Contract No. H2020 – 730842

SEMANTIC TRANSFORMATIONS FOR RAIL TRANSPORTATION

D4.2 – Software release of archetypal implementation of converters

Due date of deliverable: 31/01/2018

Actual submission date: 03/12/2018

Leader/Responsible of this Deliverable: OLTIS Group

Reviewed: Y

Document status		
Revision	Date	Description
1	15.03.2018	First draft
1.1	07.08.2018	Details on 918 and FSM standards removed General information about converter implementation updated Reviewers' comments incorporated
2	24.10.2018	Further improvements in response to the WP leader comments
3	30.11.2018	Final release
4	03/12/2018	Final version after TMC approval and quality check

Project funded from the European Union's Horizon 2020 research and innovation programme		
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: 01/11/2016 Duration: 24 months

EXECUTIVE SUMMARY

This document presents an archetypal implementation of the converter in the ST4RT project, including the description of technologies which have been chosen to be used in the converter implementation. This document also describes various inputs (e.g. annotations or master data) and outlines possible improvements.

The terminology used in this document is based on the IT2Rail and ST4RT ontologies.

ABBREVIATIONS AND ACRONYMS

Abbreviation	Description
H2020	Horizon 2020 framework programme
JAR	Java Archive file format
JAXB	Java Architecture for XML Binding
POM	Project Object Model
RDF	Resource Description Framework
RSP	Rail Service Provider
RU	Railway Undertaking
SPARQL	Simple Protocol And Rdf Query Language
ST4RT	Semantic Transformations for Rail Transportation
UML	Unified Modeling Language

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The aim of the document is to briefly describe an archetypal component intended for ontological annotation/mapping (i.e. the converter). The actual implementation of the converter includes the component software design and architecture, the development of required features identified in the preceding Task 4.1 (Tools for annotation/mapping with/to ontologies), the integration of selected tools with specific developments, the organization of software into different packages and the iterative unit and integration testing. The goal is to create a converter capable of performing semantic transformations of reservation request/response messages between systems adopting different communication standards. When two systems using different communication standards need to interoperate, such a converter must be placed between them as a mediator. This is the intended usage of the converter.

2. CORE CONVERTER ARCHITECTURE

The any-to-one centralized approach is applied in the ST4RT Converter according to the recommendation of D4.1. This approach is well-suited for managing complex and dynamic environments, like the one analysed by ST4RT, where a common shared reference ontology is available.

The main properties of this approach are:

- any-to-one: each data schema is mapped to a single reference data schema;
- centralized: the data integration is executed in a single, distinguished node.

As Figure 1 illustrates, in the proposed approach, during lifting, the source XML message is unmarshalled to a set of connected Java objects, and the RDF representation of such objects (in the form of a named RDF graph) is then produced based on the “lifting” annotations. Once the lifting phase is accomplished, the original message information is expressed according to the reference ontological model. Then, according to the target standard and the type of the message, the named RDF graph is queried to create the proper set of destination Java objects. The object attributes are filled in leveraging the “lowering” Java class annotations and information from the stored named graph. Once the set of destination Java objects has been created, they are marshalled into the desired XML message in the target standard. [1]

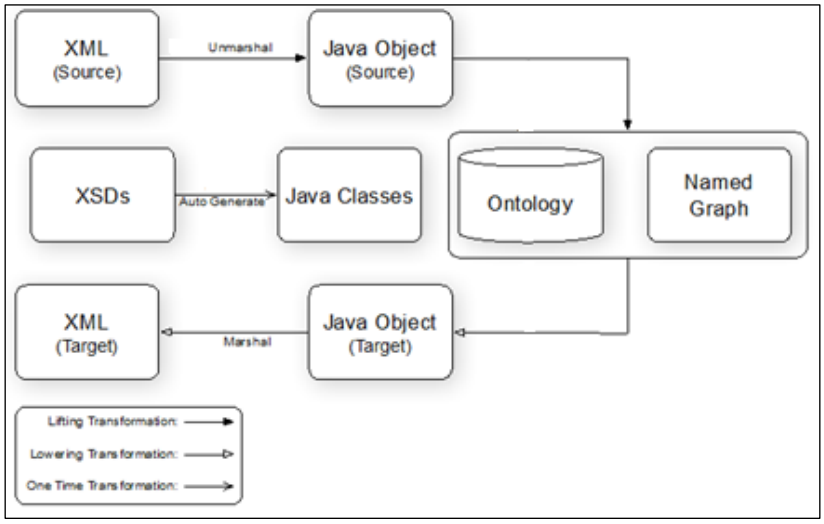

Figure 1: Lifting/lowering process

The ST4RT Converter is designed and implemented as a Java library, which can be used in any environment which is able to receive raw messages and convert them into Java objects. The core converter library doesn't include any specific knowledge about the messages specifications, therefore in order to work and to perform the conversion, it requires:

