Contract No. H2020 – 730842

SEMANTIC TRANSFORMATIONS FOR RAIL TRANSPORTATION

D3.1 Annotations for Mapping from Legacy Data Models to Ontologies

Due date of deliverable: 30/04/2017

Actual submission date: 17/05/2017

Leader/Responsible of this Deliverable: Politecnico di Milano

Reviewed: Y

Document status		
Revision	Date	Description
1	28/04/2017	First draft
2	1/05/2017	Second draft
2.1	9/05/2017	Candidate for delivery
2.2	15/05/2017	Fixed some typos and references
3	17/05/2017	Final version after TMC approval

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 deliverable first provides a first overview of the process for mapping requests for services including data built according to legacy data representations onto/from the concepts of a pivot ontology (and in particular the reference Shift2Rail ontology).

After having clarified the overall approach, which is based on the idea of annotating “schemas” (in the broad sense of the term) defining the structure of the legacy data, it provides some details concerning annotations proposed in the ST4RT project to carry out the mapping.

ABBREVIATIONS AND ACRONYMS

Abbreviation	Description
EU	European Union
FSM	Full Service Model
GA	Grant Agreement
H2020	Horizon 2020 framework programme
JAXB	Java Architecture for XML Binding
JU	Shift2Rail Joint Undertaking
RDF	Resource Description Framework
ST4RT	Semantic Transformations for Rail Transportation
SPARQL	SPARQL Protocol and RDF Query Language
TAP	Telematics Applications for Passenger service
TSI	Technical Specification for Interoperability
XML	eXtensible Markup Language
XSD	XML Schema Definition

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This document defines the basic annotations that can be used to map concepts from a legacy data representation such as the Full Service Model (FSM, [1, 2]) or the “Telematics Applications for Passenger Services - Technical Specification for Interoperability” (TAP TSI, [3, 4]) standards, to a reference ontology such as the one defined in Shift2Rail. These annotations will be the basis for creating the mechanisms to automatically convert data obeying legacy standards into data that conforms to the reference ontology, and vice-versa. The annotations are exemplified on an example FSM booking message provided by the ST4RT partners.

The rest of the deliverable is structured as follows. Chapter 2 presents an overview of the whole mapping process, to provide the context in which mappings occur. Chapter 3 presents the basic annotations used to map concepts from legacy data representations to the reference ontology, and vice-versa. Chapter 4 deals with mechanisms to manage more complicated mappings, which cannot be handled by the annotations introduced in Chapter 3.

2. OVERVIEW OF ANNOTATION-BASED TRANSFORMATIONS

Figure 1 depicts the general, high-level view of the whole mapping process:

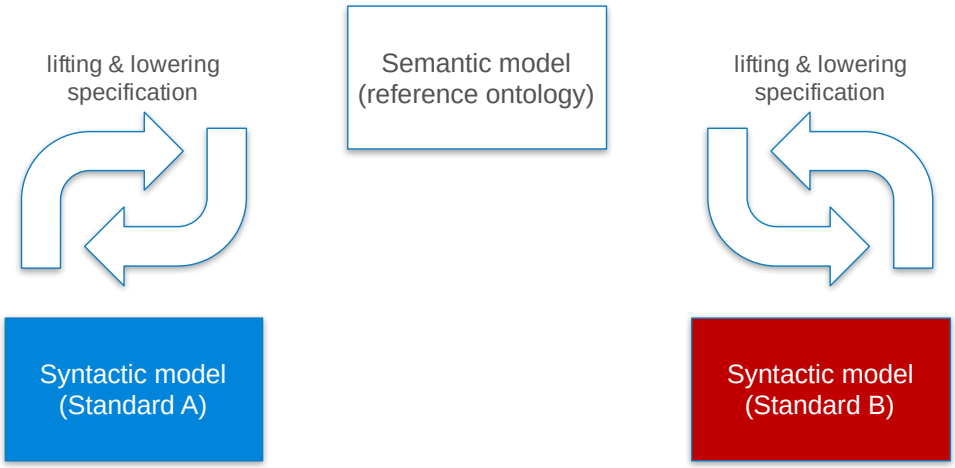

Figure 1: Overview of the mapping process

The figure shows that, if a mapping from a source standard A to a target standard B must be done, then the incoming data must be "lifted" from the standard A to the reference ontology; then, the ontology-respecting data must be "lowered" to the target standard B. If the whole mapping concerns a request/response mechanism, where a request made using standard A is mapped onto a request made using standard B, and then the response produced, which obeys standard B, is mapped onto data obeying standard A, then the lifting/lowering mechanism is done twice:

1. The incoming request is lifted from standard A to the reference ontology.
2. The "lifted" request is "lowered" from the reference ontology to standard B.
3. The response produced according to standard B is "lifted" to the reference ontology.
4. The "lifted" response is "lowered" to standard A.

