Advanced LDDTool Techniques

There are some advanced techniques available via the *<Ingest_LDD>* document and the *LDDTool* that might be useful in some complex cases.

Schematron Rules

Schematron rules can be used to check the sort of contingency relationships (i.e., "If *A*, then *B*") that XML Schema is not particularly good at. PDS also uses Schematron to define enumerated value lists rather than XML Schema largely for the diagnostic benefits - we can include the list of valid values in the Schematron error message.

Schematron rules are applied to an XML document via a two-step process involving style sheet translations (XSL). There are some subtleties to how and when Schematron rules are triggered, so if you're going to start writing Schematron rules yourself you should take some time to become familiar with the details of the standard. There are also multiple forms of the standard with a couple of significant differences in capabilities for the more advanced Schematron programmer. PDS requires ISO Schematron, which is available as a "Publicly Available Standard" (i.e., free if you promise to behave yourself) from this ISO website: http://standards.iso.org/ittf/PubliclyAvailableStandards. Search for "Schematron".

A limited capability to define additional Schematron rules as part of *LDDTool* processing is provided via the *<DD_Rule>* class. *DD_Rule* classes may be added after the *<DD_Class>* definitions in your *<Ingest_LDD>* document. Details for filling out the class are on the Filling Out the DD Rule Class page.

Choice List

The XML Schema Definitions (XSD) language is very strictly ordered, but it does provide one exception to this rule: the *choice* construct.

What Is It?

The *choice* structure is a mechanism that allows your label writers to use one or more of a list of attributes or subclasses at a particular point in a local class structure. In those cases where more than one list item may be included, the items may occur in any order, and may themselves be repeated. In other words, if I define a *choice* list containing **A**, **B**, and **C**, and allow label writers to select up to three of these to include, they can include **B** three times, or **C** then **A** and then **C** again, and either case would be considered equally valid (along with all the other permutations, of course).

So, on the one hand the *choice* construct defeats some of the validation we get from using XSD, but on the other hand sometimes you cannot practically code all the explicit contingencies you might have in certain scenarios.

On the whole, you should have some compelling reasons to resort to a *choice* construct in a local dictionary because of the impact on validation. "I keep getting errors from having my attributes out of order" is *not* a "compelling reason".

How Do I Do It?

The basic technique is to use a single *<DD_Association>* class to list all the possible attributes or subclasses that may be included at that point in the containing class being defined. Within that single *DD_Association*:

- Repeat the *<local_identifier>* tag for each attribute or class you want to include in the *choice* list.

- Somewhere in that list of *local_identifiers*, add this:
  `<local_identifier>XSChoice#</local_identifier>`

  Usage notes:

- A *choice* list defined in this way may contain only one kind of referenced object - either all attributes or all classes, but not both in the same *DD_Association*. (This is a current constraint on *Ingest_LDD* and *LDDTool*.)
- The <*minimum_occurrences*> and <*maximum_occurrences*> values you provide for this *DD_Association* refer to the total number of selections that may be made by a label writer from the list of things corresponding to the *local_identifier* values. If you specify *minimum_occurrences* of 0 (zero), then including nothing from the selection list is a valid option for a label.
- There is currently no way to specify in the *Ingest_LDD* file, or in the output XSD file, that multiple selections may be made, but no single option may be repeated. You can add a Schematron rule via the *DD_Rule* class to make that constraint.

"Exactly One"

One popular use for a *choice* list is to code an "exactly one of the following" option by a creating a multi-element *choice* list that requires exactly one of its contents be present. In order for this to work as desired, make sure that the <*minimum_occurrences*> and <*maximum_occurrences*> attributes for the *DD_Class* are both set to 1 (one).

Any Block

Any blocks are used within the shared PDS4 namespaces to pass validation control from one dictionary to another within a defined class.

What Is It?

The XSD *any* construct specifies that from this point until the closing tag of the current XML element, any elements from any namespace may be included and will not be subject to validation by the rules defined for the current namespace (they are required to be syntactically valid XML).

(You can also include elements from the current namespace, and they will be subject to validation as though they had occurred in their usual place in their label. This, however, is an excellent way to generate almost completely intractable error messages, and should be avoided.)

How Do I Do It?

As the last association in your <*DD_Class*> (yes, it absolutely ***must*** be the "last" association, since anything after an *any* statement in the schema is ignored with respect to the defining class definition), add one more <*DD_Association*> class with a <*local_identifier*> value of "XSAny#". The remaining attributes of *DD_Class* must be there, but will be ignored.

Note that:

- If a schema validator can identify what namespaces the elements in the *any* area come from, and has a schema to validate them against, it will do so and report errors accordingly. If you plan to include *any* blocks in your local dictionary classes, you should expect that your PDS standards reviewers will require that they only reference attributes and classes from namespaces for which PDS administers the defining schema.

- Using the *any* construct abdicates control of both content and validation for that part of your mission labels. If you do not have direct and significant influence over what might be thrown into this unregulated label area, you really should not be leaving this particular barn door open.

Cross-Referencing Namespace Elements

There are a couple of times when this is the preferred method for defining classes, but on the whole this should be used with extreme caution.

