Balisage: The Markup Conference 2008

Proceedings

Freedom to Constrain

where does attribute constraint come from, mommy?

Syd Bauman
Senior Programmer/Analyst
Brown University Women Writers Project
<Syd_Bauman@Brown.edu>

Balisage: The Markup Conference 2008
August 12 - 15, 2008

How to cite this paper

Bauman, Syd. “Freedom to Constrain: where does attribute constraint come from, mommy?” Presented at Balisage: The Markup Conference 2008, Montréal, Canada, August 12 - 15, 2008. In *Proceedings of Balisage: The Markup Conference 2008*. Balisage Series on Markup Technologies, vol. 1 (2008). doi:10.4242/BalisageVol1.Bauman01.

Abstract

Where should attribute constraints live? In an external schema? In the document’s own metadata? In a separate file? Several possibilities are examined, raising lots of questions and offering a few answers.

Table of Contents

It is clear that constraining document structure is a very important part of document production. We test whether or not our XML documents are properly constrained through the process of validation. “The … purpose of validation is to subject a document … to a test, to determine whether it conforms to a given set of external criteria. … Our need to test is simply explained and understood (so much so that it rarely needs to be explicated): if there exists a point in a process where it is less expensive to discover and correct problems than it is to save the work of testing and fix at later points, it is profitable to introduce a test.”[1]

Michael Sperberg-McQueen may have summed this importance up best when he advised “constrain your data early and often”, which he often did.[2] (It helped that he lived in Chicago at the time.)

So it is obvious that constraints need to be expressed in a formal language of some sort. Many such general-purpose formal languages are available, including closed schema languages like DTDs and RELAX NG, and open schema languages like Schematron and CLiX. Furthermore at least one literate encoding language exists in which such constraints along with documentation about them can be expressed. This language is called ODD (for “one document does it all”) — constraints expressed in other languages (DTDs, RELAX NG, or XML Schema; in theory others as well) can be derived from a set of constraints expressed in ODD.[3][4][5][6] Furthermore there are systems of constraint based on special-purpose languages, rather than general-purpose languages. The feature system declaration created by the Text Encoding Initiative (TEI) and now being incorporated into ISO 24610-2 is an example — a set of XML elements (the feature system declaration) that can be used to constrain the expression of another set of XML elements (the feature structure itself).[7]

So the choice of *how* to express a particular constraint is not always obvious. But a related question is perhaps just as important: *where* should these constraints be expressed? What are the consequences of expressing them in different places?

This paper will attempt to shed light on these general questions by taking an in-depth look at the possible locations for the expression of one particular kind of constraint, and the consequences of those different locations. The constraint discussed will be that of limiting the value an attribute may take to one of an enumerated list of possible values. For simplicity the presumed setting for this constraint will be in a TEI document, but the principles should be equally applicable to any other encoding language that separates the document from its metadata, including DocBook or XHTML. The locations considered will be

- the “normal” way, in the formal closed schema (RELAX NG will be used as the example)
- in a formal open schema (ISO Schematron will be used as the example)
- in the metadata element (i.e. `<teiHeader>`)
- in a separate metadata file
- in the metaschema file (i.e. the ODD file)
- no formal constraint

Each of the latter methods will be compared to and contrasted with the first.

Use Case

There are lots of reasons to wish to constrain markup constructs, in particular attribute values. One case worth considering is the markup project which has tens or hundreds of occurrences of a particular attribute in each of tens or hundreds of files, where the list of possible values for the attribute is different for each file.

Imagine, e.g., an epigraphy project transcribing thousands of inscriptions on various objects. Imagine further that the inscriptions are divided among 27 separate files, organized by some criteria other than the kind of object that bears the inscription (e.g. date the object was discovered, current museum in which it is held, whatever). That which the text bearing object is made of is recorded in a TEI manuscript description on the `material=` attribute of the `<supportDesc>` element. Possible values might include `"bronze"`, `"marble"`, `"limestone"`, `"plaster"`, `"wood"`, etc.

Such a typical humanities computing project is likely to have:

- a subject matter expert
- an XML expert
- encoders — getting the extant text into XML-encoded digital form may

be accomplished in a variety of ways:
  - typed from source
  - post-OCR editing
  - via an external vendor
- proofreaders, managers, web designers, research assistants, etc.