- ontologies: a set of RDFS/OWL ontologies used to annotate the Java classes;
- master/lookup data: a set of RDF files containing reference/master data (like code lists) to be included in the conversion process;
- SPARQL queries: an XML file containing the SPARQL queries specified in the @Query annotations inside the annotated Java classes related to incoming and destination messages.

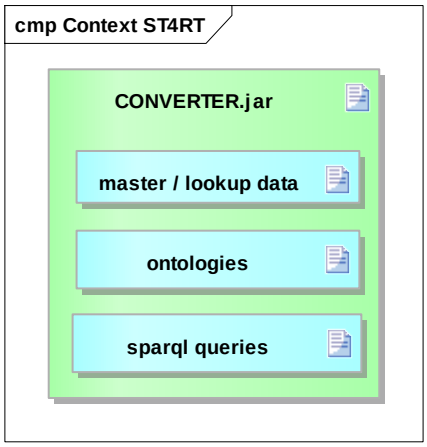

Figure 2: Converter context

3. CONVERSION PROCESS

Individual steps of the conversion process are described in the deliverable D3.3, specifically in chapters 5.1 Lifting sub-process and 5.2 Lowering sub-process. However, the need for some changes was identified during the implementation. Therefore, the original process has been updated to reflect them.

Lifting:

1. Identify root element of the created Java object and create the root element in the RDF graph
2. Process of an element/attribute
    a. If the @Link annotation is specified, then the triples corresponding to the chain of links are added, and the property (with its corresponding object) belongs to the element at the end of the chain;
    b. If the @RdfProperty has the "value" parameter, the object is given by the specified value in data type specified by "dataType" parameter (default String);
        i. If the attribute is of complex type then go inside and continue to point 2;
    c. Else, if the attribute has a complex type, the annotations of the corresponding Java class (which must be annotated with an @RdfsClass annotation) are processed recursively, and the object is the instance created by the recursive process;
    d. Else, the object is the value of the attribute.

If the attribute being processed is of complex type, it is not annotated, nor is its getter method, then the attributes to be processed are those of the referenced object. The algorithm for the processing of attributes of the referenced class starts in point 2. *Process of an element/attribute* above, but the subject to be used in the triples to be created is the one corresponding to the enclosing class, not of the referenced one.

The last part of the Lifting sub-process is the execution of @Queries/@Query associated with the class that are of "Lifting" type.

Lowering:

1. Identify RDF graph root element and create Java Object with this root element
2. Process of an element/attribute:
    a. If the value of the attribute was retrieved through a class-level @Query annotation, that value is assigned to the attribute
    b. Else, if there is an @Query annotation at the attribute level, then the value to be set to the attribute is the one which is the result of the query (which must select a single value):
        i. If the attribute is annotated by both annotations @RdfProperty and @Query then the @Query is processed, @RdfProperty is ignored;
    c. Else, if the @Link annotation is specified, then a SPARQL query is run to retrieve the value of the property at the end of the chain;
    d. Else, if the attribute has a complex type, the annotations of the corresponding Java class (which must be annotated with an @RdfsClass annotation) are processed recursively, and the value to be set is the object instance created by the recursive process;

e. Else, the value to be set is the one corresponding to the specified property.

If the attribute being processed is of complex type and neither the attribute itself, nor its setter method are annotated, then the attribute is initialized with a newly created object that does not correspond any instance in the RDF graph, and the attributes of such are processed. The algorithm for the processing of the attributes of the newly created object starts in point 2. *Process element/attribute* above. If there is no annotated attribute in this complex type for processing, the newly created object is destroyed. [2]

4. TECHNICAL DETAILS

This chapter covers the technical details of the ST4RT converter implementation, namely the list of dependencies and the converter configuration.

4.1 ANNOTATIONS

The Java Custom annotations, based on Java annotations, are used. These user-defined annotations were delivered in WP3 – see D3.3 Annex.

In order to make them ready for use (compilable), the following modifications of the originally delivered annotations had to be implemented:

1. Use of Built-In Annotations

```
@Target({ElementType.TYPE})
@Inherited
@Retention(RetentionPolicy.RUNTIME)
```

2. Import of necessary Java annotation package

```
import java.lang.annotation.*;
```

This input is necessary in any case because the converter contains instructions on how to work with these annotations.

