One Document Does it all: using the TEI to declare and document your XML system

TEI@Oxford

May 2010

The scope of ‘intelligent’ markup

What is the proper domain of the TEI?

- basic structural and functional components of text
- diplomatic transcription of historical sources, images, annotation
- links, correspondence, alignment
- data-like objects such as dates, times, places, persons, events (‘named entity recognition’)
- meta-textual annotations (correction, deletion, etc)
- linguistic analysis
- contextual metadata of all kinds
- ... and documentation of XML schemas!

Today, we focus on just the last.

Why might you need ODD?

- You need to define an XML schema to describe your resource
- You need to provide documentation about
  - the semantics of your XML schema
  - constraints, usage notes, examples
- You need to keep the two in step
- You want to share the results
  - with others
  - with yourself, long term
- you don't want to reinvent the wheel

ODD is not a new idea

- Knuth's "literate programming"
- java beans, doxygen...

The basic idea (1)

A special XML vocabulary for defining....

- schemas
- XML element types independent of a schema
- public or private groups of such elements
- patterns (MLE macros)
- classes (and subclasses) of element

And also for defining references which can pull into a schema

- named components from the above list
- objects from other namespaces

All embedded within conventional document markup elements

The basic idea (2)

An ODD processor:

- assembles all the components referenced or directly provided
- resolves multiple declarations
- may do some validity checking
- emits a schema in one or more formal languages
- emits a "plain" XML document with selected documentary components

http://www.tei-c.org/Roma

A simple example

We have <stuff>, which contains a mixture of <bit>s and <bob>s. We have never heard of the TEI and we don't want to use it. Likewise namespaces.

```
<schemaSpec  ns=""  start="stuff"  ident="simpleS">
  <elementSpec  ident="stuff">
    <desc>Root element for a very simple schema</desc>
    <content>
      <rng:oneOrMore>
        <rng:choice>
          <rng:ref  name="bit"/>
          <rng:ref  name="bob"/>
        </rng:choice>
      </rng:oneOrMore>
    </content>
  </elementSpec>
<!-- ... continues on next slide -->
</schemaSpec>
```

A simple example, contd.

```
<schemaSpec  ns=""  start="stuff"  ident="simpleS">
<!-- ... contd -->
  <elementSpec  ident="bob">
    <desc>Empty pointing element in a very simple schema</desc>
    <content>
      <rng:empty/>
    </content>
    <attList>
      <attDef  ident="href">
        <desc>supplies the URI of the thing pointed at</desc>
        <datatype>
          <rng:data  type="anyURI"/>
        </datatype>
      </attDef>
    </attList>
  </elementSpec>
  <elementSpec  ident="bit">
    <desc>textual element in a very simple schema (may have bobs
         in it)</desc>
    <content>
      <rng:zeroOrMore>
        <rng:choice>
          <rng:text/>
          <rng:ref  name="bob"/>
        </rng:choice>
      </rng:zeroOrMore>
    </content>
  </elementSpec>
</schemaSpec>
```

So what?

- We can now build a schema in RELAXNG, W3C schema, or DTD language by a simple XSLT transformation
- We can also extract documentary fragments (e.g. the descriptions of elements and attributes)

TEI provides a special element for the latter purpose:

```
<specList>
  <specDesc  key="bit"/>
  <specDesc  key="bob"  atts="href"/>
</specList>
```

which would generate something like

```
<list  type="gloss">
  <label>
    <gi>bit</gi>
  </label>
  <item>textual element in a very simple schema (may have bobs
     in it)
  </item>
  <label>
    <gi>bob</gi>
  </label>
  <item>Empty pointing element in a very simple schema</item>
</list>
```

What else might you want to say about your elements?

- alternative <desc>s or <gloss>es in different languages maybe?
- some reference usage examples
- schematron constraints
- value lists
- class memberships

Alternative descriptions

```
<elementSpec module="core" ident="p">
<gloss>paragraph</gloss>
<gloss version="2007-12-20" xml:lang="kr">■ ■</gloss>
<gloss version="2007-05-02" xml:lang="zh-tw">段落</gloss>
<desc>marks paragraphs in prose.</desc>
<desc version="2007-12-20" xml:lang="kr">■ ■ ■ ■  ■ ■ ■  ■ ■ ■ ■.</desc>
<desc version="2007-05-02" xml:lang="zh-tw">標記散文的段落。</desc>
<desc version="2008-04-05" xml:lang="ja">散文の段落を示す.</desc>
<desc version="2009-01-06" xml:lang="fr">marque les paragraphes dans un
texte en prose.</desc>
<desc version="2007-05-04" xml:lang="es">marca párrafos en prosa.</desc>
<desc version="2007-01-21" xml:lang="it">indica i paragrafi in
prosa</desc>
<!-- ... -->
</elementSpec>
```