In this document we will consider examples in which standard A is FSM, whereas standard B is TAP TSI[1].

Figure 2 details the lifting/lowering process in the case where for both the source and target standards the data representation is defined through XSD files. The proposed approach takes into account the technologies that are available to realize in practice the lifting/lowering steps. In particular, frameworks are available, such as JAXB[2], to convert XML data representations into Java objects (and vice-versa). For example, the JAXB framework takes the XSD files for a given data representation (e.g., the FSM XSD files) and automatically produces a set of Java classes that is equivalent to the whole schema definition of the given standard. The result of the generation of Java classes, which does not involve developers, is a Java class for each XML *complexType*, which includes attributes corresponding to the *XmlElement*s of the *complexType*. The generated

[1] More precisely, we will consider XML versions of messages defined in binary form in the TAP TSI standard. Such XML versions are currently not part of the standard and are proprietary standards defined by UIC.

[2] https://jaxb.java.net

Java classes are then used at runtime to automatically create marshal/unmarshal XML files to/from Java objects. In essence, these intermediate Java classes, which are produced only once and for all, provide a representation of the schema of exchanged data, and can be considered as proxy for the original XSD files. In particular, it is this Java code that is ultimately annotated to guide the mapping from the concepts in the legacy standards to those in the reference ontology, and vice-versa. Notice that, in this Java-based approach, the workflow of Figure 2 can still be applied even if, unlike FSM and TAP TSI, the data representations in the source and target standards are not defined through XSD files, but through other mechanisms. To realize the workflow in such a case, though, an initial, one-time step would be required whereby suitable Java classes are created, similar to those automatically generated by the JAXB framework, from the data representations of the standards (see, for example, http://www.jsonschema2pojo.org for a conversion of JSON Schema representations to Java classes). Other approaches, not based on the annotation of Java classes, but on the direct annotation of XSD files, for example through the SAWSDL standard [5], have been considered; they have, in the end, been rejected mainly for two reasons: the lack of supporting tools, and the apparent difficulty to use them to create more complex mappings such as those presented in Chapter 4.[3]

