Contract No. H2020 – 636078

INFORMATION TECHNOLOGIES FOR SHIFT TO RAIL

D1.4 – Semantic Query and Aggregation Engine

Due date of deliverable: 28/02/2017

Actual submission date: 25/06/2018

Leader/Responsible of this Deliverable: Riccardo Santoro - TRENITALIA

Reviewed: Y

Document status		
Revision	Date	Description
0.1	27/03/2017	First draft of the deliverable
1	22/07/2017	Added SPARQL examples with remote graphs
2	15/06/2018	Final draft of the deliverable
3	25/06/2018	Final version after TMC approval and Quality check

Project funded from the European Union's Horizon 2020 research and innovation program		
Dissemination Level		
PU	Public	X
CO	Confidential, restricted under conditions set out in Model Grant Agreement	
CI	Classified, information as referred to in Commission Decision 2001/844/EC	

Start date of project: 01/05/2015 Duration: 36 months

1. INTRODUCTION

Advanced ICT solutions that can provide a truly customer-centric, one-stop-shop experience for multimodal travel across the Single European Transport Area must have the ability to coordinate the execution of complex computational tasks that are inherently distributed on multiple heterogeneous systems, or “nodes”, of an open network with no central control. In this light, systems are interoperable if they are capable of participating in such distributed computing tasks.

The Interoperability Framework is designed to insulate applications from the complexities of handling distributed computing. Based on the principles of Linked Data and the Semantic Web[1], the Semantic Query and Aggregation engine is the component of the Interoperability Framework that gives applications access to distributed resources by providing them with an abstraction, the “web of transport”, of data that spans the world wide web. Through the semantic query and aggregation engine applications can retrieve the data resources they need to perform sophisticated computational tasks no matter where they may be located or how they may be structured.

This eliminates a powerful obstacle to interoperability that exists in conventional approaches, such as the need to pre-determine shared database schemas and transfer large data sets from pre-determined sources before customer-relevant solutions can be produced. It also enables applications to leverage new sources and information to generate insights, discover opportunities and become actors in the data economy.

Three fundamental technologies contribute to the ability to query the “web of transport”:

1. An ontology, written in the Ontology Web Language (OWL)[2] provides an explicit logical formalisation of the data in terms of their meaning. This enable machines to “understand” the data whatever their format, and to create new derived data elements by means of machine logical inferencing.

2. The Resource Description Framework (RDF) provides a representation of data that “specifically supports the evolution of schemas over time without requiring all the data consumers to be changed”[3]. In provides, in addition, the ability to link data items across the world wide web.

3. The SPARQL query language for RDF[4] provides the ability to perform queries to retrieve, insert and delete RDF linked data across the world wide web.

This document describes the usage of the three W3C standard technologies above in the implementation of the Interoperability Framework’s Semantic Query and Aggregation Engine.

[1] Cfr. https://www.w3.org/standards/semanticweb/
[2] Cfr. https://www.w3.org/OWL/
[3] Cfr. https://www.w3.org/RDF/
[4] Cfr. https://www.w3.org/TR/rdf-sparql-query/

TABLE OF CONTENTS

LIST OF FIGURES

2. ONTOLOGY

2.1 Rationale

Formats are a syntax used to provide a *representation* of certain facts, event and their relationships for machines to process. The processing of this data depends, however, on a correct *interpretation* of what facts, events or relationships are involved in the application of some technical or business logic. Since the same fact can be represented in different syntaxes the problem of interoperability may be considered as one of providing machines with some means to perform this interpretation, i.e. to recognise that different data items are, in fact, different representations of the *same* fact, i.e. with a shared machine-readable description of *what* fact the datum represents and how it relates *formally* to other facts of the domain.

2.2 Usage in IT2Rail

An ontology provides an axiomatic model of some domain of interest expressed in a formal system that is amenable to automatic machine classification and inference processing. The axioms describe domain knowledge that is independent on actual data, i.e. something that holds true for every instance of any data, e.g. a RouteLink cannot connect one StopPlace with itself, an Airport cannot be the start or end StopPlace of a RailLink, or if two items of data represent the start or end StopPlace of the same RouteLink than they are different names for the same StopPlace.
When applied to actual data the reasoner can automatically determine logical consequences or proofs such as the following:

1. What they 'mean', e.g. whether they are instances of Airports or Vehicles.
2. What relationships exist between them, e.g. this datum represents a specific Flight that connect two specific Airports.
3. Whether they are different representations (e.g. 'codes', 'formats') of the same thing, and if they are, which is the common concept or property they represent.
4. Whether the data conform to the model, i.e. whether they are compliant with the axiomatic description of the domain's logic, providing furthermore for an 'explanation' of the constraints that may be violated.

Written in the "Ontology Web Language – Description Logic" (OWL-DL) and Semantic Web Rule Language (SWRL) language the axiomatic domain knowledge can additionally be serialised using standard mechanisms such as XML or JSON, and therefore shared across distributed machines and processed by them. One particular case of such a processing is the automation of mappings between schemas or 'data formats'. For example, where an OWL classes describe different but equivalent XSD ComplexTypes, or OWL ObjectProperties or DataProperties describe different but equivalent XSD Elements or Attributes, the mapping from one XSD schema to another (or to an equivalent JSON schema) can be automated.

Also, because OWL-DL items (classes, properties and instances or 'individuals') are uniquely identified by an IRI that can be deferenced, reasoning can produce results uniquely identified by such IRIs that are hyperlinks to additional items which can be accessed over the web to obtain additional data logically related to them. For example, the identification of an Airport can be a link to an external data set describing properties of its operating hours, connected metro lines or taxi stands, which can be accessed to enrich the data.

We illustrate the approach with a few rudimentary examples in the following subsections.

2.3 EXAMPLE DOMAIN AXIOMS

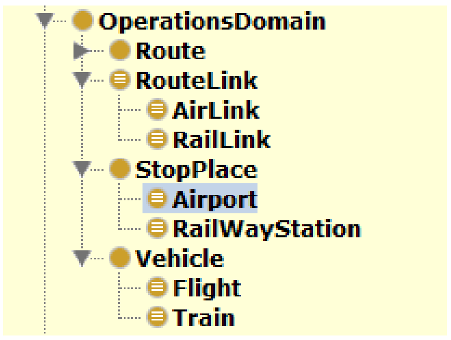
The example is based on the following axioms:

1. A Vehicle is a class of vehicles.
2. A Vehicle has a isTransportationMode property which can be either AIR or RAIL.
3. A Flight is a Vehicle whose isTransportationMode property is AIR.
4. A Train is a Vehicle whose isTransportationMode property is RAIL.
5. A StopPlace is a class of stop places.
6. A RouteLink is a class of objects with the following properties.
    a. startsAtPlace at least one StopPlace.
    b. endsAtPlace at least one StopPlace.
7. the startsAtPlace and endsAtPlace StopPlaces of a single RouteLink must be distinct.
8. If a RouteLink has startsAtPlace (or endAtPlace) properties with the same value, the values represent the same StopPlace instance (i.e. startsAtPlace and endsAtPlace are functional properties).
9. A Vehicle operates some RouteLink.
10. An AirLink is a RouteLink operated by a Flight.
11. A RailLink is a RouteLink operated by Train.
12. An Airport is a StopPlace at which an AirLink either starts or ends.
13. A RailWayStation is a StopPlace at which a RailLink either starts or ends.
14. An Airport and a RailWayStation are disjoint classes.
15. If a StopPlace is the start (or end) place of a RouteLink operated by a Vehicle whose isTransportMode property is AIR then the StopPlace has a stopPlaceMode of AIR.
16. If a StopPlace is the start (or end) place of a RouteLink operated by a Vehicle whose isTransportMode property is RAIL then the StopPlace has a stopPlaceMode of RAIL.

Given those axioms, i.e. a model of the domain, we now feed the system sample data representing StopPlaces, Vehicles, and Objects of unknown nature associated with Vehicles and StopPlaces.

2.4 Inferred Taxonomy

The reasoner automatically builds an inferred taxonomy of the classes, describing AirLink and RailLink as subclasses of RouteLink, and Airport and RailWayStation as subclasses of StopPlaces:

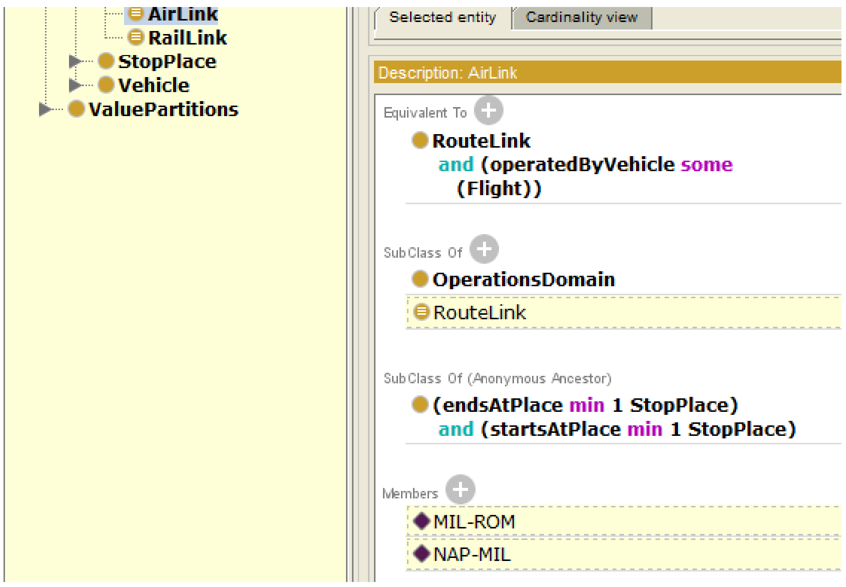


```
OperationsDomain
    Route
    RouteLink
        AirLink
        RailLink
    StopPlace
        Airport
        RailWayStation
    Vehicle
        Flight
        Train
```

Figure 1 - Inferred Taxonomy example (classes)

2.5 Inferred Classification of Instances

The reasoner automatically determines the type of data instances:



```
        AirLink
        RailLink
    StopPlace
    Vehicle
ValuePartitions
```

Selected entity | Cardinality view

Description: AirLink

Equivalent To
RouteLink
 and (operatedByVehicle some
  (Flight))

