DOCUMENTATION SNIPPETS ON MEDIATOR

re-engineered from Correndo's code

Documentation

version: 1

Paul Brandt, Eindhoven University of Technology,

Abstract

Contents

1 Package service.mediation.edoal

1.1 Gebruik van de TransformationVisitor

The visit method visit the Transformation structure and generate the RewritingRule that implements the Transformation.
De TransformationVisitor implementeert de transformatie van een entity in een ander entity.

1. Creeer nieuwe transformationVisitor()
   a) Initieert drie private variabelen:
      - lmr, rmr, als nieuw object MediationResult()
        - .patterns wordt geïnitialiseerd om te wijzen naar een nieuwe lijst van triples.
      - rr als nieuw object RewritingRule(); Een rewritingRule() zal uiteindelijk een lijst van FunctionalDependency()'s zijn.

   b) Gebruikt PropertyVisitor(), ValueVisitor()
2. Roep hierop de method visit() aan, met als parameter een EDOAL transformatie
3. Nu kan de rewritingRule() worden opgevraagd middels getRewritingrule

Note: Andere vist methoden zijn nog niet ondersteund, slechts stubs gegenereerd.

1.2 Gebruik van de PropertyVisitor class

The PropertyVisitor class implements the EDOALVisitor and MediationResultGenerator interface and produces MediationResults instances.
To that end, it encapsulates:

- MediationResult, representing the result of the mediation after the alignment visit;
- The subject (Node s), representing the subject node of the produced pattern;
- The object(Node), representing the object node of the produced pattern.

It is possible to set an external mediation result, partially composed by other MediationResultGenerator.

1.3 Gebruik van de ClassVisitor class

The ClassVisitor class implements the EDOALVisitor and MediationResultGenerator interface and produces MediationResults instances.
To that end, it encapsulates:

- MediationResult, representing the result of the mediation after the alignment visit;
- The subject (Node s), representing the subject node of the produced pattern;
- The object(Node), representing the object node of the produced pattern.

It is possible to set an external mediation result, partially composed by other MediationResultGenerator.

1.4 Gebruik van de visit method

De visit() methode genereert een RewritingRule die de transformatie daadwerkelijk uitvoert. Omdat de vorm van een rewriting rule afhankelijk is van het type entity dat wordt omgeschreven, is deze methode overloaded met net zoveel varianten als er entity types bestaan.

1.5 Gebruik MediationResult class

The Class MediationResult implements the result of a mediation applied to an EDOAL expression.
A mediationResult consists of:

- Three **Nodes** that represent the eventual replacement to be applied to:
- The subject (Node *s*)
- The object (Node *o*)
- The predicate (Node *p*)
- A **STripleList** of *patterns*; each pattern representing an EDOAL expression that provides for the way how pairs of entities are related to each other;
- An **ArrayList<FunctionalDependency>** of functional dependencies *fds* between variables that are potentially involved in the transformations that are applied by the patterns.

1.6 Gebruik van de EDOALMediator class

1.6.1 Gebruik van de Mediate(Alignment) method

The Mediate(Alignment) method will translate an EDOAL alignment (cast to an org.semanticweb.owl.align.Alignment) into (an internal alignment based on) RDF pattern rewriting rules. To that end, it performs the following operations:

1. It creates two new *forth* and *back* JenaAlignments;
2. It retrieves the **ontoA** and **ontoB** ontologies from the EDOAL alignment, and
   a) it sets *forth*.sourceOntology to ontoA, and *forth*.targetOntology to **ontoB**,
      - as RDF statement:

   b) it vice versa sets *back*.sourceOnt to ontoB, and *back*.targetOnt to **ontoA**
   c) This is done by creating RDF statements
3. It processes each alignment cell in