Usage examples

The <exemplum> element combines an XML example with some discussion of it...

```
<exemplum xml:lang="en">
  <egXML><egXML><langUsage>
      <language ident="en">English</language>
</langUsage>
</egXML>
  </egXML>
  <p>In the source of the TEI Guidelines, this element declares itself and
its content as
    belonging to the namespace <ident type="ns">http://www.tei-c.org/ns/
Examples</ident>. This
    enables the content of the element to be validated independently
against the TEI scheme.</p>
</exemplum>
```

Defining the content of an element

- We use RELAXNG directly to define content for elements and attributes
- (rather than re-invent an equally expressive language)
- Generated patterns are uniquified by means of an automatic prefix, which can be switched on or off
- Content can be constrained by means of a <valList> element ...
- ... or by means of a <datatype> element (which also uses RDELAXNG)
- Generic constraints can be expressed by means of <constraint> elements (which uses e.g. ISO Schematron)

The Durand Conundrum

Why do you bother to embed all this in a special XML language, rather than just use some standard schema language?

- TEI is the framework that holds everything together
- A single language for documentation and schema generation
- Independence of any one schema language
- A well tested and standard method
- Keeping you honest

About this wheel of yours...

The TEI does actually define elements very like yours. Why not just use them?

```
<schemaSpec
   source="/usr/share/xml/tei/odd/Source/Guidelines/en/guidelines-en.xml"
   start="div"
   ident="simpleS-2">
  <elementRef  key="div"/>
  <elementRef  key="p"/>
  <elementRef  key="ptr"/>
</schemaSpec>
```

The @*source* attribute is a URI of any kind, from which specifications are available. It could be a file name, a URL, a DOI...

Why use the TEI definitions?

- Principle of least effort
- Your resources now have a standard semantics attached to them
- (And you can explain how you've interpreted them in your own documentation)

And (if you like) you can mix and match:

```
<schemaSpec
   source="/usr/share/xml/tei/odd/Source/Guidelines/en/guidelines-en.xml"
   start="stuff"
   ident="simpleS-3">
  <elementSpec ns="" ident="stuff">
   <desc>Root element for a very simple schema</desc>
   <content>
<!-- as before -->
   </content>
  </elementSpec>
  <elementRef key="p"/>
  <elementRef key="ptr"/>
</schemaSpec>
```

In the real world, elements come in packs

A *module* is a named collection of elements. The TEI provides 22 such. To include one of them in a schema, use the <moduleRef> element:

```
<schemaSpec start="TEI" ident="testSchema-4">
  <moduleRef key="core"/>
  <moduleRef key="header"/>
  <moduleRef key="textstructure"/>
</schemaSpec>
```

Every TEI element belongs to a single module and has a unique name.

Recap

- The TEI encoding scheme consists of a number of *modules*
- Each module contains a number of *element specifications*
- Each element specification contains:
  - a canonical name (<gi>) for the element, and optionally other names in other languages
  - a canonical description (also possibly translated) of its function
  - a declaration of the *classes* to which it belongs
  - a definition for each of its *attributes*
  - a definition of its *content model*
  - usage examples and notes
- a TEI *schema* specification (<schemaSpec>) can contain
  - references to modules or elements
  - (re)declarations for elements, classes, or macros
- a TEI document containing a schema specification is called an *ODD* (One Document Does it all)

The TEI modules

analysis	Simple analytic mechanisms
certainty	Certainty and uncertainty
core	Elements common to all TEI documents
corpus	Header extensions for corpus texts
declarefs	Feature system declarations
dictionaries	Printed dictionaries
drama	Performance texts
figures	Tables, formulae, and figures
gaiji	Character and glyph documentation
header	The TEI Header
iso-fs	Feature structures
linking	Linking, segmentation and alignment
msdescription	Manuscript Description
namesdates	Names and dates
nets	Graphs, networks and trees
spoken	Transcribed Speech
tagdocs	Documentation of TEI modules
tei	Declarations for datatypes, classes, and macros available to all TEI modules
textcrit	Text criticism
textstructure	Default text structure
transcr	Transcription of primary sources
verse	Verse structures

Picking and choosing (1)