SubClass Of
OperationsDomain
RouteLink

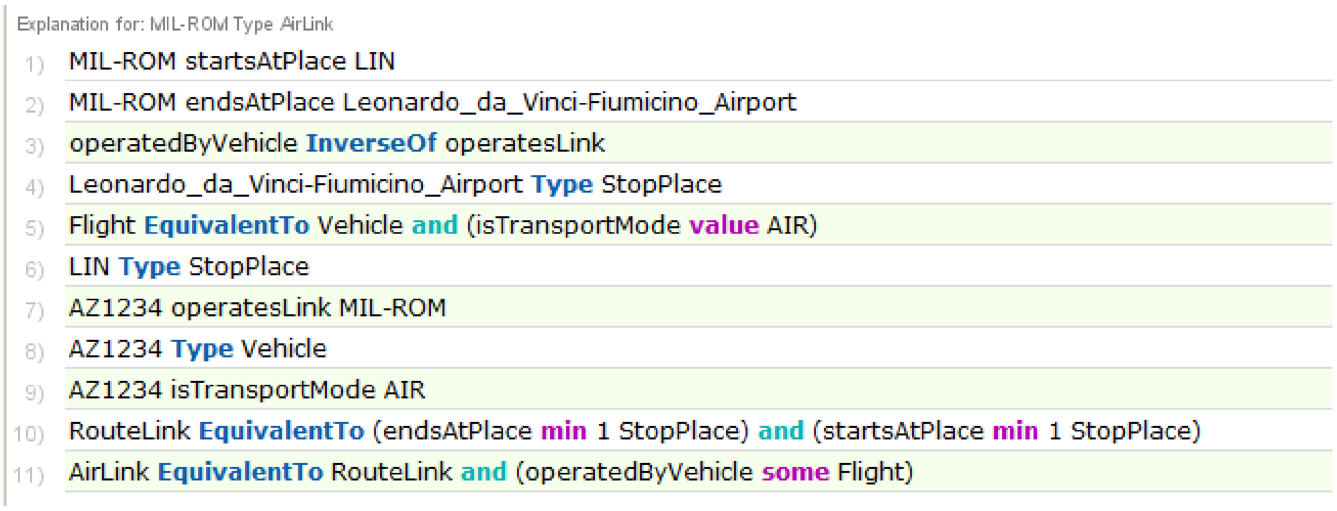
SubClass Of (Anonymous Ancestor)
(endsAtPlace min 1 StopPlace)
 and (startsAtPlace min 1 StopPlace)

Members
MIL-ROM
NAP-MIL

Figure 2 - Inferred instances example

MIL-ROM and NAP-MIL are both AirLinks, which is a subclass of RouteLink.

For example, MIL-ROM is determined to be an AirLink through the application of the axioms as follows:

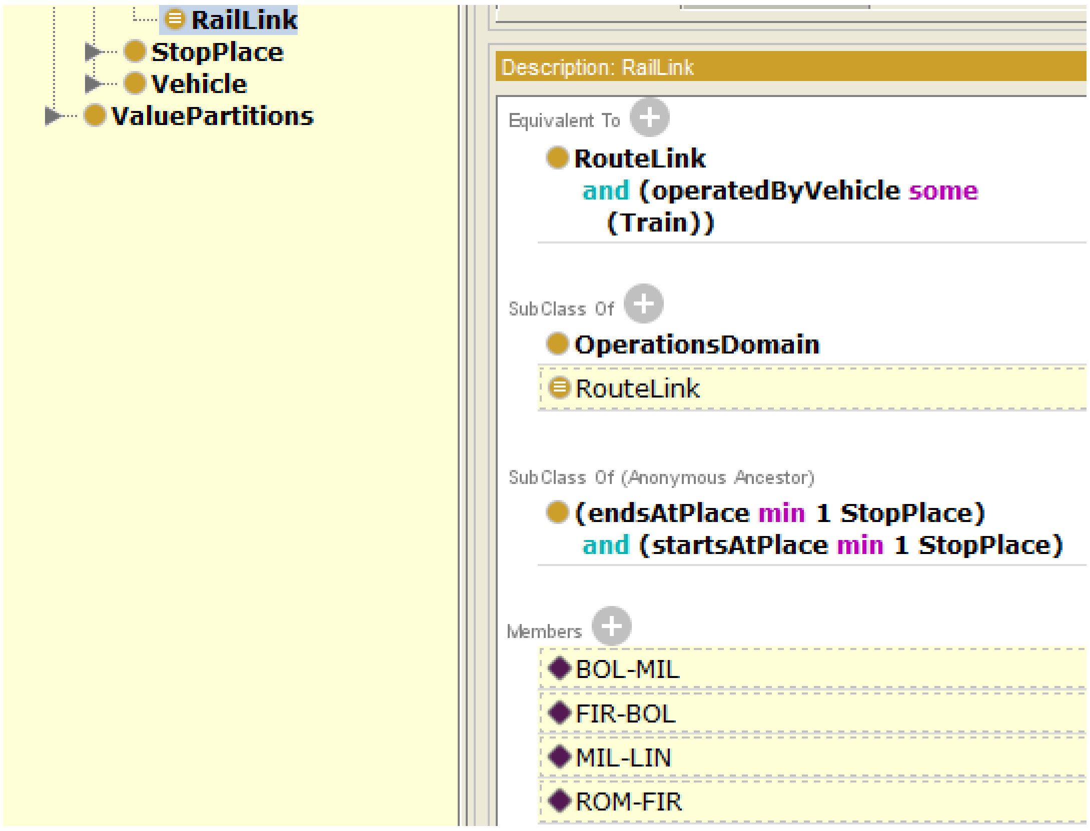


Explanation for: MIL-ROM Type AirLink

1) MIL-ROM startsAtPlace LIN
2) MIL-ROM endsAtPlace Leonardo_da_Vinci-Fiumicino_Airport
3) operatedByVehicle **InverseOf** operatesLink
4) Leonardo_da_Vinci-Fiumicino_Airport **Type** StopPlace
5) Flight **EquivalentTo** Vehicle **and** (isTransportMode **value** AIR)
6) LIN **Type** StopPlace
7) AZ1234 operatesLink MIL-ROM
8) AZ1234 **Type** Vehicle
9) AZ1234 isTransportMode AIR
10) RouteLink **EquivalentTo** (endsAtPlace **min** 1 StopPlace) **and** (startsAtPlace **min** 1 StopPlace)
11) AirLink **EquivalentTo** RouteLink **and** (operatedByVehicle **some** Flight)

Figure 3 - Automatic Inference operations

i.e. it starts and ends at LIN and Leonardo_da_Vinci-Fiumicino_Airport (lines 1 and 2), which are StopPlaces (lines 4 and 6): it is therefore a RouteLink (line 10). AZ1234 is a Vehicle (line 8) which isTransportMode AIR (line 9), which makes it a Flight (line 5), and operates this RouteLink (lines 3 and 7): the RouteLink MIL-ROM is therefore an AirLink (line 11).
RailLinks are similarly classified:



RailLink
StopPlace
Vehicle
ValuePartitions

Description: RailLink

Equivalent To
RouteLink
and (operatedByVehicle **some** (Train))

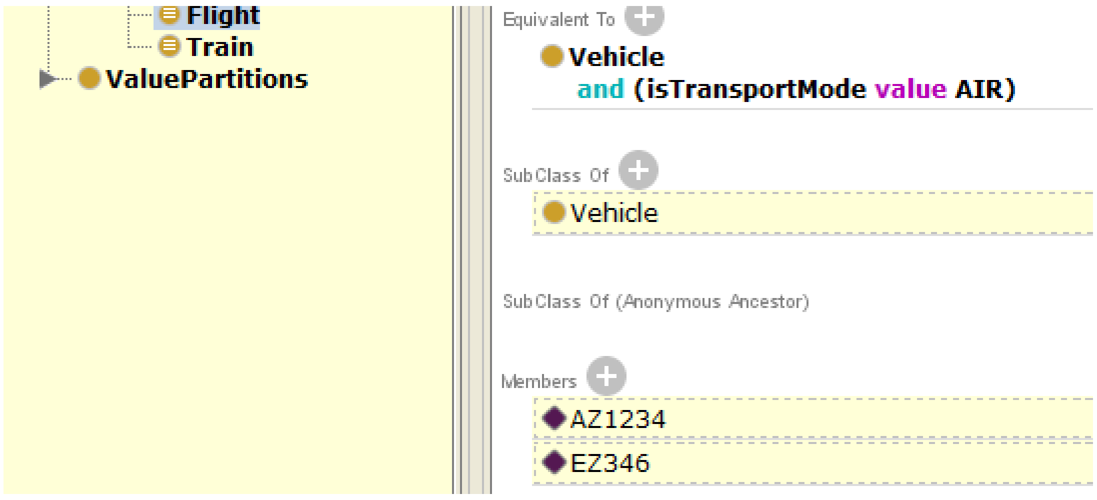
SubClass Of
OperationsDomain
RouteLink

SubClass Of (Anonymous Ancestor)
(endsAtPlace **min** 1 StopPlace)
and (startsAtPlace **min** 1 StopPlace)

Members
BOL-MIL
FIR-BOL
MIL-LIN
ROM-FIR

Figure 4 Inferred Route Links

Vehicles are similarly classified as Trains or Flights:



Flight
Train
ValuePartitions

Equivalent To

Vehicle
and (isTransportMode value AIR)

SubClass Of

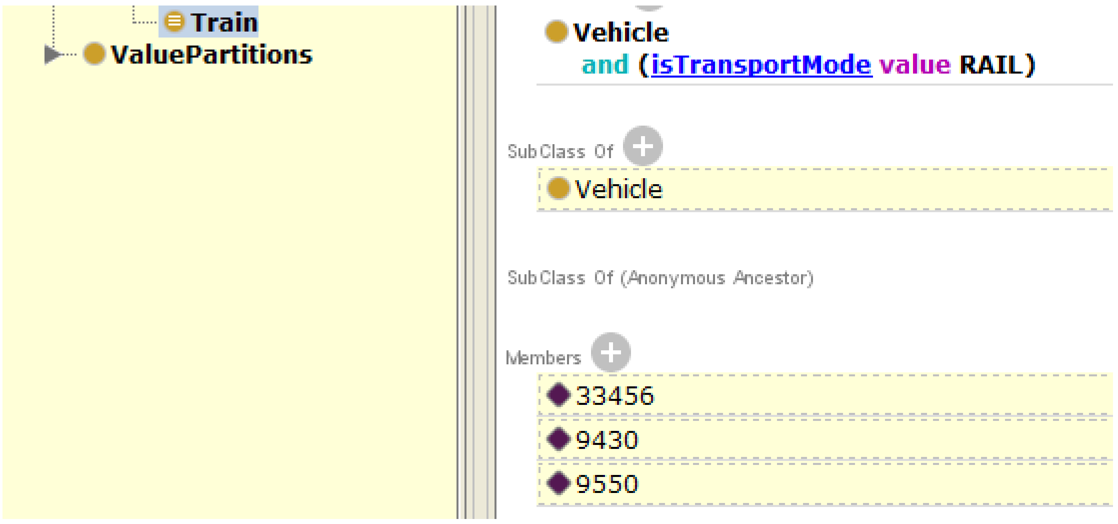
Vehicle

SubClass Of (Anonymous Ancestor)