1.6.2 Mediate(Cell, forth, back)

This method implements the generation of the rewriting rules that follow from a single EDOAL-cell. As such, the term *mediation* should be considered erroneous since it does not mediate, i.e., translate a query, but generates in stead the rewriting rules that perform the actual mediation. An EDOAL cell is defined as:

```
corresp ::= \<Cell {rdf:about=" URI "} /\>
 \<entity1\> entity \</entity1\>
 \<entity2\> entity \</entity2\>
 \<relation\> STRING \</relation\>
 \<measure\> STRING \</measure\>
 (\<transformation\> transformation \</transformation\>)*
 (\<linkkey\> linkkey \</linkkey\>)*
 \</Cell\>

entity ::= instanceExpr | classExpr | attributeExpr

attributeExpr ::= propertyExpr | relationExpr
```

A cell always describes a `\<relation\>` between two `\<entity\>`'s. The first `\<entity1\>` represents the *source* (or *from*) entity, the second one the *target* (or *to*) entity. Each entity can be *simple*, i.e., just a

class, or *complex*, i.e., a composition of multiple classes (AND, OR, NOT), or multiple relations where restrictions apply, or combinations thereof.
The 'mediation' (generation) of rewriting rules from these `\<entity\>`'s is performed in three steps:

1. Unravel every element of the source/from/entity1 and target/to/entity2 parts of the cell into an atomic RDF defined triple specification. This is deferred to the *mediateExpression*() method, but it comes down to triplicating the atomic elements that an entity consists of, e.g., a simple entity such as a class is mediated into a triple <*var*, RDF.TYPE, class_uri>, a complex [classA AND classB] expression is mediated into a list [<*var*, RDF.TYPE, classA_uri>, <*var*, RDF.TYPE, classB_uri>] of triples, where *var*'s represent unique RDF nodes. Note that the mediateExpression() method cannot support other types of compositions than AND; apparently, a list of triples represents a unification relationship, where a disjoint relationship cannot be represented yet in the list, as with the NOT relationship.
2. Unravel every element of the transformation into an atomic RDF defined triple specification. This is deferred to the *mediateTransformation*() method.
3. Since every atomic element of the correspondence cell now has been broken down to its RDF triple counterpart, these triples can be used to build a rewriting rule. This is deferred to the *addRewritingRule()* method, the form of which is dependent on (i) the relationship between both <entity>'s, and (ii) the complexity of the <entity>'s and <transformation>. The rewriting rule consists of three parts, see below, that together will be represented as a triple graph that models a rewriting rule. To that end, each triple that the parts are composed of will be added to the model graph as individual statement, composed of three graph triples each of which specifies the subject, the object and the predicate node as a triple that grounds the type of each triple-element of the original triple.

   - *LHS* represents the left hand side of the rule, and contains the triple pattern that should be matched in order for the rule to be executed. This field is filled with one of the atomic RDF triples form step 1;
   - *RHS* represents the right hand side of the rule, and contains the triple patterns into which the LHS should be translated;
   - *FD* represents the functional dependencies that exists between the yet unnamed and blank nodes that are the variables that exist in the query.

Currently, the code can only produce a rewriting rule if at least one of the <entity>'s represent a *simple* entity; any correspondence cell that carries more than one complex entity is ignored. This is due to the fact that the *LHS* cannot contain more than one triple.

1.7 Gebruik van de mediateExpression method

2 Package uk.soton.service.mediation

This package implements classes and methods that are relevant for performing a mediation.
Its foundation exists of two elements: * JENA: Because this mediation implementation is focussed on mediating data that is expressed as RDF triple, in the end, every meditation entity or operation is something about an RDF triple. *Jena* is a *Java API* which can be used to create and manipulate RDF graphs. *Jena* has object classes to represent graphs, resources, properties and literals. The interfaces representing resources, properties and literals are called Resource, Property and Literal respectively. In Jena, a graph is called a model and is represented by the Model interface. * Alignment: Mediation is performed on the basis of an (EDOAL) Alignment, which is not related to RDF or an RDF-graph. Therefore, any code that refers to the alignment part of the mediation requires an Alignment. An Alignment is an abstraction, i.e., an *Interface*, that defines the basic information contained in an ontology alignment based on RDF graph rewriting rules.

