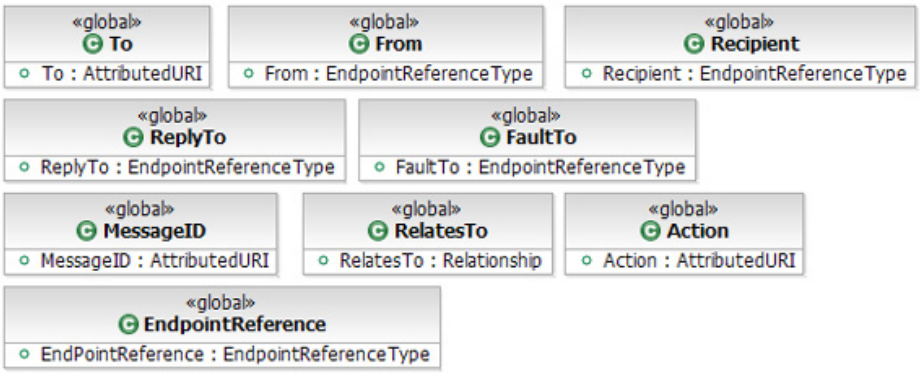
Next, look at the new complex types introduced by the schema. Again, notice that not only does the type EndpointReferenceType allow for wildcard attributes, but that it also allows for wildcard elements (using a combination of «wildcard» and «element» stereotypes). You can also see that we have identified three attributes as optional using UML multiplicity [0..1], as shown in Figure 11.

Figure 11. WS-Addressing complexTypes



Finally WS-Addressing exposes a set of global element declarations, which are the elements that the standard expects users to reuse. These are shown in Figure 12.

Figure 12. Reusable element declarations



Running this through the UML to XSD Transform results in the schema shown in Listing 6. Again, this can be compared to the standard schema from the W3C.

Listing 6. The schema resulting from the UML to XSD Transform

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://schemas.xmlsoap.org/ws/2003/03/addressing"
           xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           blockDefault="#all">
  <xs:element name="EndpointReference" type="wsa:EndpointReferenceType"/>
  <xs:complexType name="EndpointReferenceType">
    <xs:sequence>
      <xs:element name="Address" type="wsa:AttributedURI"/>
      <xs:element name="ReferenceProperties" type="wsa:ReferencePropertiesType" minOccurs="0"/>
      <xs:element name="PortType" type="wsa:AttributedQName" minOccurs="0"/>
      <xs:element name="ServiceName" type="wsa:ServiceNameType" minOccurs="0"/>
      <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>
             If "Policy" elements from namespace
             "http://schemas.xmlsoap.org/ws/2002/12/policy#policy" are used,
             they must appear first (before any extensibility elements).
          </xs:documentation>
        </xs:annotation>
      </xs:any>
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
  <xs:complexType name="ReferencePropertiesType">
```

```
    <xs:sequence>
      <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ServiceNameType">
    <xs:simpleContent>
      <xs:extension base="xs:QName">
        <xs:attribute name="PortName" type="xs:NCName"/>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="Relationship">
    <xs:simpleContent>
      <xs:extension base="xs:anyURI">
        <xs:attribute name="RelationshipType" type="xs:QName" use="optional"/>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="AttributedQName">
    <xs:simpleContent>
      <xs:extension base="xs:QName">
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="AttributedURI">
    <xs:simpleContent>
      <xs:extension base="xs:anyURI">
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:element name="MessageID" type="wsa:AttributedURI"/>
  <xs:element name="RelatesTo" type="wsa:Relationship"/>
  <xs:element name="To" type="wsa:AttributedURI"/>
  <xs:element name="Action" type="wsa:AttributedURI"/>
  <xs:element name="From" type="wsa:EndpointReferenceType"/>
  <xs:element name="ReplyTo" type="wsa:EndpointReferenceType"/>
  <xs:element name="FaultTo" type="wsa:EndpointReferenceType"/>
  <xs:element name="Recipient" type="wsa:EndpointReferenceType"/>
</xs:schema>
```

Hopefully this brief tour through RSA's ability to model XSD will allow you to capture domain-specific schema, as well as Web service messages in your model. Furthermore, it should provide you with opportunities to combine and componentize schema in a way that is much more difficult to visualize with many of today's other schema editors.