Members

AZ1234
EZ346

Figure 5 - Inferred Vehicles (Flights)



Train
ValuePartitions

Vehicle
and (isTransportMode value RAIL)

SubClass Of

Vehicle

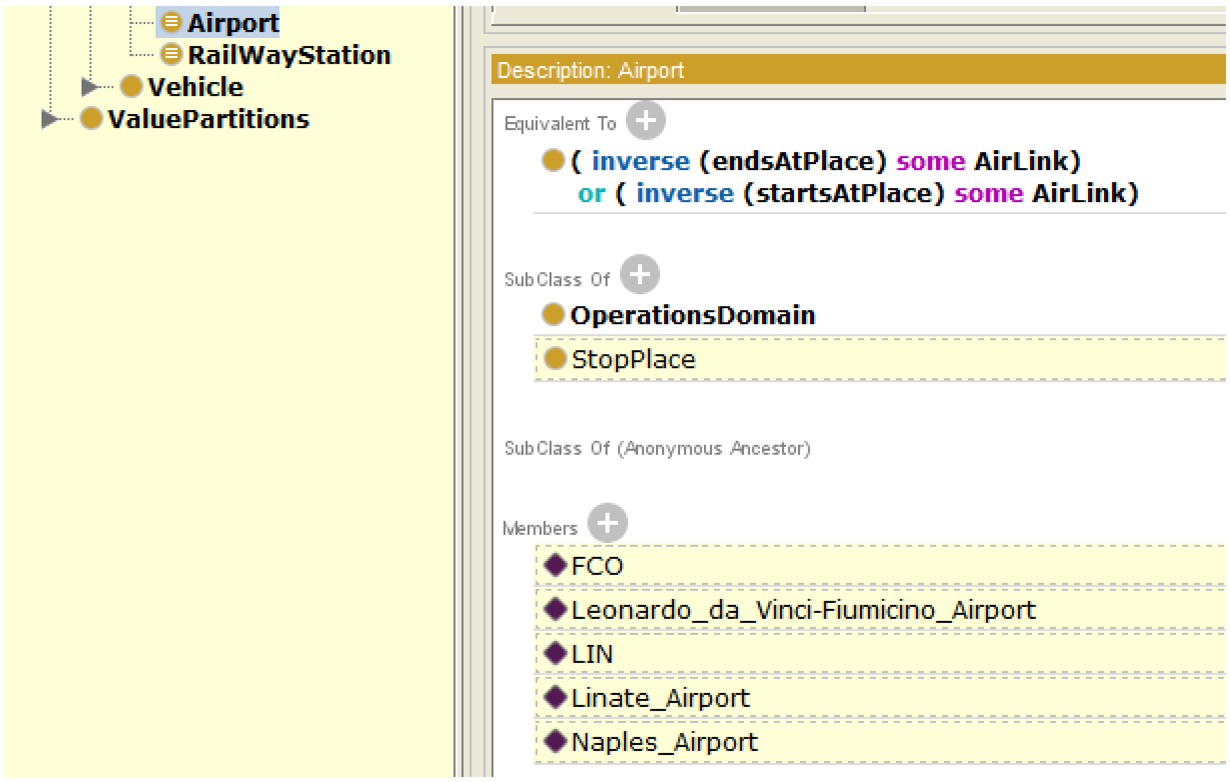
SubClass Of (Anonymous Ancestor)

Members

33456
9430
9550

Figure 6 Inferred Vehicles (Trains)

Stop Places are classified as Airports or RailWayStations:


Airport
RailWayStation
Vehicle
ValuePartitions

Description: Airport

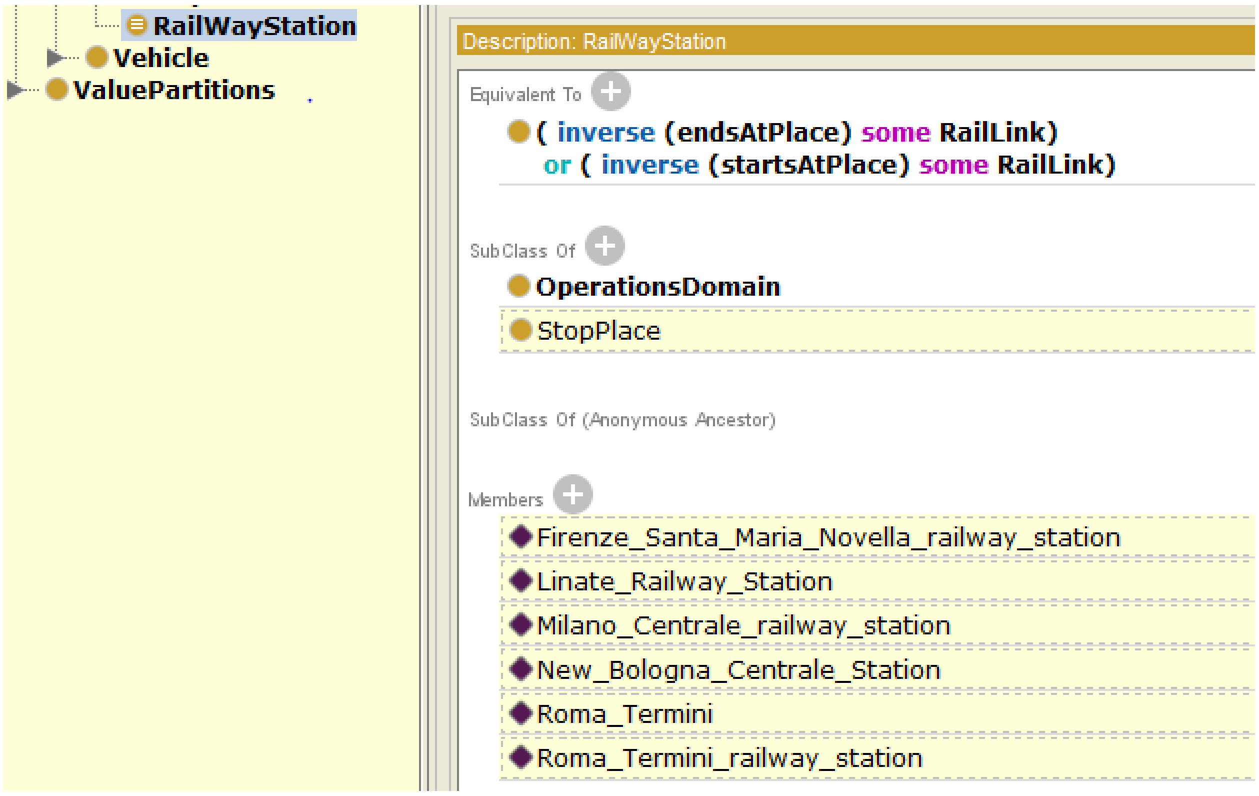
Equivalent To
(inverse (endsAtPlace) some AirLink)
or (inverse (startsAtPlace) some AirLink)

SubClass Of
OperationsDomain
StopPlace

SubClass Of (Anonymous Ancestor)

Members
FCO
Leonardo_da_Vinci-Fiumicino_Airport
LIN
Linate_Airport
Naples_Airport

Figure 7 Inferred Stop Places (Airports)


RailWayStation
Vehicle
ValuePartitions

Description: RailWayStation

Equivalent To
(inverse (endsAtPlace) some RailLink)
or (inverse (startsAtPlace) some RailLink)

SubClass Of
OperationsDomain
StopPlace

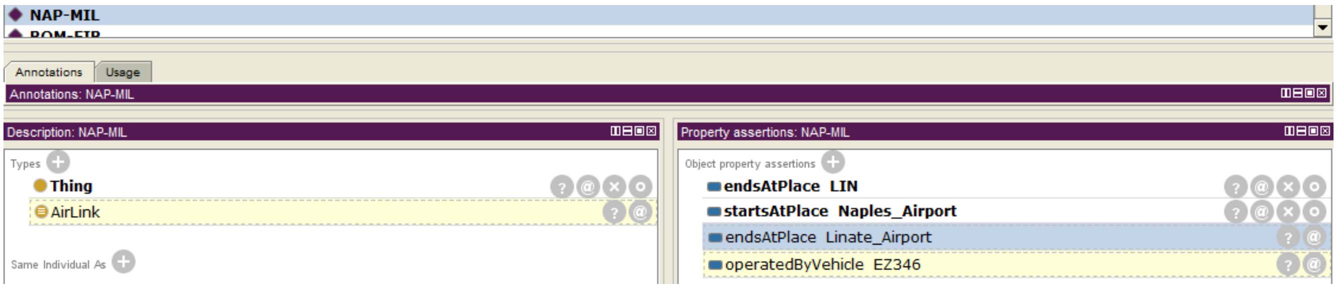
SubClass Of (Anonymous Ancestor)

Members
Firenze_Santa_Maria_Novella_railway_station
Linate_Railway_Station
Milano_Centrale_railway_station
New_Bologna_Centrale_Station
Roma_Termini
Roma_Termini_railway_station

Figure 8 Inferred Stop Places (Rail Stations)

2.6 Inferring Equivalent Instances (Representations) of a Common Concept

Given the definition of an instance (NAP-MIL) as something that startsAtPlace Naples_Airport and endAtPlace LIN (both StopPlace(s)), the reasoner infers that it is an AirLink and that LIN and Linate_Airport are the same instance of the StopPlace at which the AirLink ends:



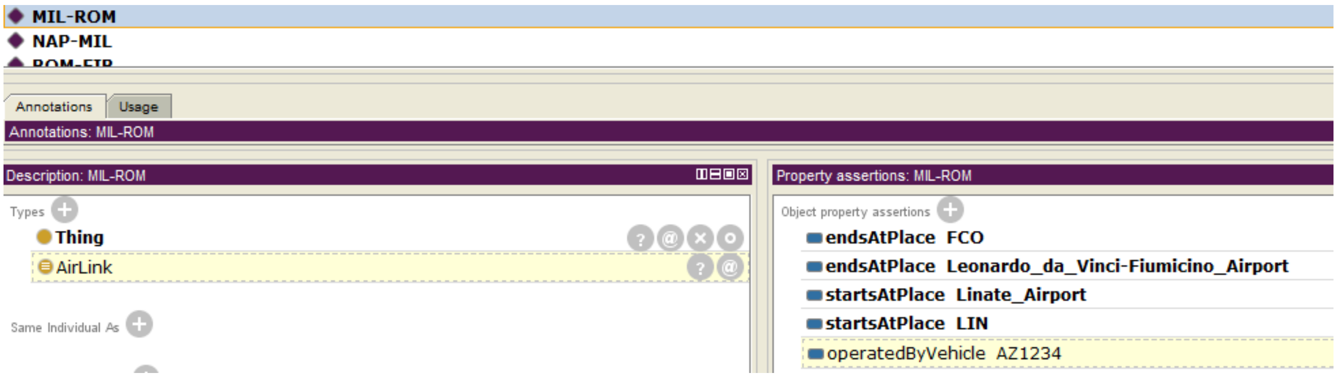
NAP-MIL
ROM-FIR
Annotations | Usage
Annotations: NAP-MIL

