Mapping Data to CIDOC-CRM

Rotterdam
2013-06-06
contact: Gerald de Jong
email: gerald@delving.eu

The goal of this document is to contribute to the refinement of a universal file format for the mapping of data to the CIDOC Conceptual Reference Model using RDF. This XML-based file format was initiated at FORTH around 2006 and has seen the development of a partial implementation prototype for transforming LIDO documents into RDF.

The XML file is intended to specify all of the transformations that are required and an accompanying piece of Java code implements policy recommendations for generating URI values to represent the CRM entities and properties.

Much thought has gone into the design of the XML mapping file format but the associated implementation is not suitable for a production environment, and also not entirely generic in several ways. We believe that the XML mapping file format should undergo a few improvements, and that there is a great need for a proper production implementation for executing any mapping built in this format.

In the following pages we analyze the current XML format, as well as the associated prototype implementation. We will make recommendations that will enhance readability and accuracy of the mapping file, ensure that URI generation is generic, and standardize the retrieval of source data elements.

Contents:

Current Mapping File Format

There are some examples of mapping files available, and the original documentation describes a previous incarnation of the format, so the following analysis is based on those sources. The most elaborate and informative example was for transforming from LIDO 0.7.

Basic XML Structure

The current format describes the transformation as a series of **map** components each containing mappings for mapping the domain to an entity, and then the range as well together with the intervening predicate or property describing the path from domain to range.

```
<mapping>
   <map>
      <domain_map/>
      <link_map>
         <range_map/>
         <path_map/>
      </link_map>
      <link_map/>
      ...
   </map>
   <map/>
   ...
</mapping>
```

Extensions

Since there are several variations required of the straightforward one-to-one mapping, extensions have been made to make the contents of **range_map** and **path_map** so that extra additions can be made during the transformations.

```
<range_map>
   <src_range>
   <target_range>
   <add_link/>
   <add_entity/>
</range_map>
```

```
<path_map>
   <src_path/>
   <target_path>
      <int_link/>
      <int_entity/>
      <int_link/>
   </target_path>
</path_map>
```

Beyond the above extensions there has been an effort to document and standardize the practice of generating URI values, and deal with the conditional creation of output triples.

Conditionals

The original documentation suggests that a full array of boolean conditional statements which can be combined into expressions. The component values reflect either presence/absence of a value, or whether a value is equal to a given constant. The only example found so far shows the latter:

```
<target_path>
   <target_path_condition>
      <if>
         <path>../../lido:roleInEvent/lido:term/text()</path>
         <has_value>produced_by</has_value>
      </if>
   </target_path_condition>
<target_path>
```

The intention was that there be conditions associated with all six possibilities:

src_domain	src_range	src_path
target_domain	target_range	**target_path**

The naming convention is like the example above, appending **_condition** to the name of the context tag.

URI Generation

A Java class called **URIPolicies** has a number of methods which are to be fed parameters containing data fetched from the source and from the mapping file, and return the best possible URI value.

```
String uriConceptual(String className, String thing)
String appellationURI(String className, String subjUri, String appellation)
...
```

Calls to these methods are then specified in the XML file within a **uri_rules** section which can appear alongside any of the source-to-target mapping sections.

```
<uri_rules>
   <uri_function>
      <name>uriForActors</name>
      <arguments>@lido:source</arguments>
      <arguments>text()</arguments>
      <arguments>../lido:legalBodyName/lido:appellationValue/text()</arguments>
      <arguments>//lido:lido/nothing</arguments>
   </uri_function>
</uri_rules>
```

```
String uriForActors(String className, String authority, String id, String name, String birthDate)
```

The reason why these URI generators require so many arguments is because the resulting URIs must be repeatable given the same source information. The raw materials for these functions are fetched using XPath, at least in the example above.

Suggested Changes

Until now there have only been a few files created in the format described above (although the one for LIDO 0.7 is large in itself, consisting of more than two-thousand lines!). We are at a point now where we have the opportunity to modify the format since the few existing files can easily be restructured.

When making changes at this juncture, it can be useful to make a clean break from the current conventions so that there is never confusion between new files and ones that have accumulated during the development and prototyping which led to the current format. But the primary reasons for the changes is to improve readability and allow for a straightforward means to automatically write as well as read the mapping files. It is the read/write combination that is required for this format to be the basis of software tools.

