Your schema and the industry-standard schema

Bob **DuCharme**

Abstract

Using an existing industry standard schema or DTD instead of developing your own provides obvious advantages. Only the luckiest find a perfect fit between a standard schema and their requirements; most need to customize the schema to take full advantage of it. They'll find that some schemas are more amenable than others to customization. Issues to consider include the modularity of a schema and the use of easily overridden parameter entities or their schema equivalents.

Table of Contents

1. Introduction

It's a truism in the XML world that when you need a DTD or schema and there's a standard one in your industry, you should use that instead of making up a new one yourself. You'll save the modeling and development work, you'll have an easier time exchanging data with business partners, and you may even find built-in support for the standard in software tools developed for use in that industry. Just because someone claims that their work is an industry standard, though, doesn't make it so, and it's always worth investigating how much traction the standard has. Are your business partners, or anyone else, really using it? Is there an active community developing with it? Are there experienced specialists who can help you with it if necessary?

How well does a specific industry standard schema meet your needs? If you're looking at a particular schema at all, there must be some fit with your business's needs. Even if everything in that schema reflects information in your content, your content probably has more to account for. Perhaps the value you add to your content, which may be what separates it from your competitors' content, would best be stored as extra elements or attributes in the content, and the standard schema doesn't have those elements and attributes. Perhaps your documents include simple workflow and versioning data that track when you received the documents and which of your processes have been run on them. Perhaps the extra data that you want to store is not something that you designed in-house but is instead another standard such as the periodical publishing industry's PRISM (Publishing Requirements for Industry Standard Metadata), which is designed to be plugged into other schemas and DTDs.

This brings up the importance of a schema's customizability relative to others. A highly customizable schema that meets 80% of your needs may be better for you than another one that meets 90% of your needs but is set in stone. While there' s nothing to prevent you from taking a schema and just editing it until it meets your needs, this is a bad strategy; if you start with a copy of version 2.1 of an industry standard and then make a lot of changes, what do you do when the standard moves to 3.0? Attempting to make all the same changes to release 3.0 will run into difficulties. When a schema designed for sharing among multiple organizations is well-designed, it provides hooks for storing customizations in a separate file so that those same customizations can be used with the upgraded version of the standard with minimal trouble. The techniques for providing these hooks varies from schema language to schema language.

2. Customizability and Schema Languages

When you define the structure of a document type's elements, you can do it by creating building blocks that can be redefined and then assembling those blocks to define your elements. You don't have to do it this way, but you can, and whether a schema does this or not is one key to its adaptability. Let's review how the three major schema languages do this, along with any extra features they offer for extensibility: DTDs, RELAX NG schemas, and W3C Schemas.

2.1. DTDs

The SGML-derived syntax of DTDs[XML], XML's original schema language, lets you define fairly arbitrary pieces of DTDs in units called parameter entities and then plug those into DTDs where you need them. ("Entities," as opposed to "parameter entities," are named pieces of actual XML documents, not DTDs.) Let's say we want to make the following declaration for a purchase order more customizable:

```
<!ELEMENT po (name,date,item,quantity)>
```

We could store its content model in a parameter entity and reference that in the element declaration.

```
<!ENTITY % po.content "name,date,item,quantity">
<!ELEMENT po (%po.content;)>
```

A customized version of the purchase order DTD could include the file holding these declarations and then redefine the `po.content` entity that is used above to define the `po` element's content model:

```
<!ENTITY % po.content "name,date,item,quantity,po-id">
<!ELEMENT po-id (#PCDATA)>  <!-- new element to add to content model -->
```

Letting someone completely redefine the content model might give them too much flexibility, though. A safer practice is to define the original content model with a reference to an empty parameter entity.

```
<!ENTITY % po.content.cust "">
<!ELEMENT po (name,date,item,quantity %po.content.cust;)>
```

Then, in the customized version of the DTD, someone can redefine the empty parameter entity to add to the content model with no chance of any damage to the existing content model:

```
<!ENTITY % po.content.cust ",po-id">
<!ELEMENT po-id (#PCDATA)>
```