You can specify elements to be excluded from those provided by a module:

```
<schemaSpec start="TEI" ident="testSchema-4a">
  <moduleRef key="core" except="mentioned quote said"/>
  <moduleRef key="header"/>
  <moduleRef key="textstructure"/>
</schemaSpec>
```

This is equivalent to the following:

```
<schemaSpec start="TEI" ident="testSchema-4b">
  <moduleRef key="core"/>
  <moduleRef key="header"/>
  <moduleRef key="textstructure"/>
  <elementSpec ident="mentioned" mode="delete"/>
  <elementSpec ident="quote" mode="delete"/>
  <elementSpec ident="said" mode="delete"/>
</schemaSpec>
```

The *@mode* parameter instructs an ODD processor how to resolve multiple declarations.

Picking and choosing (2)

You can specify just the elements you want to include:

```
<schemaSpec start="TEI" ident="testSchema-4b">
  <moduleRef key="core"/>
  <moduleRef key="header"/>
  <moduleRef key="textstructure" include="body div"/>
</schemaSpec>
```

This is equivalent to the following:

```
<schemaSpec start="TEI" ident="testSchema-4b">
  <moduleRef key="core"/>
  <moduleRef key="header"/>
  <elementRef key="div"/>
  <elementRef key="body"/>
</schemaSpec>
```

(Sadly not yet fully implemented in web Roma)

Unifying multiple declarations

As noted above, the *@mode* attribute controls what an ODD processor should do when it find multiple instances of some component.

Supposing that we have found one existing declaration, what should be done with a subsequent one for the same object?

mode value	existing dec-lara-tion	effect
add	no	add new declaration to schema; process its children in add mode
add	yes	raise error
replace	no	raise error
replace	yes	retain existing declaration; process new children in replace mode; ignore existing children
change	no	raise error
change	yes	process identifiable children according to their modes; process unidentifiable children in replace mode; retain existing children where no replacement or change is provided
delete	no	raise error
delete	yes	ignore existing declaration and its children

Specifying elements and modules

The '*-spec' elements are all members of a class att.identifiable which provides an attribute *@ident* that is used (rather than *@xml:id*) as a unique identifier for them.

To reference such a declaration, we use the *@key* attribute:

```
<elementRef key="bar"/>
<!-- implies the presence elsewhere of ... -->
<elementSpec ident="bar">
<!-- .... -->
</elementSpec>
```

Similarly:

```
<moduleRef key="foo"/>
<!-- implies the presence elsewhere of ... -->
<moduleSpec ident="foo"/>
```

But note that elements indicate the module they belong to by means of their *@module* attribute:

```
<elementSpec ident="bar" module="foo">.... </elementSpec>
```

Specification of attributes

For reasons lost in the mists of time, the element <attSpec> is actually spelled <attDef>, but otherwise, it's just the same. Within an <elementSpec> or a <classSpec>, you can supply an <attList> containing of bunch of <attDef> elements, each with an @*ident*:

```
<attList>
  <attDef ident="bax">....</attDef>
</attList>
```

Specifying value lists and datatypes

In general, the legal values for an attribute are defined by means of a <datatype> element, see later.

A common case, however, is to supply an *enumeration* (a list, open or closed, of legal values. This is done using the <valList> element, which groups a bunch of identifiable <valItem> elements: like this

```
<attDef  ident="status">
  <desc>indicates the state of the system using a predefined set
     of colour codes</desc>
  <defaultVal>green</defaultVal>
  <valList  type="closed">
    <valItem  ident="red">
      <desc>all systems shut down</desc>
    </valItem>
    <valItem  ident="orange">
      <desc>systems shut-down imminent</desc>
    </valItem>
    <valItem  ident="green">
      <desc>system status normal</desc>
    </valItem>
    <valItem  ident="white">
      <desc>system status unrecorded</desc>
    </valItem>
  </valList>
</attDef>
```

Datatypes

Typically used to constrain attribute values:

```
<attDef ident="status">
  <datatype>
    <rng:ref name="data.enumerated"/>
  </datatype>
<!-- ... implies that a vallist is supplied -->
</attDef>
<attDef ident="lastUpdated">
  <datatype>
    <rng:ref name="data.temporalExpr.w3c"/>
  </datatype>
</attDef>
```

TEI defined datatypes are actually patterns, defined by a <macroSpec>

Specifying a pattern

The <macroSpec> element is an identifiable element used to associate a name with any string. It has two typical uses in the TEI scheme:

- defining common content models
- defining TEI-specific datatypes

```
<macroSpec  ident="data.foo">
  <desc>a new datatype i just invented</desc>
  <content>
<!-- RELAXNG pattern defining the datatype -->
  </content>
</macroSpec>
<macroSpec  ident="macro.foo">
  <desc>a content model i plan to reuse often</desc>
  <content>
<!-- RELAXNG pattern defining the content model -->
  </content>
</macroSpec>
```

(DTD generation needs some extra fluff: the *@type* attribute distinguishes the two cases).

