An RDF Schema Binding for the DDI Information Model

This document describes a binding of the UML profile defined for DDI 4 to RDF Schema (RDFS) and OWL. It defines a mapping to RDFS/OWL for each modelling construct supported in the UML profile. It also discusses some additional topics, such as the incorporation of mappings to other RDF vocabularies, and the use of Named Graphs for packaging and re-use of instance information.

Table of Contents

Introduction 1

Instance packaging and Named Graphs 2

Modules and namespaces 3

Skeleton 3

Classes 4

Associations 4

Compositions 4

Generalization 5

Source cardinality 5

Target cardinality 5

Literal properties (UML attributes) 5

Cardinalities on literal properties 6

Datatype mapping 6

Mappings to other vocabularies 6

TBD issues 6

# Introduction

This document describes an RDFS/OWL binding of the UML profile developed in the “Future of DDI Lifecycle” workshop. This binding is part of a bigger picture for producing an updated, more modular, multi-target-platform version of the DDI specification. The production process for this specification is intended to include these steps:

* A platform-independent **information model** for DDI is modelled in an UML tool.
* This model conforms to a simplified **UML profile** that constrains the model to the use of a limited set of modelling constructs. For example, modelling constructs such as interfaces or stereotypes are not used.
* The model is exported as an **XMI file**.
* **Additional documentation**, including textual definitions and examples, is provided in another format, in a way that snippets of documentation can be associated with items in the information model via identifiers such as class names.
* **Target specifications** are generated by some scripted conversion process in a number of target formats, including RDFS/OWL, XML Schema, and HTML.

This document describes how the RDFS/OWL target specification can be created from the inputs.

We divide the UML profile into a number of modelling constructs (classes, associations, cardinalities, etc.), and describe the RDF triples that need to be generated for each instance of such a modelling construct in the information model.

The complete RDF output is obtained by merging all the generated RDF triples. Some additional RDF triples may be merged from additional files, to address metadata for the RDF schema itself, mappings to other RDF vocabularies, etc.

# Instance packaging and Named Graphs

Here we will briefly discuss one important difference between this RDF Schema binding and previous versions of DDI-XML.

It is often desirable to package parts of a DDI description, with the intent of re-using it in multiple places, or with the intent of versioning it.

In XML, this is typically done by giving an identifier to a subtree, and by referencing that subtree from all relevant places.

Many elements in DDI-XML exist mainly for the purpose of allowing the grouping of information into such re-usable form, or the referencing of such groups. They do not form part of the domain that DDI describes. They are merely artefacts of the requirement to package bits of DDI information for re-use or versioning.

In RDF, a different approach is used. Information expressed in XML is a tree. Information in RDF is a graph, or more often a *set* of multiple graphs, where each graph is named with a URI. To package parts of an RDF graph, the following approach can be used:

* The triples to be packaged are put into a separate graph.
* This graph is named with a URI under the control of the organization doing the packaging and publishing.
* The items (variables, questions, concepts, etc.) described within that graph are given URIs that *dereference to the named graph*. For example, an organization might publish re-usable concepts in a graph named <http://example.com/concepts>, and one concept described therein may be named <http://example.com/concepts#concept123>.
* Another organization desiring to reference that concept can do so by using that URI. The URI encodes not just the name of the concept, but also the location where the graph defining it can be found.

By using named graphs as a packaging mechanism, and by understanding URIs not just as unique identifiers for items but also as locators for their definitions, the model itself can be kept free of constructs for packaging, and can be focused solely on representing the domain.

# Modules and namespaces

We assume that the DDI Information Model will be organized as a number of *packages*.

We assume here that one RDF Schema file will be produced for each package.

To derive an RDF Schema representation for a package, a namespace URI and mnemonic namespace prefix has to be assigned. For example, let us assume that we will have a module called “DDI Core”. Its namespace information could be:

|  |  |
| --- | --- |
| **Title** | DDI Core |
| **Namespace URI** | http://rdf-vocabulary.ddialliance.org/core# |
| **Namespace prefix** | ddicore: |

All RDF files that define or use this module will start with a *namespace declaration*:

@prefix ddicore: <http://rdf-vocabulary.ddialliance.org/core#>.

This allows us to reference particular items in the module (e.g., classes) via short prefixed names like ddicore:Scheme.

**Open question**: Do these RDF namespace URIs need to be aligned with XML namespaces used in the XML binding? Should the URIs be made part of the “platform-independent” part of the information model?

# Skeleton

The following snippet provides some skeleton triples for describing the vocabulary generated from a module:

@prefix ddicore: <http://rdf-vocabulary.ddialliance.org/core#>.

<http://rdf-vocabulary.ddialliance.org/core> a owl:Ontology;

rdfs:label “DDI Core Vocabulary”;

rdfs:comment “””This is the DDI Core Vocabulary, an RDF

Schema vocabulary that defines foundational concepts

For describing the domain of statistics.”””.