Description: NAP-MIL
Types
Thing
AirLink
Same Individual As

Property assertions: NAP-MIL
Object property assertions
endsAtPlace LIN
startsAtPlace Naples_Airport
endsAtPlace Linate_Airport
operatedByVehicle EZ346

Figure 9 Inferred Equivalent Instance of Airport (Linate)

This is a result of LIN and Linate_Airport having been both defined as end Stop Places of the same MIL-ROM object, which is an AirLink:



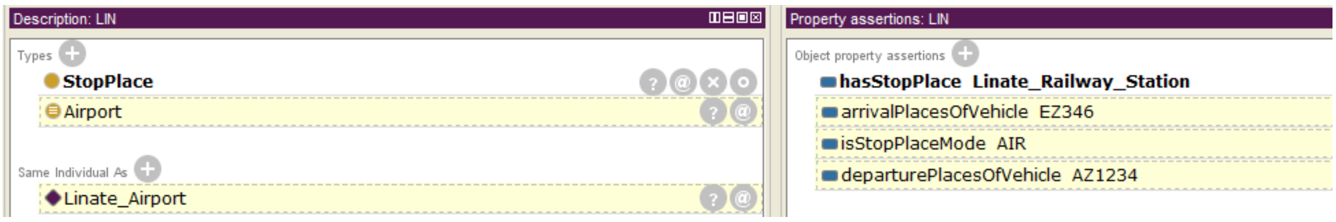
MIL-ROM
NAP-MIL
ROM-FIR
Annotations | Usage
Annotations: MIL-ROM

Description: MIL-ROM
Types
Thing
AirLink
Same Individual As

Property assertions: MIL-ROM
Object property assertions
endsAtPlace FCO
endsAtPlace Leonardo_da_Vinci-Fiumicino_Airport
startsAtPlace Linate_Airport
startsAtPlace LIN
operatedByVehicle AZ1234

Figure 10 Inferred Instance of Airport (Fiumicino)

In fact, FCO and Leonardo_da_Vinci-Fiumicino_Airport are also defined as start StopPlace for the same AirLink. Since the startsAtPlace and endsAtPlace relationships are defined as functional in the model, the system determines that the instances represent the same Stop Place (in addition to other inferred properties, such as its transportationMode, or the Vehicles that arrive or depart at this StopPlace.
Here's the eqivalence of LIN with Linate_Airport:

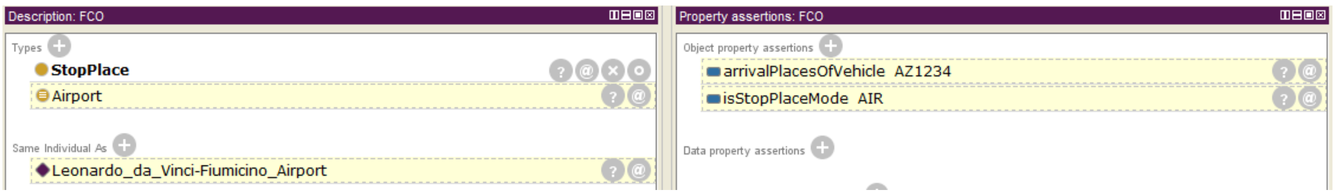


Description: LIN
Types
StopPlace
Airport
Same Individual As
Linate_Airport

Property assertions: LIN
Object property assertions
hasStopPlace Linate_Railway_Station
arrivalPlacesOfVehicle EZ346
isStopPlaceMode AIR
departurePlacesOfVehicle AZ1234

Figure 11 – Inferred equivalentce of LIN and Linate_Airport

Here is the equivalence of FCO with Leonardo_da_Vinci-Fiumicino_Airport:


Description: FCO
Types
StopPlace
Airport
Same Individual As
Leonardo_da_Vinci-Fiumicino_Airport

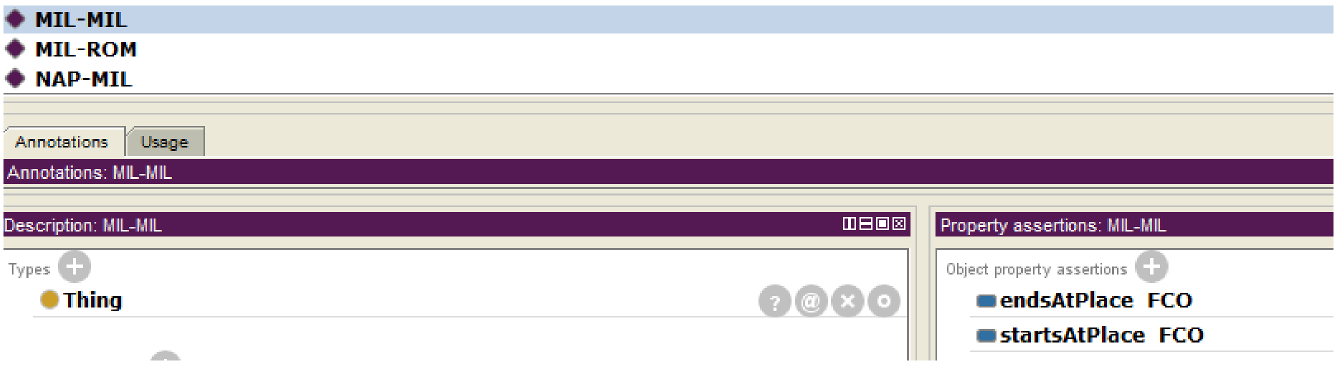
Property assertions: FCO
Object property assertions
arrivalPlacesOfVehicle AZ1234
isStopPlaceMode AIR
Data property assertions

Figure 12 – Inferred equivalence of FCO and Leonardo_da_Vinci-Fiumicino_Airport

The system is therefore able to determine automatically the equivalence of two different ‘codes’ for the same instance when they are in the range of a functional property just once in the entire knowledge base.

2.7 Checking for inconsistencies

We define (erroneously) an object that starts and ends at the same StopPlace:

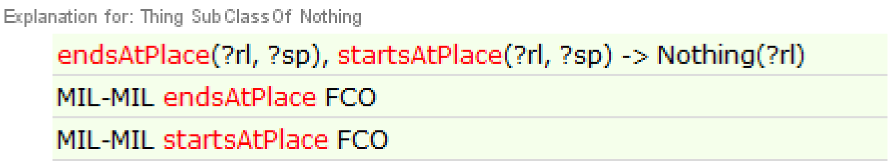

MIL-MIL
MIL-ROM
NAP-MIL

Annotations | Usage
Annotations: MIL-MIL

Description: MIL-MIL
Types
Thing

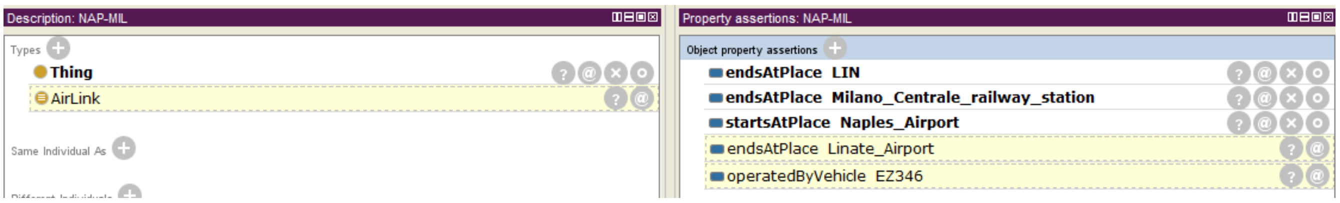
Property assertions: MIL-MIL
Object property assertions
endsAtPlace FCO
startsAtPlace FCO

The system verifies that this is not allowed in the model:


Explanation for: Thing SubClassOf Nothing
endsAtPlace(?rl, ?sp), startsAtPlace(?rl, ?sp) -> Nothing(?rl)
MIL-MIL endsAtPlace FCO
MIL-MIL startsAtPlace FCO

The constraint is expressed by a SWRL rule specifying that something (?rl) that starts and ends at the same StopPlace (?sp) is to be an instance of the (empty) class Nothing.
As an additional example, we specify that NAP-MIL endsAtPlace LIN and also endsAtPlace Milano_Centrale_raiwlay_station. Since endsAtPlace is a functional property, we would expect that LIN and Milano_Centrale_railway_station are inferred to be the same instance However this would be an error, as LIN is an Airport and Milano_Centrale_railway_station is a RailWayStation. We check for the reasoner to report the inconsistency:

This is the wrong assertion:



Description: NAP-MIL

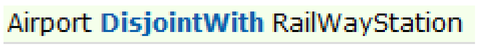
Types
- Thing
- AirLink

Same Individual As

Property assertions: NAP-MIL

Object property assertions
- endsAtPlace LIN
- endsAtPlace Milano_Centrale_railway_station
- startsAtPlace Naples_Airport
- endsAtPlace Linate_Airport
- operatedByVehicle EZ346

The reasoner detects the inconsistency:
endsAtPlace is functional property, therefore LIN (an Airport) and Milano_Centrale_railway_station (a RailWayStation) are initially considered identical instances. However, the axiom:



Airport **DisjointWith** RailWayStation

is violated and the inconsistency is detected.