A similar technique lets people add attributes to the customized version of an ATTLIST declaration. The following declares and references an empty `item.att.cust` parameter entity.

```
<!ENTITY % item.att.cust "">

<!ATTLIST item id    ID      #REQUIRED
               color NMTOKEN "white"
               %item.att.cust;
>
```

A customized version of the DTD with the `item` attribute list declaration can redeclare the `item.att.cust` entity to add more attributes to the list, and this will be substituted for the `%item.att.cust;` entity reference instead of the original declaration:

```
<!ENTITY % item.att.cust "supplierCode NMTOKEN 's0000'
                          flavor       CDATA   #IMPLIED"
>
```

One weak point of a DTD's approach to DTD component modularity is the dependence on string substitution. If the redeclaration of `po.content.cust` above had a value of "po-id" instead of ",po-id" it would cause an error, because the comma is necessary as a delimiter between the "quantity" ending of the original `po` content model and the new `po-id` element being added to it. An advantage of the RELAX NG and W3C Schema languages is that their more structural approach is architecturally more solid and therefore more scalable.

Another DTD extensibility option is the use of the keyword ANY as a content model, which allows an element to have any well-formed content without causing a parsing error. While this is lax enough to be of little use to DTD developers, we'll see that refinements of this ability in the RELAX NG and W3C Schema languages can provide important contributions to the adaptability of a schema.

2.2. RELAX NG

The RELAX NG schema language is all about specifying patterns for allowable XML structures. A schema itself is considered a pattern. Like the parameter entities of a DTD, RELAX NG named patterns let you define structures to be referenced elsewhere in a schema:

```
<define name="po.content">
  <interleave>
    <ref name="name"/>
    <ref name="date"/>
    <ref name="item"/>
    <ref name="quantity"/>
  </interleave>
</define>

<element name="po">
  <ref name="po.content"/>
</element>
```

By redefining a named pattern in a separate file, you can customize a schema without editing the schema file itself. When a RELAX NG parser finds the same named pattern defined in more than one place, it expects to find a `combine` attribute that specifies the relationship between the new version and the old one. A value of "choice" means that an element using that named pattern must conform to either one or the other named patterns, and a value of "interleave" means that the new pattern is to be combined with the original one. For example, the following adds `po-id` to the `po.content` content model as declared above:

```
<define name="po.content" combine="interleave">
  <ref name="po-id"/>
</define>
```

Note that, unlike with DTDs, no special slot had to be inserted in the original definition (as with the `%po.content.cust;` entity reference in the DTD example) to enable the addition of the revision from elsewhere. The original schema must still define a named pattern to allow the customization, because the customization must be able to identify the component being customized.

A RELAX NG Schema lets you specify `anyName` as part of a content model. This is already an improvement over the DTD `ANY` keyword, which can only be used as a complete content model itself, not combined with other names to specify parts of a content model. The RELAX NG `anyName` name class offers further refinement of what it specifies by letting you add an `except` child with a pattern identifying a name or names to disallow.

As an example of its use, let's look at how an `entry` is specified in Atom 1.0[ATOM], the recent IETF (Internet Engineering Task Force) version of the RSS format used to notify applications of updates to news stories, weblog entries, and other content. The following shows the definition of this element using RELAX NG's compact syntax for brevity:

```
element atom:entry {
   atomCommonAttributes,
   (atomAuthor*
    & atomCategory*
    & atomContent?
    & atomContributor*
    & atomId
    & atomLink*
    & atomPublished?
    & atomRights?
    & atomSource?
    & atomSummary?
    & atomTitle
    & atomUpdated
    & extensionElement*)
}
```

At first I was surprised to see that the content model was not defined by a pattern that could be redefined, because the ability to add new, application-specific children to the `atom:entry` element can greatly speed the adoption of this new standard. (Keep in mind that in the `entry` definition above, the names `atomCategory`, `atomContent`, and the others shown above are pattern names, not element names.) Then I looked at the definition of the `extensionElement` pattern that ends the `entry` content model:

```
<define name="simpleExtensionElement">
  <element>
    <anyName>
      <except>
        <nsName ns="http://www.w3.org/2005/Atom"/>
      </except>
    </anyName>
    <text/>
  </element>
</define>
```