Background

Open vs Closed vs Extensible Schemas

Formal schema languages can generally be categorized into one of two types: open or closed. A closed schema language like RELAX NG specifies a complete document grammar. Only those documents that meet all of the constraints of the grammar are considered valid; all others are rejected as invalid.

An open schema language, like Schematron, specifies particular rules. Documents that violate the specified rules are rejected as invalid; all others are accepted as valid.

One can think of closed schema languages as a white list spam filter, and closed schema languages as a black list spam filter. Using a white list (closed schema language) only e-mail from the addresses specified get through, all others are rejected as spam. Using a black list (open schema language) any e-mail that is on the list of problematic addresses is rejected as spam, all others are allowed through.

Of course the situation is not as simple as that. One can specify some open constructs in many closed schema languages, and one can write sufficiently tight rules in most open languages that they behave like a closed language.

For example, validation against the following complete RELAX NG grammar will permit any XML document as long as it has a `<foo>` element with a `bar=` attribute as the first child of the root element.

```
start = element * { any_attribute*, foo, any_element* }
any_attribute = attribute * { text }
any_element = element * { any* }
any = ( any_attribute | any_element | text )
any_sans_bar = ( attribute * - ( bar ) { text } | any_element | text )
foo = element foo { attribute bar { text }, any_sans_bar* }
```

Conversely, validation against the following Schematron rule will permit only those documents that have one `<platypus>` element with a `bill=` attribute that has the value `"duck"` as the only child of the root `<enigma>` element.

```
<pattern>
  <rule context="/*">
    <assert test="name(.)='enigma'">Root element must be "enigma"</assert>
    <report test="@*">Root "enigma" element can not have attributes</report>
    <assert test="count(child::*)=1">"enigma" can only have one child
    ("platypus")</assert>
    <assert test="count(child::platypus)=1">"enigma" can only have one
    "platypus" child</assert>
    <report test="child::text()[not(normalize-space(.)='')]">"enigma" is
    not allowed to have text, just "platypus"</report>
  </rule>
  <rule context="/enigma/platypus">
    <assert test="@*[name(.)='bill']">"platypus" must have a bill=
    attribute</assert>
    <report test="@*[not(name(.)='bill')]">"platypus" must not have any
    attributes other than bill=</report>
    <report test="child::*">"platypus" must be empty (i.e., can not have
    child elements)</report>
    <assert test="string-length( normalize-space(.) ) = 0">"platypus"
    must be empty (i.e., can not contain text)</assert>
  </rule>
  <rule context="/enigma/platypus/@bill">
    <assert test="normalize-space(.)='duck'">The value of bill= of
    "platypus" must be 'duck'</assert>
  </rule>
</pattern>
```

These reverse uses of open and closed schema languages may be thought of as analogous to black-list or white-list spam filters that permit wildcards.

Neither of the above examples are particularly good ways of performing the desired validation, but they serve as proofs-of-concept that when we refer to a schema language as “open” or “closed”, we may be referring to its default, and not its only, behavior.

There is one further twist worth mentioning. Some modular XML document systems, including DocBook and TEI, permit a user of the system to generate (closed) schemas that contain not only the element and attribute declarations native to the system, but also additional declarations for constructs added by the user.

Literate Encoding

Literate programming is a style of programming intended to make computer documentation better by, among other things, placing the documentation and source code in the same computer file. The TEI has applied this concept to the schemas used to validate documents to help ascertain whether or not they conform to the TEI Guidelines. The source code from which the schemas are generated and the prose documentation that make up the bulk of the TEI Guidelines are stored in one computer document.

In order to facilitate this, and in order to help make it easy to extract formal schemas in any of a variety of popular languages, the formal constraints are (for the most part) expressed in the TEI language, rather than any particular schema language.

Thus the TEI Guidelines proper (some 32 chapters of prose documentation), formal schemas expressed in RELAX NG, the XML DTD language, or the W3C Schema language, and reference documentation for those schemas, are all extracted from the same single document. We say that this “one document does” it all, and thus it is referred to as an ODD document.

In the Closed Schema (RELAX NG file)

how

Many are probably quite familiar with the mechanism for constraining an enumerated attribute in a formal closed schema language. E.g., in RELAX NG (compact syntax), the possible values of the `type=` attribute (in this case, of the `<name>` element) could be constrained with a construct like

```
attribute type { "person" | "place" | "ship" | "sword" }
```

A variety of readily available off-the-shelf software will test whether or not a document is valid with respect to a RELAX NG schema.

advantages

This method is extremely common for a reason: it makes a lot of sense. In many, many cases XML document structure is already governed by an external closed schema. These external schemas, at least when written in one of the three major languages (DTD, RELAX NG, W3C XML Schema) are generally easy to read and process. They describe the constraint in a standard formal language that has wide software support, including open source validators.

