Contract No. H2020 – 826172

SEMANTICS FOR PERFORMANT AND SCALABLE INTEROPERABILITY OF MULTIMODAL TRANSPORT

D4.3 A lightweight solution to automate the generation of ontologies, mappings and annotations (F-REL)

Due date of deliverable: 30/06/2020

Actual submission date: 31/08/2020

Leader/Responsible of this Deliverable: CEF

Reviewed: Y

Document status		
Revision	**Date**	**Description**
0.1	20/05/2020	First draft of the table of contents
0.2	21/07/2020	Lifecycle management section
0.3	30/07/2020	Added XSD2OWL section
0.4	31/07/2020	Added Collaborative Ontology Engineering section
0.5	05/08/2020	Mapping automation section
0.6	06/08/2020	Asset Manager as NAP companion and Generic Converter sections
1.0	06/08/2020	Document ready to be reviewed
2.0	31/08/2020	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: 01/12/2018 Duration: 25 months

EXECUTIVE SUMMARY

This document describes the improvements in the F-Rel version of the SPRINT Interoperability Framework. We describe how our tools help establishing the Shift2Rail IP4 ecosystem, focusing on improving interoperability by providing automation in different areas:

- Collaborative ontology engineering
- Reuse of data schemas expressed as XML Schemas to help create ontologies
- Finding similarities and correspondences in different standards and specifications to create better mappings
- Accessing multiple data and metadata sources avoiding the necessity of moving data between TSPs
- Data and metadata sharing according to given governance processes
- Using shared data and metadata to automatically create Converters
- Automatically convert data in other formats using available Converters

The implementation of the features described in this document will then be documented in D5.5 and validated in D5.6.

ABBREVIATIONS AND ACRONYMS

Abbreviation	Description
API	Application Programming Interface
AVMS	Automatic Vehicle Monitoring
BPMN	Business Process Model and Notation
C-REL	Core Release
CCTV	Closed-circuit television
CRM	Customer Relationship Management
DCAT	Data Catalog Vocabulary
DNS	Domain Name System
DNS-SD	DNS Service Discovery
DPI	Dynamic passenger information
DRM	Driver Relationship Management
EBSF	European Bus System of the Future (EU-funded project)
EIF	European Interoperability Framework
EU	European Union
FMS	Vehicle Fleet Management System
FSM	Finished State Machine
FOAF	Friend of a friend is a machine-readable ontology
H2020	Horizon 2020 framework programme
HTTP	HyperText Transfer Protocol
IF	Interoperability framework
IP	Internet Protocol
IP4	Innovation Program 4
ISO	International Organization for Standardization
ITxPT	Information Technology for Public Transport
ITS	Information
JSON	JavaScript Object Notation

MaaS	Mobility as a Service
NAP	National Access Point
ORM	Object Relational Mapping
RDF	Resource Description Framework
REST	Representational State Transfer
S2R	Shift2Rail Joint Undertaking
SCXML	State Chart XML
SOAP	Simple Object Access Protocol
SPARQL	Protocol and RDF Query Language
WP	Work Package
XML	eXtensible Markup Language
XSD	XML Schema Definition
EIP	Enterprise Integration Pattern

TABLE OF CONTENTS

Executive Summary 2

Abbreviations and Acronyms 3

Table of Contents 5

List of Figures 6

List of Tables 6

1. Introduction 7

2. Automation helping the ontology, mappings and annotations development process 8

  2.1 Collaborative ontology engineering 8

  2.2 Automation in ontology development 11

  2.3 Automation in mapping and annotation creation 14

3. SPRINT Asset Manager and integrated support for automation 19

  3.1 Lifecycle and access request processes 20

    3.1.1 Lifecycle management 20

    3.1.2 Access request for assets 21

  3.2 Scalable Converter synthesis 22

  3.3 Asset Manager-Converter runtime integration 25

  3.4 Asset Manager as a companion of National Access Points 29

    3.4.1 Automating metadata aggregation from multiple NAPs 30

    3.4.2 Automating contributions to NAPs 33

4. Scalable/distributed RDF-based data access 35

5. Conclusions and next steps 37

6. References 38

LIST OF FIGURES

Figure 1 OnToology Workflow 9
Figure 2 Example of Jenkinsfile 9
Figure 3 Example of ontology repository with Jenkinsfile 10
Figure 4 Jenkins dashboard 10
Figure 5 version control systems support by OnToology 11
Figure 6 XSD2OWL architecture 12
Figure 7 Schema netex_facility_support 13
Figure 8 Ontology result from netex_facility_support 14
Figure 9 Basic lifecycle management process 21
Figure 10 Process to request the right to access an asset in the Asset Manager Store 21
Figure 11 Deployment and Service in Kubernetes 23
Figure 12 Converter Deployment Kubernetes Manifest 23
Figure 13 Converter Service Kubernetes Manifest 24
Figure 14 Converter interaction with the Mappings Resolver 26
Figure 15 Interaction between the Asset Manager and the Converter mediated by Resolvers 27
Figure 16 Definition of the assets required by a Converter 28
Figure 17 Simplified interaction between the Asset Manager and the Converter using Exploration API 29
Figure 18 Chimera pipeline used to convert metadata from a specific NAP and store it in the RDF repository 31
Figure 19 SPARQL query for the “get assets” Exploration API 33
Figure 20 EIP-based description of an IF-based publication flow 34
Figure 21 NAP-aware lifecycle process 35

LIST OF TABLES

Table 1 XML/XSD to Ontology (OWL) Translation Rules 16

1. INTRODUCTION

The establishment of an IP4 ecosystem requires solving many technical and organisational problems. In this document, we document our efforts in addressing some of those challenges and in offering ways to automate them.

An ecosystem fostering interoperability must take care of ensuring a consistent data modeling, which is a fundamental task which ensures a proper digital representation of the domain. This first task is addressed by offering a collaborative ontology engineering environment, which leverages on best practices in team-based source code editing and in automatic generation of ontology diagrams and documentation. This is complemented by automatic generation of ontology drafts using non-ontological sources, which allows reusing data models already prepared in other formats as the starting point for the ontology engineering work. Those topics will be described in Section 2.1 and Section 2.2.

To enable interoperability, the two ends of a communication channel must be able to understand the information coming from the other end. For a TSP, this means mapping the information coming from an external system to its own data model. Since the transportation domain is very complex and many standards and specifications are in use since years, the creation of mappings is a cumbersome task. In Section 2.3 we propose the F-Rel version of a tool to suggest possible mappings between standards and specifications, to lighten the burden of understanding the differences between two domain representations and finding correspondences.

Establishing an ecosystem means creating an environment where different actors can share information while maintaining sovereignty over their data. To this extent, the F-Rel version of the SPRINT Interoperability Framework described in Section 3 will feature a refinement of the Asset Manager, which is a tool to let all the actors of the ecosystem share data and metadata according to governance processes. The tool also shows how such data and metadata can be exploited to achieve better automation, providing automatic creation of Converters and automatic data conversion.

Once ontologies and mappings are created and shared, the only missing piece to obtain interoperability is a set of software artifacts to execute such mappings, actually transform messages and enact interoperability. In Section 3.3 we describe how our Chimera framework, which can be used to build Converters with a pipeline-based approach, can interact with the Asset Manager to download new mappings, ontologies and dataset, therefore providing a way to dynamically support many conversion processes in a single artifact.

2. AUTOMATION HELPING THE ONTOLOGY, MAPPINGS AND ANNOTATIONS DEVELOPMENT PROCESS

2.1 COLLABORATIVE ONTOLOGY ENGINEERING

The collaborative construction of ontologies has become a central paradigm of modern ontology engineering. This understanding of ontologies and ontology engineering processes is the result of intense theoretical and empirical research within the Semantic Web community. That is why in the context of Shift2Rail, collaborative development, it is generally recognized that, in order to be useful, but also economically viable, ontologies should be governed, developed and maintained in a community-led manner, with the help of comprehensive environments that provide dedicated support for collaboration and user participation. Wikis and similar communication and collaboration platforms that allow ontology stakeholders to exchange ideas and discuss modeling decisions are probably the most important technological components of such environments. In addition, process-based methodologies help the ontology engineering team throughout the ontology life cycle and provide best practices and guidelines to optimize the results of ontology development in real world transport projects.

To help in the collaborative construction of ontologies, OnToology is a proof-of-concept tool able to work with several types of version control systems (tested on platforms like GitHub, GitLab and Bitbucket), obtaining good results in the documentation generation and quality evaluation of the ontologies. OnToology applies mechanisms such as pipelines with continuous integration tools (e.g. Jenkins) where each user can create a task, add a configuration file (jenkinsfile) inside the repository where the ontology is located and automatically deploy all the workflow.

Figures 1-5 depict the workflow of OnToology and its corresponding outputs. Mainly, this workflow is divided in three parts:

Developers: Ontology developers who work in a collaborative mode.

Continuous Integration: which is basically OnToology, where are all the steps that will be executed to generate the documentation and evaluation of the ontologies.

Production: Corresponds to the continuous deployment.