2.1 Alignment Interface

The Interface Alignment defines the basic information contained in an ontology alignment based on RDF graph rewriting rules. The name Alignment is misleading since its main object of interest is a set of rewriting patterns that latches onto RDF triples; therefore, following the terminology of Euzenat, this “Alignment” in fact represents the result of an EDOAL alignment, and should have been denoted as a **Mediation**. In spirit of reverse engineering and understanding, we will remain to use the illogical name. The main orientation of an alignment is from left to right.
It contains the following elements:

- Relation: this encodes the different kind of alignments that is supported (equivalence, subsumption, etc.). Currently it has only specified the Equivalence relation, which seems in contradiction with the mail from Corendo, d.d.August 28, 2015.
- getPatterns():
  public Hashtable<Triple,List<Triple>> getPatterns();
  public Hashtable<Triple,Hashtable<Node, FunctionalDependency>> getFunctionalDependencies();
  public void addRewritingRule(RewritingRule rule);
  public void setSourceOntologyURIs(List<String> sourceOntologyURIs);
  public List<String> getSourceOntologyURIs();
  public void setTargetOntologyURIs(List<String> targetOntologyURIs);
  public List<String> getTargetOntologyURIs();
  public void setTargetDatasetURIs(List<String> targetDatasetURIs);
  public List<String> getTargetDatasetURIs();
  /**
  - Match the input lhs to the managed alignments.
  -
  - (**???**) t Triple to match against the LHS of all the patterns contained
  - in the alignment.
  - (**???**) null if any of the LHS matched, a matching result otherwise */ public MatchingResult matchLHS(Triple t);

  /**
  - The Class MatchingResult represent the result of a matching between a BGP triple and this ontology alignment.
  - (**???**) Gianluca Correndo <gc3@ecs.soton.ac.uk> */ public class MatchingResult{
    /** The rhs. */ private List<Triple> rhs;
    /** The binding. */ private Hashtable<Node,Node> binding;
    /** The fdependencies. */ private Hashtable<Node,FunctionalDependency> fdependencies;
    /**
    - Instantiates a new matching result.
    -
    - (**???**) rhs the Right Hand Side
    - (**???**) binding the binding between variables
    - (**???**) fdependencies the functional dependencies */ public MatchingResult(List<Triple> rhs, Hashtable<Node,Node> binding, Hashtable<Node,FunctionalDependency> fdependencies){ this.rhs = rhs; this.binding = binding; this.fdependencies = fdependencies; } //Getters /**

- Gets the Right Hand Side.
-
- (**???**) the Right Hand Side */ public List<Triple> getRhs() { return rhs; }

/**

- Gets the binding between variables.
-
- (**???**) the binding */ public Hashtable<Node, Node> getBinding() { return binding; }

/**

- Gets the functional dependencies.
-
- (**???**) the functional dependencies */ public Hashtable<Node,FunctionalDependency> getFunctionalDependencies(){ return this.fdependencies; }

/**

- Gets the functional dependency relevant for a variable.
-
- (**???**) var the variable
- (**???**) its functional dependency */ public FunctionalDependency getFunctionalDependency(Node var){ return this.fdependencies.get(var); }
  } }

2.1.1 mail Correndo

From: Gianluca Correndo [mailto:gianluca_correndo@yahoo.com] **Sent:** vrijdag 28 augustus 2015 9:53
To: Brandt, P. (Paul) **Subject:** Re: Github Mediation - error in Junit
Hi Paul, I hope that, although dead, it will prove useful code. The alignments are by no means restricted to the equivalence relation. That is, if I remember correctly, a constraint of EDOAL language. This constraints make sense for EDOAL because the language is used to encode the alignments resulting from an ontology mapping tool, which most of the time can only detect when two classes or properties are equivalent. The kind of alignments I can handle are actually based on graph rewriting, basically I define a graph pattern as left hand side, and when I match it on a query BGP, I rewrite it using the rule's right hand side, renaming variables for instantiating correctly the pattern. These are syntax-based rules, so any closure due to reasoning is not taken into account. But you can, using syntactic rules, define: * class equivalence * property equivalence * property value patterns * property chains * class conjunction expressions * property conjunction expression
The downsize of it is that expressing such rewriting rules in RDF is cumbersome. I used reification and the typical "simple" rules are not that easy to read (see resources/kettle_internal.ttl)
Hope that helps Gianluca

2.2 JenaAlignment class

This class represents the foundation to the mediator. Since anything in the end will be represented as RDF triple, there needs to be something that is capable of representing RDF and processing RDF. Jena is the de-fact java API to address RDF, and therefore a very low-level 'operating system' to W3C's Resource Description Framework (RDF). In Jena, a model can depict both a specification of the universe of discourse, as a specification of the state of affairs, about that part in reality that we are concerned with. Technically, a model is represented as a directed graph where Nodes represent either a subject or an object and Edges represent the properties (a.k.a. relations) between the subject and object.