These languages typically provide the capability to specify a variety of structural and content constraints on XML documents. In particular, they provide the capability needed here: to constrain the set of possible values of the `type=` attribute to one of a list of possibilities. [8]

disadvantages

In many cases, the person or persons who write and maintain the external schema is not the same as the person or persons who create the XML instances (or the programs that write the XML instances) that conform to it.

In these cases, those who create the instances often do not have either the necessary knowledge (e.g., knowing the schema language) or capability (e.g., having read-write access to the schema) to make changes to it.

Furthermore in many cases (whether the instance creator is the same as the schema maintainer or not), a single external schema governs the validity of dozens or even tens of thousands of XML instances. But the desired constraints on a particular attribute may be different in different instances. Typically in these cases the schema limits the attribute to one of a set that is the union of all possible values in all governed documents. Here adding the additional constraint of “only these values in *this* document” requires making a separate schema that is like the original in all respects except for the declaration of the `type=` attribute of `<name>`.

In the Open Schema (ISO Schematron)

Many are probably quite familiar with the mechanism for constraining an enumerated attribute in a formal open schema language. E.g., in Schematron (DSDL part 4), the possible values of the `type=` attribute of the TEI `<name>` element could be constrained with a construct like

```
<pattern>
  <rule context="tei:name/@type">
    <assert test="normalize-space(.)='person'
               or normalize-space(.)='place'
               or normalize-space(.)='ship'
               or normalize-space(.)='sword'">
      Names can only be of people, places, ships, or swords
    </assert>
  </rule>
</pattern>
```

While the use of open vs closed schemas have a lot of advantages and disadvantages to the schema designer, with respect to this particular question, the advantages and disadvantages are primarily the same: while the constraint can be expressed in a formal, widely supported language, and can be tested with readily available tools, it is still in a separate file that may support many documents, that may not be accessible, and that uses a language that may be foreign to those who would like to change it.

There is one additional disadvantage of Schematron in particular with respect to RELAX NG: it is harder to annotate the Schematron schema than the RELAX NG schema. RELAX NG deliberately permits elements from other namespaces to be mixed in with the RELAX NG specifications, and defines where annotations relating to particular structures should go. Furthermore, because the four tokens against which we are trying to validate are expressed as four separate elements (in the XML syntax), there is a place to annotate each separately (the `<a:documentation>` element follows the `<rng:value>` element to which it refers). Schematron also has a built-in documentation feature (a `<p>` element), but because all four tokens are tucked into a single XPath expression, it is a bit harder to discuss them individually. This is partially confounded because `<p>` is not permitted in `<rule>`, `<assert>`, or `<report>`, making it difficult to put the documentation close to the code. This is partially alleviated because elements from foreign namespaces are permitted in those spaces, and inside `<p>`. Thus something like the following construct could be used to provide documentation of such a constraint.

```
<pattern>
  <p class="annotation">The various values for <tei:att>type</tei:att> of
    <tei:gi>name</tei:gi> came about as follows: <tei:list type="gloss">
      <tei:label>
        <tei:val>person</tei:val>
      </tei:label>
      <tei:item>Added 2007-04-17 when we removed <tei:gi>persName</tei:gi></tei:item>
      <tei:label>
        <tei:val>place</tei:val>
      </tei:label>
      <tei:item>Added 2007-04-17 when we removed <tei:gi>placeName</tei:gi></tei:item>
      <tei:label>
        <tei:val>ship</tei:val>
      </tei:label>
      <tei:item>Added 2007-04-17 in order to accommodate the various ship names</tei:item>
      <tei:label>
        <tei:val>ship</tei:val>
      </tei:label>
      <tei:item>Added 2007-10-02 when we found a reference to "Excalibur" that the
        professor needed to annotate</tei:item>
    </tei:list>
  </p>
  <rule context="tei:name/@type">
    <tei:note><tei:att>type</tei:att> of <tei:gi>rs</tei:gi> is matched elsewhere.</tei:note>
    <assert test=".='person' or .='place' or .='ship' or .='sword'"> Names may only be
      of people, places, ships, or swords </assert>
  </rule>
</pattern>
```