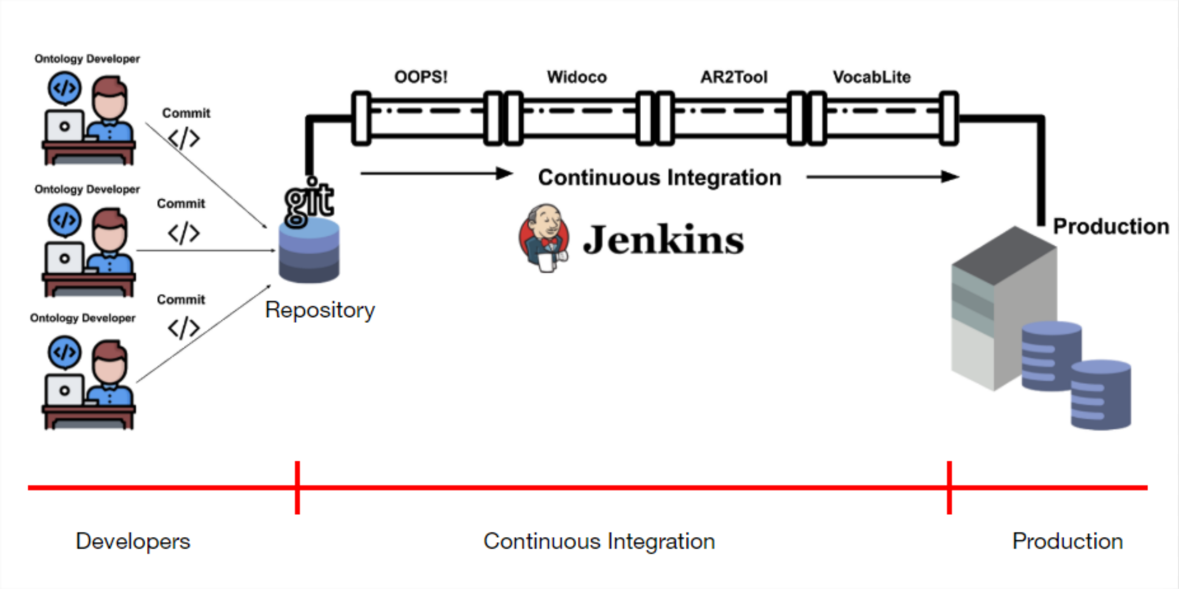
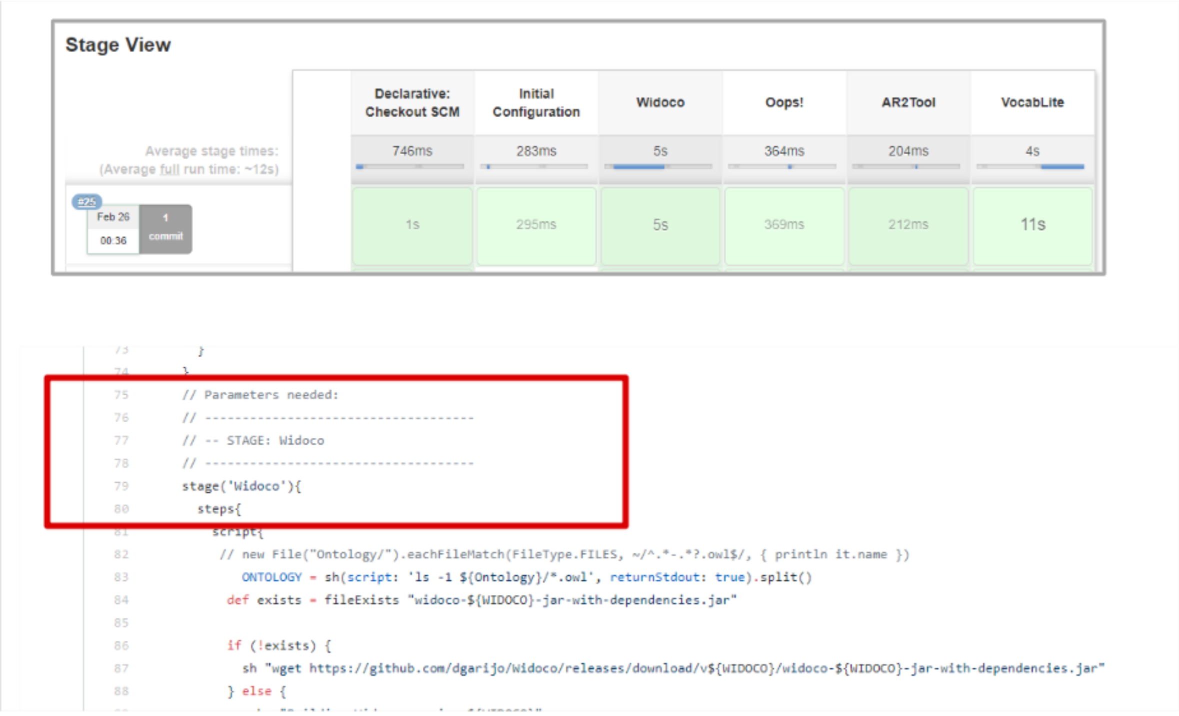


Figure 1 OnToology Workflow

Figure 2 shows a picture about the integration of different tools included in OnToology. For this example, a configuration file, called Jenkinsfile, has been defined in Figure 3. This file contains the definition of the pipeline and is checked into source control. This file specifies the stages for each tool which will be executed/deployed by the Jenkins server. An example of a stage for Widoco [1] is enclosed by a red box in Figure 2.



Stage View

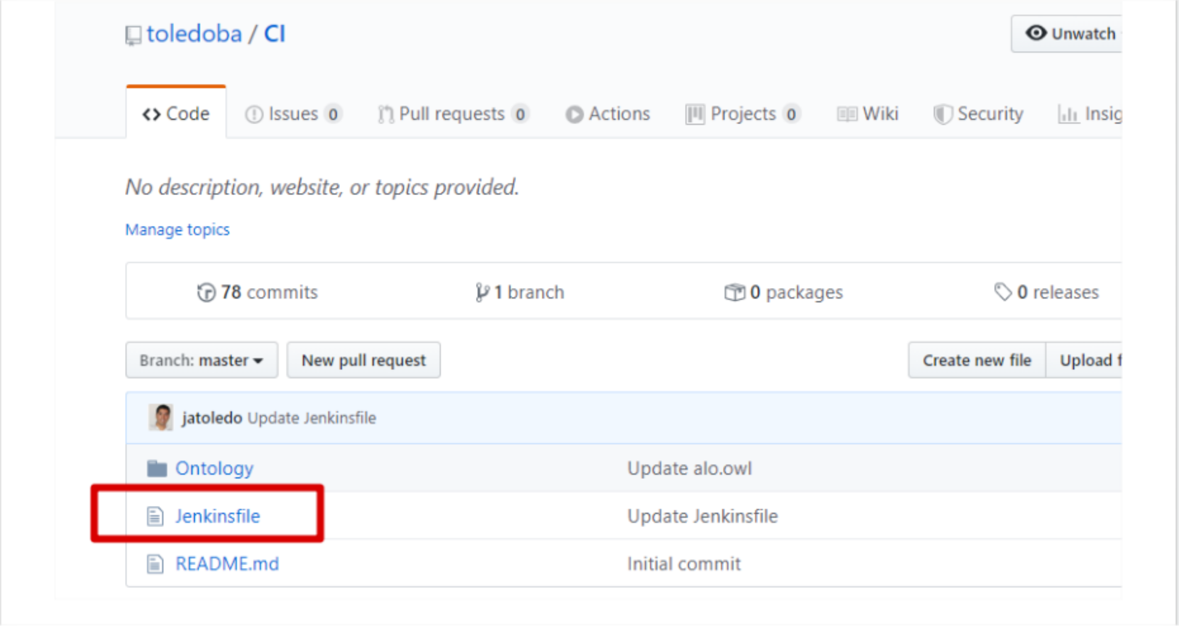
	Declarative: Checkout SCM	Initial Configuration	Widoco	Oops!	AR2Tool	VocabLite
Average stage times: (Average full run time: ~12s)	746ms	283ms	5s	364ms	204ms	4s
#25 Feb 26 00:36 1 commit	1s	295ms	5s	369ms	212ms	11s

```
      }
    }
    // Parameters needed:
    // ------------------------------------
    // -- STAGE: Widoco
    // ------------------------------------
    stage('Widoco'){
      steps{
        script{
          // new File("Ontology/").eachFileMatch(FileType.FILES, ~/^.*-.*?.owl$/, { println it.name })
            ONTOLOGY = sh(script: 'ls -1 ${Ontology}/*.owl', returnStdout: true).split()
          def exists = fileExists "widoco-${WIDOCO}-jar-with-dependencies.jar"

          if (!exists) {
            sh "wget https://github.com/dgarijo/Widoco/releases/download/v${WIDOCO}/widoco-${WIDOCO}-jar-with-dependencies.jar"
          } else {
```

Figure 2 Example of Jenkinsfile

In the current configuration of Figure 3, we have put the Jenkinsfile in the root of our repository and the directory where the ontology is found is called "Ontology". This configuration can be modified, but we highly recommend everyone to use the same structure.



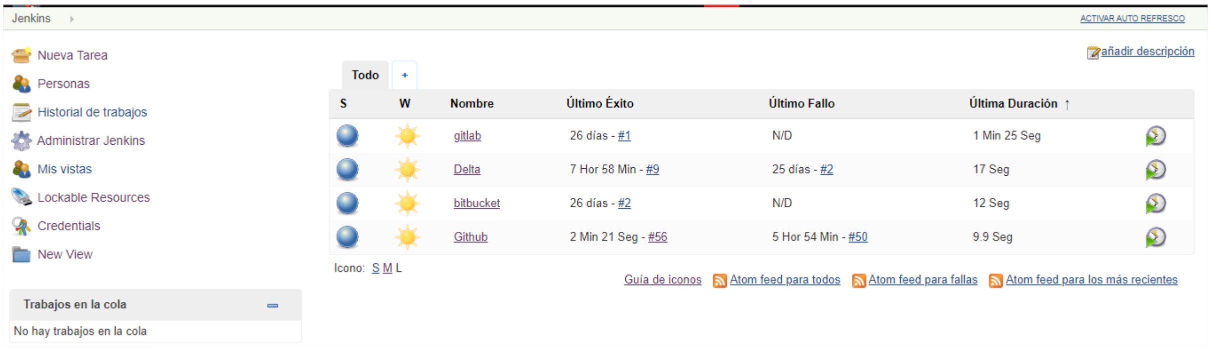
No description, website, or topics provided.

jatoledo Update Jenkinsfile

Ontology	Update alo.owl
Jenkinsfile	Update Jenkinsfile
README.md	Initial commit

Figure 3 Example of ontology repository with Jenkinsfile

The dashboard shown in Figure 4 integrates the ontologies, and the names of repositories are names that identify our ontologies.



S	W	Nombre	Último Éxito	Último Fallo	Última Duración ↑	
●	☀	gitlab	26 días - #1	N/D	1 Min 25 Seg	
●	☀	Delta	7 Hor 58 Min - #9	25 días - #2	17 Seg	
●	☀	bitbucket	26 días - #2	N/D	12 Seg	
●	☀	Github	2 Min 21 Seg - #56	5 Hor 54 Min - #50	9.9 Seg	

Trabajos en la cola

No hay trabajos en la cola

Figure 4 Jenkins dashboard

Finally, as you can observe in Figure 5, our pipeline only depends on Git version control system; therefore it could work with any version control platform.