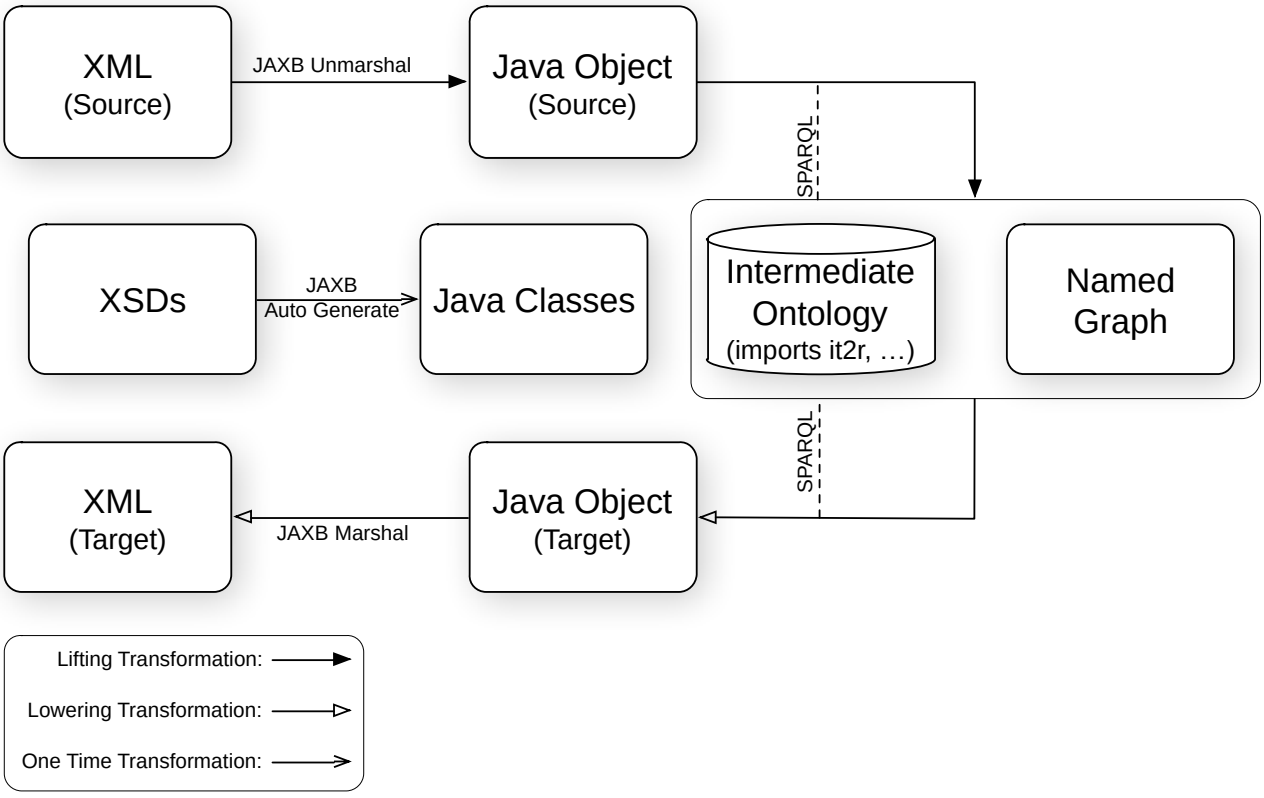

Figure 2: Approach workflow in the case where source and target standards are defined through XSD files as in the case of FSM/TAP TSI

As Figure 2 shows, in the proposed approach, during lifting, the source XML message is unmarshalled to a java object, and a named graph is produced based on the "lifting" annotations

[3] A possibility which we leave for future consideration is to build mechanisms to translate XSD-based annotations into Java-based ones, in order to be able to rely also on the former kind of annotations, in addition to the latter.

(and the corresponding SPARQL queries included therein, as remarked by the dotted line labeled “SPARQL” in the figure, see also Section 3.1) and on the intermediate ontology. Based on the target standard and the type of the message, the proper java object is created. The object attributes are filled using the java class annotations and information from the stored named graph (a set of RDF triples). Finally, the resulting object is marshalled to an XML message in the target standard, also using the “lowering” annotations and corresponding SPARQL queries (see also Section 3.2).

Let us remark that the “Intermediate Ontology” depicted in Figure 2 is defined for mapping purposes, and is typically an extension of the general-purpose ontology defined in Shift2Rail. For example, it needs to include relations that are necessary strictly for mapping purposes, such as those between countries and the actual identifiers used in various standards to refer to them (e.g., “IT” for Italy, and so on).

3. BASIC ANNOTATIONS

This chapter describes the basic annotations, which are used when concepts in the legacy data representation have a straight correspondence with the concepts in the reference ontology. Chapter 4 covers the case where the correspondence is not direct, and the mapping is more complicated.
The annotations introduced in this chapter are compatible with (and extend) those defined in the Pinto framework[4] for mapping Java Beans to RDF triples and vice-versa.

Several types of java annotations are defined in this approach.

- Class Annotations

  The used name spaces, the name, type and URI of the named graph, and the corresponding entity in the named graph are defined using this type of annotation on the given java class.

- Field Annotations

  This kind of annotations is used when an attribute has an immediate corresponding entity, that is essentially the object of a property, in the named graph. These annotations introduce the value of an attribute of a Java object as an object of an RDF triple (lifting) and vice-versa (lowering).

- Method Annotations

  The transformer looks for the given attribute getter method for lifting and its corresponding setter method for lowering. When an attribute lacks an annotated getter/setter method for lifting/lowering, the field annotation is taken into account.

- Parameter Annotations

  Parameter annotations are used when SPARQL queries are required for producing the values during lowering.