In the Metaschema (ODD file)

how

The same constraint might be expressed, at a slightly higher level of abstraction and combined with some documentation, using the ODD literate encoding language:

```
<attDef ident="type">
  <valList type="closed">
    <valItem ident="person">
      <desc>The name refers to a person</desc>
    </valItem>
    <valItem ident="place">
```

```
      <desc>The name refers to a political or man-made region, for example
        a city, country, hamlet, town, or neighborhood. For geographical
        places such as rivers or valleys, use <gi>geogName</gi></desc>
    </valItem>
    <valItem ident="ship">
      <desc>The name refers to a ship, whether sea-worthy, interplanetary,
        or interstellar</desc>
    </valItem>
    <valItem ident="sword">
      <desc>The name refers to a sword</desc>
    </valItem>
  </valList>
</attDef>
```

There exists software that will “tangle” ODD specifications like the above into formal declarations in one of several schema languages, including RELAX NG. Then any of the same variety of readily available off-the-shelf software could be used to test validity.

Furthermore, there exists software that will “weave” the same specification above into easily readable hyperlinked documentation.

advantages

The advantages of literate programming are well understood, and include more easily readable and understandable source code, and that documentation (because it is right next to the source code) is more likely to match the program and be updated when the source code changes.[9] These advantages apply here as well. In addition, at least for those familiar with TEI, there is the advantage that the language used to describe the constraints is a TEI language, so schema designers are likely to be familiar with at least the documentation paradigm for the specialized schema-description elements, if not the elements themselves; in addition, they are likely familiar with the generic TEI elements (like `<desc>`, above) that are used in addition to the specialized elements.

disadvantages

The disadvantages of the external schema (whether open or closed) are present here as well. Furthermore, an extra processing step is required to generate (i.e. “tangle”) a schema that itself can be used to validate instances using off-the-shelf software. In addition, at least for those who are not intimately familiar with TEI, there is the disadvantage that the language used to describe the constraints is primarily a TEI language, so schema designers may not be familiar with the specialized schema-description elements.

In the Metadata (`<teiHeader>`)

how — pointing

It should be quite feasible to develop a mechanism for expressing the list of possible values of an attribute in the same document in a rather abstract way. For example:

```
<codeGrp elementTypes="name rs" attributes="type">
  <codeDef xml:id="person">The name or string refers to a
    person</codeDef>
  <codeDef xml:id="place">The name or string refers to a
    political or man-made region, for example a city, country,
    hamlet, town, or neighborhood. For geographical places such as
    rivers or valleys, use <gi>geogName</gi></codeDef>
  <codeDef xml:id="ship">The name or string refers to a ship,
    whether sea-worthy, interplanetary, or
    interstellar</codeDef>
  <codeDef xml:id="sword">The name or string refers to a
    sword, <foreign xml:lang="fr">main-gauche</foreign>, switchblade,
    or other edged weapon</codeDef>
</codeGrp>
```

Given this encoding in the `<teiHeader>`, the `<name>` element could have `type=` values of `"#person"`, `"#place"`, etc. Software could be developed to validate that the value of `type=` of `<name>` is a URI that points to an element whose parent `<codeGrp>` has `"name"` in its `elementTypes=` list and `"type"` in its `attributes=` list. (I believe that Schematron code could probably be used for this test, but have not yet demonstrated this.) Note that the check does not specify the element type of the child of `<codeGrp>`. This gives the flexibility to have special-purpose `<codeDef>`-like elements that might provide structured information about the value. E.g., one can well imagine the TEI's `<handNote>` element being used in this way.

advantages

This mechanism has significant potential advantages, particularly in cases where one schema is used for many files which may have different attribute constraint requirements. For most users it is much easier to change something in the same file they are working on, rather then needing to make changes to an external schema, particularly an external schema that may be in a language the user does not know or in a file to which the user does not have write access, and particularly changes that might inadvertently invalidate other existing instances. Thus the encoder, as opposed to the schema-designer, can add, remove, or change a value quite easily.

Another advantage is that the information about to what values the attribute is constrained, and what those values mean, is an integral part of the document. This means that this information will survive in the situation where a document instance is sent along without its schema or documentation.

Furthermore the list of values in different files at a given project could be slightly different.