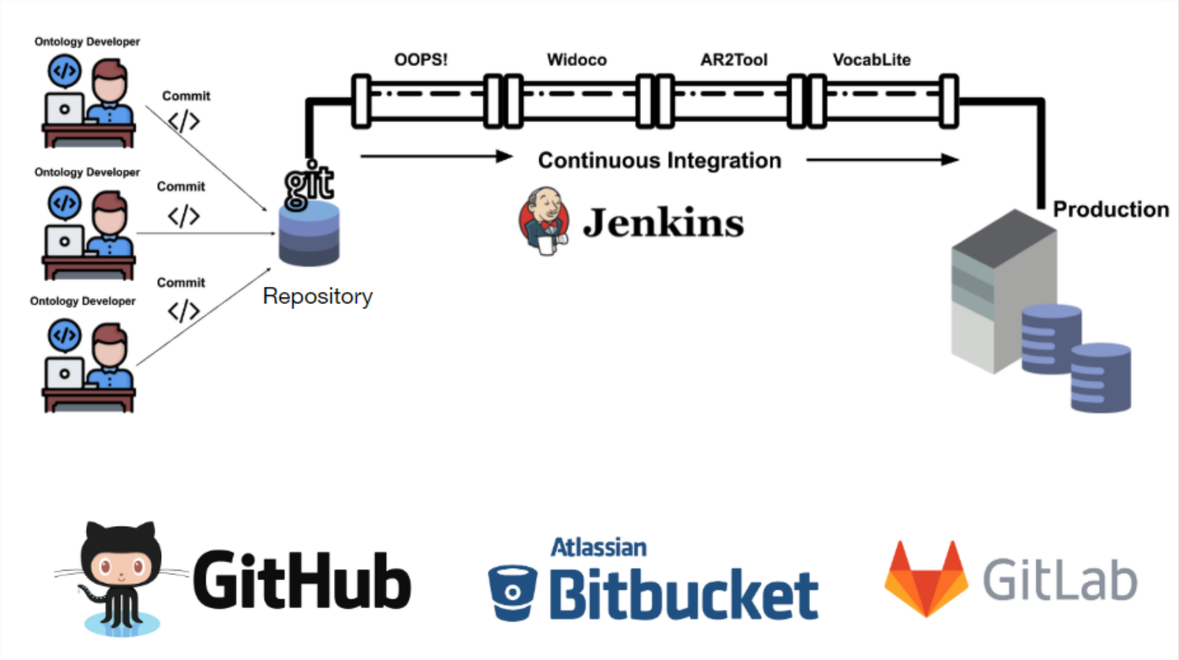


Figure 5 version control systems support by OnToology

2.2 AUTOMATION IN ONTOLOGY DEVELOPMENT

As in many other domains, in the transport domain, much of the data is in XML (Extensible Markup Language) format. However, the use of XML requires that applications interpret the format of each data source they access to transform XML into OWL (Ontology Web Language) if we want to semantically represent data. Generally, this process of transformation from XML to OWL has been performed manually or using semi-automatic components. In the context of Shift2Rail, we want to automatically generate conceptual models from semi-structured models. The automation of ontology development from existing XML Schemas can speed up and simplify the match and merge processes with S2R ontologies. In this section, we introduce a tool called XSD2OWL which allows the automatic transformation from the XML Schema to OWL by means of the integration of many XML data. For F-REL, we will focus on transforming representations of the XML schema components of NeTEx[1] and GTFS bench XSD[2].

XSD2OWL can be applied for XML semantics reuse and it is based on mapping from XML Schema constructs to the OWL ones that are semantically more appropriate. XML schemas are used in grammars as the source from which the semantics they capture implicitly are going to be formalized and made explicit. In general, the transfer of XML metadata to the ontology is not made explicit when XML metadata instantiating these schemas is mapped.

---

[1] https://github.com/NeTEx-CEN/NeTEx

[2] https://github.com/jatoledo/xsd2owl/tree/master/GTFS_XSD

It is important to note that XSD2OWL is an extension of Ontmalizer[3]. For more details about Ontmalizer please refer to SPRINT deliverable D4.2.

Figure 6 shows the workflow that follows XSD2OWL illustrating a real use case on the original NeTEx sources. This workflow can be applied on other sources such as GTFS-Madrid-Bench XML.

The XSD2OWL workflow consists of the following processes:

Mapping XSD Files. Once the NeTEx XSD file has been selected by the user, XSD2OWL syntactically analyzes this file to create tokens which will be used in the next processes.

Extracting simple types. XSD2OWL gets simple types from the tokens previously received in the first process and it converts them to OWL constructs.

Extracting complex types & elements. XSD2OWL gets complex types and elements from the tokens previously received in the first process and it converts them to OWL constructs. The OWL file is finally produced in this process.

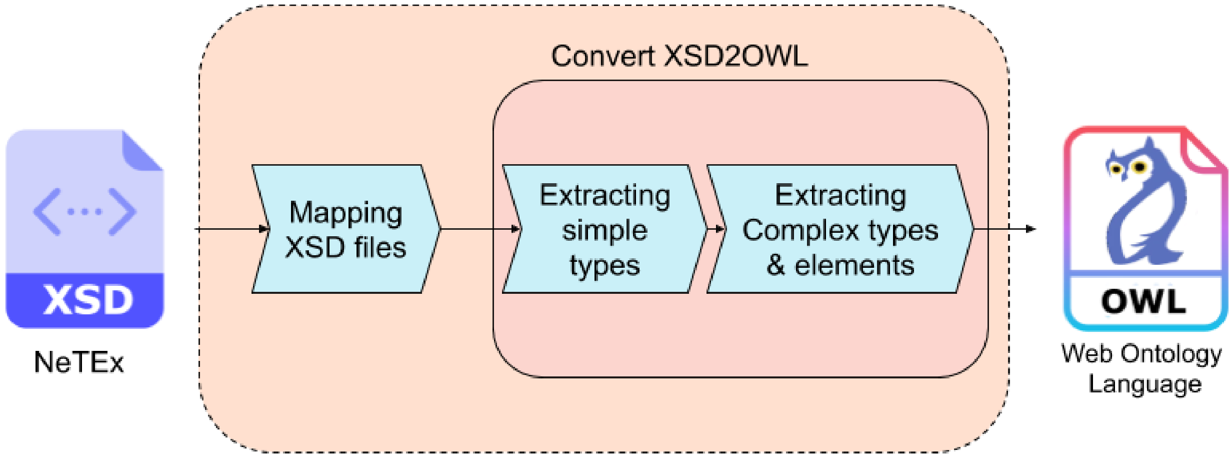
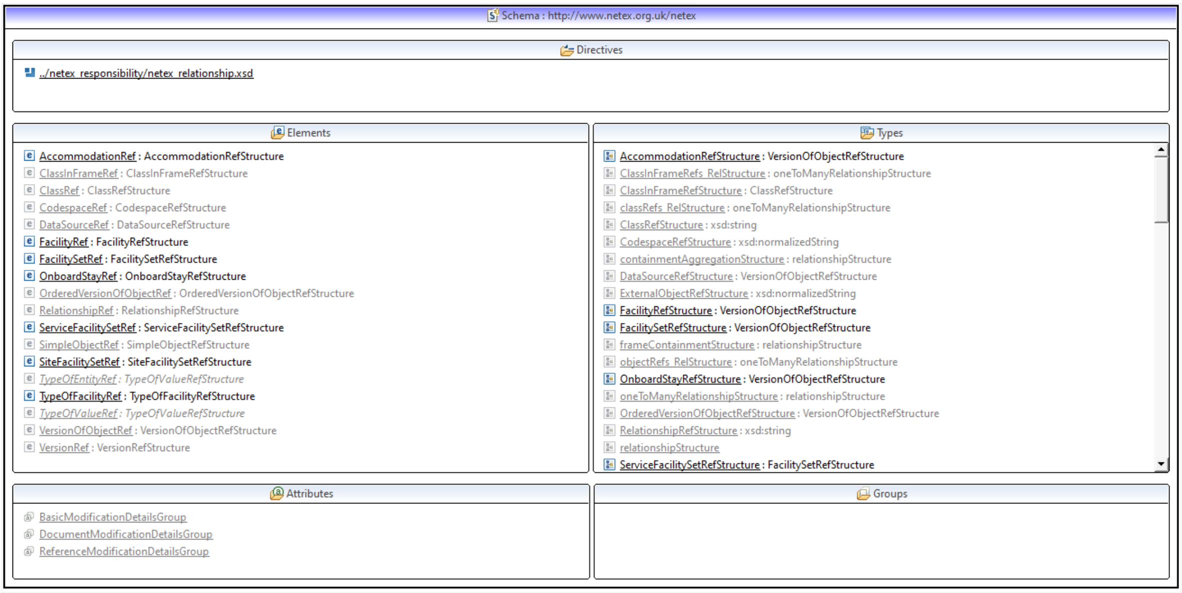


Figure 6 XSD2OWL architecture

We can observe in Figure 7 an example of a NeTEX Schema file which will be transformed by XSD2OWL.