Figure 3 depicts a Java class and highlights where each type of annotation can be introduced in the class. The next sections describe each type of annotation, and illustrate them through more detailed examples.

```
@SampleClassAnnotation
class Foo {
    @SampleFieldAnnotation
    private String bar;
    @SampleMethodAnnotation
    public void setName(@SampleParameterAnnotation() String s) {
        bar = s;
    }
}
```

Figure 3: Skeleton of a Java class with placeholders for annotations

[4] https://github.com/stardog-union/pinto

3.1 LIFTING ANNOTATIONS

This section defines the used annotations in this approach and provides usage examples.

@NamedGraph

This annotation defines a named graph with a type, and a URI. A named graph type can be either *instance* when the java object is bounded to a named graph with the same URI as the corresponding individual being persisted, or *static* when the java object persists to a specific named graph.

@NamedSpaces

Name spaces are defined using this annotation.

Figure 4 shows part of a *ContactInformation* class that persists to a static named graph with the given URL. The name spaces used in other annotations in *ContactInformation* class are defined by a set of strings.

```
@NamedGraph(type = NamedGraph.NamedGraphType.Static,
value = "http://www.semanticweb.org/Mehdi/ontologies/2017/2/st4rt/messages.graph")
@NamedSpaces({
    "st4rt","http://www.semanticweb.org/Mehdi/ontologies/2017/2/st4rt#",
    "it2r","http://it2rail.eu/ontology#",
    "foaf","http://xmlns.com/foaf/0.1#",
    "dbp", "http://dbpedia.org/property#",
    "customer", "http://www.it2rail.eu/ontology/customer#"})
@RdfsClass("customer:Passenger")
public class ContactInformation
    extends FSMID
{
```

Figure 4: @NamedGraph and @NameSpaces example

@RdfsClass

The java class is mapped to an entity in the corresponding named graph using this annotation.

The expected RDF being produced by the @RdfsClass annotation of Figure 4 is

```
_:contactInformation rdf:type customer:Passenger
```

@RdfProperty

The annotated attribute is mapped to the object (third element of an RDF) of an object/data property, where the RDF is a type “subject property object”, and the subject is the entity related to the enclosing java class.

```
@RdfProperty("infrastructure:isInCity")
@XmlElement(name = "City")
protected String city;
```

Figure 5: @RdfProperty example

For example, the expected RDF to be produced by the annotation of Figure 5 is

```
_:contactInformation infrastructure:isInCity cityName
```

where *cityName* is the value of *city* attribute. This annotation has two attributes: the name of the property, and the *isList* Boolean value, which needs to be set to true if the attribute to be mapped is a list.

@Sparql

This annotation, as a field annotation, complements @RdfProperty when the value to be mapped is the result of a SPARQL query, instead of the annotated attribute value. Similarly to the @RdfProperty, the mapping target of this annotation is the object of an object/data property, but the value is the result of a SPARQL query.

```
public @interface Sparql {
/**
 *
 * name: Name of the query to be looked up in sparql_queries.xml.
 * inputs: List of input parameters (class attributes),
 * s.t. nth string in the list is replaced with ??inputn.
 */
    public enum QueryType {
        Lifting,
        Lowering,
    }
    String name();
    String[] inputs() default {};
    String[] outputs() default {};
    public QueryType type() default QueryType.Lifting;
}
```

Figure 6: @Sparql class

Figure 6 shows @Sparql class with *name*, *inputs*, *outputs*, and *type* attributes. Note that this annotation can appear as a method annotation for lifting a single attribute, a parameter annotation for lowering a single attribute (explained in Section 3.2) and as a class annotation for lifting/lowering multiple attributes (explained in Section 3.3).

For lifting a single attribute this annotation requires name of the query to be looked up in the query collection, and a set of strings as input parameters.

```
@Sparql(name = "getAlpha2ByName", inputs = {"country"})
@RdfProperty("st4rt:isInCountry_Alpha2")
@XmlElement(name = "Country")
protected String country;
```

Figure 7: Lifting @Sparql example

Figure 7 shows an attribute whose value must be mapped to the object of an RDF triple whose subject and property are `_:contactInformation` and `st4rt:isInCountry_Alpha2`, respectively. The additional @Sparql annotation signifies that the attribute value must be fetched from the *getAlpha2ByName* SPARQL query. Figure 8 depicts the definition of the *getAlpha2ByName* given in XML terms. The definition introduces the name of the query, the list of parameters of the query, and the definition of the query itself (plus a comment for the user). When the parameter is used in the query, its name is prefixed by ??. When the query presented in Figure 8 is run for attribute *country* of Figure 7, `??input1` must be replaced by the value of the *country* attribute before the SPARQL query is run.