Moreover, the particular system shown here has the advantage that it uses a mechanism most users are already familiar with: `xml:id=` and relative URIs (i.e., bare name fragment identifiers). It is worth noting, though, that there is no requirement that the URIs be bare name fragment identifiers, which permits this system to quickly and easily be changed to that which is discussed in section “In the Metadata (separate file)”.

disadvantages

This system has obvious inefficiencies when multiple, perhaps thousands, of document instances share the same constraints — the same information is repeated in each file.

Another significant disadvantage of this method is that we are using a non-standard language for constraint and documentation. The question, then, is whether or not this system is demonstrably significantly better than what can be obtained using standard languages.[10]

Lastly the fact that this system uses the URI pointing mechanism produces a disadvantages, one of which is severely problematic:

- of minor annoyance is that the user needs to encode a hash-mark (“`#`”, U+0023) at the beginning of each value;
- the fact that values are restricted to XML Names could be a problem in some situations;
- but far more problematic, because `xml:id=` needs to be unique within the document, any given possible attribute value can only occur on one attribute (although that attribute could be on multiple elements) — furthermore, no other element elsewhere in the document can use the same string as one of these attribute values as its identifier.

how — co-reference

Those last disadvantages that are the result of using `xml:id=` and URIs could be circumvented by matching the attribute values, rather than using a true pointer (e.g. ID/IDREF or URI). In the `<teiHeader>` the enumeration of the possible attribute values would look almost the same, but would use a different attribute for storing the actual value.

```
<codeGrp elementTypes="name rs" attributes="type">
  <codeDef attrVal="person">The name or string refers to a
    person</codeDef>
  <codeDef attrVal="place">The name or string refers to a
    political or man-made region, for example a city, country,
    hamlet, town, or neighborhood. For geographical places such as
    rivers or valleys, use <gi>geogName</gi></codeDef>
  <codeDef attrVal="ship">The name or string refers to a ship,
    whether sea-worthy, interplanetary, or
    interstellar</codeDef>
  <codeDef attrVal="sword">The name or string refers to a
    sword, <foreign xml:lang="fr">main-gauche</foreign>, switchblade,
    or other edged weapon</codeDef>
</codeGrp>
```

Software could be developed to validate that the value of `type=` of `<name>` is a string that matches the `attrVal=` attribute of an element whose parent `<codeGrp>` has `"name"` in its `elementTypes=` list and `"type"` in its `attribute=` list. (I believe that Schematron code could probably be used for this test, but have not yet demonstrated this. Certainly XSLT 1.0 can transform this into simple Schematron; this I have demonstrated, see Appendix A.) Note that the check does not specify the element type of the child of `<codeGrp>`. This gives the flexibility to have special-purpose `<codeDef>`-like elements that might provide structured information about the value. E.g., one can well imagine the TEI's `<handNote>` element being used in this way.

This system avoids the disadvantages of using `xml:id=`, and yet has several advantages over external schema files. E.g., encoders can quickly and easily add values to closed lists, in a manner that does not run the the risk that they might break the rest of the schema. I find the case of the encoder who wishes to quickly and easily express stricter constraints on her attribute values in a given file than those that come with the generic external schema very compelling.

In the Metadata (separate file)

In the method described in section "how — pointing" the values of the `type=` attribute of `<name>` are URIs. Because of this, it would be feasible to store the `<codeGrp>` element with `xml:id=` attributes in a project-wide "attribute_definitions.xml" file. While this has the advantage of flexibility and reusability, it presents the sizable disadvantage that the attribute values would now depend on details of system features external to the document. E.g., the ability to validate `<name type="../attribute_definitions.xml#sword">` breaks if the current file is moved to a sub-directory.

Furthermore, if the `<codeGrp>` is stored in a separate file, the maintenance issues are almost the same as those for a separate closed schema (e.g., a RELAX NG grammar), open schema (e.g., a Schematron schema), or metaschema (e.g., a TEI ODD): those who have reason to change the constraints expressed may not have the write-permissions necessary to do so, and if they do may be at risk for invalidating files other than the one being worked on.

So in some cases (in particular, the scenario sketched out in section “Use Case”) it makes lots of sense to leave the formal constraints for some aspects of a document in the metadata section of that document itself, e.g. in the `<teiHeader>`. But having convinced ourselves there is a need to be able to express constraints in a different *place* than is usual, why require a separate formal construct to express the constraint? Why not include RELAX NG, Schematron, or ODD markup constructs in the `<teiHeader>` directly?[11] This is worthy of consideration, but is outside the scope of the current paper.