4.2 CONVERTER DEPENDENCIES

The three main dependencies of the ST4RT converter are JAXB, Apache Jena and Empire.

Java Architecture for XML Binding (JAXB) is a technology used to transform XML messages to Java objects and back. Since the W3C XML stack comprises a standard to specify the message structure (XML Schema), using JAXB allows generating the Java classes corresponding to the various elements of the messages structures. Such generated classes come with a set of Java annotations which are used to keep track of the link between classes and attributes in the Java objects and the elements and the attributes in the XML message.

At runtime, JAXB includes methods for unmarshalling / reading XML file and creating Java object and for marshalling / writing into XML file from Java object.

App.java | BerthReques... | PreBookRequ... | PreBookResp... | Fare.java | RSPPrice.java | 918_request.

```
// This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference Implementation, v2.2.8-b130911.1802

package cz.og.uic918;

import java.math.BigInteger;

 * Data to request an allocation of berths.
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "BerthRequestType", propOrder = {
    "requestedTrain",
    "passengers",
    "berths",
    "clazz",
    "coachType",
    "compartmentType",
    "contingent",
    "connectingDoor",
    "passengerSex",
    "travelersWithCar",
    "positionOfPlaces",
    "positionOfCompartment",
    "tariff"
})
public class BerthRequestType {

    @XmlElement(name = "RequestedTrain", required = true)
    protected RequestedTrainType requestedTrain;
    @XmlElement(name = "Passengers")
    @XmlSchemaType(name = "integer")
    protected int passengers;
    @XmlElement(name = "Berths")
    protected List<NumberOfBerthsType> berths;
    @XmlElement(name = "Class")
    protected String clazz;
    @XmlElement(name = "CoachType", defaultValue = "0")
    protected Integer coachType;
    @XmlElement(name = "CompartmentType", defaultValue = "0")
    protected Integer compartmentType;
    @XmlElement(name = "Contingent", defaultValue = "0")
    protected Integer contingent;
    @XmlElement(name = "ConnectingDoor", defaultValue = "false")
    protected Boolean connectingDoor;
    @XmlElement(name = "PassengerSex")
```

Figure 3: Example of pure Java class with JAXB annotations

Apache Jena [4] has been used inside the converter to deal with RDF triples manipulation. It has been used to load master data inside the named RDF graph, and to query it via SPARQL queries.

The Empire framework [3] has been used as a development starting point, since it exploits specific Java annotations to perform object-to-RDF transformation. The set of annotations originally supported by Empire has been enriched to cover the cases described in D3.1 and D3.3. More specifically, the following annotations have been added to Empire:

- @Link
- @Queries
- @Query

5. THE ST4RT USE CASE

This chapter will provide the basic information required to use the ST4RT converter to deal with messages coming from a new specification/standard.

To integrate a new specification/standard, the following steps must be accomplished:

a. Concepts and relations related to the standard must be formalised in an ontology, or an already existing ontology must be found;
b. XML Schema of the standard must exist;
c. JAXB (Java classes) must be generated from that standard;

d. Such Java classes must be annotated using ST4RT annotations and linked to ontology concepts and relations;
e. In case @Query annotations are used, the related SPARQL queries must be written (to select values or reshape the RDF graph);
f. Master data file must contain appropriate records (items).

In the following, we will use the ST4RT use case (TAP/TSI 918 XML to FSM) as an example and will show how to build a converter using ST4RT technologies.

5.1 PROJECT STRUCTURE

To ease the comprehension of the concepts, we will start from a sample Maven project structure. The example project *st4rt-service* implements a REST interface where the messages of chosen standards are forwarded and the converter is invoked. Our converter service requires a set of various libraries:

1. *st4rt-918-messages* contains annotated Java Classes (JAXBs) for 918 standard;
2. *st4rt-annotations* contains the definition of the ST4RT annotations;
3. *st4rt-convertor* contains the core of conversion, includes the Empire framework, and reads annotated elements in JAXBs;
4. *st4rt-fsm-messages* contains annotated Java Classes (JAXBs) for FSM standard;
5. *st4rt-ontology* contains all non-code data, namely the ontologies and the RDF master data.

