Towards a Mapping from ERDF(S) to Take Vocabulary

Ion-Mircea Diaconescu[1], Adrian Giurca[1], Gerd Wagner[1], Jens Dietrich[2]
E-Mail: {M.Diaconescu, Giurca, G.Wagner}@tu-cottbus.de,
J.B.Dietrich@massey.ac.nz

[1] Brandenburg University of Technology, Germany
[2] Massey University, New Zealand

Abstract. This paper presents a mapping solution from ERDF(S) to Take vocabulary. The work is related to an investigation of integrating ERDF Derivation Rules into Take inference engine. Some steps are required to finalize this task: a mapping between ERDF(S) and Take vocabulary, integration of ERDF knowledge base in Take, and empowering Take inference engine to deal with both Closed World Assumption and Open World Assumption, specifically with Open and Closed predicates and two types of negation: strong negation and weak negation.

1 Introduction and Motivation

The paper presents an approach of a mapping from ERDF(S)[3, 4, 2] to Take [6] vocabulary. This mapping is part of our project to extend Take with support for ERDF Derivation Rules.

Influenced by Mandarax [8], Take is a backward inference engine, designed to deal with objects as facts. In the actual implementation, Java objects are used. The engine use a polymorphic negation: (1) *strong negation* for `JPredicate` and `PropertyPredicates` - predicates generated for boolean methods respectively boolean properties from Java Beans. Those are computed without using rules. (2) *negation as failure* for `SimplePredicate` - predicates defined by rules and used in rules, facts, queries or external fact stores. Take compiles rules to an optimized Java code before the inference process is started, this having as primary advantage a better scalability.

Since explicit negative information is not provided (with exception of Boolean properties), and the inference is under *CWA* hypothesis, false information can be inferred in some cases. For instance, reasoning on top of a FOAF[3] facts base, a rule set can conclude that two persons does not `foaf:knows` one each other since no occurrence of a `foaf:knows` property is found in at least one FOAF file of the two persons refereing the other person (CWA). Since it is not mandatory to refer all known persons in your FOAF file (not relating a known person in your FOAF file does not means that you don't know him), the derived conclusion

[3] FOAF Specification - `http://xmlns.com/foaf/spec/`

might be not appropriate. In this example, it is obvious that `foaf:knows` can be represented as an *Open Property*.

Based on Partial Logic [7], ERDF comes to solve this problem by providing the possibility of expressing negated facts, and two types of negation in rules: *strong negation* and *weak negation*. Also it defines new types of classes and properties. In [9] detailed information and use cases are provided about those extensions and the rule language.

We are not aware of any substantial work regarding the mapping from ERDF (and implicitly RDF(S)[5]) vocabularies to Java classes. Such a mapping is a key point towards mapping RDF/ERDF descriptions to Java objects. The next paper section deals with this issue tailored to the Take inference engine mechanism.

2 Mapping ERDF Schemas to Take Vocabulary

2.1 Resolving URI's to Java Identifiers

Since in ERDF Schemas classes and properties are defined by URI's, first step is to define a mapping from URI's to Java identifiers. The following rules define this mapping:

1. Using MD5[4] hashes, each URI is mapped to an unique ID. It will be prefixed with an '_' (underscore), since Java identifiers cannot start with digits.
2. Optional, at the end of the identifier the local name (where is possible) is added. Additional, comments may be generated to improve readability.
3. The reversibility is obtained by storing each URI expressing a class or a property and its generated Java identifier.

Example 1. Mapping URI's to Java identifiers

```
http://example.org/voc/Person ⇒ _35EDA2C34D57C09DCDB4D1544C674779_Person
http://example.org/v/Person ⇒ _DD3A8FFF3850417783DC9651D86B0395_Person
```

Using MD5 hashes, for each different URI a different signature is obtained. Two or many URI's may express the same concept (such as *Person* from the above example), but since different URI's are used, a distinct class or property is generated for each of them. The mapping from Java identifiers to ERDF classes/properties is implemented by using a Java `Map`.