Appendix A. <codeGrp> to Schematron

The following XSLT 1.0 stylesheet is a proof-of-concept demonstration for transforming the `<codeGrp>` elements discussed above into Schematron that could be used to validate that an XML instance used only the mentioned possible values of the attribute specified.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Tranform my mythical <codeGrp> elements into a Schematron schema -->
<!-- Copyleft 2008 Syd Bauman -->
<!-- Last updated: 2008-08-31 -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sch="http://purl.oclc.org/dsdl/schematron">

  <xsl:template match="/">
    <!-- only mess with <codeGrp> elements; if there are none, we do nothing -->
    <!-- Note that we presume each <codeGrp> has both elementTypes= and  -->
    <!-- attriubtes= specified and that their values are lists of one or more -->
    <!-- XML Names. No error-checking for this here, schema validation should -->
    <!-- have already flagged any that don't have both required attributes or -->
    <!-- have inappropriate values. -->
    <xsl:if test="//codeGrp">
      <!-- if there is one (or more) we write out a Schematron schema -->
      <sch:schema>
        <sch:ns uri="http://www.tei-c.org/ns/1.0" prefix="tei"/>
        <!-- and process each <codeGrp> into it -->
        <xsl:apply-templates select="//codeGrp"/>
      </sch:schema>
    </xsl:if>
  </xsl:template>

  <!-- Each <codeGrp> becomes a Schematron <pattern> -->
  <xsl:template match="codeGrp">
    <sch:pattern>
      <!-- append a blank to the GI list for easier parsing later -->
      <xsl:variable name="elementTypes" select="concat(normalize-space(@elementTypes),' ')"/>
      <!-- append a blank to the attribute name list for easier parsing later -->
      <xsl:variable name="attributes" select="concat(normalize-space(@attributes),' ')"/>
      <!-- Each GI/attribute pair becomes a Schematron <rule> -->
      <!-- A little more detail: each paired combination of -->
      <!-- 1. a GI listed on my elementTypes= attribute, and -->
      <!-- 2. an attribute name listed on my attributes= attribte -->
      <!-- becomes a <rule>. We do this by processing each GI in  -->
      <!-- a recursive template, which in turn calls another recursive -->
      <!-- template for the list of attributes. -->
      <xsl:call-template name="elementTypes">
        <xsl:with-param name="gis" select="$elementTypes"/>
        <xsl:with-param name="attrs" select="$attributes"/>
      </xsl:call-template>
    </sch:pattern>
  </xsl:template>

  <!-- Each GI listed on the elementTypes= attribute gets processed separately -->
  <xsl:template name="elementTypes">
    <xsl:param name="gis"/>
    <xsl:param name="attrs"/>
    <!-- Taking advantage of that ending blank, parse off the 1st GI -->
    <xsl:variable name="this_gi" select="substring-before($gis,' ')"/>
    <xsl:variable name="rest" select="substring-after($gis,' ')"/>
    <!-- call attributes template to do the work for this particular GI -->
    <xsl:call-template name="attributes">
      <xsl:with-param name="gi" select="$this_gi"/>
      <xsl:with-param name="attrs" select="$attrs"/>
    </xsl:call-template>
    <!-- and do the same thing (via recursion) for the rest of the GIs, if any -->
    <xsl:if test="string-length($rest) > 1">
      <xsl:call-template name="elementTypes">
        <xsl:with-param name="gis" select="$rest"/>
        <xsl:with-param name="attrs" select="$attrs"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>