3. RDF AND LINKING TO EXTERNAL DATA SOURCES

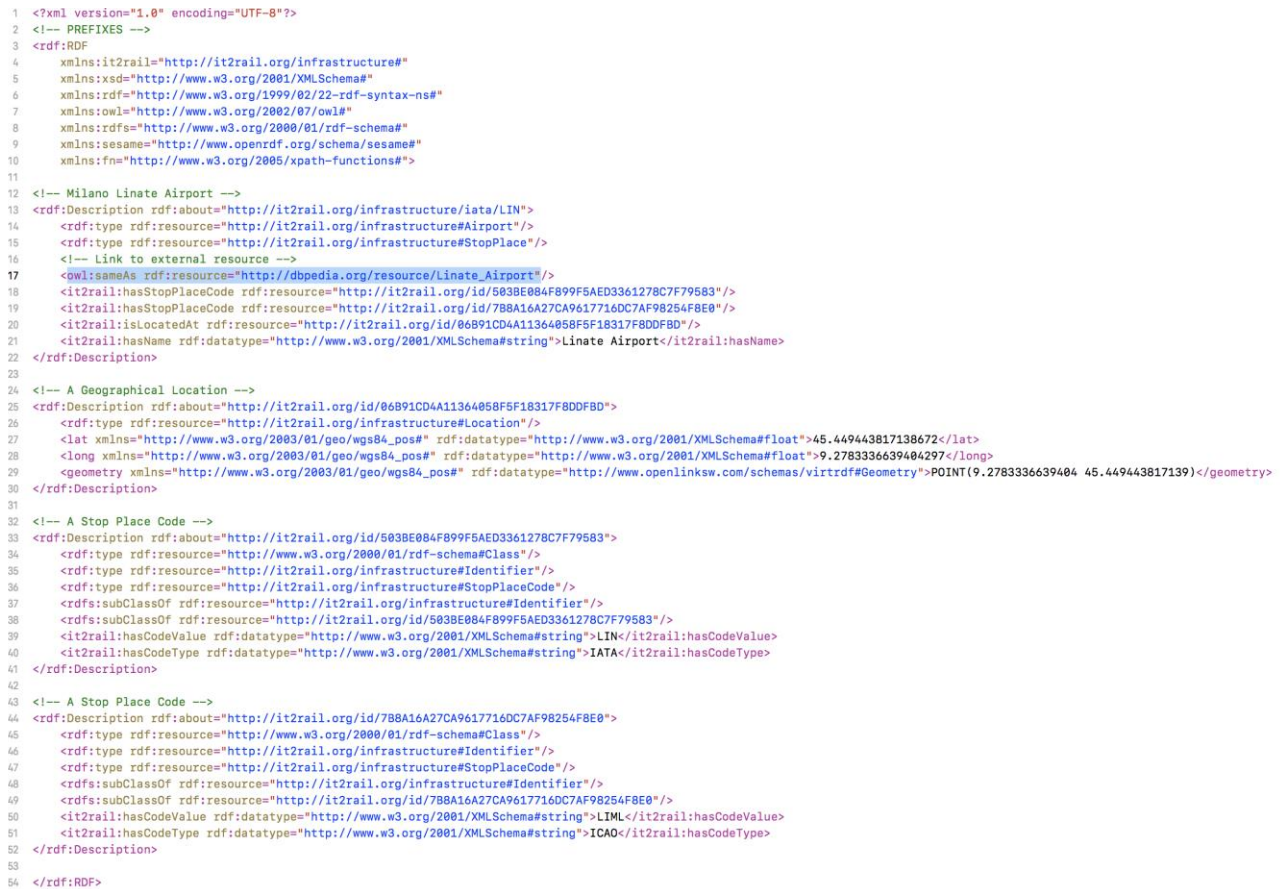
The Interoperability Framework leverages the following feature of the Resource Definition Framework (RDF) to create, maintain and query the “web of transport” abstraction for IT2Rail project applications:

“*RDF extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a “triple”). Using this simple model, it allows structured and semi-structured data to be mixed, exposed, and shared across different applications.*

This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand visual explanations”[5].

The following figure provides the RDF representation of the Linate Airport in Milan:

[5] https://www.w3.org/RDF/



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- PREFIXES -->
<rdf:RDF
    xmlns:it2rail="http://it2rail.org/infrastructure#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:sesame="http://www.openrdf.org/schema/sesame#"
    xmlns:fn="http://www.w3.org/2005/xpath-functions#">

<!-- Milano Linate Airport -->
<rdf:Description rdf:about="http://it2rail.org/infrastructure/iata/LIN">
    <rdf:type rdf:resource="http://it2rail.org/infrastructure#Airport"/>
    <rdf:type rdf:resource="http://it2rail.org/infrastructure#StopPlace"/>
    <!-- Link to external resource -->
    <owl:sameAs rdf:resource="http://dbpedia.org/resource/Linate_Airport"/>
    <it2rail:hasStopPlaceCode rdf:resource="http://it2rail.org/id/503BE084F899F5AED3361278C7F79583"/>
    <it2rail:hasStopPlaceCode rdf:resource="http://it2rail.org/id/7B8A16A27CA9617716DC7AF98254F8E0"/>
    <it2rail:isLocatedAt rdf:resource="http://it2rail.org/id/06B91CD4A11364058F5F18317F8DDFBD"/>
    <it2rail:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Linate Airport</it2rail:hasName>
</rdf:Description>

<!-- A Geographical Location -->
<rdf:Description rdf:about="http://it2rail.org/id/06B91CD4A11364058F5F18317F8DDFBD">
    <rdf:type rdf:resource="http://it2rail.org/infrastructure#Location"/>
    <lat xmlns="http://www.w3.org/2003/01/geo/wgs84_pos#" rdf:datatype="http://www.w3.org/2001/XMLSchema#float">45.449443817138672</lat>
    <long xmlns="http://www.w3.org/2003/01/geo/wgs84_pos#" rdf:datatype="http://www.w3.org/2001/XMLSchema#float">9.2783336639404297</long>
    <geometry xmlns="http://www.w3.org/2003/01/geo/wgs84_pos#" rdf:datatype="http://www.openlinksw.com/schemas/virtrdf#Geometry">POINT(9.2783336639404 45.449443817139)</geometry>
</rdf:Description>

<!-- A Stop Place Code -->
<rdf:Description rdf:about="http://it2rail.org/id/503BE084F899F5AED3361278C7F79583">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    <rdf:type rdf:resource="http://it2rail.org/infrastructure#Identifier"/>
    <rdf:type rdf:resource="http://it2rail.org/infrastructure#StopPlaceCode"/>
    <rdfs:subClassOf rdf:resource="http://it2rail.org/infrastructure#Identifier"/>
    <rdfs:subClassOf rdf:resource="http://it2rail.org/id/503BE084F899F5AED3361278C7F79583"/>
    <it2rail:hasCodeValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string">LIN</it2rail:hasCodeValue>
    <it2rail:hasCodeType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">IATA</it2rail:hasCodeType>
</rdf:Description>

<!-- A Stop Place Code -->
<rdf:Description rdf:about="http://it2rail.org/id/7B8A16A27CA9617716DC7AF98254F8E0">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    <rdf:type rdf:resource="http://it2rail.org/infrastructure#Identifier"/>
    <rdf:type rdf:resource="http://it2rail.org/infrastructure#StopPlaceCode"/>
    <rdfs:subClassOf rdf:resource="http://it2rail.org/infrastructure#Identifier"/>
    <rdfs:subClassOf rdf:resource="http://it2rail.org/id/7B8A16A27CA9617716DC7AF98254F8E0"/>
    <it2rail:hasCodeValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string">LIML</it2rail:hasCodeValue>
    <it2rail:hasCodeType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">ICAO</it2rail:hasCodeType>
</rdf:Description>

</rdf:RDF>
```

Figure 13 - RDF representation of Linate Airport in local It2Rail graph

The RDF fragment above describes a portion of the local IT2Rail semantic graph with four nodes:

- The Milano Linate Airport node itself, which references
  - A geographical Location node through property isLocatedAt
  - Two Stop Place Code nodes through two instances of property hasStopPlaceCode

However, it also contains a reference to an external node, i.e. http://dbpedia.org/resource/Linate_Airport through the property owl:sameAs. The property owl:sameAs is a logical identity operator understood by the reasoner to signify that the two URIs http://it2rail.org/infrastructure/iata/LIN in the local graph and http://dbpedia.org/resource/Linate_Airport on the world wide web identify the same node instance. RDF data associated with http://dbpedia.org/resource/Linate_Airport provides therefore additional details on the resource that is accessible by a machine retrieving the local graph node identified by http://it2rail.org/infrastructure/iata/LIN: the two sets of RDF data located at different machines enrich each other so that both become part of the “web of transport” data.

4. SEMANTIC QUERY AND AGGREGATION

Summing up the discussion in the preceding chapters 2 and 3, the “web of transport” abstraction offered to IT2Rail applications is realised through the following mechanisms:

- The “meaning” of data is captured as formal and explicit axiomatic description of “knowledge” about the application domain known as the domain’s “ontology”. The ontology is written in the machine-readable standard language OWL.
- Facts about the world, i.e. “data”, are described as a linked graph in the standard RDF representation, i.e. as a set of “triples” in which objects are identified by an URI that constitutes a link across the world wide web.
- The application to RDF data of machine reasoning based on the ontology results in the production of additional triples in the graph representing extra data that are logical consequences of the known facts. For example, a certain datum may be found to be an instance of an Airport, and this becomes an extra data element in the graph associated with the original datum

As a result an RDF graph spanning the world wide web is constructed which acts as the shared data base for all IT2Rail applications. Semantic Query and Aggregation provides the means to access the data base and to package results for use in the applications.

To this end the standard the SPARQL query language for RDF[6] is used, which is a fundamental element of the semantic web toolset.

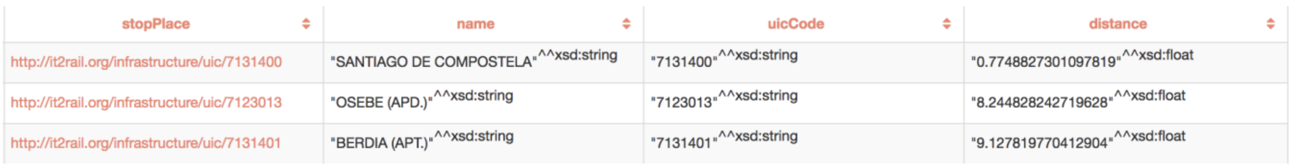
The following figure shows a SPARQL query performed on the IT2Rail “web of transport” data.

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX gdb-geo: <http://www.ontotext.com/owlim/geo#>
PREFIX dbo: <http://dbpedia.org/ontology/>
prefix it2rail: <http://it2rail.org/infrastructure#>
PREFIX omgeo:   <http://www.ontotext.com/owlim/geo#>

SELECT DISTINCT ?stopPlace ?name ?uicCode ?distance

    WHERE {

    service <http://lod.openlinksw.com/sparql>

    { dbr:Santiago_de_Compostela geo-pos:lat ?citylat ; geo-pos:long ?citylong }

    ?stopPlace a it2rail:RailStation;
                it2rail:hasName ?name;
                it2rail:isLocatedAt/geo-pos:lat ?placelat;
                it2rail:isLocatedAt/geo-pos:long ?placelong;
                it2rail:hasStopPlaceCode/it2rail:hasCodeValue ?uicCode

    bind(omgeo:distance(?citylat, ?citylong, ?placelat, ?placelong) as ?distance)
    filter(?distance < 10)

} ORDER BY ?distance
```