Our proposals here will therefore include changing the names of tags (easy find-replace in older examples) and some of the structure (requires manual adjustment). The philosophy behind any changes will be the DRY Principle (Don't Repeat Yourself). Defaults are chosen such that a great deal of redundancy is removed. In fact, the LIDO mapping file used here was reduced from more than 2500 lines to just over 1600 which represents a reduction of one third!

We should also establish a clear version strategy for moving our group's mapping file format forward, so that there is a clear roadmap for those making implementations now and in the future. Software for reading and writing can then be made backwards-compatible and used as a tool for managing version upgrades to existing mappings.

Basic XML Structure: Entity and Property

The basic structure is of course essentially correct, except some DRY renaming would make it more easily human-readable, and a clear indication of the contents of the different sections using the **entity** and **property** tags has a number of advantages.

```
<mapping>
  <map>
    <domain_map/>
    <link_map>
      <range_map/>
      <path_map/>
    </link_map>
    <link_map/>
    ...
  </map>
  <map/>
  ...
</mapping>
```

```
<mappings version="0.9.0">
  <mapping>
    <domain>
      <source/>
      <entity/>
    </domain>
    <link>
      <path>
        <source/>
        <property/>
      </path>
      <range>
        <source/>
        <entity/>
      </range>
    </link>
    <link/>
    ...
  </mapping>
  <mapping/>
  ...
</mappings>
```

The purpose of these mappings is to create triples of the form **entity-property-entity** where both are named according to the CIDOC-CRM (with their associated E??_Name and P??_Name tags), so it makes sense to simply given them these names.

The clear delineation of **entity** has the added advantage that it can be used to encapsulate the URI generation mechanism. Likewise, **property** can capture the conditional mechanism. Both of these which will be covered in a later sections.

A version number is added so that compatibility with the software which executes the mapping can be ensured.

Extensions: Additional Node

In the design of the extension structure, decisions were made which make automatic interpretation of the XML more difficult than it needs to be. The extension to create additional nodes is intended to turn **domain-path-range** to **entity-property-entity-Property-Entity** (capitalized are new), so the following structural change is suggested to make this perfectly clear.

<range_map> <src_range/> <target_range/> **<add_link/>** **<add_entity/>** </range_map>	<range> <source/> <entity/> **<additional_node>** **<property/>** **<entity/>** **</additional_node>** </range>

The suggested **additional_node** section here is clearly delineated and its content resembles **link** so it adds symmetry. Also, since it is a clearly delineated section beside **source** and **entity**, it is easier for the interpreter of the XML to see as an optional whole section.

In the original, **add_link** was used to give a CRM property name like **P2_has_type**, which prompts renaming to **property** in this context.

Extensions: Internal Node

When a domain-path-range is to result in two new triples with an internal node entity inserted in between. So this maps **domain-path-range** to **entity-property-Entity-Property-entity** (capitalized are new), an extra section is added to the target.

<path_map> <src_path/> <target_path> <int_link/> **<int_entity/>** **<int_link/>** </target_path> </path_map>	<path> <source/> **<property/>** **<internal_node>** **<entity/>** **<property/>** **</internal_node>** </path>

Similar to the case of an additional node, the internal_node section is delineated from the original target path in a separate **internal_node** section. This makes it easier for the interpreter in the same way, with clearly optional pair of items, this time in the opposite order.

Conditionals

In the original specifications there was a description of the conditional creation of triples guided by a full boolean expression mechanism. This means that any combination of and/or/not statements can be constructed to decide whether a link should be made or not.

The core condition is either a determination of **presence/absence** or a **comparison** of a fetched value with a given constant value. The structure so far used for this in practice is very limited and somewhat awkward since it consists only (so far) of **has_value** determinations in an **if** construction. The current prototype implementation does not handle all expressions, only these.

Deciding whether to make a link amounts to deciding whether the domain entity should be attributed a given property which points to another entity. It is in the **property** or "path" connecting the entities that the conditional can then be placed.

It would make sense to put the condition in the position between the property tags as shown below, and to capture the base component boolean in a simple **exists** tag which optionally contains a **value** attribute to which the result is to be compared.

```
<target_path_condition>
   <if>
      <path>...</path>
      <has_value>...</has_value>
   </if>
</target_path_condition>

<src_path_condition>
   <if/>
</src_path_condition>

<src_domain_condition/>

<target_domain_condition/>

<src_range_condition/>

<target_range_condition/>
```

```
<property tag="P?">
   <exists value="...">...</exists>
</property>

<property tag="P?">
   <and>
         <exists>...</exists>
         <exists value="...">...</exists>
   </and>
</property>

<property tag="P?">
   <or>
      <exists>...</exists>
      <and>
         <exists value="...">...</exists>
         <not>
            <exists>...</exists>
         </not>
       </and>
   </or>
</property>
```

Further, if the **exists** tag is considered to be a function returning a **boolean** response, it can be combined easily with tags for the boolean operations when they are structured properly. The tags **and**, **or**, and **not** also then become functions that return true or false and do so by combining the values of their subtag entries.

