Expressing Business Rules

Issue

A community-wide XML Schema is created. How should members of the community constrain the XML Schema to meet their business rules?

Examples of Business Rules

1. Level 1 managers can sign off on purchase requests under $10K.

2. We only accept Visa, American Express, and MasterCard.

3. Flights booked under 50% capacity shall be cancelled.

Definition of Business Rules

A business rule is a local constraint. It is an additional data constraint on top of a community's structure and content data rules.

It is Important to Declaratively Express Business Rules

With XML Schema 1.0 there was limited support for expressing business rules. Many developers resorted to implementing them in procedural code and database code. This resulted in business rules that were opaque to business managers and customers, and were difficult to change and assess. This made it difficult for businesses to be agile.

It is important to declaratively express business rules. Make business rules visible, understandable, and modifiable.

XML Schema 1.1 can express many business rules. Thus, business rules that were previously codified in procedural or database logic may now be declaratively expressed.

Example of a Business Rule that is Declaratively Expressed

The following XML instance document is a purchase request:

```
<purchase-request>
    <item>Widget</item>
```

```
    <cost>1500</cost>
    <signature-authority>Level1</signature-authority>
</purchase-request>
```

Validating the instance document involves checking that it uses the correct elements, the elements are arranged in the proper fashion, and the content of each element is correct. These structure and content rules are specified at the community level and are readily expressed in XML Schema 1.0:

```
<element name="purchase-request">
  <complexType>
    <sequence>
      <element name="item" type="Name"/>
      <element name="cost" type="nonNegativeInteger"/>
      <element name="signature-authority">
        <simpleType>
          <restriction base="string">
            <enumeration value="Level1"/>
            <enumeration value="Level2"/>
            <enumeration value="Level3"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>
</element>
```

Members of the community create and exchange purchase requests conforming to the XML Schema. However, each member may have additional constraints they wish to impose on the community schema. For example, one member needs to enforce this business rule:

> Level 1 managers can only sign off on purchase requests under $10K.

Below are three approaches the member may use to enforce its business rule.

Approach #1: Override and Constrain

One approach is for the member to create an XML Schema that overrides the element declaration for purchase-request and constrains it using the new XML Schema 1.1 <assert> element.

Here's the structure of the <assert> element:

```
<assert  test="xpath"/>
```

Either XPath 1.0 or XPath 2.0 may be used.

Here's how to express the business rule:

```
<assert  test="(po:signature-authority eq 'Level1') and
                (xs:integer(po:cost) lt 10000)"/>
```

Read as: If the value of <signature-authority> is 'Level1' then the value of <cost> must be less than 10K.

The <assert> element is positioned within an XML Schema 1.1 document in the same location as attributes.

The new XML Schema 1.1 <override> element replaces the <redefine> element, which has been deprecated. The <override> element is more powerful; for example, it can be used to redefine an element declaration, which was not possible with the <redefine> element.

The following XML Schema snippet shows how the purchase-request element in the community-wide XML Schema is constrained by the member's business rule:

```
<override schemaLocation="purchase-request.xsd">
  <element name="purchase-request">
    <complexType>
      <sequence>
        <element name="item" type="Name"/>
        <element name="cost" type="nonNegativeInteger"/>
        <element name="signature-authority">
          <simpleType>
            <restriction base="string">
              <enumeration value="Level1"/>
              <enumeration value="Level2"/>
              <enumeration value="Level3"/>
            </restriction>
          </simpleType>
        </element>
      </sequence>
      <assert  test="(po:signature-authority eq 'Level1') and
                     (xs:integer(po:cost) lt 10000)"/>
    </complexType>
  </element>
</override>
```

The member's XML Schema is identical to the community's XML Schema except the member has added an additional constraint to meet its business rule.

Using this approach the member validates XML instance documents against its own XML Schema, not the community-defined XML Schema.

Advantage

This approach scales well: Each member can constrain the community-agreed-to XML Schema, without *a priori* co-ordination with the entire community. This has the additional benefit that members can change their business rules as often as desired.

Only one schema needs to be created, updated, distributed, and stored.

Disadvantage

Interoperability is jeopardized if members change or delete the element declarations in the community-agreed-to XML Schema. For example, a member overrides the purchase request and changes this element declaration:

```
<element name="item" type="Name"/>
```

to this:

```
<element name="product" type="Name"/>
```

While "product" may be the term that the member uses, the community has agreed on the term "item." To facilitate interoperability it is important that each member not alter the community-agreed-to XML Schema.

Approach #2: Create a Separate XML Schema for Business Rules

In the first approach a member validates XML instance documents against its own XML Schema. In this second approach a member validates XML instance documents against the community XML Schema; if it passes validation then the instance document is validated against a second XML Schema which expresses the business rule. Thus this second approach uses a validation pipeline.

Here is the community-agreed-to XML Schema for purchase requests:

```
<element name="purchase-request">
  <complexType>
```

```
      <sequence>
        <element name="item" type="Name" />
        <element name="cost" type="nonNegativeInteger" />
        <element name="signature-authority">
          <simpleType>
            <restriction base="string">
              <enumeration value="Level1" />
              <enumeration value="Level2" />
              <enumeration value="Level3" />
            </restriction>
          </simpleType>
        </element>
      </sequence>
    </complexType>
  </element>
```

Each member uses that XML Schema to validate XML instance documents. The schema checks that instances are structurally valid, but does not check member-specific business constraints.