Figure 14 - SPARQL Query Example 1

[6] https://www.w3.org/TR/rdf-sparql-query/

The System is instructed to retrieve from an extended graph comprised of rdf data stored in the local It2rail repository and the linked open data cloud[7] reachable at endpoint http://lod.openlinksw.com/sparql all Rail Stations located within a 10 Km radius from the geographical coordinates of Santiago de Compostela, ordered by increasing distance. For each Rail Station in the result set the station's URI, its name, it code and the distance from the identified point. The city's geographical coordinates are retrieved from the remote graph and used to identify the Rail Stations stored in the local graph. The query generates the following results:



stopPlace	name	uicCode	distance
http://it2rail.org/infrastructure/uic/7131400	"SANTIAGO DE COMPOSTELA"^^xsd:string	"7131400"^^xsd:string	"0.7748827301097819"^^xsd:float
http://it2rail.org/infrastructure/uic/7123013	"OSEBE (APD.)"^^xsd:string	"7123013"^^xsd:string	"8.244828242719628"^^xsd:float
http://it2rail.org/infrastructure/uic/7131401	"BERDIA (APT.)"^^xsd:string	"7131401"^^xsd:string	"9.127819770412904"^^xsd:float

Figure 15 – SPARQL Sample Query Results 1

As a second example we describe a query that leverages the owl:sameAs property of the ontology:

```
PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX gdb-geo: <http://www.ontotext.com/owlim/geo#>
PREFIX dbo: <http://dbpedia.org/ontology/>
prefix it2rail: <http://it2rail.org/infrastructure#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?name ?facility  ?placelat ?placelong ?elevation
                  ?city ?country ?description
WHERE   {

     ?stopPlace a ?facility;
               it2rail:hasName ?name;
               it2rail:isLocatedAt/geo-pos:lat ?placelat;
               it2rail:isLocatedAt/geo-pos:long ?placelong;
               it2rail:hasStopPlaceCode/it2rail:hasCodeValue "LIN"^^xsd:string

        service <http://lod.openlinksw.com/sparql> {
               ?stopPlace  dbo:elevation ?dbelevation;
                           dbo:city ?city;
                           rdfs:comment ?description.

        }
     bind(xsd:float(?dbelevation) as ?elevation)
     filter(lang(?description)= 'fr')
}
```

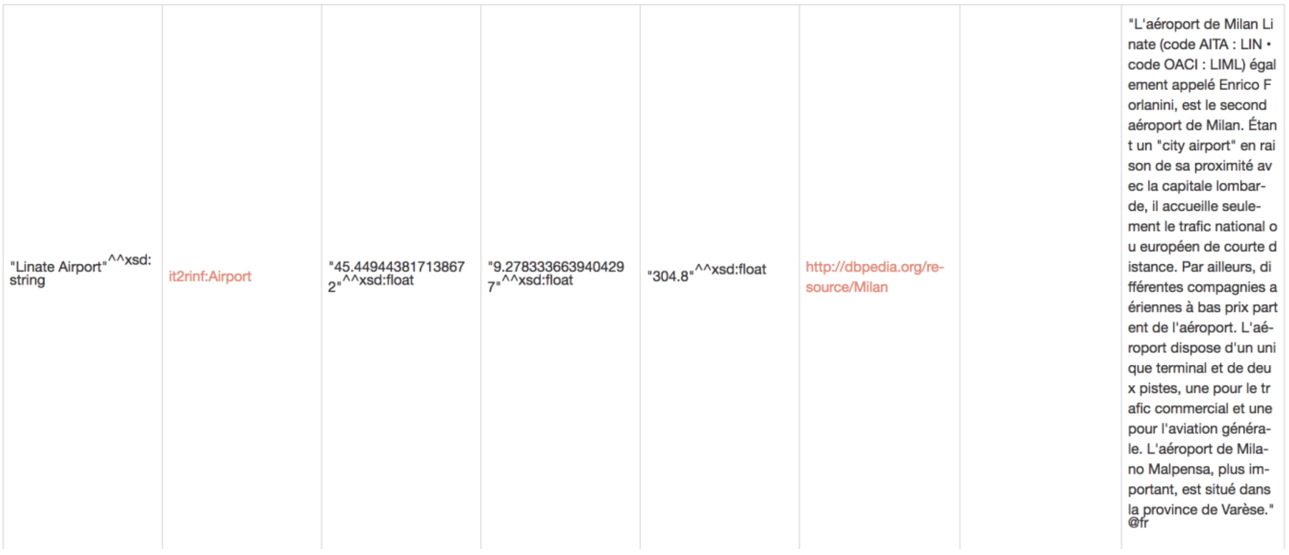
Figure 16 - SPARQL Query Example 2

The query retrieves the entity in the local rdf graph whose stop place code value is the string LIN. It also looks up the linked open data repository to get additional information on the same entity, namely its elevation expressed as a float data type, city and the description in French.

The results are aggregated from properties in the local graph, i.e. the entity's name, type of facility, latitude and longitude, and elevation, city and description from the remote graph.

[7] https://lod-cloud.net/

The following figure shows the result:



"Linate Airport"^^xsd:string	it2rinf:Airport	"45.449443817138672"^^xsd:float	"9.2783336639404297"^^xsd:float	"304.8"^^xsd:float	http://dbpedia.org/resource/Milan		"L'aéroport de Milan Linate (code AITA : LIN • code OACI : LIML) également appelé Enrico Forlanini, est le second aéroport de Milan. Étant un "city airport" en raison de sa proximité avec la capitale lombarde, il accueille seulement le trafic national ou européen de courte distance. Par ailleurs, différentes compagnies aériennes à bas prix partent de l'aéroport. L'aéroport dispose d'un unique terminal et de deux pistes, une pour le trafic commercial et une pour l'aviation générale. L'aéroport de Milano Malpensa, plus important, est situé dans la province de Varèse."@fr

Figure 17 - SPARQL Query Result 2

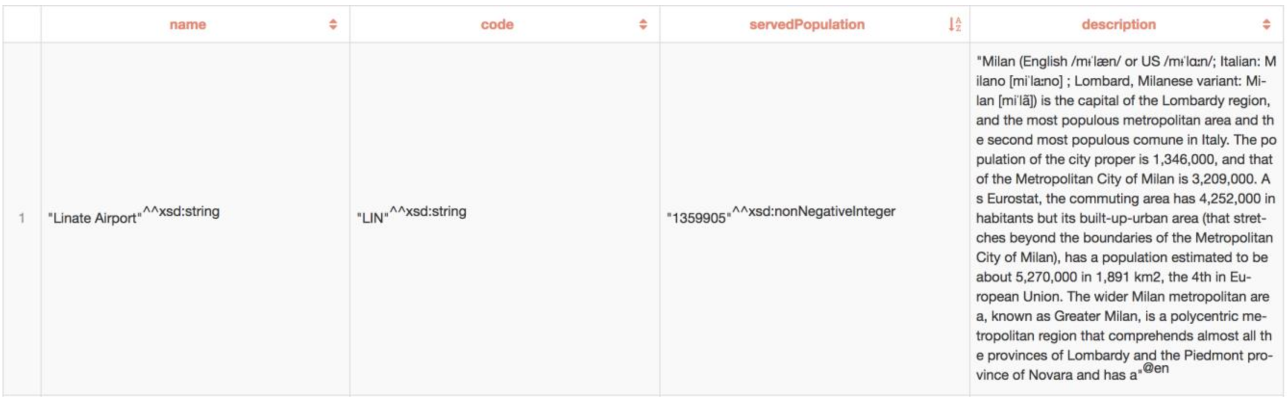
Because of the owl:sameAs property found in the graph for Linate Airport, as shown in Figure 13 above, the remote graph is queried for resource http://dbpedia.org/resource/Linate_Airport by which http://it2rail.org/infrastructure/iata/LIN is known in the local graph.

It should be noted also that the city result is itself a resource in the remote graph identified by the URI http://dbpedia.org/resource/Milan, and can therefore be used to retrieve additional information using the same capability. For example, the following query finds the Airport in the local graph that serves the city identified by postal code 20121–20162 in the remote graph, returning the description of the city with that postal code and the total residing population thus served by the airport:

```
PREFIX dbo: <http://dbpedia.org/ontology/>
prefix it2rail: <http://it2rail.org/infrastructure#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT   ?name ?code ?servedPopulation ?description
WHERE   {

        service <http://dbpedia.org/sparql> {
                ?stopPlace  dbo:city ?servedCity.
                ?servedCity dbo:populationTotal ?servedPopulation;
                            dbo:postalCode "20121–20162";
                            rdfs:comment ?description
        }
        ?stopPlace a it2rail:Airport;
                it2rail:hasName ?name;
                it2rail:hasStopPlaceCode/it2rail:hasCodeValue ?code
        filter(lang(?description) = 'en')

}
```

Figure 18 - SPARQL Query for Airport in area code for Milan

The following result is returned by the query:



	name	code	servedPopulation	description
1	"Linate Airport"^^xsd:string	"LIN"^^xsd:string	"1359905"^^xsd:nonNegativeInteger	"Milan (English /mɨˈlæn/ or US /mɨˈlɑːn/; Italian: Milano [miˈlaːno] ; Lombard, Milanese variant: Milan [miˈlã]) is the capital of the Lombardy region, and the most populous metropolitan area and the second most populous comune in Italy. The population of the city proper is 1,346,000, and that of the Metropolitan City of Milan is 3,209,000. As Eurostat, the commuting area has 4,252,000 inhabitants but its built-up-urban area (that stretches beyond the boundaries of the Metropolitan City of Milan), has a population estimated to be about 5,270,000 in 1,891 km2, the 4th in European Union. The wider Milan metropolitan area, known as Greater Milan, is a polycentric metropolitan region that comprehends almost all the provinces of Lombardy and the Piedmont province of Novara and has a"@en

Figure 19 – SPARQL Query Result for Airport in area code Milan

The important thing to note in these examples is that the sample queries are designed to respond to very different use cases but the data is not. They are in fact *semantic* queries, based on the data's *meaning*, not on the data's *structure*. That is what makes it possible to use data in a travel and transportation application even if it was not originally designed for it, e.g. from the linked open data (lod) graph. Also, the examples show that it is not necessary to obtain and transfer copies of these data, and that it is not, therefore, necessary to convert them to any specific format either for them to be used in a local application.

Since multiple use cases can be supported on the same semantic graph as specific SPARQL queries, a generic Semantic Query and Aggregation component capable of performing such queries has been developed as part of the Interoperability Framework. Different queries, all supported by the same engine, correspond to the different "packaged resolvers" delivered as services of the Interoperability Framework to IT2Rail applications.