URI Generation

The prototype implementation does not currently suggest the best way forward for properly defining and executing the URI policies, but with some small modifications we can be sure that the process will be future-safe.

It is only entities which require the generation of URI values, so rather than have an unattached section in the mapping XML file it makes sense to encapsulate the URI generation into the **entity** tag.

```
<range_map>
   <src_range/>
   <target_range/>
   <uri_rules>
      <uri_function>
         <name/>
         <arguments/>
         <arguments/>
         ...
      </uri_function>
   </uri_rules>
</range_map>
```

```
<range>
   <source/>
   <entity tag="E?">
      <uri_function name="...">
         <arg name="...">....</arg>
         <arg name="...">....</arg>
         ...
      </uri_function>
   </entity>
</range>
```

There are two other changes to the actual structure of the URI method call which would greatly contribute to the accuracy and readability of the result. First if all the **name** of the function is isolated as an attribute so that the content can be clearly just a list of arguments. Secondly, the **arguments** are named rather than just being a sequence of un anonymous values. Recall that we are calling methods like this one:

```
String uriForActors(String className, String authority, String id, String name, String birthDate)
```

Clearly naming the parameters will avoid errors that would have otherwise arisen, and they would certainly no longer require nonsensical arguments in order to signal the absence of a value.

```
<uri_function>
   <name>uriForActors</name>
   <arguments>...</arguments>
   <arguments>...</arguments>
   <arguments>....</arguments>
   <arguments>//lido:lido/nothing</arguments>
</uri_function>
```

```
<uri_function name="Actor">
   <arg name="authority">....</arg>
   <arg name="identifier">....</arg>
   <arg name="name">....</arg>
</uri_function>
```

The functions themselves should then be set up to fetch their named parameters as XPath expressions in the source explicitly and they must have clear strategies in place for when values happen to be absent. When there is only one argument, the name can be omitted since there is no ambiguity.

Some functions require some extra information which is not available from the source document, such

as the **entityTag** (which entity is being created) and the **domainURI** (which URI was used for domain in this set of links). Since functions will be fetching names values from their "environment", they can fetch these just as well. The values can be made available by default so that any URI function can fetch them.

Also, since many of the URI function uses only fetchable values and one argument that is the XPath statement text(), this can be considered the default scenario and the argument omitted entirely. Taking it step by step, we could imagine this evolution:

```
<uri_function name="Literal">
   <arg name="note">text()</arg>
</uri_function>
```

```
<uri_function name="Literal">
   <arg>text()</arg>
</uri_function>
```

```
<uri_function name="Literal"/>
```

These simplifications, based on properly chosen defaults, make the result much more readable and therefore amenable to being used as a document for discussion.

Defining URI Functions

In the prototype code, a Java class is used to define the URI functions and they are called using the Java Reflection API. The disadvantage of this approach is that it explicitly depends on the Java programming language, and the parameters cannot be assigned by name.

Also, the naming conventions among these methods is not entirely consistent with different orderings like **uriConceptual** and **appellationURI**, which should be rectified by using a well-considered and perfectly consistent convention. Each method should also be outfitted with a version number such that there is never any ambiguity about which function is being executed.

The level of complexity in the URI methods is really minimal, since they simply attempt to build a URI string from a number of identifier strings as raw materials. When certain parameters are missing, they can make different choices about the URI they generate, and when not enough information is present to create a useful URI, they default to generating a generic unique identifier or UUID.

As the prototype shows, many of the defined functions have parameters which are not used at all in the body of the function, so they are irrelevant but included for perceived consistency. In reality the prototype did not achieve a generic implementation, so these workarounds did not accomplish the intended goal.

A better approach than a Java class for these definitions would be to explicitly define and publish function definitions in **pseudo-code** that anyone can understand, not only programmers. This way any programming language can be used as an implementation. Each pseudo-code function should be accompanied by an exhaustive series of examples showing input and output so that different implementations can be verified.