  <!-- Each attibute name on the attributes= attribute gets processed in combination -->
  <!-- with the current GI -->
  <xsl:template name="attributes">
    <xsl:param name="gi"/>
    <xsl:param name="attrs"/>
    <!-- Taking advantage of that ending blank, parse off the 1st attribute -->
    <xsl:variable name="this_attr" select="substring-before($attrs,' ')"/>
    <xsl:variable name="rest" select="substring-after($attrs,' ')"/>
    <!-- make a rule out of it -->
    <xsl:element name="sch:rule">
      <xsl:attribute name="context">
        <!-- There must be a better way to do this ... -->
        <xsl:text>tei:</xsl:text>
        <xsl:value-of select="$gi"/>
        <xsl:text>/@</xsl:text>
        <xsl:value-of select="$this_attr"/>
      </xsl:attribute>
      <xsl:variable name="numVals" select="count(child::*/@attrVal)"/>
      <!-- if I have no children with attrVal= specified, then don't -->
      <!-- generate any assertions (luckily an emtpy <rule> is valid -->
      <!-- in Schematron). -->
      <xsl:if test="$numVals > 0">
        <xsl:element name="sch:assert">
          <!-- Probably would be better to generate this test (i.e., the expression -->
          <!-- that is the value of this output test= attribute) only once per attrVal=, -->
          <!-- rather once for each attrVal= for each GI/attr combination. -->
          <xsl:attribute name="test">
            <xsl:for-each select="child::*/@attrVal">
              <xsl:text>.='</xsl:text>
              <xsl:value-of select="."/>
              <xsl:text>'</xsl:text>
              <xsl:if test="$numVals > 1  and  position() != last()">
                <xsl:text> or </xsl:text>
              </xsl:if>
            </xsl:for-each>
          </xsl:attribute>
        </xsl:element>
      </xsl:if>
    </xsl:element>
    <!-- and do the same thing (via recursion) for the rest of the attributes, if any -->
    <xsl:if test="string-length($rest) > 1">
      <xsl:call-template name="attributes">
        <xsl:with-param name="gi" select="$gi"/>
        <xsl:with-param name="attrs" select="$rest"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

---

[1] Piez, Wendell, “Beyond the ‘descriptive vs. procedural’ distinction”, presented at Extreme Markup Languages 2001, Montréal, Canada. http://www.idealliance.org/papers/extreme/proceedings/html/2001/Piez01/EML2001Piez01.html.

[2] Sperberg-McQueen, C. Michael. Oral conversation, and multiple oral presentations throughout the 1990s. See, e.g., http://www.w3.org/People/cmsmcq/2001/darmstadt.html.

[3] Burnard, Lou and Syd Bauman, eds. “4.3.2 Floating Texts.” *TEI P5: Guidelines for Electronic Text Encoding and Interchange*. Version 1.1.0. 2008-07-04. TEI Consortium. http://www.tei-c.org/release/doc/tei-p5-doc/html/DS.html#DSFLT 2008-08-30

[4] Burnard, Lou and Syd Bauman, eds. “23.4 Implementation of an ODD System.” *TEI P5: Guidelines for Electronic Text Encoding and Interchange*. Version 1.1.0. 2008-07-04. TEI Consortium. http://www.tei-c.org/release/doc/tei-p5-doc/en/html/USE.html#IM 2008-08-30

[5] Sperberg-McQueen, C. Michael and Lou Burnard. “The Design of the TEI Encoding Scheme.” *Computers and the Humanities* 1995. 29 (1) p. 17–39.

[6] Burnard, Lou, Sebastian Rahtz. “RelaxNG with Son of ODD”, presented at Extreme Markup Languages 2004, Montréal, Canada. http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Burnard01/EML2004Burnard01.pdf.

[7] Burnard, Lou and Syd Bauman, eds. “18 Feature Structures” *TEI P5: Guidelines for Electronic Text Encoding and Interchange*. Version 1.1.0. 2008-07-04. TEI Consortium. http://www.tei-c.org/release/doc/tei-p5-doc/en/html/FS.html 2008-08-30

[8] DTDs impose greater restrictions on what the members of that list can be

than the others: each possible value must be an XML Name.

[9] Knuth, Donald. *Literate Programming*, ISBN 0-9370-7380-6.

[10] What some call *Syd's rule*, and I have begun to call my *wheel re-invention prevention convention*: "unless your method is significantly and demonstrably superior to the standard, you should be using the standard.".

[11] Indeed, James Cummings and I have suggested this on more than one occasion. See, e.g., http://lists.village.virginia.edu/pipermail/tei-council/2005/005627.html.

Author's keywords for this paper: XML; attribute; TEI; ODD; constraint

Syd Bauman
<Syd_Bauman@Brown.edu>
Senior Programmer/Analyst
Brown University Women Writers Project

Syd Bauman is the technical person at the Brown University Women Writers Project, where he has worked since 1990, designing and maintaining a significantly extended TEI-conformant schema for encoding early printed books. He has served as the North American Editor of the Text Encoding Initiative Guidelines, has an AB from Brown University in political science, and has worked as an Emergency Medical Technician since 1983.

Balisage Series on Markup Technologies