For example, if `Italy` is the value of *country* attribute, the following RDF must be produced

```
_:contactInformation st4rt:isInCountry_Alpha2  "IT"
```

```
    <query>
        <name>getAlpha2ByName</name>
        <inputs>
            <input>
                input1
            </input>
        </inputs>
        <comment>
            Input: Country name (??input1)
            Output: ??input1's Alpha2 code
        </comment>
        <sparqlquery><![CDATA[
        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX st4rt: <http://www.semanticweb.org/Mehdi/ontologies/2017/2/st4rt#>
PREFIX countries: <http://www.bpiresearch.com/BPMO/2004/03/03/cdl/Countries#>
SELECT ?alpha2
    WHERE {
    ?subject countries:countryCodeISO3166Alpha2 ?alpha2 ;
        countries:countryNameISO3166Short ??input1 }
]]></sparqlquery>
    </query>
```

Figure 8: getAlpha2ByName SPARQL query

Notice that SPARQL queries referenced by @Sparql annotations could be used by different annotated Java classes. As a consequence, this deliverable assumes that they are stored in an external file which is known and accessible by the mapper (say, a *sparql_queries.xml* file). However, the exact configuration mechanisms through which such files are made available to the mapper are outside the scope of this deliverable.

3.2 LOWERING ANNOTATIONS

@Sparql is also one of the lowering annotations and must be applied on the parameters of the setter methods. Having the set of RDF triples related to a specific java object, the result of the SPARQL query is passed as a parameter to the setter methods of the java class. For example, in Figure 9 the result of the *setCountry* query (whose definition is shown in Figure 10) must be set to the *country* attribute.

```
public void setCountry(@Sparql(name = "setCountry") String value) {
        this.country = value;
}
```

Figure 9: Lowering @Sparql example

```
    <query>
        <name>setCountry</name>
        <comment>
            Fetches country name of the given passenger's Alpha2.
        </comment>
        <sparqlquery><![CDATA[
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX st4rt: <http://www.semanticweb.org/admin/ontologies/2017/2/st4rt#>
PREFIX customer: <http://www.it2rail.eu/ontology/customer#>
PREFIX countries: <http://www.bpiresearch.com/BPMO/2004/03/03/cdl/Countries#>
SELECT ?name
    WHERE {
    ?p rdf:type customer:Passenger .
    ?p st4rt:isInCountry_Alpha2 ?alpha2 .
    ?country countries:countryCodeISO3166Alpha2 ?alpha2 .
    ?country countries:countryNameISO3166Short ?name }
]]></sparqlquery>
    </query>
```

Figure 10: setCountry SPARQL query

Given the following RDF triple

```
_:contactInformation st4rt:isInCountry_Alpha2  "IT"
```

the *setCountry* query returns "Italy"^^xsd:string.

3.3 General Annotations

@Sparqls is a class annotation that contains a set of @Sparql annotations, each of which has a type that is either *Lifting* or *Lowering*. This annotation is defined merely to allow more than one @Sparql annotation to be applied on a single class. @Sparqls as a class annotation can be used to lift/lower a group of attributes viable to be lifted or lowered together by a single SPARQL query.

```
@RdfsClass("customer:Passenger")
@Sparqls({
    @Sparql(name = "LiftingContactInformation", inputs = {"country"}),
    @Sparql(name = "LoweringContactInformation", outputs = {"gender", "country"}, type = QueryType.Lowering)})
public class ContactInformation
    extends FSMID
{
```