This lets you put an element of any name there, as long as it's not in the same namespace as Atom itself, because you wouldn't want an entry or a feed inside of an entry. James Clark used the same technique to define an XSLT 1.0 RELAX NG schema so that literal result elements could be added in the appropriate places in an XSLT stylesheet but that inappropriate XSLT elements could not go in those same places. [XSLT.RNG]

2.3. W3C Schema

A key different between the approach of W3C Schemas[XSD] and the other schema languages is that while W3C Schemas let you define an element's structure when you declare the element (a concept known as defining an element to be of an "anonymous type") it is much more common to define a named type and the declare an element to be of that type. Types don't have to be the simple data types known in typical programming languages such as integer, string, or Boolean; you can define an `address` type, complete with attributes and subelements, and then declare both `billingAddress` and `shippingAddress` elements to be of the `address` type. W3C Schema extensibility mechanisms are usually built around options for defining these types, but they still ultimately define what can go in which elements.

You can derive a new type from any complex named type (that is, from any non-anonymous type with attributes or subelements) that doesn't have a `final` attribute to specifically prohibit it. Deriving, which can either extend or restrict the model of the base type, is loosely based on the concept of class inheritance in object-oriented development. When extending a type, you name the type to extend, and then specify the new parts of the content model to add to the end of the original--and it must be added at the end--and restriction restates the content model minus the parts to omit.

The discussion of DTD parameter entity substitution earlier showed how parameter entities can be used to store subsets of an element type's content model or attribute list. Instead of following the string substitution model, W3C Schemas have specific constructs for each of these: the `group` and `attributeGroup` elements. When one of these is used to define a component, that component can be redefined, giving you a more robust version of the parameter entity technique demonstrated earlier in DTDs.

W3C Schema substitution groups let you define elements that can be substituted for specific other elements. For example, a schema customized for U.S. use could define a `zipCode` element to be one that can be used instead of the `postalCode` element in the base schema. The use of the `complexType` element's `final` and `block` attributes can prevent this, but as with the prevention of derived types, it's good news for extensibility that extensible elements, types, and groups are the default that must be explicitly overridden, unlike with DTDs and RELAX NG, in which a lack of extensibility is the default and the schema author must take explicit steps to make a schema extensible. (In RELAX NG, the near-universal use of the optional named pattern feature means that most schemas are extensible in practice.)

Like RELAX NG, W3C Schemas' equivalent of the DTD `ANY` keyword gives you more granular control over what can constitute "any" at the specified point in your content model. While W3C Schemas lack the full power of a pattern matching language for specifying the set of allowed names, the choices are still an improvement over the DTD `ANY` keyword: you can specify that the element inserted is from any namespace, from a specific namespace, or from any namespace other than the target namespace being defined, as with the Atom and XSLT examples mentioned above.

3. Modularity

All three schema languages provide what is often called an "include" mechanism that lets one schema file identify another whose contents should be treated as part of the one doing the including. (W3C Schemas offer three variations on this: `include`, `import`, and `redefine`.) In the examples above, one schema file would include another and then redefine components of the included schema file.

Another reason to use inclusion in any schema language is to implement modularity. A large, complex standard schema may group declarations into several files, with the master file being little more than a short file with instructions to include the other modules. This kind of modularized approach was the key difference between XHTML 1 and XHTML 1.1, which provides an excellent example of why such modularity is a good thing.

A schema that takes this modular approach offers two benefits for people adapting it for customized use. One the one hand, a customized version may only aim to support a subset of the full standard. For example, if you're developing content for delivery on mobile phone web browsers, you may want only the core HTML markup without using the elements that allow fancier user interface features. This is what WAP (Wireless Application Protocol) 2.0 did with the XHTML Mobile Profile. On the other hand, you may have new markup customized for your application that you want to add to the core, so you can write new modules and include them in the master file along with the existing standard modules. (You would still need the techniques described earlier to integrate new elements and attributes with the content models of the existing elements.)

More likely, you'd want to do both: pick a subset of the available modules and then include new modules, specialized for your application, into your schema file. This is what the PRISM standard for magazine and journal metadata did when they created their PAM (Prism Aggregator Message) content DTD for use with magazine articles.[PRISM] So, while some people saw no new features in the transition from XHTML 1.0 to XHTML 1.1[XHTML1.1], the PRISM group saw an obvious improvement to take advantage of.