2.2 Mapping Classes and Properties

This section describes the mapping from ERDF classes and properties to Take vocabulary. The following rule expresses the general mapping from ERDF vocabulary to Java vocabulary:

[4] The MD5 Message-Digest Algorithm - http://www.ietf.org/rfc/rfc1321.txt

Rule 1 *(a) For each ERDF class a Java class is generated; (b) Each ERDF property maps to a Java property.*

Example 2. Mapping classes and properties from ERDF(S) to Java Vocabulary

```
<erdf:Class erdf:about="http://example.org/v/Person"/>
<rdf:Property rdf:about="http://example.org/v/name">
  <rdfs:domain rdf:resource="http://example.org/v/Person" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
</rdf:Property>

// Class related URI: http://example.org/v/Person
public class _DD3A8FFF3850417783DC9651D86B0395_Person {
  // Property related URI: http://example.org/voc/name
  private Collection<String> _4D9F16067649DF5A90773F7D832D9122_name;

  public Collection<String> get_4D9F16067649DF5A90773F7D832D9122_name() {
    return this._4D9F16067649DF5A90773F7D832D9122_name;
  }
}
```

Each ERDF property maps in Java as a collection containing all values of that property. This allows to express multi-valued ERDF properties. Accessors are generated for each property. Adding an element to collection must be done by checking some constraints depending by the type of the added value (datatype or object type).

2.3 Solving rdfs:range, rdfs:domain and rdfs:subClassOf relations

In ERDF multiple ranges are allowed for a property. Types are mapped according with their nature: (1) *datatype* mappings and (2) *object type* mappings.

Datatypes If all property ranges are datatypes then, according with RDF Semantics [1], the property type is the intersection of all those types. For compatible datatypes this intersection is defined, for all the rest the intersection will be an empty set and therefore the property will not be generated and an exception is thrown. Only datatypes defined by XML Schema datatypes [5] are allowed. A mapping from XML datatypes to Java types is defined [6]. All datatypes in Java will be expressed by using corresponding wrapper classes (e.g.`Integer` for `int`) and collections representing properties are parameterized with the corresponding wrapper class. The RDF datatype, `rdf:XMLLiteral`, maps to `String` type in Java.

Example 3. Mapping ranges for datatypes

```
<rdf:Property rdf:about="http://example.org/v/salary">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#double" />
```

[5] XML Schema Datatypes - http://www.w3.org/TR/xmlschema-2/
[6] XML datatypes mapping - http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html

```
  <rdfs:domain rdf:resource="http://example.org/v/Employee" />
</rdf:Property>

public class _f2989a52fddb17fa72653625cd9c0374_Employee {
  private Collection<Integer> _93ad619912a976c6c56623d3c6b73491_salary;

  public Collection<Integer> get_93ad619912a976c6c56623d3c6b73491_salary() {
    return this._93ad619912a976c6c56623d3c6b73491_salary;
  }
}
```

Since the Take engine manages itself properties having type `Collection`, we just have to define a `get` method which returns the collection. Before adding new values for a specific property, some constraints have to be verified (e.g. do not have the same value many times). An implementation of `Collection` (i.e. extending `ArrayList` by overriding the `add` method) is provided.

Example 4. Implementing DatatypeArrayList

```
public class DatatypeArrayList extends ArrayList<Object> {
  public boolean add(Object value) {
    if(!this.contains(value)) {super.add(value); return true;}
    return false;
  }
}
```

Object types All types which are not XML datatypes are considered object types. In this case, a proper intersection solution cannot be defined using an automatic method. When a single *range* is defined for a ERDF property, the collection will be parameterized with that type. If many ranges are used for the same property, then the collection representing the property is parameterized with the Java super type `Object`. The intersection of ranges is resolved at runtime by the `add` method, checking if the value wanted to be added is instance of all types expressed by its `rdfs:range` occurrences. For object types, we define a new `Collection` implementation, `ObjectArrayList`, which will be used to instantiate all object properties from our Java classes.

Example 5. Implementing ObjectArrayList

```
public class ObjectArrayList extends ArrayList<Object>{
  private ArrayList<String> types;

  public ObjectArrayList(ArrayList<String> types) {this.types = types;}

  public boolean add(Object value) {
    boolean valid = true;
    for(int i=0;i<types.size();i++) {
      try {
        valid = valid && (Class.forName(types.get(i)).isInstance(value));
      }catch(Exception e){System.out.println(e.toString());}
    }
    if(valid) {super.add(value); return true;}
    return false;
  }
}
```

Mapping rdfs:domain Unlike in Java, ERDF properties are global and applies to all classes defined as their domains. The proposed mapping creates a property with the same name in each class appearing in their `rdfs:domain` occurrences.