What Is It?

A namespace cross-reference invokes an attribute or class defined in another dictionary (we will call these *external elements*) for use in a class that you are defining in your own dictionary. This external element will appear in labels as it appears in its own namespace, preserving its own namespace references, just interpolated into your class.

The XML technique involved is to change the namespace of reference at some local level. There are a couple of different ways to do this, including using namespace abbreviation prefixes (like "pds:" or "disp:") on the tags.

How Do I Do It?

At the point in the *DD_Class* definition where you want to include the external element (attribute or subclass), add a *<DD_Attribute>* class as you would if the element was in this dictionary, with one exception: form the *<local_identifier>* as "*ns.name*", where *ns* is the PDS4 reserved namespace abbreviation for the external dictionary, and *name* is the name of the attribute or class you want to reference as it appears in the dictionary XSD file. Note that the namespace abbreviation separator in this case is a full stop ('.'), and *not* the usual colon (':').

When Should I Do It?

There are a small number of standard places where you *should* do this in order to take advantage of useful conventions established in the PDS-controlled namespaces. These are:

Referencing Other PDS4 Archive Products

If you want to create a link from a class you're defining to a PDS4 product somewhere else in the archive, use the *<Internal_Reference>* class defined in the *pds:* dictionary. (Note that you will also need to define values for the *<pds:reference_type>* attribute in your own Schematron file to go with this use.) *Internal_Reference* has a standard format recognized by PDS4 tools that enables direct access to the referenced product via a PDS registry, which is a handy capability to be able to offer end-users. To use it, add a *<DD_Attribute>' class with a* <local_identifier> *value of "pds.Internal_Reference" and a* <reference_type> *of "component_of".*

Cross-referencing Classes in the Same Label

If you need to explicitly link, for example, a set of parameters to a data object, use the *<pds:Local_Internal_Reference>* to reference the *<pds:local_identifier>* of the target class (in the example case, a data object class like *<Array_2D_Image>*. As with *pds:Internal_Reference*, you'll need to define values for the *<pds:local_reference_type>* attribute as well. As for *pds:Internal_Reference*, you include a *<DD_Attribute>* class with a *local_identifier* of "pds.Local_Internal_Reference".

Referencing an External Source

If you need to include a formal reference to an external source (perhaps a published article, or a source product from a non-PDS4 archive), include the *<pds:External_Reference>* class using the same method as in the previous two cases. In this case there is no

corresponding *pds:reference_type* to define. The advantage of using this class for external references is that it makes it possible for PDS tools to retrieve these references and report them to bibliographic services like the ADS to help in generating citation counts for the referenced works.

When You Should Not Do This

Sometimes, in very limited circumstances, there may be compelling reasons for a mission dictionary to directly reference elements defined in other namespaces. Here's why you should not do this in most typical mission dictionaries:

- You have no control over changes in the referenced namespace.
- You must assume the author of the other namespace has no knowledge of or interest in your cross-references.
- You must assume there is no commitment on the part of the author of the other namespace to maintain their elements in a way that will not invalidate your cross-reference, technically or informationally.
- You are requiring end-users to go outside of your own dictionary to understand classes defined in your dictionary.

Even references to discipline dictionaries may be problematic if that dictionary has not specifically designed and designated its classes for this sort of use. If you really think you need to pull in elements from dictionaries you do not control, talk to your PDS node consultant *first*.

And then do not do it.

If you find yourself managing a large mission in which teams or instruments are each creating their own dictionaries, in addition to the overarching mission dictionary, then you might have both good reasons and sufficient control to reasonably reference the mission dictionary from the instrument dictionaries, or conversely. In which case, have at it - and contact your PDS node consultant if you need advice.

More Reasons Why You Should Not Do This

This is an *extremely* experimental capability that has not been fleshed out as yet. In particular, note that:

- The namespace abbreviation in itself is not sufficient to identify a namespace, and even within the reserved PDS4 namespace abbreviations, it carries no information about versioning. All versions of a namespace have the same reserved abbreviation.
- If you write and validate your dictionary based on assumptions related to a particular version of the namespace you're referencing, you have no way at present to prevent erroneous linking to a different and potentially incompatible version of the same external namespace in labels. Worse, the labels may attempt to reference two different versions of the same namespace. At best, this generates validation warnings that may or may not be safely ignored. At worst, validation will be completely and wordlessly undermined as the software environment decides which version of the namespace should apply.
- LDDTool decides which version of the core dictionary or a discipline dictionary it will pull its definitions from - you cannot alter that. Which version of any particular namespace that it will reference depends on what is programmed into the specific version of the LDDTool you're using. This may not be the same version of the discipline dictionary referenced directly in labels for other reasons. In fact, if you have updated your version of LDDTool, it might not even be the same version that was referenced the last time you ran the tool.
- Some classes you include with the *[namespace_id].[element]* syntax will be included by type, others by reference - and which method is employed depends on what is in that other schema. Which method is used also affects which namespace each part of the definition

lives in, which in turn affects how your Schematron rules should be formulated. If you are not *extremely* detail-oriented in your approach to testing your output schemas, an error in namespace can go undetected indefinitely.