Each member creates a supplementary XML Schema to express its member-specific business constraints (i.e., its business rules). Here is the XML Schema that one member created to express its business rule:

```
  <element name="purchase-request">
    <complexType>
      <sequence>
        <any maxOccurs="unbounded" processContents="lax" />
      </sequence>
      <assert   test="(po:signature-authority eq 'Level1') and
                      (xs:integer(po:cost) lt 10000)" />
    </complexType>
  </element>
```

The <any> element is used since the content of <purchase-request> has already been expressed by the community XML Schema. The <assert> element expresses the member's business rule:

> Level 1 managers can only sign off on purchase requests under $10K.

Each member validates XML instance documents against two (or more) XML Schemas in a pipeline fashion. XProc is the XML Pipeline language and is recommended for implementing this validation pipeline.

Advantage

This approach scales well: No *a priori* co-ordination with the entire community is needed. Members can change their business rules as often as desired.

There is no danger of impacting interoperability by a member using an altered version of the community's XML Schema.

The division of labor yields enhanced productivity.

Disadvantage

Multiple schemas must be created, updated, distributed, and stored. Machinery (such as XProc) must be employed to orchestrate the multiple validations.

Approach #3: Express Each Member's Business Rules in the Community XML Schema

In the above two approaches the business rule implementations were distributed across the members. In this approach the business rules are centralized inside the community XML Schema.

In the community XML Schema an attribute is added to <purchase-request> to indicate which member the purchase request is intended for:

```
<purchase-request member="Company1">
        <item>Widget</item>
        <cost>1500</cost>
        <signature-authority>Level1</signature-authority>
</purchase-request>
```

The constraints on <purchase-request> vary depending on the value of the member attribute. This variability is accomplished using the new XML Schema 1.1 <alternative> element. Here's its format:

```
<alternative test="xpath" type="type" />
```

If the *xpath* expression evaluates to true then *type* is used.

The <alternative> element enables conditional types. For example, suppose that the community consists of 3 members, which we refer to as Company1, Company2, and Company3. Each member has a different business rule regarding how much a Level1 manager is allowed to sign off:

- In Company1 a Level1 manager can sign off on purchases up to $10K.
- In Company2 a Level1 manager can sign off on purchases up to $15K.
- In Company3 a Level1 manager can sign off on purchases up to $20K.

Three <alternative> elements are used when declaring purchase-request:

```
<element name="purchase-request">
      <alternative test="@member eq 'Company1'"
                   type="po:company1-purchase-request" />
      <alternative test="@member eq 'Company2'"
                   type="po:company2-purchase-request" />
      <alternative test="@member eq 'Company3'"
                   type="po:company3-purchase-request" />
 </element>
```

Read as:

If member="Company1" then use this as the type for purchase-request:
company1-purchase-request.

If member="Company2" then use this as the type for purchase-request:
company2-purchase-request.

If member="Company3" then use this as the type for purchase-request:
company3-purchase-request.

Here is the first type:

```
<complexType name="company1-purchase-request">
      <complexContent>
        <extension base="po:purchase-request-type">
          <assert   test="(po:signature-authority eq 'Level1') and
                          (xs:integer(po:cost) lt 10000)" />
        </extension>
      </complexContent>
 </complexType>
```

This type extends purchase-request-type and adds to it an <assert> element which expresses the business rule: Level1 managers can sign off on purchases up to $10K.

Here is purchase-request-type:

```
   <complexType name="purchase-request-type">
      <sequence>
        <element name="item" type="Name"/>
```

```
        <element name="cost" type="nonNegativeInteger"/>
        <element name="signature-authority">
          <simpleType>
            <restriction base="string">
              <enumeration value="Level1"/>
              <enumeration value="Level2"/>
              <enumeration value="Level3"/>
            </restriction>
          </simpleType>
        </element>
      </sequence>
      <attribute name="member" use="required">
        <simpleType>
          <restriction base="string">
            <enumeration value="Company1" />
            <enumeration value="Company2" />
            <enumeration value="Company3" />
          </restriction>
        </simpleType>
      </attribute>
    </complexType>
```

Notice the member attribute.

Here is the second type:

```
<complexType name="company2-purchase-request">
      <complexContent>
        <extension base="po:purchase-request-type">
          <assert   test="(po:signature-authority eq 'Level1') and
                          (xs:integer(po:cost) lt 15000)" />
        </extension>
      </complexContent>
 </complexType>
```

Notice that this type extends purchase-request-type and adds to it an <assert> element which expresses the business rule: Level1 managers can sign off on purchases up to $15K.

Here is the third type:

```
<complexType name="company3-purchase-request">
      <complexContent>
        <extension base="po:purchase-request-type">
          <assert   test="(po:signature-authority eq 'Level1') and
```

```
                    (xs:integer(po:cost) lt 20000)" />
        </extension>
      </complexContent>
 </complexType>
```

Notice that this type extends purchase-request-type and adds to it an <assert> element which expresses the business rule: Level1 managers can sign off on purchases up to $20K.

Advantage

Only one schema needs to be created, updated, distributed, and stored.

The business rules of each member are centrally documented within the community XML Schema, so everyone knows everyone's business rules.

Disadvantage

It doesn't scale: the community XML Schema needs to be changed each time a new member is added and each time a member changes its business rule.

It's not good for rapidly changing business rules.

Approach #2 Revisited

Recall the second approach: each member of the community creates their own supplementary XML Schema to express their business rules and a validation pipeline is employed. Rather than using XML Schema for the supplementary schema, Schematron may be used. An advantage of using Schematron is that it provides better, customized output messages. An advantage of using XML Schema is that it is supported by web service description languages, WSDL.

Zip File of Examples

Here is a zip file of the examples described in this document.

Acknowledgement

The following people contributed to the formation of this document:

- George Cristian Bina
- Roger Costello
- Stephen Green
- Ken Holman
- Michael Kay

Last Updated: November 21, 2010