In Maven terms, the *pom.xml* of the project where the converter should be used must contain the following dependency on the core converter:

```
<dependency>
    <groupId>eu.st4rt</groupId>
    <artifactId>st4rt-convertor</artifactId>
    <version>1.6.0</version>
</dependency>
```

The core converter itself contains these additional dependencies:

```
<dependency>
    <groupId>eu.st4rt</groupId>
    <artifactId>st4rt-annotations</artifactId>
    <version>1.0.0</version>
</dependency>

<dependency>
    <groupId>eu.st4rt</groupId>
    <artifactId>st4rt-fsm-messages</artifactId>
    <version>1.0.0</version>
</dependency>

<dependency>
    <groupId>eu.st4rt</groupId>
    <artifactId>st4rt-918-messages</artifactId>
    <version>1.0.0</version>
</dependency>
```

```
<dependency>
    <groupId>eu.st4rt</groupId>
    <artifactId>st4rt-ontology</artifactId>
    <version>1.0.0</version>
</dependency>
```

5.2 Converter Invocation

The key class to invoke the converter is *Convertor*, which is included in the *st4rt.convertor* package inside the *st4rt-converter* package. The Convertor class contains the path to the file where the master data are stored. The ontology model is created in memory from this file. The SPARQL queries then require values selected from this model.

```
common_model.read(getClass().getResourceAsStream("/Berth.owl"), null, FileUtils.langXML);
```

Then the package with converter class can be imported to the package/s where the converter is to be used.

```
import st4rt.convertor.Convertor;
```

5.3 Annotated Classes

The ST4RT annotations, described in D3.3 and its Annex, need to be added to the Java classes generated by invoking the *xjc* utility (included in standard Java Runtime Environment) on both input and destination XML schemas (in this case the FSM and 918-XML schemas). We put the FSM-related annotated classes into the *st4rt-fsm-messages* package, and annotated Java classes related to 918-XML are placed in *st4rt-918-messages* package. The definitions of annotations are placed in *st4rt-annotation* package.

An example of such annotations is shown in Figure 4: Example of additionally annotated JAXB.

```java
package model.fsmJC;

import annotation.Queries;
import annotation.Query;
import annotation.Query.QueryType;
import com.clarkparsia.empire.SupportsRdfId;
import com.clarkparsia.empire.annotation.*;

import javax.xml.bind.annotation.*;
import javax.xml.bind.annotation.adapters.CollapsedStringAdapter;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import java.util.ArrayList;
import java.util.List;

 * The PreBookRequest message is used in different cases that will impact the
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "PreBookRequest", namespace = "http://servicemodel.pts_fsm.org/2015/10/29/rsp.booking.messages", propOrder = {
        "bookingIdRef", "carrierOfferIdRefsList", "carrierOfferContainer", "fulfilmentMethod", "paymentMode",
        "passengerPreference", "contactInformation", "directBookingParameters" })
@XmlRootElement(name = "PreBookRequest")

@NamedGraph(type = NamedGraph.NamedGraphType.Static)
@Namespaces({ "st4rt", "http://st4rt.eu/ontologies/st4rt#", "it2r",
    "http://it2rail.eu/ontology#", "foaf", "http://xmlns.com/foaf/0.1/#",
        "shopping", "http://www.it2rail.eu/ontology/shopping#",
        "travel", "http://www.it2rail.eu/ontology/travel#",
        "transport", "http://www.it2rail.eu/ontology/transport#",
        "infrastructure", "http://www.it2rail.eu/ontology/infrastructure#",
        "customer", "http://www.it2rail.eu/ontology/customer#",
        "product", "http://www.it2rail.eu/ontology/product#",
        "xmlschema", "http://www.w3.org/2001/XMLSchema#"
})
@RdfsClass("st4rt:ProvisionalBookingRequestMessage")
@Queries({
    @Query(name = "LiftingProvisionalBookingRequest1", type = QueryType.Lifting),
    @Query(name = "LiftingProvisionalBookingRequest2", type = QueryType.Lifting),
    @Query(name = "LiftingProvisionalBookingRequest3", type = QueryType.Lifting),
    @Query(name = "LiftingProvisionalBookingRequest4", type = QueryType.Lifting),
    @Query(name = "LoweringProvisionalBookingRequest1", type = QueryType.Lowering),
    @Query(name = "LoweringProvisionalBookingRequest2", type = QueryType.Lowering)
    })
public class PreBookRequest
extends Request implements SupportsRdfId {

    private SupportsRdfId mIdSupport = new SupportsRdfIdImpl();

    public RdfKey getRdfId() {

    public void setRdfId(RdfKey theId) {
//  private String provisionalBookingRequestMessageID;

    protected String bookingIdRef;
    protected List<String> carrierOfferIdRefsList;

    @XmlElement(name = "CarrierOfferContainer")
    @RdfProperty(propertyName = "st4rt:hasItineraryOffer")
    protected SaleCarrierOfferContainer carrierOfferContainer;
```