---

[3] https://github.com/srdc/ontmalizer



../netex_responsibility/netex_relationship.xsd

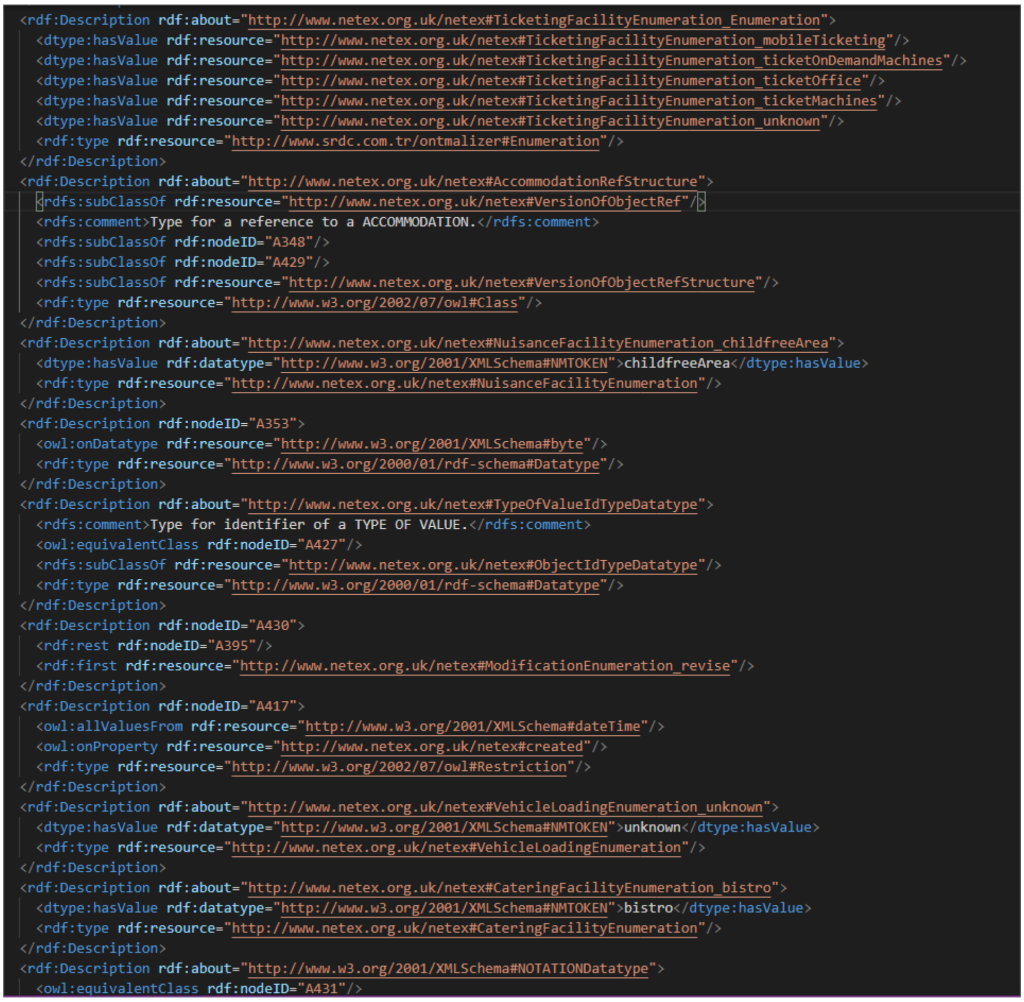
AccommodationRef : AccommodationRefStructure
ClassInFrameRef : ClassInFrameRefStructure
ClassRef : ClassRefStructure
CodespaceRef : CodespaceRefStructure
DataSourceRef : DataSourceRefStructure
FacilityRef : FacilityRefStructure
FacilitySetRef : FacilitySetRefStructure
OnboardStayRef : OnboardStayRefStructure
OrderedVersionOfObjectRef : OrderedVersionOfObjectRefStructure
RelationshipRef : RelationshipRefStructure
ServiceFacilitySetRef : ServiceFacilitySetRefStructure
SimpleObjectRef : SimpleObjectRefStructure
SiteFacilitySetRef : SiteFacilitySetRefStructure
TypeOfEntityRef : TypeOfValueRefStructure
TypeOfFacilityRef : TypeOfFacilityRefStructure
TypeOfValueRef : TypeOfValueRefStructure
VersionOfObjectRef : VersionOfObjectRefStructure
VersionRef : VersionRefStructure

AccommodationRefStructure : VersionOfObjectRefStructure
ClassInFrameRefs_RelStructure : oneToManyRelationshipStructure
ClassInFrameRefStructure : ClassRefStructure
classRefs_RelStructure : oneToManyRelationshipStructure
ClassRefStructure : xsd:string
CodespaceRefStructure : xsd:normalizedString
containmentAggregationStructure : relationshipStructure
DataSourceRefStructure : VersionOfObjectRefStructure
ExternalObjectRefStructure : xsd:normalizedString
FacilityRefStructure : VersionOfObjectRefStructure
FacilitySetRefStructure : VersionOfObjectRefStructure
frameContainmentStructure : relationshipStructure
objectRefs_RelStructure : oneToManyRelationshipStructure
OnboardStayRefStructure : VersionOfObjectRefStructure
oneToManyRelationshipStructure : relationshipStructure
OrderedVersionOfObjectRefStructure : VersionOfObjectRefStructure
RelationshipRefStructure : xsd:string
relationshipStructure
ServiceFacilitySetRefStructure : FacilitySetRefStructure

BasicModificationDetailsGroup
DocumentModificationDetailsGroup
ReferenceModificationDetailsGroup

Figure 7 Schema netex_facility_support

The file in Figure 7 was transformed to OWL file and the Figure 8 depicts the output generated by XSD2OWL



```
<rdf:Description rdf:about="http://www.netex.org.uk/netex#TicketingFacilityEnumeration_Enumeration">
  <dtype:hasValue rdf:resource="http://www.netex.org.uk/netex#TicketingFacilityEnumeration_mobileTicketing"/>
  <dtype:hasValue rdf:resource="http://www.netex.org.uk/netex#TicketingFacilityEnumeration_ticketOnDemandMachines"/>
  <dtype:hasValue rdf:resource="http://www.netex.org.uk/netex#TicketingFacilityEnumeration_ticketOffice"/>
  <dtype:hasValue rdf:resource="http://www.netex.org.uk/netex#TicketingFacilityEnumeration_ticketMachines"/>
  <dtype:hasValue rdf:resource="http://www.netex.org.uk/netex#TicketingFacilityEnumeration_unknown"/>
  <rdf:type rdf:resource="http://www.srdc.com.tr/ontmalizer#Enumeration"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.netex.org.uk/netex#AccommodationRefStructure">
  <rdfs:subClassOf rdf:resource="http://www.netex.org.uk/netex#VersionOfObjectRef"/>
  <rdfs:comment>Type for a reference to a ACCOMMODATION.</rdfs:comment>
  <rdfs:subClassOf rdf:nodeID="A348"/>
  <rdfs:subClassOf rdf:nodeID="A429"/>
  <rdfs:subClassOf rdf:resource="http://www.netex.org.uk/netex#VersionOfObjectRefStructure"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.netex.org.uk/netex#NuisanceFacilityEnumeration_childfreeArea">
  <dtype:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#NMTOKEN">childfreeArea</dtype:hasValue>
  <rdf:type rdf:resource="http://www.netex.org.uk/netex#NuisanceFacilityEnumeration"/>
</rdf:Description>
<rdf:Description rdf:nodeID="A353">
  <owl:onDatatype rdf:resource="http://www.w3.org/2001/XMLSchema#byte"/>
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Datatype"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.netex.org.uk/netex#TypeOfValueIdTypeDatatype">
  <rdfs:comment>Type for identifier of a TYPE OF VALUE.</rdfs:comment>
  <owl:equivalentClass rdf:nodeID="A427"/>
  <rdfs:subClassOf rdf:resource="http://www.netex.org.uk/netex#ObjectIdTypeDatatype"/>
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Datatype"/>
</rdf:Description>
<rdf:Description rdf:nodeID="A430">
  <rdf:rest rdf:nodeID="A395"/>
  <rdf:first rdf:resource="http://www.netex.org.uk/netex#ModificationEnumeration_revise"/>
</rdf:Description>
<rdf:Description rdf:nodeID="A417">
  <owl:allValuesFrom rdf:resource="http://www.w3.org/2001/XMLSchema#dateTime"/>
  <owl:onProperty rdf:resource="http://www.netex.org.uk/netex#created"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.netex.org.uk/netex#VehicleLoadingEnumeration_unknown">
  <dtype:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#NMTOKEN">unknown</dtype:hasValue>
  <rdf:type rdf:resource="http://www.netex.org.uk/netex#VehicleLoadingEnumeration"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.netex.org.uk/netex#CateringFacilityEnumeration_bistro">
  <dtype:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#NMTOKEN">bistro</dtype:hasValue>
  <rdf:type rdf:resource="http://www.netex.org.uk/netex#CateringFacilityEnumeration"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.w3.org/2001/XMLSchema#NOTATIONDatatype">
  <owl:equivalentClass rdf:nodeID="A431"/>
```

Figure 8 Ontology result from netex_facility_support

2.3 AUTOMATION IN MAPPING AND ANNOTATION CREATION

Heterogeneous data sources are the main threat to interoperability in the transportation domain (For more detailed discussion See D2.2, D2.3 and D3.2). The multitude of transportation actors are generating and operating upon various types of data represented in a wide range of data models, vocabularies, format and standards. To overcome this challenge, SPRINT has developed Converter technology that can seamlessly transform the desired data from one standard/representation to another. In this direction, the Mapping Tool has been developed to automate the mapping process and has been introduced in SPRINT deliverable D4.2. The main product of Mapping Tool, which constitutes one of the essential inputs to the SPRINT converter, is a set of “mappings”. The so-called mappings are a one-to-one translation of concepts/terms in one standard to their equivalents in the other standard.

The first version of the Mapping Tool has been constructed merely on the basis of the semantic mapping of the concepts. The core design idea was to identify the semantically similar terms in two standards (source and target standard) using natural language processing and machine learning algorithms and technologies. Details of the algorithm and procedure of such mapping have been presented in deliverable D4.2 and the results of the validation of the first deployable Mapping Tool software are reported in deliverable D5.3.

In the rest of this section we introduce an improved version of the algorithm that exploits the structure of data in the source and target standards to further drive a new and more precise set of mappings. The heuristic behind the algorithm is as follows:

> *"There is a higher probability that two terms in two different standards refer to an identical concept if their syntactical positions in the structure of such standards are also equivalent in addition to the semantic of the terms."*

In this direction, the algorithm keeps track of the position of each concept in a structure and includes it in the similarity calculation, in addition to the semantics of the term. More precisely, only the terms with similar syntactic positions would undergo the natural language processing through Word2Vec technique. For example, ontologies represent knowledge as a set of Classes and the relations (properties) among them. Hence, the basic elements in an ontology are Classes (to which individuals belong) and object and datatype Properties, which correspond to binary relations between the elements of the domain (i.e., instances of classes) and those of their range (i.e., instances of classes or of datatypes). Accordingly, if the source (O1) and target (O2) standards are both ontologies (in OWL format for instance), the algorithm extracts all Classes, Datatype and Object Properties in each ontology (where C1 and P1 stand for the set of all Classes and Properties in O1, whereas C2 and P2 stand for the set of all Classes and Properties in O2). It then exclusively matches[4] the equivalent structural elements to each other; that is, C1 would be matched against C2 and P1 against P2.