5. SEMANTIC QUERY AND AGGREGATION ENGINE

As shown in the fragment in Figure 13 describing Linate Airport, semantic graphs described in RDF are actually a 'dialect' of XML that use a specific namespace and schema for rdf, indicated by the rdf: prefix and rdf:<tagname> tags. They could therefore in principle be exchanged and manipulated using standard tooling for XML. However, a number of frameworks have been developed to facilitate the development of applications in standard environments such as JAVA. Open source such frameworks are available[8] that additionally include the ability to handle secure communications and interactions with remote semantic graph stores ("triple stores") from multiple vendors provided they handle the standard SPARQL protocol. Of particular interest for the IT2Rail project is the fact that large graphs accessible through SPARQL exist already over the world wide web, as shown in the examples in chapter 4. In order to exploit the availability of these existing graphs the Semantic Query and Aggregation Engine design has been based on the following principles:

1. The Engine must be built on open source frameworks and be able to operate on any existing or future semantic graph under the sole condition that it be compliant with W3C semantic web standards.
2. The Engine must be independent of any implementation of a local or remote triple store, and must be able to interact to multiple such local or remote triples stores simultaneously.
3. The Engine must provide the ability to perform any SPARQL compliant query.
4. The Engine must be able to find and execute a SPARQL compliant query identified by name from a local or remote repository.
5. The Engine must be able to import the domain's ontology from a local or remote ontology repository.
6. The triple stores, ontology repository or SPARQL query repositories used by the Engine must be controlled by configuration.
7. The Engine must handle, if requested, automatic serialisation of Java classes to RDF and vice-versa, i.e. generate SPARQL queries from Java classes and return SPARQL results as Java classes.

Taken together, these principles must ensure that specific Interoperability Framework "packaged resolvers" such as Location Resolver or NeTEX producer, can be built to use the same Engine with a specific configuration, i.e. as services that only specialise the specific SPARQL queries and Triple Stores the use for their particular specialised task. As a consequence, the following additional principle must be applied in its design:

8. The Engine must be packaged as a Java Archive (JAR) file so that it can be included as a dependency in the creation of "packaged resolvers".

5.1 RDF FRAMEWORK

The IT2Rail RDF framework provides Java Persistence API Architecture (JPA)-like capabilities extending it with the ability to operate on triple stores, i.e. on database systems designed for storage and retrieval of RDF statements ("triples") through semantic queries.
Similarly, with how in JPA annotations on java classes provide automatic object-relational mappings from these classes to relational database schemas and vice-versa, the RDF framework provides automatic mappings to/from RDF statements. This enables the developer to delegate the mechanics of connection, input/output and storage/retrieval of the data across the network to the framework, while concentrating on the manipulation in pure java of objects and relationships which, unlike the JPA entities, represent logical statements connected to the domain's axiomatic description of the

[8] Cfr. RDF4J http://rdf4j.org/, JENA https://jena.apache.org/

domain's ontology. For example, triples generated by a SPARQL query as a result of logical inference, i.e. new logical statements inferred based on the axioms and the existing data, are automatically instantiated as java objects and relationships and are therefore available for "ordinary" programming.
The IT2Rail RDF framework is itself an extension and a merging of the two open source frameworks Empire (https://github.com/mhgrove/Empire) and Pinto (https://github.com/stardog-union/pinto), both licensed under the Apache License 2.0. The extensions consist in:

1. Porting to the Eclipse RDF4J framework (http://rdf4j.org/)
2. Merging of Empire and Pinto RDF mapping functionality
3. It2Rail-specific extensions, including multiple additional datatype conversions

The it2rail RDF framework for the FREL release is built and installed in the IT2Rail project's MAVEN repository for use in higher level development.

5.1.1 Empire Configuration

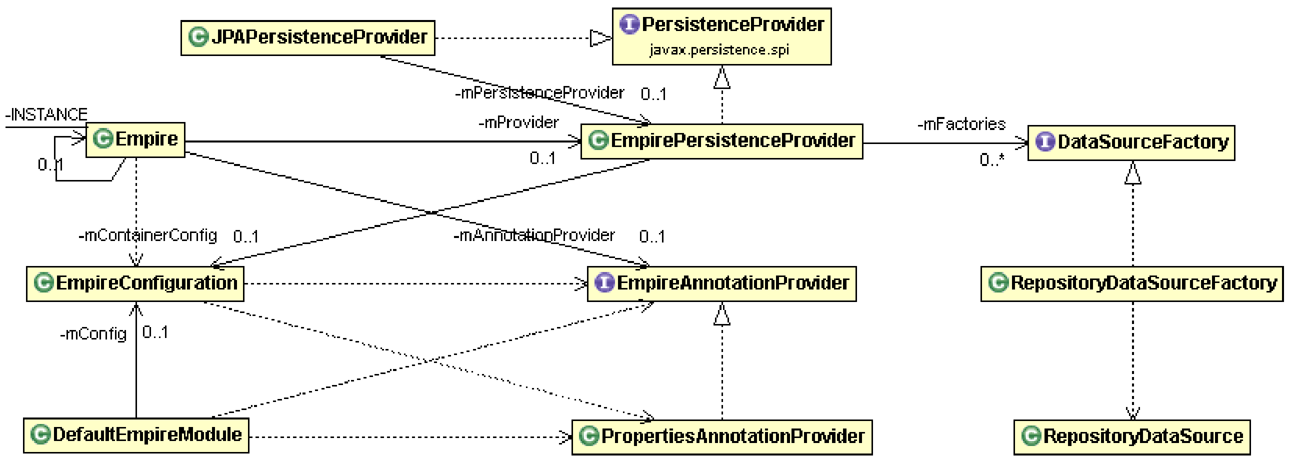


Figure 20 - RDF Framework configuration model

The Empire class is a container that holds the RDF framework's configuration instantiated at start up time. The configuration consists principally of three items:

1. The EmpireConfiguration containing a list of persistence unit descriptors and the name of a file of annotated java classes.
2. The Namespace and NamedQueries annotations used by the RdfGenerator and RdfQueryFactories classes, described below, to perform SPARQL queries and automatic mapping to/from RDF. This configuration is generated at initialisation time by an implementation of the EmpireAnnotationProvider interface.
3. The DataSourceFactory to be used by the EntityManager, described below, to create access to one or more triple store providers for storing and retrieving triples. This configuration is generated at initialisation time by an implementation of the JPA PersistenceProvider interface.

Concrete implementations of the EmpireAnnotationProvider and EmpirePersistenceProviders are injected at initialisation time through the Guice framework[9], i.e. by defining bindings in a Guice Module. The figure above displays the providers injected by default through the DefaultEmpireModule (not shown in the figure): EmpirePersistenceProvider implements the JPA PersistenceProvider while PropertiesAnnotationProvider implements EmpireAnnotationProvider.

[9] https://github.com/google/guice/wiki/ExternalDocumentation

5.1.2 Default Empire Configuration

The following default configuration is created at initialisation time by the call

```
Empire.init(new OpenRdfModule)
```

under the control of Guice modules:

1. Empire Configuration persistence unit descriptors are read from the configuration file whose name is associated with the System Property "empire.configuration.file". The configuration file must be accessible by the classLoader.
2. Namespace and NamedQueries are read through EmpireAnnotationProvider from the annotations of java classes listed in a file whose name is the value of "annotation.index" in the configuration file. The file listing classes and named queries must be accessible by the classLoader.
3. The EmpirePersistenceProvider implementation of the JPA PersistenceProvider is injected.
4. The concrete implementation RepositoryDataSourceFactory of the DataSourceFactory interface is injected from the binding specified in the Guice module OpenRdfModule.

The following is a sample empire configuration file with two persistence unit descriptors:

```
0.name=it2rail
0.factory= rdf4j
0.url =http://192.168.150.139:7200
0.repo=IT2RAIL

1.name=wikidata
1.factory = rdf4j
1.sparql_endpoint = https://query.wikidata.org/
```

5.1.3 Entity Manager

Once an Empire configuration is set up the NameSpace and NamedQueries annotations, the persistence unit descriptors and a concrete DataSourceFactory are available to create instantiate one or more Entity Managers used to perform storage and retrieval of RDF statements and conversions of java classes to/from RDF triples.
The IT2Rail RDF framework's Entity Manager is an implementation of the JPA EntityManager interface, i.e. it provides the same methods to persist, merge, remove and find java entities to/from triple stores as those provided for relational data bases.