Figure 4: Example of additionally annotated JAXB with RDF annotations

Invocation of SPARQL queries is defined at appropriate Java class.

```java
@RdfsClass("st4rt:ProvisionalBookingRequestMessage")
@Queries({
        @Query(name = "LiftingProvisionalBookingRequest1", type = QueryType.Lifting),
        @Query(name = "LiftingProvisionalBookingRequest2", type = QueryType.Lifting),
        @Query(name = "LiftingProvisionalBookingRequest3", type = QueryType.Lifting),
        @Query(name = "LiftingProvisionalBookingRequest4", type = QueryType.Lifting),
        @Query(name = "LoweringProvisionalBookingRequest1", type = QueryType.Lowering),
        @Query(name = "LoweringProvisionalBookingRequest2", type = QueryType.Lowering)
        })
```

There are two types of queries according to the phase of processing:

1. Lifting
2. Lowering

If *Lifting* phase is running, then the queries for *Lowering* type are ignored and vice versa.

Queries not containing type part are by default of *Lifting* type.

```java
@Target({ElementType.FIELD, ElementType.METHOD}) @Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface Query {

    /**
     * The type of query to be executed that can be either Lifting or Lowering
     */
    enum QueryType {
        Lifting,
        Lowering
    }

    /**
     * @return the name of corresponding query that should be looked up in queries.xml
     */
    String name();

    /**
     * @return the attributes names whose values are required for running this query.
     */
    String[] inputs() default {};

    /**
     * @return the attributes names whose values must be set using the query result
     */
    String[] outputs() default {};

    /**
     * @return the type of the query whose default value is Lifting.
     */
    public QueryType type() default QueryType.Lifting;

}
```

Figure 5: Definition of Query annotation, Lifting type as default

The supporting files for the conversion from FSM to 918 and vice versa (like sparql_queries.xml file, etc.) are placed in *st4rt-ontology* package.

5.4 SPARQL QUERIES

The file *sparql_queries.xml* (included in *st4rt-ontology* package) contains queries needed to read the values and reshape the RDF graph structure when necessary. In our example converter project we will place such file in the package st4rt-ontology. If a new query needs to be added, modified or deleted, then it must be done in that file. In case the path to the queries file needs to be changed, it must be configured in the source code, in the file QueryUtil.java.

```java
InputStream is = QueryUtil.class.getResourceAsStream("/sparql_queries.xml");
```

Example of a query used during Lowering phase where one element (stopPlaceId) from FSM is divided into two elements (country and localCode) in 918:

```
<query>
<name>918_LW_Divide1CodeInto2</name>
      <comment>
            Input: country code + station code together = as one value
            Outputs: country and station
      </comment>
        <outputs>
            <output>
                country
            </output>
            <output>
                localCode
            </output>
      </outputs>
```

```
        <sparqlQuery><![CDATA[
                PREFIX st4rt: <http://st4rt.eu/ontologies/st4rt#>
                PREFIX owl: <http://www.w3.org/2002/07/owl#>
                PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

                SELECT ?country ?localCode
                WHERE {
                  ??this st4rt:hasStopPlaceId ?code.
                  BIND ( SUBSTR(?code, 1, 2) AS ?country).
                  BIND ( SUBSTR(?code, 3) AS ?localCode).
                }
        ]]></sparqlQuery>
</query>
```

The following name convention is used in prefixes:

- 918_LW – query in 918 message for Lowering phase;
- 918_LF – query in 918 message for Lifting phase;
- FSM_LW - query in FSM message for Lowering phase;
- FSM_LF - query in FSM message for Lifting phase.

There are two forms of SPARQL queries:

1. SELECT – returns values;
2. CONSTRUCT – returns reshaped RDF graph.

Apache Jena is used to process such queries.