As shown in the above example, the algorithm follows an intuitive flow when the mapping takes place between standards with the same format since the structure is identical. The main challenge, however, arises when the source and target standards come from two different origins, since in such cases a "structural mapping" is also required. To this end, the algorithm is relying on a predefined set of "translation rules" that states which syntactical positions in the two formats should be considered as corresponding. The most widely used knowledge representation formats in the transportation domain are ontologies (captured in OWL, ttl, etc. files) and XML/XSDs formats. Accordingly, the core of our algorithm focuses on translation rules between XML/XSD files and Ontologies and vice versa.

Our research in this field can be considered as a branch of the broader research domain concerned with the automation of ontology management and aimed at the generation of ontologies from non-ontological sources such as XML/XSD (see for example section 3.2 of D4.2). However, in our case the translation rules do not need to be overly sophisticated and accurate, since we only use them as a heuristic to unearth similarities among concepts and not to actually construct new ontologies from scratch. Accordingly, as listed in Table 1, our XML/XSD to Ontology translation rules are a simplified version of the state-of-the-art translation rules used in ontology generation.

---

[4] By matching we are referring to the whole process of extracting the semantic similarity using word2vec.

Table 1 XML/XSD to Ontology (OWL) Translation Rules

XSD	OWL	
	Type	Name
<xsd:complexType name=**"A"**> <xsd:complexContent> <xsd:extension base=**"B"**> *Where B is another Complex Element*	Class	**A**
	is SubClassof:	**B**
<xsd:complexType name=**"A"**> <xsd:complexContent> <xsd:extension <xsd:element name=**"E1"** type= **"B"** > *Where B is another Complex Element*	Object Property	**has_E1**
	Domain (Class)	**A**
	Range (Class)	**B**
<xsd:complexType name=**"A"**> <xsd:complexContent> <xsd:extension <xsd:attribute name=**"Atr1"** type= **"B"** > *Where B is another Complex Element*	Object Property	**has_Atr1**
	Domain(Class)	**A**
	Range(Class)	**B**
<xsd: complexType name=" **A">** <attribute name=**"Atr1"** type="**D**"/> *Where D is a DataType*	DataType Property	**has_ Atr1**
	Domain (Class)	**A**
	Range (Data Type)	**D**
<xsd:complexType name=**"A"**> <xsd:complexContent> <xsd:extension <xsd:element name=**"E1"** type= **"D"** > *Where D is a DataType*	DataType Property	**has_E1**
	Domain (Class)	**A**
	Range (Data Type)	**D**

The revised proposed algorithm for the generation of suggested mappings between terms of different standards is the following.

```
Input:     X: XSD file
           O: OWL file
Output:    P: set of pairs ⟨xt, ot⟩ of terms (where xt ∈ X and ot ∈ O)

Procedure:

1.   X_CT_names = ∅
     X_EA_names = ∅
     X_EA_types = ∅
2.   foreach item i ∈ X
3.       if i is name of ComplexType
             X_CT_names = X_CT_names + i
         else if i is name of element or attribute of ComplexType
             X_EA_names = X_EA_names + i
         else if i is name of type of element or attribute
             X_EA_types = X_EA_types + i
4.   X_cand_classes = X_CT_names ∪ X_EA_types
     X_cand_obj_props = ∅
     X_cand_dtype_props = ∅
5.   foreach p ∈ X_EA_names
6.       if range of p is ComplexType
             X_cand_obj_props = X_cand_obj_props + p
         else if range of p is datatype
             X_cand_dtype_props = X_cand_dtype_props + p
7.   O_classes = ∅
     O_obj_props = ∅
     O_dtype_props = ∅
8.   foreach class c ∈ O
         O_classes = O_classes + c
9.   foreach object property op ∈ O
         O_obj_props = O_obj_props + op
10.  foreach datatype property dp ∈ O
         O_dtype_props = O_dtype_props + dp
11.  mapped_classes = word2vec_mapping(X_cand_classes, O_classes)
12.  mapped_obj_props =
             word2vec_mapping(X_cand_obj_props, O_obj_props)
13.  mapped_dtype_props = word2vec_mapping(X_cand_dtype_props, O_dtype_props)
14.  foreach ⟨x_p, o_p⟩ ∈ mapped_obj_props
```

```
          let   ctd be the ComplexType to which x_p belongs
                d be the domain of o_p
                ctr be the (ComplexType) type of  x_p
                r be the range of o_p
          mapped_obj_props = mapped_obj_props ∪ {⟨ctd, d⟩, ⟨ctr, r⟩}
15.  foreach ⟨x_p, o_p⟩ ∈ mapped_dtype_props
          let   ctd be the ComplexType to which x_p belongs
                d be the domain of o_p
                ctr be the (datatype) type of  x_p
                r be the range of o_p
          mapped_dtype_props = mapped_dtype_props ∪ {⟨ctd, d⟩, ⟨ctr, r⟩}
16.  foreach ⟨x_c1, o_c1⟩ ∈ mapped_classes
          foreach ⟨x_c2, o_c2⟩ ∈ mapped_classes
             let X_props = { x_p |   x_p ∈ X_cand_obj_props ∪ X_cand_dtype_props
                                     and
                                     x_c1 is the ComplexType to which x_p belongs
                                     and
                                     x_c2 is the type of x_p }
                 O_props = { o_p |   o_p ∈ O_obj_props ∪ O_dtype_props
                                     and
                                     o_c1 is the domain of o_p
                                     and
                                     o_c2 is the range of o_p }
             foreach x_p ∈ X_props
                foreach o_p ∈ O_props
                   if o_p ∈ O_obj_props
                        mapped_obj_props = mapped_obj_props ∪ + ⟨x_p, o_p⟩
                   else
                        mapped_dtype_props = mapped_dtype_props ∪ + ⟨x_p, o_p⟩
17. return mapped_classes ∪ mapped_obj_props ∪ mapped_dtype_props
```

The algorithm takes as input a pair of files (an XSD file and an OWL file), which are the standards to be mapped to one another. First (steps 1-6) it goes through the terms of the XSD file, and it builds three sets of terms: those that are candidates to be mapped to OWL classes (X_cand_classes); those that are candidates to be mapped to OWL object properties (X_cand_obj_props); and those that are candidates to be mapped to OWL datatype properties (X_cand_dtype_props). To do this, it essentially applies the rules of Table 1. Then, it goes through the terms of the OWL file, and it retrieves three other sets (lines 7-10): the one with the names of the OWL classes (O_classes), the one with the names of the OWL object properties (`O_obj_props`), and the one with the names of the OWL datatype properties (`O_dtype_props`).

Then, it separately applies the word2vec technique described in Deliverable D4.2 to three pairs of sets of terms: those containing the (candidate) classes (line 11), those containing the (candidate) object properties (line 12), and those containing the (candidate) datatype properties (line 13).

In steps 14-16 the mappings returned by the word2vec-based algorithm are further enriched using the structure of the OWL ontology as guidance. More precisely, for each pair of properties x_p (from the XSD file), o_p (from the OWL file), also their domains and ranges are mapped to one another (steps 14 and 15, where step 14 focuses on object properties, and step 15 on datatype properties). In addition, if there are 4 elements, x_c1, x_c2 (from the XSD file), o_c1, o_c2 (from the OWL file) such that (i) x_c1 and o_c1 (resp., x_c2 and o_c2) have already been mapped to one another, (ii) there is a candidate property x_p in the XSD file such that x_c1 is the ComplexType to which x_p belongs and x_c2 is the type of x_p, and (iii) there is a property o_p in the OWL file such that o_c1 and o_c2 are, respectively, the domain and range of o_p, then x_p and o_p are mapped to one another.

Finally, line 17 returns all identified mappings, where each mapping is a pair of terms, one from the XSD file, and one from the OWL file.

The algorithm presented above is the core of the Mapping Tool that will be deployed as part of the demonstration platform being developed in WP5. The tool will be based on the mapping suggestions created through the mechanism described above, and it will allow users to revise them, confirm them if they are suitable, modify them if necessary, and then generate annotations to be used by the converter technology that is being developed within the SPRINT project.

3. SPRINT ASSET MANAGER AND INTEGRATED SUPPORT FOR AUTOMATION

The F-Rel version of the Asset Manager will provide a set of new features, as well as fixes and improvements in the user interface. In this section we will describe the new features related to automation. C-Rel demonstrated that the Asset Manager is not bound to the simple role of being a catalogue of assets. Lifecycle management integrated with a continuous integration and deployment tool enables reacting to changes in assets, and therefore enables the usage of the Asset Manager as a “command and control center” for an IF-based ecosystem. Some of the actions which can be implemented as a “reaction” to publishing an asset indeed are:

- deploying a service onto a cloud platform;
- instructing a monitoring tool to observe the behavior of a remote resource;
- automatically generating documentation for data models, ontologies or services;
- tracking dependencies to ensure that new versions of data models, ontologies or services do not cause disruptions in other services.

In this section we will describe two new possible ways to exploit the Asset Manager and its automation features, and we will describe improvements in lifecycle management and in the generation of scalable artifacts to be deployed onto a cloud platform.

We will show how we can build a Converter which dynamically exploit the Asset Manager to obtain new mappings and therefore to cover a potentially unlimited number of format transformations, and we will design how the Asset Manager can become a companion to the National Access Points, which are becoming mandatory in the European transportation domain.

3.1 LIFECYCLE AND ACCESS REQUEST PROCESSES

3.1.1 Lifecycle management

The Asset Manager is an application which can enforce a given governance process. To this extent, together with CONNECTIVE, we agreed on a basic governance scheme and process to be tested during F-Rel. The governance structure proposed by CONNECTIVE divides the users of the AM into three main groups: Consumers, Contributors and Administrators. A Contributor is the expected user of the Asset Manager Publisher application, which as the name suggests allows publishing new assets. A Consumer is instead the main user of the Asset Manager Store, which allows accessing information about assets. The Administrators are in charge of maintaining the IP4 ecosystem, built using the IF, and therefore must assess the quality of the information in assets and ensure that the overall ecosystem is “stable”. By CONNECTIVE request, the Administrator group is then split into several sub-groups, each one responsible of a specific asset type. The motivation behind this request is that different asset type serve different purposes inside the IP4 ecosystem, and therefore their contents must be assessed by different people. As an example, the publication of a “Journey planning” asset (whose content can be a GTFS file) could trigger via an automation job an update in the Meta network, and therefore the group responsible for the Meta network maintenance should decide whether to approve such publication.

