An Introductory Developer Documentation for Mopy

General Overview

Mopy maps classes and properties defined in an ontology onto conventional objects, common in object orientated programming languages such as Python. This is achieved by parsing ontology documents (RDF schemas) and generating Python code for each class in the ontology. The parser/generator (genpy.py) provides Mopy's flexibility, since any change of the ontology can be immediately reflected in the model by running the generator with the modified schemas. Although some examples are provided, it is advised to follow the relevant sources parallel to reading this document.

Dependencies

Python: Mopy is written for Python 2.4, also tested using Python 2.5

rdflib: Mopy is using rdflib (http://rdflib.org/), a Python RDF library, for parsing documents and serialising the data contained in the model back to RDF.

Mopy's Model

As described in the installation notes (http://motools.sourceforge.net/mopyreadme.html) mopy does not come with a pre-generated model. However, a set of schemas describing the music ontology and related ontologies are included together with a parser/generator, which is used for producing its model (model.py) from the Music Ontology definition. All Python classes corresponding to music ontology classes are defined in this file. A typical Mopy class consits of the following methods:

Each class definition contains a doc string obtained form literal values of *rdfs:comment* for the class. An initialisation function sets up a dictionary holding *ProperySet* objects. These objects can hold a set of properties declared for a given class in the ontology. Managed access to the contents of these objects is implemented as python class attributes declared after the *__init__* function. For more details, refer to the documentation of managed attributes (i.e. >>>print property.__doc__). Additional utility functions are provided: First, to prevent using or adding a foreign attribute to a class (by overloading the python class's *__setattr__* method), and for providing a string representation of the class.

Parser/Generator

The generator (genpy.py) is responsible for producing the model described above. In the main section the ontology documents are loaded into a conjunctive graph (provided by rdflib). This is used for extracting the information about classes and properties defined in the ontologies. Property descriptions are extracted from the graph using the objects of rdfs:comment predicates:

```
for p in graph_props:
    doc = "\n".join(spec_g.objects(p, RDFS.comment))
```

Mapped ontology class definitions are created by a Generator class which extracts information from the the conjunctive graph and generates python code for a particular ontology class.

Class definitions are consisting of the python class names and parent names e.g.:

```
class mo___Recording(event___Event):
```

The ontology class hierarchy is obtained from the rdf graph using the *getParents* method of the generator. The primary source of this information is the objects of *rdfs:subClassOf* predicates for a given class.

```
for parent in self.graph.objects(self.c, RDFS.subClassOf):
    if not isinstance(parent, BNode):
        p.append(parent)
```

It is also necessary to handle parents named in RDF collections, follow *sameAs* links and handle orphaned classes, with no parents declared. (See source code for details.)

Properties for each class are extracted in a similar manner from the ontologies. However, it is necessary to note an important difference between the RDF data model and programming object models. While in most programming languages class declarations also contain attribute or method declarations, classes and properties in ontologies are declared independently. A class URI declared explicitely in a propertiy's domain (*rdfs:domain*) or in an RDF collection means that the property in question can be a valid property of the class. Assuming we have a list of all *properties*, this can be obtained as follows:

```
for property in properties
    self.graph.triples((property,RDFS.domain,class)
```

If a class is named in a property's range (meaning that the property can take this class as its value) we need to add its inverse properties to the class. Additionally, we need to add sub properties and properties declared in classes pointed by *owl:sameAs* links. Finally, we need to extract the types allowed to be taken as values of each property. As the example shows below: the allowed type of *mo:composer* is *foaf:agent*.

```
self._props["composer"] =
PropertySet("composer","http://purl.org/ontology/mo/composer", foaf___Agent,
False)
```

MusicInfo

The MusicInfo class (MusicInfo.py) can hold a collection of python music ontology objects. This class can be used as a temporary storage for a set of instances we are working with. It provides utility functions: for example it merges already existing objects.

RDF interface

The rdf interface (RDFInterface.py) provides functions for importing and exporting RDF data to an from the Python model. It uses rdflib to load an RDF document into a conjunctive graph which is used to parse the data into a collection of mopy objects stored in a MusicInfo object.