The following sections will walk through a number of UML modelling constructs, showing how each is translated.

# Classes

## Input

A class in a package with a class name

## Output

ns:MyExample a rdfs:Class, owl:Class;

rdfs:label “my example”@en;

rdfs:isDefinedBy <http://rdf-vocabulary.ddialliance.org/core>.

## Comments

* There is a correspondence between packages and namespace prefixes. In the example, everything in the same package as our class will use the ns: prefix. If we reference terms in other namespaces, they’d get other prefixes. As always in RDF, the prefix stands for some full URI. The mapping between packages, prefixes and full URIs needs to be configured somewhere.
* By convention, class identifiers in RDF start with an uppercase letter.
* The label is intended to be human-readable, therefore the camel-case capitalization is removed. This may either be done algorithmically, or the label could be injected from the

# Associations

## Input

An association between two classes with an association name

## Output

ns:relatedTo a rdf:Property, owl:ObjectProperty;

rdfs:label “related to”@en;

rdfs:domain ns:ClassOne;

rdfs:range ns:ClassTwo;

rdfs:isDefinedBy <http://rdf-vocabulary.ddialliance.org/core>.

## Comments

* By convention, property identifiers in RDF start with a lowercase letter.
* See *Classes* above for notes on namespaces and labels

# Compositions

## Input

A composition that relates a parent class to a child class with a composition name

## Output

ns:child a rdf:Property, owl:ObjectProperty;

rdfs:label “child”@en;

rdfs:domain ns:Person;

rdfs:range ns:Person;

rdfs:isDefinedBy <http://rdf-vocabulary.ddialliance.org/core>.

## Comment

* See *Classes* above for notes on namespaces and labels
* The output is identical to the generated RDF for associations. We do not make a distinction between them.

# Generalization

## Input

A superclass related to a subclass

## Output

ns:Subclass rdfs:subClassOf ns:Superclass.

## Comments

* None

# Source cardinality

**TODO.** *I’m not sure how to express this in OWL, needs a bit of research. —RC*

# Target cardinality

## Input

A cardinality for a named association or composition on a given class. Cardinalities can be 1…n, 0…1, 1…1, or in other words “minimum one”, “maximum one”, and “exactly one”.

## Output

ns:MyClass rdfs:subClassOf [

a owl:Restriction;

owl:onProperty ns:myProperty;

owl:minCardinality 1;

].

ns:MyClass rdfs:subClassOf [

a owl:Restriction;

owl:onProperty ns:myProperty;

owl:maxCardinality 1;

].

ns:MyClass rdfs:subClassOf [

a owl:Restriction;

owl:onProperty ns:myProperty;

owl:cardinality 1;

].

## Comments

* No triples are generated for the unconstrained cardinality (0…n).

# Literal properties (UML attributes)

## Input

An attribute associated with a class, consisting of a name and datatype

## Output

ns:numberOfMissingLimbs a rdf:Property, owl:DatatypeProperty;

rdfs:label “number of missing limbs”@en;

rdfs:domain ns:OwningClass;

rdfs:range xsd:integer;

rdfs:isDefinedBy <http://rdf-vocabulary.ddialliance.org/core>.

## Comments

* See *Classes* above regarding namespaces and comments.
* See the section on *Datatype Mapping* for the supported datatypes.

# Cardinalities on literal properties

See *target cardinality* above.

# Datatype mapping

**TODO**

* See <http://www.w3.org/TR/rdf11-concepts/#xsd-datatypes> and following sections for the default datatypes in RDF; these include a large selection of XML Schema types, as well as rdf:HTML, rdf:XMLLiteral, and rdf:langString.
* Additional user-defined datatypes are possible.

# Mappings to other vocabularies

Additional RDF triples may be merged into the resulting RDF file in the process. For example, an additional RDF file could be written with mappings to other RDF vocabularies.

Such mappings could take the following forms (where one term is from a DDI module namespace, the other term from some third-party vocabulary):

ns1:SomeClass rdfs:subClassOf ns2:OtherClass.

ns1:SomeClass owl:equivalentClass ns2:OtherClass.

ns1:someProperty rdfs:subPropertyOf ns2:otherProperty.

ns1:someProperty owl:equivalentProperty ns2:otherProperty.

# TBD issues

Here we list some further issues that still need to be considered.

* Will the UML model contain “abstract” classes? If so, a way of representing that information in RDF Schema will be needed. The options include: (i) not define a class, but use a union of the concrete subclasses wherever the abstract class is referenced; (ii) define a class, and use some convention to communicate that this class is not to be used in instance data (e.g., appending “(abstract)” to the rdfs:label).
* Is there a requirement that the ordering of the values of some compositions or associations is retained? In other words, do we need “ordered collections” or something like that? This would be somewhat awkward to represent in RDF, but in practice, retaining order is sometimes important.