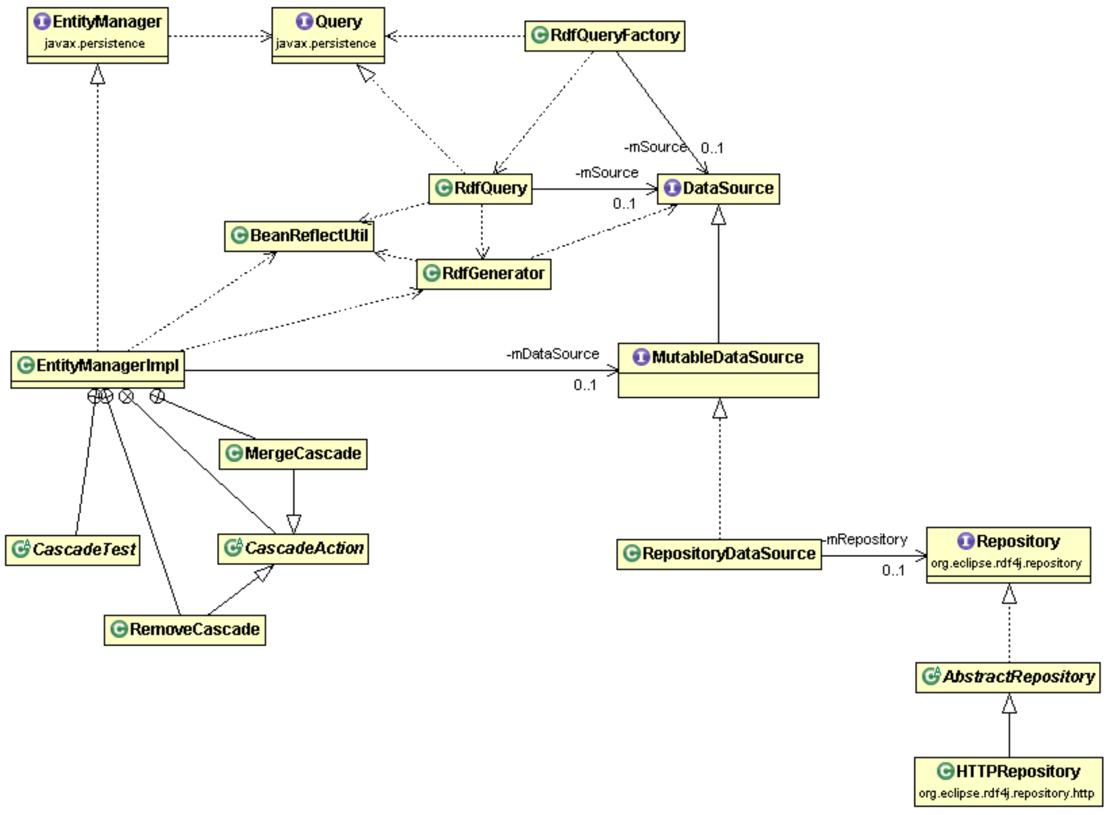


Figure 21 - RDF Framework Entity Manager

In the figure above the EntityManager, on the left, handles the usual JPA operations, including managing cascading as determined by normal JPA CascadeType values. Objects are persisted/retrieved from the concrete DataSource implementation created by the RepositoryDataSourceFactory injected at Empire configuration time. In the figure, to the right, the DataSource interface implementation created by the RepositoryDataSourceFactory is the class RepositoryDataSource, which itself has a reference to a HTTPRepository of the Eclipse RDF4J framework to access a remote triple store. All persistence operated commanded by the EntityManager are executed as method invocations on the actual interface to the remote triple store. Between the Entity Manager and the Repository the RDFGenerator performs the serialisation of java objects to/from RDF triples.
In analogy with the JPA API Architecture, which provides capabilities to generate custom SQL queries from java classes, the IT2rail RDF framework includes the RdfQueryFactory utility class to generate and validate SPARQL queries against the target triple store concrete Repository interface implementation.
An Entity Manager is instantiated through the call

```
EntityManager aManager = Persistence.createEntityManagerFactory(persistenceUnit)
                                                                  .createEntityManager();
```

Where `persistenceUnit` is the name of a persistence unit descriptor created at Empire configuration time, i.e. “it2rail” or “wikidata” in the example configuration file shown above.

5.1.4 RDF Generator

The RDFGenerator provides utility classes and methods to execute serialisation of java classes to/from RDF triples using annotations on the class.

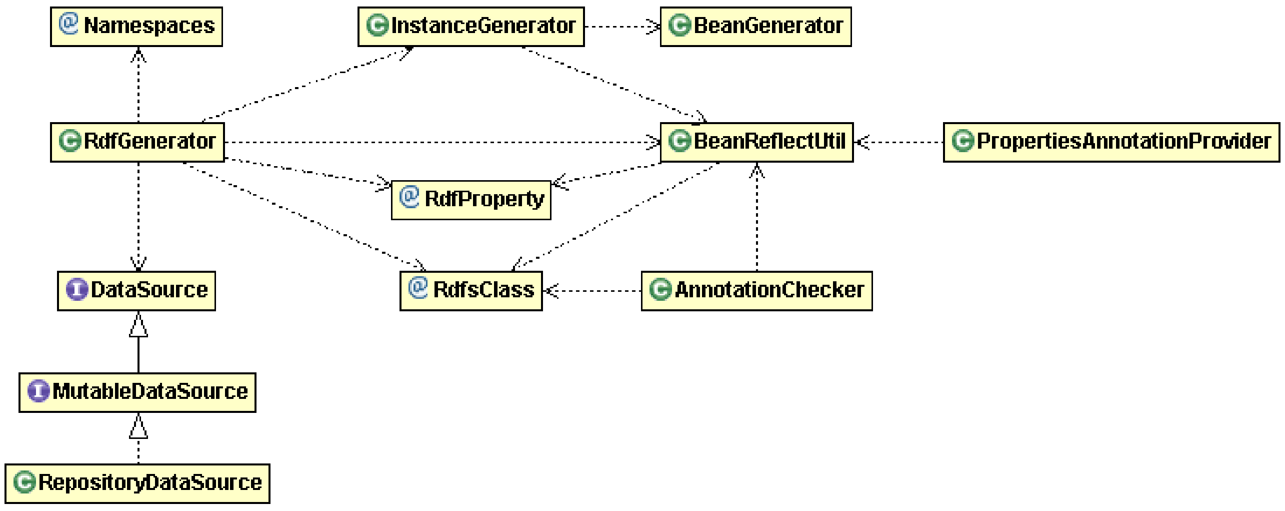


Figure 22 - RDF Serialisation

The fundamental mechanism of the serialisation is the following:

1. Serialisation **to** RDF (“lifting” from the specific data format to the ‘ontology’ language)
    a. The class @RdfsClass annotation determines the rdf:type property of an instance of this class.
    b. All getter methods of a class that contain an @RdfProperty annotation are used to read property values using reflection. The value of the @RdfProperty annotation becomes a predicate, and the property value obtained from the getter becomes the object of the triple.
2. Serialisation **from** RDF (“lowering” from the ‘ontology’ language to the specific data format)
    a. The object of the rdf:type predicate in a triple determines the instantiated java class.
    b. For all predicates in the triple the setter methods in the class that have a matching @RdfProperty annotation are invoked to set the class’ corresponding value to the object of the triple’s predicate. The setter methods can be ‘inferred’ from the annotated getter methods if they conform to the usual naming convention for java getters/setters.

5.2 Semantic Graph Manager

The Semantic Graph Manager is built on top of the It2Rail RDF framework and provides an implementation of the Semantic Query and Aggregation Engine for use by Interoperability Framework “packaged resolver” services.
The Semantic Graph Manager for the FREL release is built and installed in the IT2Rail project’s MAVEN repository for use in packaged resolver development.

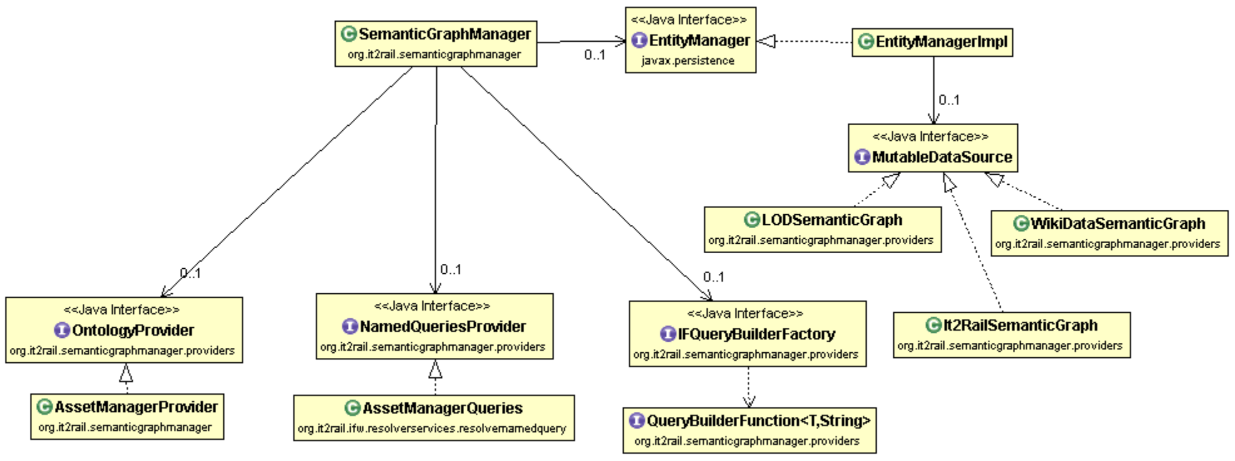

SemanticGraphManager
org.it2rail.semanticgraphmanager
<<Java Interface>>
EntityManager
javax.persistence
EntityManagerImpl
0..1
<<Java Interface>>
MutableDataSource
LODSemanticGraph
org.it2rail.semanticgraphmanager.providers
WikiDataSemanticGraph
org.it2rail.semanticgraphmanager.providers
It2RailSemanticGraph
org.it2rail.semanticgraphmanager.providers
<<Java Interface>>
OntologyProvider
org.it2rail.semanticgraphmanager.providers
<<Java Interface>>
NamedQueriesProvider
org.it2rail.semanticgraphmanager.providers
<<Java Interface>>
IFQueryBuilderFactory
org.it2rail.semanticgraphmanager.providers
AssetManagerProvider
org.it2rail.semanticgraphmanager
AssetManagerQueries
org.it2rail.ifw.resolverservices.resolvenamedquery
QueryBuilderFunction<T,String>
org.it2rail.semanticgraphmanager.providers

Figure 23 - Semantic Graph Manager model

As can be seen in the figure above, the Semantic Graph Manager adds to the underlying it2rail RDF framework references to an instantiated EntityManager, to an OntologyProvider interface, to a NamedQueriesProvider interface and to an IFQueryBuildFactory interface.

A particular “packaged resolver” Semantic Graph Manager is obtained by supplying a specific Guice injection module containing the specific bindings requested

1. to the desired persistence unit configuration for which an EntityManager must be created.
2. to the desired DataSourceFactory for the specific packaged resolver.
3. to the desired NamedQueriesProvider interface implementation for the particular resolver.
4. to the desired OntologyProvider interface implementation for the particular resolver.
5. to the desired IFQueryBuilderFactory implementation requested for the particular packaged resolver.

The first two bindings are propagated to the underlying it2rail RDF framework to create the Empire configuration described in chapter 4.1.2, while the third, fourth and fifth are specific to the Semantic Graph Manager:

1. a NamedQueryProvider interface implementation provides access to a set of stored SPARQL queries templates from which individual queries can be instantiated. Since semantic queries can represent first order predicate logic ‘rules’, the ability to store them for re-use, versioning, etc. as ‘assets’ in the Interoperability Framework Assets Manager is an important feature of the design and the implementation. Also since providers can be injected through the Guice framework it is possible use different sets/versions of named queries, for example for testing and production, and/or to store them in different media, e.g. as files packaged with the Semantic Graph Manager’s JAR archive, on distributed web servers or the triple store itself. In Figure 21 above the concrete implementation is the AssetManagerQueries provider that accesses the Interoperability Framework’s Asset Manager for stored queries.
2. an OntologyProvider interface implementation provides access to the domain’s ontology. In Figure 21 above the concrete implementation is the AssetManagerProvider class that provides access to the AssetManager for the stored ontology.
3. An IFQueryBuilderFactory interface implementation, also injected through the Guice framework, provides the means to produce an implementation of the Function<T,String> functional interface available in Java 1.8. The concrete injected implementation implements a specific functional interface to create a SPARQL query from a request of generic type T. Used in conjunction with the named queries templates, this feature allows for the development of multiple “packaged resolvers” on the same Semantic Graph Manager by supplying it with the appropriate Guice Module.