4. Some Schemas

We've seen some of the techniques used by XHTML and Atom to allow extensibility. Let's look at some extensibility options in other popular schemas.

4.1. SOAP 1.2

The following shows an excerpt from a W3C Schema excerpt for a SOAP envelope.[SOAP1.2] The basic document envelope, `Envelope`, consists of an optional `Header` followed by a `Body` element.

```
<!-- Envelope, header and body -->
<xs:element name="Envelope" type="tns:Envelope" />
<xs:complexType name="Envelope" >
  <xs:sequence>
    <xs:element ref="tns:Header" minOccurs="0" />
    <xs:element ref="tns:Body" minOccurs="1" />
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax" />
</xs:complexType>

<xs:element name="Header" type="tns:Header" />
<xs:complexType name="Header" >
  <xs:sequence>
    <xs:any namespace="##other" processContents="lax"
           minOccurs="0" maxOccurs="unbounded"  />
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax" />
</xs:complexType>

<xs:element name="Body" type="tns:Body" />
<xs:complexType name="Body" >
  <xs:sequence>
    <xs:any namespace="##any" processContents="lax"
           minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax" />
</xs:complexType>
```

The `Header` element's definition demonstrates the W3C Schema technique for allowing any element (or attribute) from outside of the target namespace, and the `Body` element shows how to declare that any well-formed element at all can be in an element's content. For a document designed to be an envelope with a payload, it makes perfect sense to do this in the body part so that systems can ship whatever they want in the payload, but it is a bit surprising to see so much flexibility in the header, where envelope-oriented documents typically have specific metadata elements. However, the rest of the schema (not shown here) defines elements and types that can be used when creating the children of the `Envelope` element's two potential children, so the schema does provide more guidance to SOAP envelope content than what you see here.

4.2. DocBook

DocBook [DocBook] began as an SGML DTD for technical documentation, and the XML version has been around for nearly as long as XML itself. It has always been modular, letting a customized version pick a subset of DocBook's modules. The most immediate example of this practice is the schema used for this paper and the others being written for the XML 2005 conference. [ConfSchema]

DocBook also makes it easy to customize nearly any of its elements. The following shows how the `title` element's content model and attribute list are declared in the DTD version of DocBook (RELAX NG and W3C Schema versions are also available; the DocBook Technical Committee plans to make RELAX NG the normative version in the future, generating the others from that):

```
<!ENTITY % local.title.char.mix "">

<!ENTITY % title.char.mix
      "#PCDATA
      |%xref.char.class;   |%gen.char.class;
      |%link.char.class;   |%tech.char.class;
      |%base.char.class;   |%docinfo.char.class;
      |%other.char.class;  |%inlineobj.char.class;
      |%ndxterm.class;
      %local.title.char.mix;">

<!ELEMENT title %ho; (%title.char.mix;)*>

<!ENTITY % local.title.attrib "">

<!ATTLIST title
      %pagenum.attrib;
      %common.attrib;
      %title.role.attrib;
      %local.title.attrib;
>
```

To add a "myCustElement" element as something that could be included inline in a `title` element, you would redefine the `local.title.char.mix` entity as "| myCustElement". To add a new "myCustAttribute" attribute to the `title` element, you could redefine the `local.title.attrib` entity as "myCustAttribute NMTOKEN #IMPLIED".

The inclusion of an empty "local" entity reference in the content model and attribute list for the `title` element is the kind of arrangement that you'll find throughout DocBook, and this was in the SGML version of DocBook before XML was invented. It will be interesting to see how the DocBook Technical Committee takes advantage of RELAX NG's extensibility options when they make that language the normative language for the schema.

4.3. SVG

Several of the larger, more complex standards offer a leaner, "lite" version that is simpler and easier to implement. In fact, the original reason for XML's existence was to be a subset of SGML that would be easier to implement. DocBook and XHTML both have "lite" versions, and the modularity described earlier makes these versions easier to specify.

The W3C's SVG (Scalable Vector Graphics) Working Group takes an interesting approach with SVG's schema, which uses the RELAX NG language[XML]: instead of defining a fully-featured version and then leaving some parts out to make the simpler subset, they've made the "tiny" version the core and made the full version an extension of that.