Together with CONNECTIVE, we decided to draw a BPMN process to model the lifecycle of a generic asset. This means that in the F-Rel version of the Asset Manager this will become the default process to manage publishing, and that it will become the base process for further customizations related to specific asset types.

When the contributor asks for the publication of an asset, the Asset Manager locks the asset information disallowing further modifications and sends an approval request to the administrators of the specific asset type. The asset state is then changed according to the administrators’ decision. If the publication has been rejected, the author of the asset is notified (both inside the application and via email) and the asset editing is “unlocked”, allowed further interventions by the asset author. If otherwise the asset publication has been allowed, the metadata is sent to the RDF repository and all the automation jobs linked to the specific asset type are started. As last steps, the asset editing is “unlocked” and the asset author receives a notification (both inside the application and via email) about the successful publication. Such process is shown as a BPMN diagram in Figure 9, and will be implemented for F-Rel.

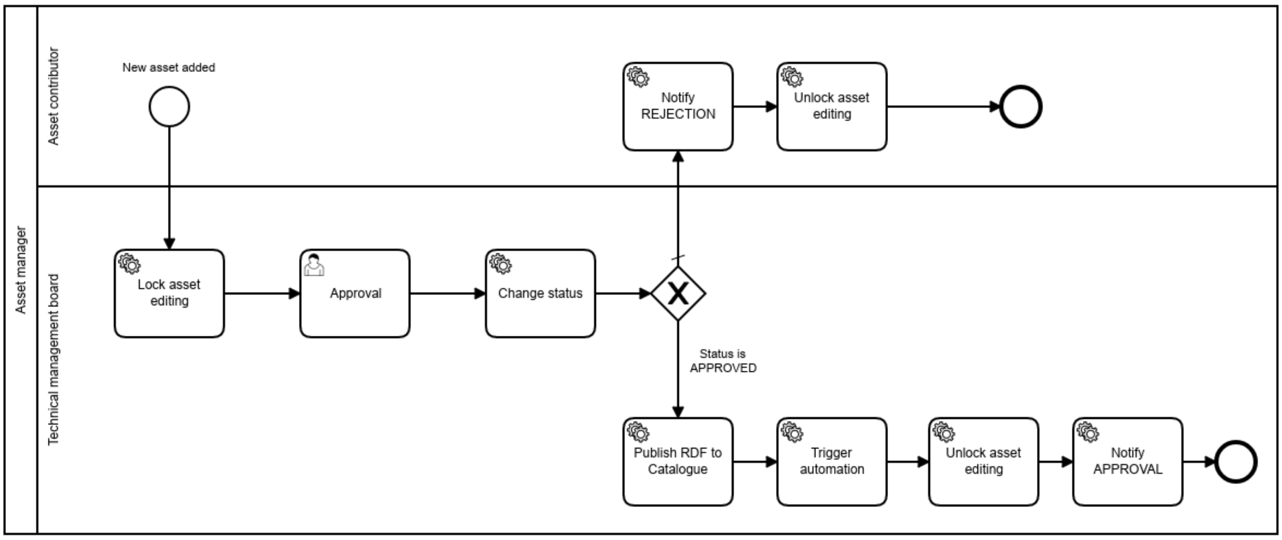


Figure 9 Basic lifecycle management process

3.1.2 Access request for assets

As a default policy for accessing information about assets in the Asset Manager Store, we decided that the basic information about all assets is public and visible to all users. Such information just states that an asset “exists”, and that it has been published by a specific TSP at a certain point in time. We don’t disclose any other information in public. If an Asset Manager Store user wants to access the full set of metadata of an asset, together with its attachments, he needs to explicitly ask to the asset owner. Such access request is performed according to the BPMN process described in Figure 10. The request is sent to the asset owner via the Asset Manager Publisher application. If the request is allowed, then the user is authorized and is notified (both via the Store and via email) that he can access the asset information. Otherwise, he is just notified (again, both via the Store and via email) that the access request has been rejected.

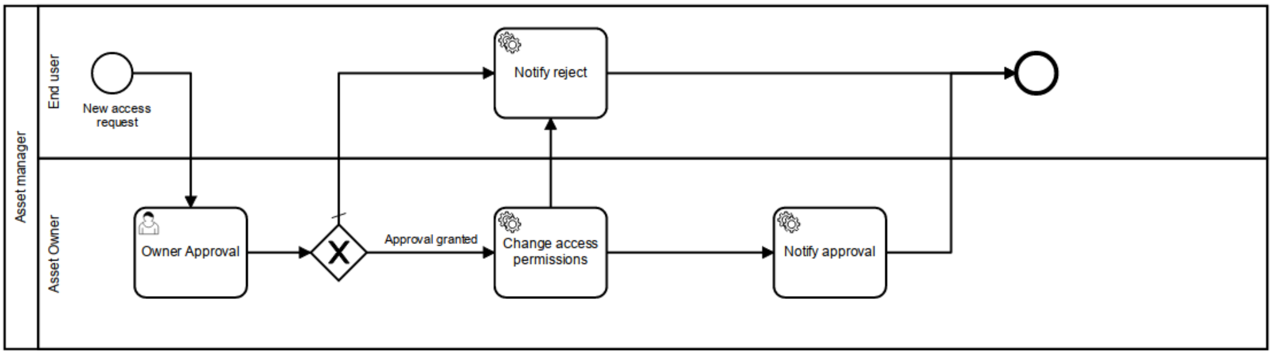


Figure 10 Process to request the right to access an asset in the Asset Manager Store

3.2 SCALABLE CONVERTER SYNTHESIS

The C-Rel version of the Asset Manager supported the automatic creation of Converters, given their description in terms of required ontologies, datasets and mappings. The Asset Manager, via its integrated continuous integration/delivery component (Jenkins), is able to gather all required files, configure a Converter, package it and “attach” it to the Converter asset. The Docker image of the Converter is made available to the user together with a Docker Compose file that allows to easily deploy one Converter in a Container Runtime Environment on a machine. In C-Rel we demonstrated the scalability features of the proposed approach. The user was able to manually scale the Converter on a single machine by exploiting replicas of the Converter container, while a reverse proxy was created to equally distribute the requests to the replicas running on the machine.

For F-Rel, we continue to explore the possibility of delivering scalable artifacts exploiting Kubernetes features. Modern software architectures are composed by a set of (micro)services running on containers that interacts among them to implement the application logic. To facilitate the deployment and management of these type of architectures, container orchestrators have been developed. Kubernetes is a cloud orchestrator initially developed by Google in 2014, and its role is to manage containers running on a cluster composed of multiple nodes. Kubernetes specification defines multiple abstractions that can be used by the developer to define the desired deployment in a declarative way leaving to the orchestrator the responsibility of reaching and maintaining the declared state. The Asset Manager, in the F-Rel release, will automatically also generate the Kubernetes manifests to deploy the Converter on a Kubernetes cluster taking advantage of its features, in particular considering scalability.

Porting the Converter onto a Kubernetes environment is an activity which reuses all the previous work carried out in the context of C-Rel. A Pod is the implementation-unit in a Kubernetes cluster usually running one container. The Docker container specification, which is already generated by the automatic Converter synthesis, is the starting point for obtaining a Kubernetes configuration to run a Pod. The same process which generate the Docker Compose package can be therefore extended to generate both a Docker Compose configuration and a complete Kubernetes configuration.

Figure 11 represents a Kubernetes cluster composed of two nodes. The first abstraction needed to configure the Converter on Kubernetes is the *Deployment.* A *Deployment* declares the desired state for a set of Pods defining, in particular, the number of replicas of the Pod that should be deployed. In the first step in the Figure 11, a *Deployment* declaring one replica of the Pod is installed on the cluster and the orchestrator deploys one Pod in one of the two nodes. The number of replicas of the *Deployment* can be changed at any time and the orchestrator takes care of the required actions to reach the target number scaling up or down the number of Pods deployed.

In Figure 12, we reported an example manifest for a *Deployment* of the Converter with one replica. The manifest defines the containers running in a replica of the Pod, in this case the *repository/chimer-example* image, the amount of resources (memory and CPU) required in the node where the Pod replica is the deployed and the maximum amount of resources that are made available to a Pod replica. In the example, the Pod running the Converter exposes its interface on port 8888.