Figure 11: @Sparqls as a class annotation

Figure 11 shows an example of a @Sparqls annotation that lowers attributes *gender* and *country* together using the *LoweringContactInformation* query of Figure 12. The SPARQL query returns two values; the *outputs* annotation, an ordered set of attribute names, tells the converter to invoke the setter methods of *gender* and *country*, using the results of the SPARQL query, respectively. When a @Sparql annotation has the *Lifting* type the converter retrieves the values of the attributes present in the *inputs* set by invoking their getter methods and prepares the query to be executed.

```
<query>
        <name>LoweringContactInformation</name>
        <comment>
            Lowering query for ContactInformation attributes.
        </comment>
        <outputs>
            <output>
                gender
            </output>
            <output>
                countryName
            </output>
        </outputs>
        <sparqlquery><![CDATA[
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX st4rt: <http://www.semanticweb.org/Mehdi/ontologies/2017/2/st4rt#>
PREFIX it2r: <http://www.it2rail.eu/ontology#>
PREFIX customer: <http://www.it2rail.eu/ontology/customer#>
PREFIX foaf: <http://xmlns.com/foaf/0.1#>
SELECT ?gender ?countryName
    WHERE {
    ?p rdf:type customer:Passenger .
    ?p <http://xmlns.com/foaf/0.1/gender> ?gender
    ?p st4rt:isInCountry_Alpha2 ?alpha2 .
    ?country countries:countryCodeISO3166Alpha2 ?alpha2 .
    ?country countries:countryNameISO3166Short ?countryName }
]]></sparqlquery>
</query>
```

Figure 12: Lowering query for multiple attributes

4. COMPLEX MAPPINGS

When the concepts in the legacy data representation and those in the intermediate ontology have structural differences, more complex mappings than those allowed by the annotations defined in Section 3 are required. To tackle these cases, a possibility is to first introduce suitable Java code that creates the necessary concepts from those present in the legacy standard, and then exploit the restructuring provided by the added Java code to facilitate the mapping through the annotations of Chapter 3. This avenue is explored in Section 4.1. A second possibility, which is briefly exemplified in Section 4.2, is to introduce suitable concepts in the intermediate ontology, and then create mappings that do not tightly match the reference ontology.

4.1 TRANSFORMATIONS AT JAVA CLASS LEVEL

Let us illustrate through an example how the restructuring of concepts can be achieved at java class level.

```
@XmlElement(name = "Origin")
protected StopPlace origin;

@XmlElement(name = "StopPlace")
protected List<StopPlace> stopPlace;

@XmlElement(name = "Destination")
protected StopPlace destination;
```

Figure 13: Fragment of Java class FSM Segment class

Figure 13 shows a fragment of a Java class, *Segment*, capturing the notion of “Segment” in the FSM standard. An FSM Segment has two StopPlaces as attributes *origin* and *destination*, and a list of StopPlaces as attribute *stopPlace*. Conversely, the IT2Rail ontology includes the notion of Segment as a list of RouteLinks, where each RouteLink has two StopPlaces as its *isStartingAt* and *isEndingAt* properties. For example, If origin=stp1, stopPlace={stp2, stp3}, and destination=stp4, {rl1, rl2, rl3} could be the list of RouteLinks, where the following relationships hold:

```
rl1 :isStartingAt stp1; :isEndingAt stp2 .

rl2 :isStartingAt stp2; :isEndingAt stp3 .

rl3 :isStartingAt stp3; :isEndingAt stp4 .
```

For this kind of mapping in which the legacy standard does not include entities that are directly equivalent to ones in the IT2Rail ontology, Java classes can be manually produced as if the XSD files corresponding to the legacy data representation had those entities.

In the aforementioned example, we create a *RouteLink* class shown in Figure 14, and we add an attribute (*routeLinks*) and its corresponding getter/setter methods to class *Segment* mentioned above.

```java
@RdfsClass("infrastructure:RouteLink")
public class RouteLink
{

    @RdfProperty("it2r:isStartingAt")
    protected StopPlace isStartingAt;

    @RdfProperty("it2r:isEndingAt")
    protected StopPlace isEndingAt;

    public StopPlace getIsStartingAt() {
        return isStartingAt;
    }

    public StopPlace getIsEndingAt() {
        return isEndingAt;
    }

    public void setIsStartingAt(StopPlace value) {
        this.isStartingAt = value;
    }

    public void setIsEndingAt(StopPlace value) {
        this.isEndingAt = value;
    }

}
```