The annotation @Queries/@Query has to be placed before a class or attribute if the SPARQL query needs to be used.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "StationCode918Type", namespace = "http://www.uic-asso.fr/xml/passenger/02")
@RdfsClass("infrastructure:StopPlace")
@Queries({
        @Query(name = "918_LW_Divide1CodeInto2", outputs = {"country", "localCode"}, type = Query.QueryType.Lowering)})
public class StationCode918Type
    extends StationCodeType
{
```

5.5 ONTOLOGY

There are two ontologies that have been examined for the ST4RT Converter. These ontologies are not directly processed by the converter, but they are used for the manual annotation process in order to know the names of objects and corresponding properties.

5.5.1 IT2Rail Ontology

The existing IT2Rail ontology was created as the automatic interpretation of data exchanges between functional applications irrespective of syntactic formats or protocols. The actual implementation of the ontology is formed by a secure and stable repository of a persistent machine-readable, explicit and formal description. The ontology contains the list of terms together with description of their meaning, as it was consolidated from WP2-6 Ontology deliverables. The main source for the formal ontology is the UML-based model which has been developed by the Capella tool.

5.5.2 ST4RT Ontology

The ST4RT ontology extends the IT2Rail ontology with concepts belonging mostly to 918 standard.

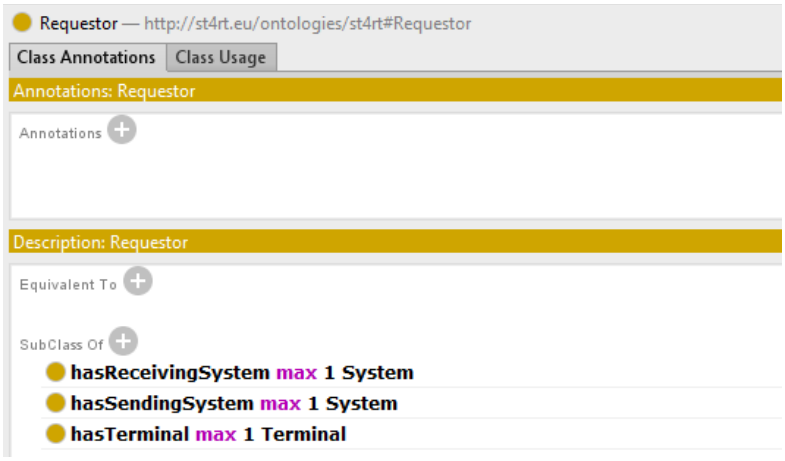

Figure 6: 918 Requestor in ontology

5.6 MASTER DATA

Master / lookup data are part of the ontology files.

There are two cases when master data are needed:

1. Different naming / coding for the same concept

   The transformation between two different formats with different values for the same “item” cannot be correctly done without help of extra mapping file which states that this ***valueA*** in standard **A** means this ***valueB*** in standard **B**.