This is done for separately for each module. For example, the "tiny" version of SVG's font module has the following in the tiny-font.rng schema file:

```
  <define name='SVG.glyph.class'>
    <notAllowed/>
  </define>
```

Why define something and then disallow it? For the same reason that you'd create an empty parameter entity in a DTD: as an easily-redefined placeholder to make customization easier. The full version of SVG 1.2 has a file called font.svg, which includes the font declarations from the tiny version with this line:

```
  <include href="../Tiny-1.2/tiny-font.rng"/>
```

It then redefines the SVG.glyph.class pattern like this:

```
  <define name="SVG.glyph.class" combine="choice">
    <choice>
      <ref name="SVG.Animation.class"/>
      <ref name="SVG.Structure.class"/>
      <ref name="SVG.Conditional.class"/>
      <ref name="SVG.Image.class"/>
      <ref name="SVG.MultiImage.class"/>
    <!-- (15 additional ref elements removed) -->
    </choice>
  </define>
```

A similar approach lets the other modules build around the tiny versions.

4.4. NITF

NITF (News Industry Text Format) is very popular for newspaper content. The following shows a few declarations from version 3.2 of their DTD[NITF]:

```
<!ELEMENT nitf (head?, body)>
<!ELEMENT head (title?, meta*, tobject?, iim?, docdata?,
                pubdata*, revision-history*)>

<!ATTLIST head
   %global-attributes;
  >

<!ELEMENT meta EMPTY>

<!ATTLIST meta
   %global-attributes;
   http-equiv  NMTOKEN  #IMPLIED
   name     NMTOKEN  #IMPLIED
   content  CDATA       #REQUIRED
```

```
   >

<!ENTITY % global-attributes '
   id    ID                    #IMPLIED
   '>
```

There's very little room for customization here. (Elsewhere in the DTD, there is a `general-text` parameter entity that could be redefined to allow new inline elements on most content elements, as with the DocBook `title` element earlier, but little if anything at higher structural levels.) A typical reason for customization would be to allow the addition of metadata specific to your shop's workflow, and the logical place for it would be in the NITF `head` element (compare the structure of the SOAP `Envelope` element that we saw earlier) but NITF offers no way to customize the `head` element's content model to add something new. You could redefine the `global-attributes` entity to hold a new attribute, but that would add the same attribute to the 65 other elements whose attribute list declarations reference this entity. The optional `meta` element offers no place in its content model to add children, so the use of its `name` and `content` attributes is about the only option for adding named information inside an NITF document.

There's another option for bundling customized information with schemas like this: defining a wrapper element that includes the schema's document element and other elements that you want to add. For example, let's say I want to add the elements `myEl1` and `myEl2` to an NITF document. I could declare the following DTD:

```
<!ELEMENT myWrapper (myHeader,nitf)>
<!ELEMENT myHeader (myEl1,myEl2)>
<!ELEMENT myEl1 (#PCDATA)>
<!ELEMENT myEl2 (#PCDATA)>

<!ENTITY % nitfdtd SYSTEM "nitf-3-2.dtd">
%nitfdtd;
```

It declares a container element called `myWrapper` that includes a header I've defined called `myHeader` and the `nitf` element that is the document element of a standard NITF document. The `myHeader` element contains my new fields `myEl1` and `myEl2`. When NITF gets upgraded, I can plug the new version into the entity declaration at the bottom of this DTD. Then, I'll be able to create NITF documents that conform to the new version and include my customized new fields.

NITF's `global-attributes` parameter entity does define an optional ID attribute for many of the elements. This gives you a further hook for storing information specific to a document or to its individual elements. If an element has a unique ID value of "i643" and you can reference the document itself with an unambiguous identifier, then you can take information that you might otherwise have added as an attribute or subelement of that document and store it outside of that document in another document or in a database. The unique ID value gives you a key to identify what the information describes. This ability to store information about any resource or subresource that has an identifier is behind much of the philosophy underlying RDF (Resource Description Format).

5. Picking and Extending a Schema

How important is extensibility when picking a schema? That depends on several factors. Maybe a given standard really has everything you need. Maybe you're in an industry in which one XML format is so dominant that not supporting it would be foolish. For example, if you publish a newspaper, I can't say that you shouldn't use NITF because it isn't extensible. It's one of the most commonly used standards in a large industry that I know of.