In itself, RDF is not capable of describing reality in highly accurate semantics; it can only discern an object from a subject and a relation between them. At the same time, within the context of the mediator it is not used as vehicle to specify semantics: in stead, it is used as a means to represent the syntax of the semantics, and to make translations between various syntactical representations of the same semantics.
The Jena Alignment class specify a lot of things, the most important ons being:

- **inner : Model:** This holds the graph that represents the rewriting rules that apply.
- **patterns : <Triple, List<Triple>>:** This holds the index of rules that apply, where the first triple identifies the pattern for which a rule is available (in **inner**) that implements the translation to the list of triples from the second part of the pattern. Note that this is only an index.
- **fdependencies : <Triple, <Node, FunctionalDependency>>:** This holds the index of functional dependencies that are contained in the alignment. The first triple identifies the pattern that incorporates a functional dependency between variables, e.g., var = func(parameters) (in **inner**) /** The RDF Node root of the alignment document. */ private Resource root;

2.2.1 addRewritingRule() method

3 Package align.impl.edoal.Transformation

This implements a transformation of an Entity into another Entity. The transformation is specified usually through function and can go one way or both.

3.1 ValueExpression()

De ValueExpression() interface class wordt gebruikt als vehikel om de entiteiten uit de ontologie te selecteren. Dit representeert daarmee de zgn. EntityLanguage van Euzenat, en zou een SPARQL of SPIN of RDF of … moeten kunnen zijn.
Vermoedelijk kan voor elke EntityLanguage die je in gedachte hebt, een interface implementatie worden gemaakt.

4 EDOAL concepts

4.1 EDAOL Expression

Refer to (**???**, p.332)

4.2 Correspondence relation

The EDOAL correspondence relation is defined in http://alignapi.gforge.inria.fr/format.html, as follows (Relation Element):
The relation element only contains the name identifying a relation between ontology entities. This relation may be given:

- through a symbol: > (subsumes), < (is subsumed), = (equivalent), % (incompatible), HasInstance, InstanceOf, ~> (non transitive implication), where:
  - > and <, and
  - HasInstance and InstanceOf are each others inverse;
- through a fully qualified class name of the relation implementation. If this class is available under the Java environment, then the relation will be an instance of this class.

Hence, `<relation>=<\relation>` and `<relation>fr.inrialpes.exmo.align.impl.rel.EquivRelation</r` are equivalent.
We assume the `%` relation to represent the disjoint relation. The semantics of each of the relation, in the context of the knowledge that is available and represented by the data, needs further elaboration. Especially we need to look into the purpose of query transformation: what use is it to us?

4.2.1 Semantics of subsumption '<'

In order to better explain the implications of a subsumption relation, we introduce the following example. Applications A and B make observations about a parking lot. The parking lot can contain cars, motor cycles, buses and more. Assume that application B is not interested in the type of vehicle, as long as it carries a registration plate it suffices, e.g., it will incorporate cars, motor cycles, buses and trucks into its world. Application A is only interested in cars, motor cycles and bikes. Ontology A acknowledges the existence of cars, motor cycles and bikes. Ontology B acknowledges vehicles that carry a registration plate.
[TODO: formele definitie van ontoA en ontoB invoegen conform Guizzardi]
Assume the following alignment:

1. ontoA:car < ontoB:vehicle
2. ontoA:motorCycles < ontoB:vehicle
3. ontoA:bike % ontoB:vehicle

This alignment expresses that both cars and motor cycles are considered vehicles that carry registration plates, as opposed to bikes that don't and hence cannot be considered a vehicle. Since we follow the mathematical practice that one shall not assume what has not been specified explicitly, we cannot conclude from this alignment that only cars and motor cycles are vehicles that carry a registration plate. This is even more relevant since what is and what is not acknowledged to exist in the observed world, and hence, taken into consideration, is very much dependent on the purpose of the applications.
Now we can address the semantics of a query, as well as a statement that carries a mere fact.