   Data from master data (file) are selected via SPARQL queries (SELECT queries).

	FSM	**918 request**	**918 reply**
Accommodation type:	SINGLE	L	1
Class :	FIRST_CLASS	1	1

Table 1: Example of different values with the same meaning

2. Different objects

   The transformation between two different formats with different objects. These two objects must be linked to be derivable one from another. The link between these objects is placed in master data file. Data from master data (file) are selected via SPARQL queries and the RDF graph can be reshaped (SELECT, CONSTRUCT queries).

General	**FSM**	**918**
Sender	Operator / RU / RSP	Used reservation system
Recipient	Operator / RU / RSP	Used reservation system

Table 2: Example of different objects

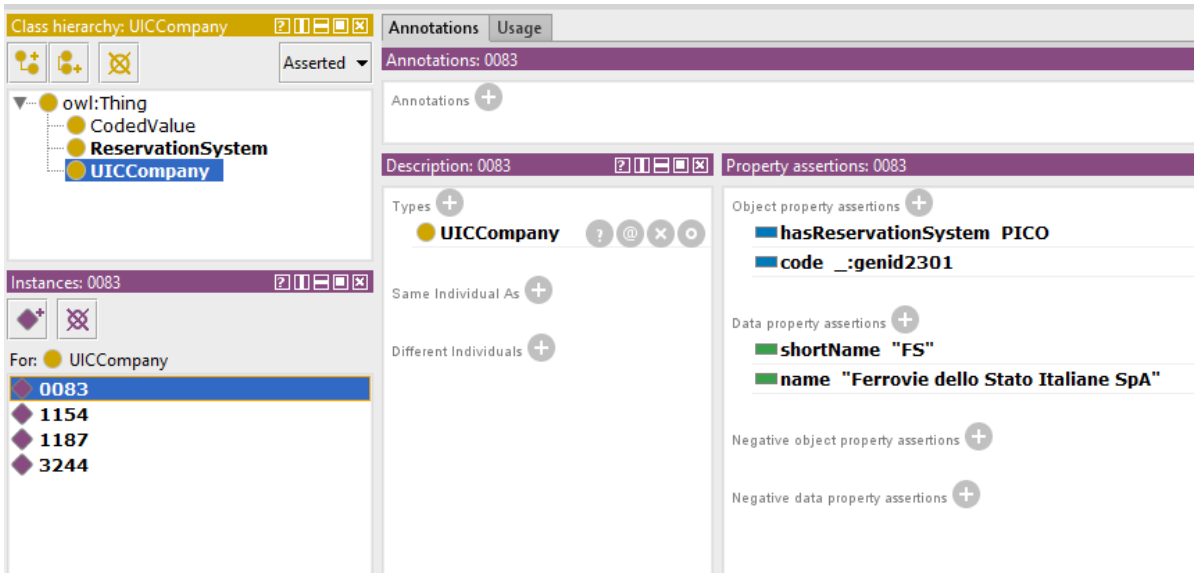

Figure 7: Link between two different objects (a company and a reservation system)

5.7 RDF GRAPH

No external database or repository is used for the conversion process. All the data are stored in memory as an RDF graph.

The RDF graph is generated according to annotations used in JAXBs (in st4rt-918-messages and st4rt-fsm-messages). Neither IT2Rail nor ST4RT ontology is required for the conversion, the ontologies are used only as templates that enable to name RdfsClasses/RdfProperties and to set up the links between such RdfsClasses/RdfProperties by using SPARQL queries.

When the class processing is finished, a search for values or reshaping of the RDF graph can be executed. It depends on whether such class is annotated by @Queries / @ Query annotations. If the @Queries / @ Query annotations are found, then the corresponding SPARQL query is searched (by name and by type) in the file sparql_queries.xml and executed. An example of the RDF graph is in Figure 8.

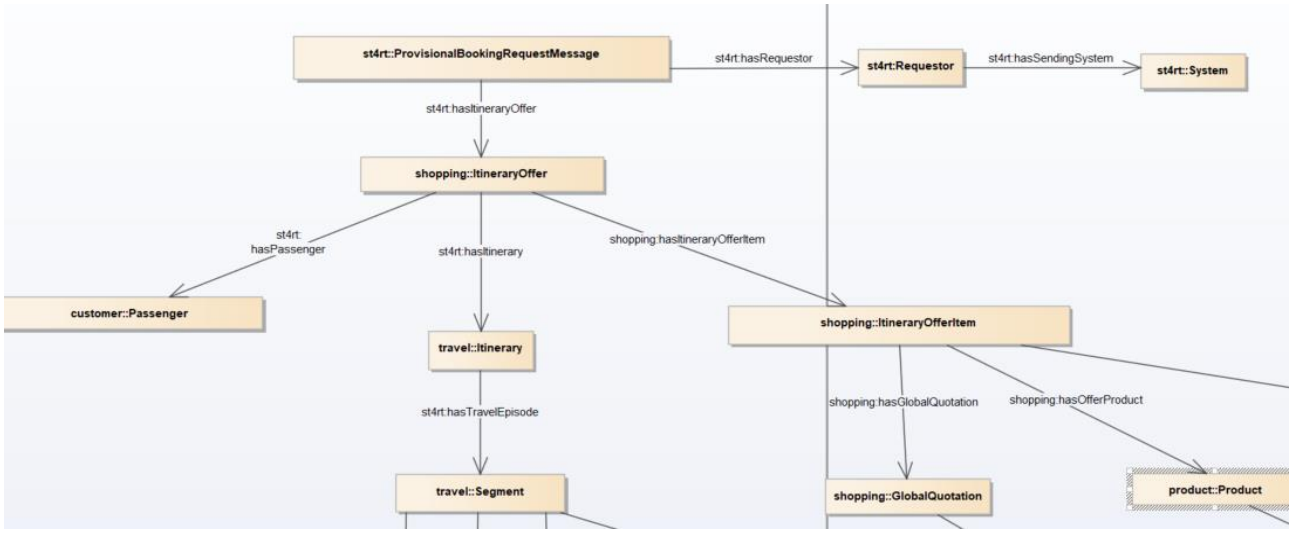

Figure 8: Part of RDF graph