If your analysis of your content reveals that you need to track more information than a particular standard allows for, the first step in investigating the adaptability of the standard is to look through the schema itself. Digging through a schema and evaluating its extensibility can be difficult, though; an extensible schema can be more difficult to read

than a less flexible one because of the role that indirection plays in extensibility. For example, extensive declaration and referencing of named schema components leads to components that reference components that reference components, making it more difficult to learn which elements contain which other elements. Another problem with schema extensibility features is that their position among the more arcane features of the relevant specification means that they can be inconsistently implemented among the different tools that claim support for that schema language. This is mostly a problem with W3C Schemas.

The second step in evaluating a schema's extensibility--and don't knock yourself out in the first step before proceeding to this step--is to find a mailing list where the schema is discussed and ask people about their experiences adapting the schema. (If you can't find a mailing list where such things are discussed, then consider this lack of activity as less reason to use that schema.) Try a Google search on the name of the standard and the word "extensibility"; you'd be surprised how easily you can find discussions of the fine points of extending a particular standard.

Microformats are a new trend that holds some promise for work on extensibility. XHTML is the most popular schema being extended to create microformats, but the principle can apply elsewhere: take a specific module of a popular schema and use its more open-ended features, such the `class` attribute so common in XHTML or the `role` attribute found in most DocBook elements, to identify the semantics of your information. Doing this instead of making up new elements for your information makes it easier to use your documents with existing tools. While this is generally only applied to small, self contained handfuls of information such as outline and calendar information (hence the "micro" part of the name "microformats"), the extensive work going on in this area holds promise for anyone interested in schema extensibility and adaptability.

Bibliography

[ATOM] *The Atom Syndication Format*, 15 August 2005. Available at http://www.ietf.org/internet-drafts/draft-ietf-atompub-format-11.txt.

[ConfSchema] *XML 2005 Conference Schema*. Available at http://2005.xmlconference.org/node/91.

[DocBook] *DocBook documentation [http://www.docbook.org]*.

[NITF] *News Industry Text Format document Type Definition - Version 3.2*, 10 October 2003. Available at http://www.nitf.org/IPTC/NITF/3.2/dtd/nitf-3-2.dtd.

[PRISM] *PRISM and PAM specifications*. Available at http://www.prismstandard.org/specifications/.

[SOAP1.2] *soap-envelop schema*. Available at http://www.w3.org/2002/12/soap-envelope/.

[SVG] *RNG for SVG 1.2*, 27 October 2004. Available at http://www.w3.org/Graphics/SVG/1.2/rng/.

[XHTML1.1] *XHTML 1.1 - Module-based XHTML*, 31 May 2001. Available at http://www.w3.org/TR/2001/REC-xhtml11-20010531/.

[XML] *Extensible Markup Language (XML) 1.0 (Third Edition)*, 04 February 2004. Available at http://www.w3.org/TR/2004/REC-xml-20040204/.

[XSD] *XML Schema Part 1: Structures Second Edition*, 28 October 2004. Available at http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[XSLT.RNG] *XSLT Relax NG Schema*. Available at http://www.thaiopensource.com/relaxng/xslt.rng.

Biography

Bob **DuCharme**
consulting software engineer
LexisNexis [http://www.lexisnexis.com/]
Charlottesville
Virginia
United States of America
http://www.snee.com/bob

Bob DuCharme is the author of Manning Publications' "XSLT Quickly," Prentice Hall's "XML: The Annotated Specification" and "SGML CD," and McGraw Hill's "Operating Systems Handbook." He writes the monthly "Transforming XML" column for XML.com and has contributed to Dr. Dobb's Journal, perl.com, XML Magazine, XML Journal, IBM developerWorks, XML Developer, O'Reilly Books' "XML Hacks," and Prentice Hall's "XML Handbook." A consulting software engineer at LexisNexis, Bob received his BA in religion from Columbia University and his masters in computer science from New York University. Bob's O'Reilly Network weblog [http://www.oreillynet.com/pub/au/1191] is dedicated to linking-related topics.