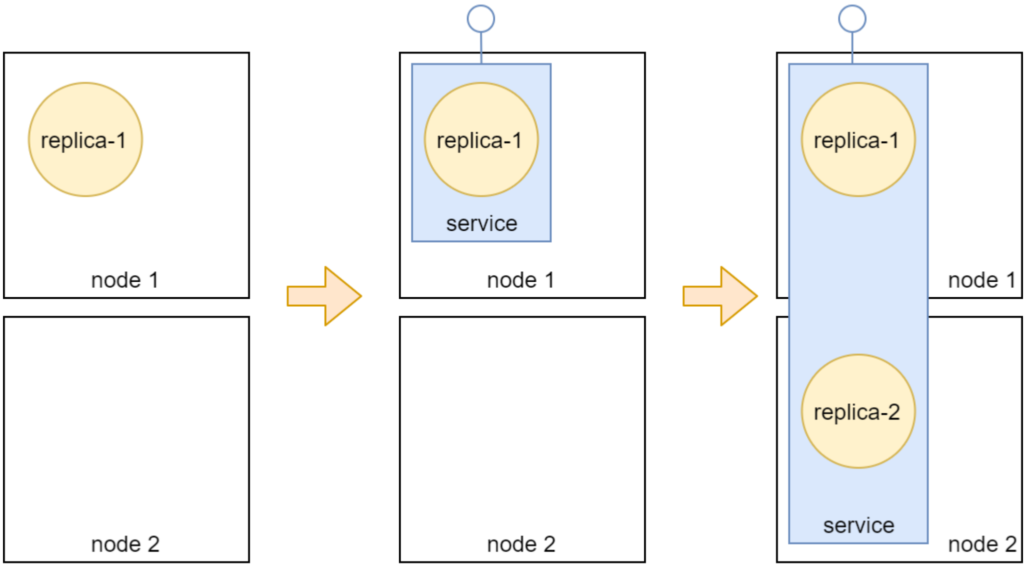
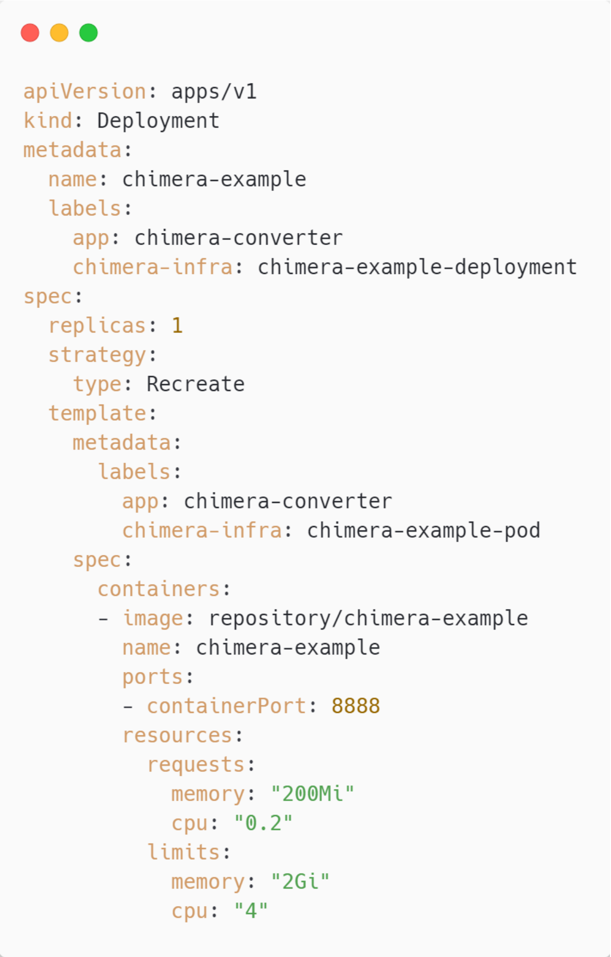
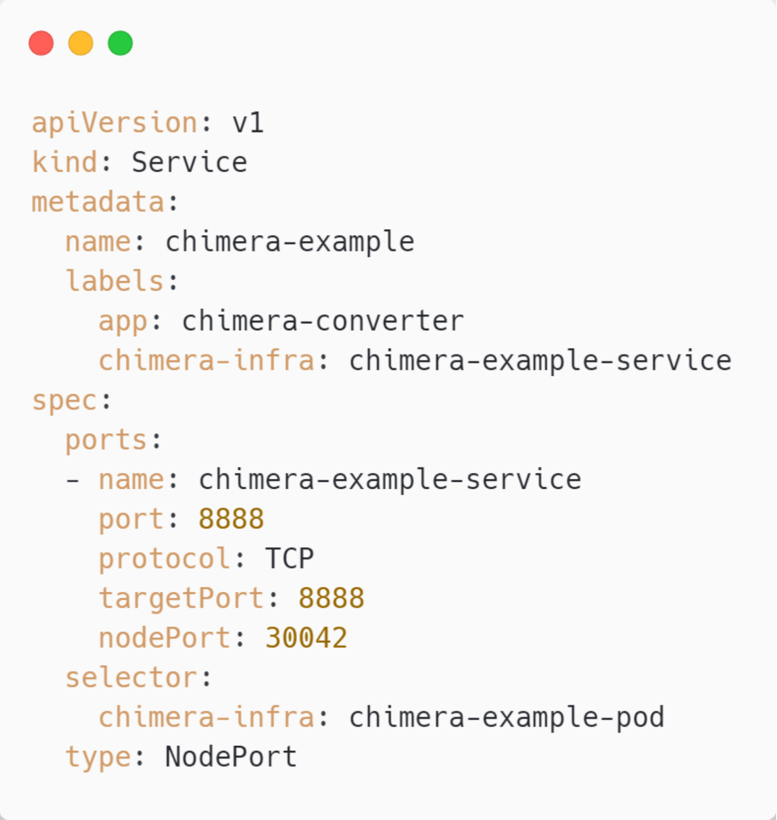


Figure 11 Deployment and Service in Kubernetes



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: chimera-example
  labels:
    app: chimera-converter
    chimera-infra: chimera-example-deployment
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: chimera-converter
        chimera-infra: chimera-example-pod
    spec:
      containers:
      - image: repository/chimera-example
        name: chimera-example
        ports:
        - containerPort: 8888
        resources:
          requests:
            memory: "200Mi"
            cpu: "0.2"
          limits:
            memory: "2Gi"
            cpu: "4"
```

Figure 12 Converter Deployment Kubernetes Manifest



```yaml
apiVersion: v1
kind: Service
metadata:
  name: chimera-example
  labels:
    app: chimera-converter
    chimera-infra: chimera-example-service
spec:
  ports:
  - name: chimera-example-service
    port: 8888
    protocol: TCP
    targetPort: 8888
    nodePort: 30042
  selector:
    chimera-infra: chimera-example-pod
  type: NodePort
```

Figure 13 Converter Service Kubernetes Manifest

The manual scaling of the Deployment can then be achieved using the command:

```
$ kubectl scale --replicas=3 deployments/chimera-example
```

With this command we ask the orchestrator to deploy three replicas of the Converter onto the cluster, each one with the resource constraints defined before.

The second abstraction to configure the Converter on the Kubernetes cluster is the *Service*. A *Service* is an abstraction that allows to group logically a set of Pods defining the policies to access them. In particular, a *selector* is defined to identify the Pods and a *policy* is specified to expose them within the cluster or on the external network. The *Service* abstraction offers the possibility of obtaining automatic load distribution managed by Kubernetes and dispatching requests among the different Pods composing the *Service*. This feature removes the need of configuring a reverse-proxy as done in C-Rel for the docker-compose deployment. As shown in the second and third step in Figure 11, a *Service* groups the a set of Pods within the cluster on the different nodes, if more replicas are deployed, it automatically adapts to integrate them.

In Figure 13 a *Service* Kubernetes manifest for the Converter *Deployment* defined in Figure 12 is reported. The label assigned to the Pod is used as a *selector* for the *Service.* The orchestrator exposes the *Service* on the port 30042 of each cluster node distributing requests among the different Pods composing the *Service.* In the case considered, it forwards requests on port 8888 of the different Pods. The configuration provided also ensures that scaling the *Deployment* also the number of Pods associated to the *Service* scales accordingly.

The Kubernetes configuration provided offers the possibility of taking advantage of the orchestrator to automatically handle the scalability of the Converter. Kubernetes defines the abstraction of *Horizontal Pod Autoscaler (HPA)* that is a controller that can automatically scale horizontally the number of pods in a *Deployment* based on an observed metric and a set of pre-defined target values for the metric. Intuitively, the algorithm determines the number of replicas to be deployed using the following formula that correlates the current and desired metric value to the number of replicas to be deployed.

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue /
                        desiredMetricValue )]
```

Deploying the metric-server[5] a set of pre-defined metrics (Memory, CPU, ecc.) can be used to configure an HPA. However, also custom metrics exposed by the application itself can be used to configure the autoscaler.

For example, we can attach an HPA to the defined Converter *Deployment* monitoring the CPU usage:

```
$ kubectl autoscale deployments/chimera-example --min=1 --max=5 --cpu-percent=80
```

The orchestrator will then create a Resource Controller which will check continuously the CPU usage of the Converters. If they will use more than 80% of the CPU it will spawn additional replicas, up to maximum 5 replicas. The HPA can also scale down the number of replicas to avoid occupying resources currently not used, in the examples to minimum 1 replica.

The examples shown in this section demonstrates how it is possible, without any intervention on the programming side, to exploit the manifests generated by the Asset Manager to deploy the Converter onto a cluster running a Kubernetes orchestrator. Moreover, we showcased how this type of configuration allows to take advantage of the features of such orchestrator to obtain resource-efficient deployment in production.

3.3 ASSET MANAGER-CONVERTER RUNTIME INTEGRATION

The C-Rel version of the Asset Manager focused on automatically generating Converter deployable artifacts. We demonstrated that a Jenkins job can fetch the description of a Converter, retrieve from the Asset Manager all the required files (ontologies, mappings, datasets), and build a stand-alone package (both as executable JAR archive and Docker compose package) which has no further dependency on the Asset Manager itself. In a sense, the Asset Manager “statically compiles” the Converter package, which is then completely independent.

In F-Rel, we already described our plans to extend such concept to the creation of Kubernetes templates that can enable the definition of services and autoscaling on a cluster or cloud environment. The creation of a static package anyway is not the only way to exploit a component providing a “single source of truth” for interoperability, which is the role of the Asset Manager. The Converter framework that we created allows for a very wide array of solutions, and it is possible to create a “generic Converter” which dynamically accesses the Asset Manager to discover which

---

[5] https://github.com/kubernetes-sigs/metrics-server

assets are required to enable a successful conversion between two messages belonging to different specifications/standards. Since the role of a Resolver is to provide additional features to support other services, to achieve this goal Resolvers should be created to support the discovery phase. Then a conversion pipeline should be created to support dynamically querying the Resolver, downloading artifacts and caching them. This workflow is depicted in Figure 14 using the Enterprise Integration Patterns (EIP) notation.