To be able to convert one standard to another one, the annotations of both corresponding JAXBs must contain the same names for the elements with the same context. The use of a name from ontology in FSM (PreBookRequest) and 918 (ReservationRequest) JAXBs' root elements is illustrated in the Figure 9:

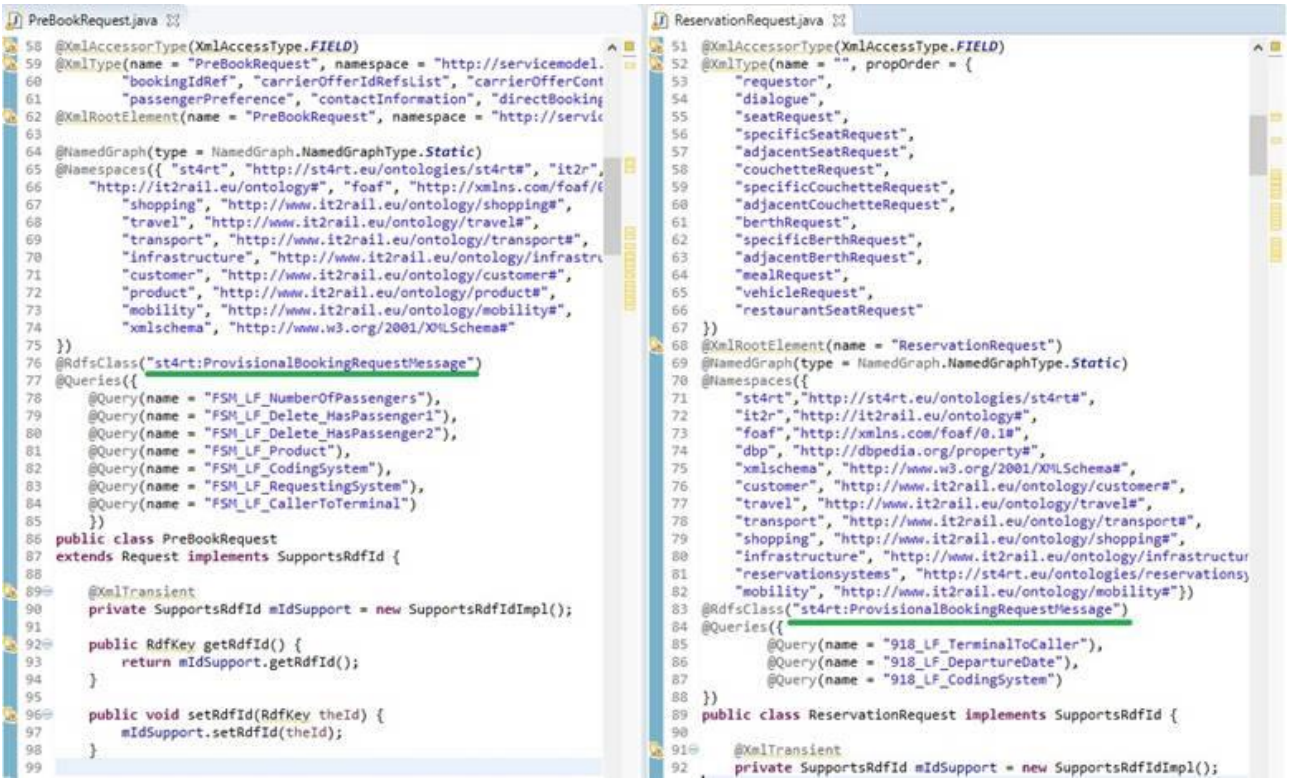

Figure 9: Example of corresponding JAXBs' elements

The elements defined in the ST4RT ontology have the "st4rt" namespace.

6. CONCLUSIONS

This deliverable describes the technical details of the first official release of the core converter. Testing of the developed features will take place in the next stage of the project and will be documented in D5.4. In detail, D5.4 will document the results of the integration of the ST4RT Converter with the HEROS testing environment.

In terms of access to the knowledge base, the ST4RT Converter has been implemented as self-contained. This means that the deployed converter must access both ontologies and master data by using their local copies. Each time a new version of the ontology is released, the converter needs to be repackaged and re-deployed. Therefore, accessing remote resources by their URLs (downloading them), as well as direct connecting to a semantic graph may become significant improvements of the core converter implementation.

REFERENCES

[1] ST4RT D4.1 Analysis of state-of-the-art ontology conversion tools. Available at: http://www.st4rt.eu

[2] ST4RT D3.3 Mapping between standards and reference ontology. Available at: http://www.st4rt.eu

[3] The Empire framework. Available at: https://github.com/mhgrove/Empire

[4] Jena Ontology API. Available at: https://jena.apache.org/documentation/ontology

[5] Apache Maven Project: Welcome to Apache Maven. Available at: https://maven.apache.org

[6] Apache Maven Project: Introduction to the POM. Available at: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html