For example, the following method is not very understandable, and certainly not for a non-programmer:

```
public String appellationURI(String className, String subjUri, String appellation) {
    tem = reading("appellationURI");
    String uri = tem.get(0);
    if (subjUri == null || appellation.equals("")) {
            return uuid("");
    } else {
            if (!className.equals("Appellation")) {
                for (int i = 1; i < tem.size(); i++) {
                    uri = uri + tem.get(i);
                }
                uri = uri + subjUri + "@" + appellation;
            } else {
                uri = uri + "Appellation/" + subjUri + "@" + appellation;
            }
            return encode(uri);
    }
}
```

The actual nature of the pseudo-code which would explain what the functions do in a way that everyone can understand is not handled here, but it should enable building these functions in any programming language. There are only several handfuls of these functions, so with a clear one-page official description of each, which includes an exhaustive set of examples, implementation could be made easy.

Querying the Source

In the original documentation about this mapping format from 2006 there seemed to be an indication that that the source was expected to be data directly from the tables of a relational database. On the other hand, the later introduction of URI functions were only ever based on querying using **XPath**, assuming that the source is XML.

We propose that this be resolved, if it hasn't been already, to work solely on the basis of XPath and XML, since the reasons are clear. Assuming W3C standards is much more future-safe than building any dependency on a particular relational database vendor or SQL dialect. Also, it makes sense to consider any path from a relational database to be a **pipeline** in order to separate the concerns of complete data extraction and actual mapping to CRM triples. First enable dumping of database content to XML (if it is not stored in XML, which is becoming more common) and then execute the mapping file to generate triples.

In the event that a decision is made later on to generate triples directly from the tables of a relational database, we can simply expand the definition of how the **source** tag is interpreted. Now it is assumed to be an XPath expression locating segments in a source XML document, which is a simplification that the pipeline approach affords us. The source tag could perhaps be elaborated with attributes or sub-tags that describe other kinds of queries.

Conclusions

We have analyzed the existing mapping format and made suggestions for improving its readability as well as the ease with which it can be written as well as read using software. We have also argued that a clear commitment be made to use XML as source format so that standard XPath can be used for querying, and that URI functions be officially described in a pseudo-code language and accompanied by a comprehensive set of example input and output values.

Implementation

Initial work has already been done to enable software to work with the suggested format, and it has been tested on the elaborate mapping file built for LIDO. The work initially done to build the existing mapping file was **preserved** in this process, and the file in the new format can be viewed online (http://goo.gl/FjmE3) and compared to the original (http://goo.gl/Xnf4V).

The current code reads the new format in a way that makes it construct a useful tree of typed objects in memory, rather than a generic Document Object Model. The object tree (http://goo.gl/yv7K5) will be used as a kind of pre-configured machine, which itself becomes the main component of the mapping engine. This was the technique employed for code generation in the current Delving SIP-Creator.

The values of entity and property tag attributes are controlled by enumeration classes (http://goo.gl/ulXki and http://goo.gl/mHUjA) which were already useful in catching numerous inconsistencies in the existing mapping file such as:

`<target_domain>39_Actor</target_domain>`	`<int_link>P12B_was_present_at</int_link>`
`<int_entity>E39_Actor</int_entity>`	`<property tag="P12i_was_present_at"/>`

The associated unit test locks down the reading and **writing** functionality to ensure that it is completely accurate by comparing the written result to the original.

```
@Test
public void lidoToCRM() throws IOException, ParserConfigurationException, SAXException {
    URL mappingFile = getClass().getResource("/rdf/lido-to-crm.xml");
    MapToCRM.Mappings mappings = MapToCRM.readMappings(mappingFile.openStream());
    String xml = MapToCRM.toString(mappings);
    String[] fresh = xml.split("\n");
    List<String> original = IOUtils.readLines(mappingFile.openStream());
    int index = 0;
    for (String originalLine : original) {
        originalLine = originalLine.trim();
        String freshLine = fresh[index].trim();
        Assert.assertEquals("Line " + index, originalLine, freshLine);
        index++;
    }
}
```

With verified reading/writing in place, we have the foundation for creating a software tool for **editing** as well as executing the mapping described in the XML file.

What is next?

The next steps will be to build up the code of the object tree such that it takes an input document and generates the triples that the mapping describes. Once the mapping can actually be **executed**, the remaining challenge of building a user-friendly graphic interface begins.

The interaction design of this interface should be worked out by those experienced with building mappings, together with those building the code such that the two points of view are merged.