Figure 14: FSM RouteLink class

The transformations creating the new RouteLink concept take place in the getter/setter methods of class *Segment*, as shown in Figure 15. During lifting, the *stopPlace* list is duplicated (in attribute *allStopPlaces*) and the values of the *origin* and *destination* attributes are inserted at the beginning and the end of the list, respectively. After the ordered list *allStopPlaces* is created, an ordered list of *RouteLink* instances with proper *isStartingAt,* and *isEndingAt* attributes is produced. Accordingly, during lowering we have similar transformations. For example, *isStartingAt* of the first RouteLink in attribute *routeLinks* is mapped to *origin*, and *isEndingAt* of the last RouteLink in *routeLinks* is mapped to *destination.*

```
@RdfProperty(value = "st4rt:hasRouteLinks", isList = true)
public List<RouteLink> getRouteLinks() {
    if (routeLinks == null) {
        List<RouteLink> routeLinks1 = new ArrayList<RouteLink>();
        List<StopPlace> allStopPlaces = new ArrayList<StopPlace>();
        allStopPlaces.add(this.origin);
        if (this.stopPlace != null && this.stopPlace.size() > 0)
            allStopPlaces.addAll(this.stopPlace);
        allStopPlaces.add(this.destination);
        for (int i = 0 ; i < allStopPlaces.size() - 1;){
            RouteLink rl = new RouteLink();
            rl.setIsStartingAt(allStopPlaces.get(i));
            rl.setIsEndingAt(allStopPlaces.get(++i));
            routeLinks1.add(rl);
        }
        this.routeLinks = routeLinks1;
    }
    return this.routeLinks;
}
@XmlElement(name = "RouteLinks")
@RdfProperty(value = "st4rt:hasRouteLinks", isList = true)
public void setRouteLinks(List<RouteLink> value) {
    this.routeLinks = value;
    setOrigin(this.routeLinks.get(0).getIsStartingAt());
    setDestination(this.routeLinks.get(this.routeLinks.size() - 1).getIsEndingAt());
}
```

Figure 15: Transformation at java class level

4.2 Introduction of ad-hoc Concepts in the Intermediate Ontology

When the structure of the concepts in the legacy data representations does not match that in the reference ontology, a second possibility is to introduce suitable concepts in the ontology used for the transformation (see Figure 2), thus deviating from the reference ontology. In the case of the notion of Segment in the FSM standard, the intermediate ontology can be modified such that every element in Segment has its own equivalent to avoid complex mappings (in this case, property `hasIntermediateStopPlaces` is introduced to match the notion of list *stopPlace*. Then, as shown in Figure 16, *origin, stopPlace,* and *destination* are simply mapped to the corresponding properties by using the basic annotations of Section 3.

```
@XmlElement(name = "Origin")
@RdfProperty("travel:hasOrigin")
protected StopPlace origin;

@XmlElement(name = "StopPlace")
@RdfProperty(value = "st4rt:hasIntermediateStopPlaces", isList = true)
protected List<StopPlace> stopPlace;

@XmlElement(name = "Destination")
@RdfProperty("travel:hasDestination")
protected StopPlace destination;
```

Figure 16: Modification on the IT2Rail ontology

5. CONCLUSIONS AND FUTURE WORK

The annotations and mapping mechanisms presented in this document are the first step towards the creation of modules that can automatically convert messages conforming to legacy standards to/from a reference ontology.

The next steps that will be taken in the ST4RT project to make the creation of the conversion modules a reality are the following:

- A complete example of annotated Java classes concerning the FSM and TAP TSI standards will be produced, to validate and fine-tune the annotation mechanisms presented in this document.
- The intermediate ontology which is at the core of the transformation (as depicted in Figure 2) will be defined, which will possibly entail modifications to the reference Shift2Rail ontology.
- The detailed steps and algorithms of the mapping workflow will be defined; this might also lead to modifications and extensions to the annotation mechanisms, for example to optimize the mapping procedures from the point of view of the efficiency (e.g., to reduce the number of SPARQL queries to be run during the mapping process).

REFERENCES

[1] ETTSA, *Taking rail ticket distribution to the next level: railways and ticket vendors launch the 'Full Service Model' initiative*, Press Release, retrieved from http://www.ettsa.eu/uploads/Modules/Mediaroom/131001-press-release-full-service-model.pdf

[2] TAP TSI Services Governance Association (TSGA), *Full Service Model – Who we are and what we do*, https://tsga.eu/fsm

[3] UIC, *Telematics Applications for Passenger Services Technical Specifications for Interoperability* (TAP TSI), http://tap-tsi.uic.org

[4] ERA, *Telematics Applications for Passenger Services Technical Specifications for Interoperability* (TAP TSI), http://www.era.europa.eu/Document-Register/Pages/TAP-TSI.aspx

[5] W3C, *Semantic Annotations for WSDL and XML Schema*, http://www.w3.org/TR/sawsdl/