4.2.1.1 Query semantics

Consider the following queries, Q_1 and Q_2, within the context of the above example:
Q_1: ˜ SELECT x? WHERE { x? typeOf ontoA:car } ˜
Q_2:

```
SELECT y? WHERE {
    y? typeOf ontoB:vehicle
}
```

Notwithstanding the relation that exists between Q_1 and Q_2, both queries do not express the same thing. In fact, if we apply them on a specific parking lot, by virtue of the alignment that holds, the answers to Q_1 (denoted as A_1) will refer to the subset of cars that exist in the answer A_2 to Q_2 that also contain the motor cycles at least: $A_1 \subset A_2$. This implies that a direct translation from Q_1 in Q_2 would be erroneous. A translation from Q_2 into Q_1 *is* possible, however results in an incomplete translation, since the set A_1 does not contain the individuals that are implied by Q_2.
Now consider the additional query Q_3:
Q_3:

```
SELECT x? WHERE {
    {x? typeOf ontoA:car} UNION {x? typeOf ontoA:motorCycles}
}
```

Again, a relation exists between Q_3 and Q_2, but they still do not express the same thing. Consider both translations:

- ontoA ← ontoB, i.e., Q_2 into Q_3: one might think this translation is complete, but it still is not because its necessary specification that `ontoA:car` ∪ `ontoA:motorCycle` ≡ `ontoB:vehicles` is lacking. Therefore, within the context of this alignment, although we can translate Q_2 into Q_3, we need to be aware of the fact that the translation is incomplete and hence $A_2 \subset A_3$.
- ontoA → ontoB, i.e., Q_3 into Q_2: since $A_3 \supset A_2$ it would be erroneous again to directly perform this translation because it would provide application A with data that are outside the semantics that ontoA assumes for Q_3.

In conclusion, the semantics of a subsumption relation has consequences for the way how queries are translated and how answers to the queries are interpreted. For applications that follow
The best way to proceed for this case would be to translate Q_3 into Q_2, however, filter the resulting answer A_2 into A_3' such that only cars and motor cycles are passed. Dus als de alignment uitgebreid zou zijn met:

- (auto's UNION motoren) = motorvoertuigen

dan zou q3 wél een volledige vertaling betreffen van q2.
Dit heeft consequenties voor het service protocol, namelijk, ontoA en ontoB weten niet van elkaars uitdrukkingskracht. Derhalve is het applicatie B niet bekend dat de resultaten van q3 op basis van zijn formulering van q2, alle motorvoertuigen op de parkeerplaats betreffen. Dit is alleen bekend aan de mediator. Dus is het aan het service protocol om deze informatie (betreffende de volledigheid van het antwoord) te verschaffen.
{answers} < {query} versus {answer} = {query}

4.2.1.2 Statement semantics

5 Code snippets

5.1 Jena: get object in specific triple

According to: https://jena.apache.org/tutorials/rdf_api.html

```
String hasRelationProp = "http://ecs.soton.ac.uk/om.owl#hasRelation";
Property prop = ((JenaAlignment)m.getJenaAlignment()).getModel().getProperty(hasRelationPro

StmtIterator iter = ((JenaAlignment)m.getJenaAlignment()).getModel().listStatements(
        null, prop, (RDFNode) null );
ResIterator ri = ((JenaAlignment)m.getJenaAlignment()).getModel().listSubjectsWithProperty(

if (ri.hasNext()) {
    System.*out*.println("The database contains hasRelations:");
    while (ri.hasNext()) {
        System.*out*.println("Obj:  " + ri.nextResource().getProperty(prop).getObject().toS
        System.*out*.println("Res:  " + ri.nextResource().getProperty(prop).getResource().t
    }
} else {
    System.*out*.println("No \<hasRelation\> were found.");
}
```