Example 6. Mapping properties having multiple domains

```
<erdf:Class erdf:about="http://example.org/v/Person" />
<erdf:Class erdf:about="http://example.org/v/City" />
<rdf:Property rdf:about="http://example.org/v/name" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
  <rdfs:domain rdf:resource="http://example.org/v/Person" />
  <rdfs:domain rdf:resource="http://example.org/v/City" />
</rdf:Property>

public class _dd3a8fff3850417783dc9651d86b0395_Person {
  private Collection<String> _ba6a578397a5a376500712b8cb11e9e9_name;
}
public class _0545f96c913977fbf95b0c52de54d6af_City {
  private Collection<String> _ba6a578397a5a376500712b8cb11e9e9_name;
}
```

Mapping rdfs:subClassOf Since Java does not allows multiple inheritance, but ERDF supports that by using `rdfs:subClassOf` property, we need to emulate multiple inheritance by using Java interfaces. The following rule defines the multiple inheritance approach:

Rule 2 *(a) For each class implied in an inheritance chain, an empty interface is generated; (b) Each class from the inheritance chain must implements all interfaces corresponding to its superclasses; (c) Each class implied in the inheritance chain contain a copy of all properties (and corresponding accessors and comments) of its superclasses.*

This methodology allows the usage of the Java operator `instanceof` for emulating multiple inheritance. All generated interfaces does not contains methods and are used only for the above purpose.

2.4 Expressing Closed, Open and Partial Properties and classes

ERDF specializes its classes and properties as *closed*, *open* or *partial*. Such a classification is important during the inference process (OWA or CWA may applies depending on the classification). The mapping uses Java annotations as a solution for classifying classes and properties. The following annotations are defined: (1) `@closed` - meaning *closed* class and/or property; (2) `@open` - to express *open* class and/or property; (3) `@partial` - denoting *partial* class and/or property. The default annotation is `@open`. Using Java annotation provides us an elegant solution for identifying types of classes and properties.

Example 7. Annotating properties

```
<erdf:Class rdf:about="http://example.org/v/Person" />
<erdf:ClosedProperty rdf:about="http://example.org/v/authorOf" />
  <rdfs:range rdf:resource="http://example.org/v/Publication" />
```

```
  <rdfs:domain rdf:resource="http://example.org/v/Person" />
</erdf:ClosedProperty>

public class _dd3a8fff3850417783dc9651d86b0395_Person {
  @closed
  private Collection<_64742878a633276917290732b0301d3b_Publication>
                     _5470da641d7738e11e2e1ef15e3c23a8_authorOf;
}
```

3 Conclusion and Future work

The paper provides a mapping solution from ERDF Schemas to Take vocabulary. Some issues such as the intersection of ERDF property ranges are considered and a concrete distinction between datatypes and object types is made.

Further work includes empowering Take engine with support for reasoning in both *OWA* and *CWA* taking in account classes and properties types and considering two kinds of negation, namely *strong negation* and *negation as failure*.

References

1. RDF Semantics. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-mt/.
2. A. Analyti, G. Antoniou, C. V. Damasio, and G. Wagner. Extended RDF as a Semantic Foundation of Rule Markup Languages. *Journal of Artificial Intelligence Research*, 32:37–94, 2008.
3. Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Negation and Negative Information in the W3C Resource Description Framework. *Annals of Mathematics, Computing and Teleinformatics*, 1(2):25–34, 2004.
4. Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Stable Model Theory for Extended RDF Ontologies. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *Proceedings of the 4th International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science (LNCS)*, pages 21–36, Galway, Ireland, 6-10 November 2005. Springer-Verlag.
5. D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation February 2004. http://www.w3.org/TR/rdf-schema/.
6. Jens Dietrich, Jochen Hiller, and Bastian Schenke. Take - A Rule Compiler for Derivation Rules. In *Proceedings of the International RuleML Symposium on Rule Interchange and Applications (RuleML-2007)*, volume 4824 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.
7. Heinrich Herre, Jan O. M. Jaspars, and Gerd Wagner. Partial Logics with Two Kinds of Negation as a Foundation for Knowledge-Based Reasoning. In D.M. Gabbay and H. Wansing, editors, *What is Negation?* Kluwer Academic Publishers, 1999.
8. Mandarax. project website. http://www.mandarax.org/.
9. Gerd Wagner, Adrian Giurca, Ion-Mircea Diaconescu, Grigoris Antoniou, and Carlos Viegas Damasio. ERDF Implementation and Evaluation. Technical report, March 2008.