A macro can be referenced explicitly, using the <rng:ref> syntax, or embedded in the usual way, using <macroRef>

Schematron constraints

- An element specification can also contain a <constraintSpec> element which contains rules about its content expressed as ISO Schematron *constraints*

```
<elementSpec ident="div" module="teistructure" mode="change"
   xmlns:s="http://purl.oclc.org/dsdl/schematron">
  <constraintSpec ident="cartoon" scheme="isoschematron">
    <constraint>
      <s:assert test="@type='cartoon' and .//tei:graphic">a cartoon must
include a graphic
      </s:assert>
    </constraint>
  </constraintSpec>
</elementSpec>
```

However...

- You can only add such rules by editing your ODD file: Roma doesn't know about them.
- Not all schema languages can implement these constraints.

Using the TEI Class System

When defining a new element, we need to consider

- its name and description
- what attributes it can carry
- what it can contain
- where it can appear in a document

The TEI class system helps us answer all these questions (except the first).

Attribute Classes

- Attribute classes are given (usually adjectival) names beginning with att.; e.g. *att.naming*, *att.typed*
- all members of `att.naming` inherit from it attributes @*key* and @*ref*; all members of `att.typed` inherit from it @*type* and @*subtype*
- If we want an element to carry the @*type* attribute, therefore, we add the element to the `att.typed` class, rather than define those attributes explicitly.

A very important attribute class: att.global

All elements are a member of att.global; this class provides, among others:

@xml:id	a unique identifier
@xml:lang	the language of the element content
@n	a number or name for an element
@rend	how the element in question was rendered or presented in the source text.

All TEI elements are members of this class by default.

Model Classes

- Model classes contain groups of elements which are allowed in the same place. e.g. if you are adding an element which is wanted wherever the <bibl> is allowed, add it to the model.biblLike class
- Model classes are usually named with a Like or Part suffix:
  - members of model.pLike are all things which 'behave like' paragraphs, and are permitted in the same places as paragraphs
  - members of model.pPart are all things which can appear *within* paragraphs. This class is subdivided into
    - model.pPart.edit elements for simple editorial intervention such as <corr>, <del> etc.
    - model.pPart.data'data-like' elements such as <name>, <num>, <date> etc.
    - model.pPart.msdesc extra elements for manuscript description such as <seal> or <origPlace>

Basic Model Class Structure

Simplifying wildly, one may say that the TEI recognises three kinds of element:

divisions	high level major divisions of texts
chunks	elements such as paragraphs appearing within texts or divisions, but not other chunks
phrase-level elements	elements such as highlighted phrases which can occur only within chunks

There are 'base model classes' corresponding with each of these, and also with the following groupings: three:

inter-level elements	elements such as lists which can appear either in or between chunks
components	elements which can appear directly within texts or text divisions

And yes, there is a class model.global for elements that can appear *anywhere* — at any hierarchic level.

Specifying a class

The <classSpec> element is used to declare a class. Its *@type* attribute indicates whether this is an attribute or a model class

For a model class, the class specification is purely documentary. For an attribute class it contains an <attList>, which specifies the attributes it provides.

Elements are classified (i.e. classes are referenced) by means of the <memberOf> child of the <classes> element inside an <elementSpec> (and Classes can also be *members-of* other classes

```
<classSpec  ident="model.foo"  type="model">
  <desc>The foo class consists solely of elements with silly names made
     up for didactic purposes</desc>
</classSpec>
```

```
<classSpec  ident="att.foo"  type="atts">
  <desc>The foo class provides the attribute <att>bar</att>
  </desc>
  <attList>
    <attDef  ident="bar">
<!-- ... -->
    </attDef>
  </attList>
</classSpec>
```

Conclusions

ODD provides a wide range of facilities... all of which have been found useful in editing and maintaining the TEI Guidelines.

Over the last couple of years we have also experimented with the usability of ODD outside the TEI, which has greatly influenced its evolution.

It's probably time for a major re-appraisal and evaluation as we progress towards the next generation of ODD