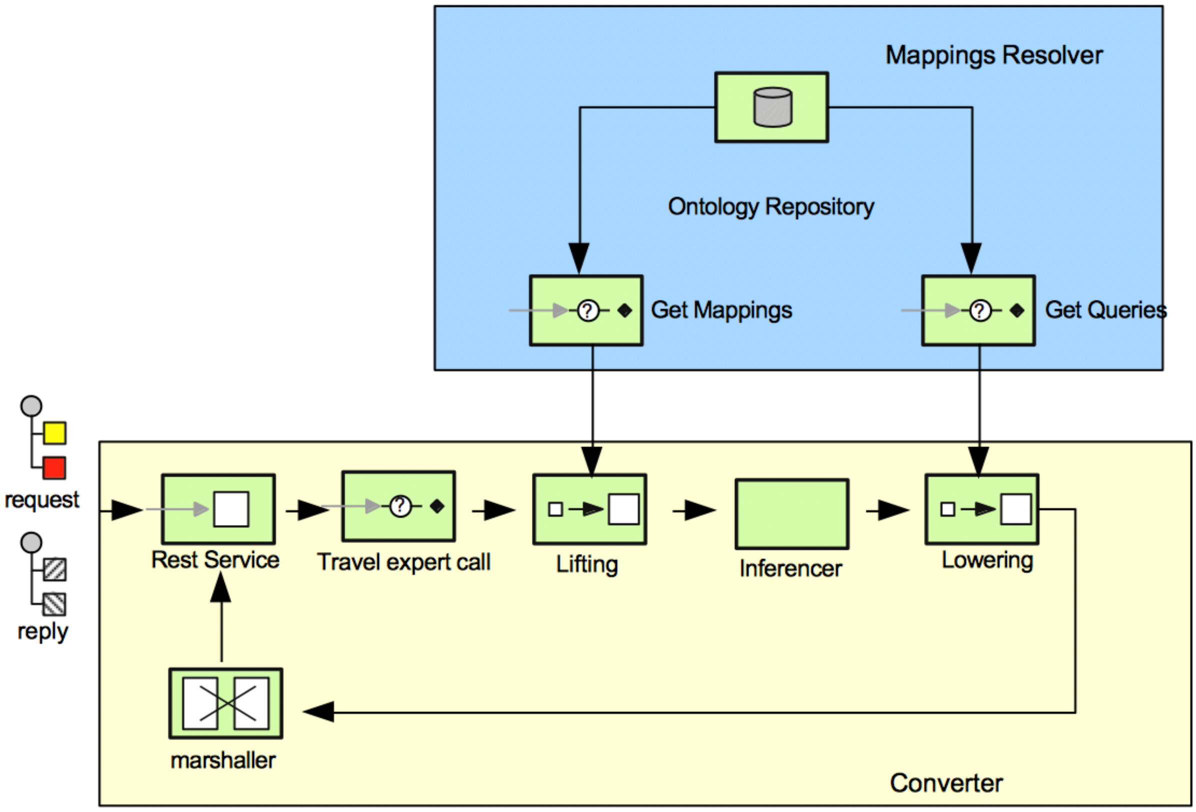


Figure 14 Converter interaction with the Mappings Resolver

The C-Rel version of the Chimera framework assumes that the Converter configuration is static and known at deployment time. The design of a generic Converter requires therefore modifying the Chimera framework, introducing more configuration blocks and extending the features of the existing ones to let them interact with Resolvers to obtain new configurations. The resulting conversion pipeline is depicted in Figure 15, where the new blocks in the conversion pipeline are highlighted in green. The new blocks are:

- Converter Finder: given the source format and the destination format, it performs a call to a Converter Resolver looking for existing Converter configurations (which include the identifiers of the Mappings, Ontologies, Datasets and Data Enrichment queries). The output of this block will be then used by the other “Initializer” blocks.
- Lifting Initializer: it performs a call to a Mappings resolver, looking for Lifting Mappings which enable extracting knowledge from the incoming message into RDF according to a specific ontology.

- Lowering Initializer: it performs a call to a Mappings resolver, looking for Lowering Mappings which enable extracting knowledge from RDF (according to a specific ontology) into the desired destination format.

Moreover, the Inference Enricher and the Data Enricher must be modified to dynamically obtain Ontologies and Data Enrichment SPARQL queries performing calls to the Mappings Resolver.

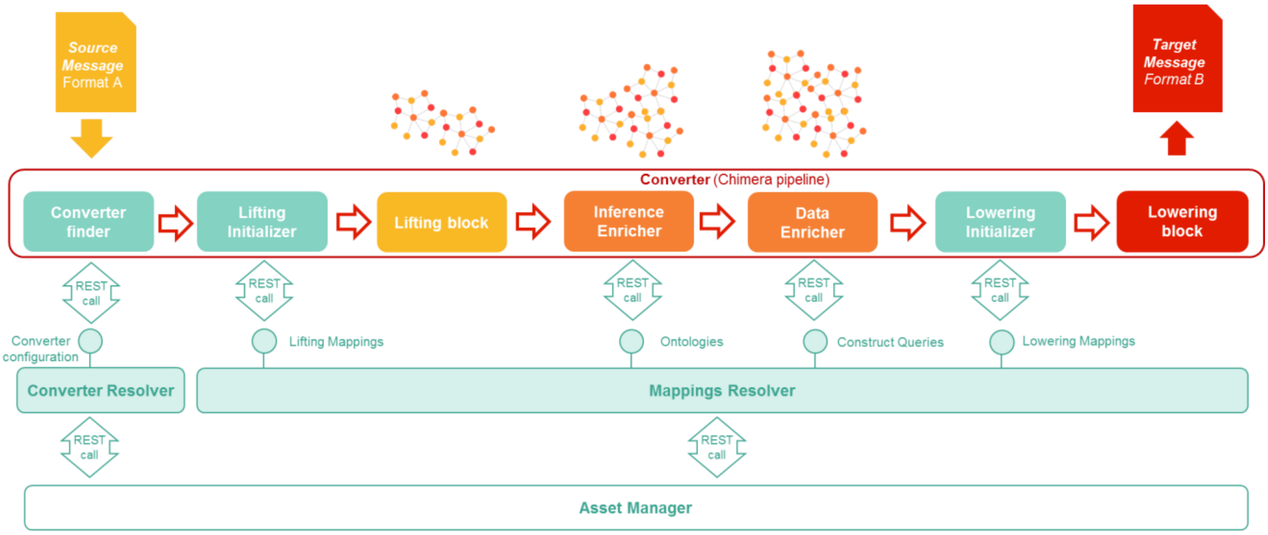
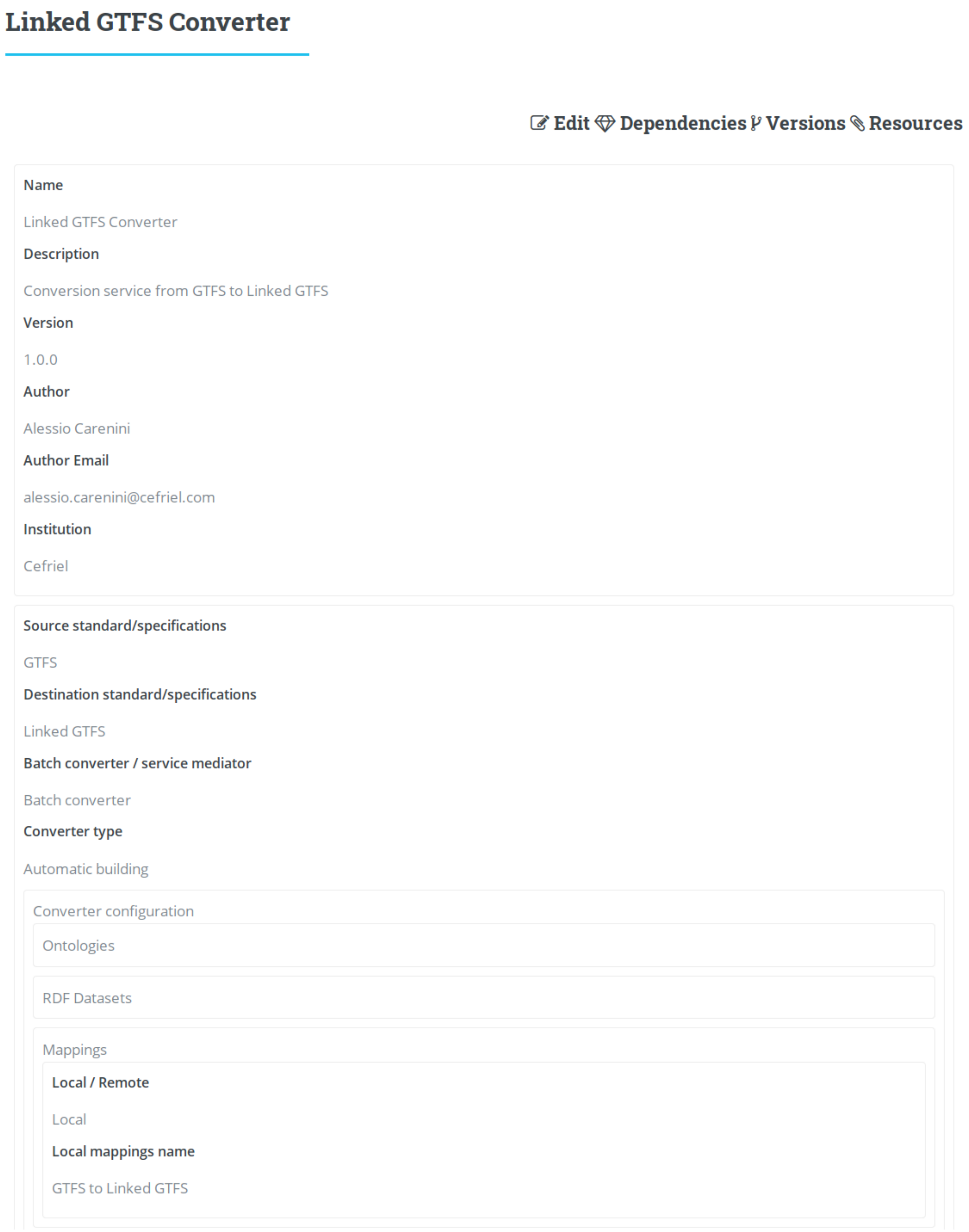


Figure 15 Interaction between the Asset Manager and the Converter mediated by Resolvers

The key part in the interaction between the Converter and the Asset Manager is the Converter finder, which obtains Converter metadata from the Converter Resolver. In a sense, the “Converter” asset metadata contains a “recipe” which states which Mappings, RDF Datasets and Ontologies are required for a proper conversion between different formats/standards/specifications. An example of such metadata can be seen in Figure 16, which contains a Converter definition from the GTFS format to the Linked GTFS format. Such metadata can be queried by the Converter resolver, and the feature can be exposed as an API, thus creating a Converter Resolver. After calling the “Converter finder” block, then, the other blocks inside the conversion pipeline will be aware that they will need to call the Mappings Resolver using parts of the Converter metadata information to obtain their required files.



Linked GTFS Converter

Edit Dependencies Versions Resources

Name

Linked GTFS Converter

Description

Conversion service from GTFS to Linked GTFS

Version

1.0.0

Author

Alessio Carenini

Author Email

alessio.carenini@cefriel.com

Institution

Cefriel

Source standard/specifications

GTFS

Destination standard/specifications

Linked GTFS

Batch converter / service mediator

Batch converter

Converter type

Automatic building

Converter configuration

Ontologies

RDF Datasets

Mappings

Local / Remote

Local

Local mappings name

GTFS to Linked GTFS

Figure 16 Definition of the assets required by a Converter

Since the Converter resolver and the Mappings resolver are APIs based on pre-packaged queries, implementing dedicated Resolvers is not strictly required. The Asset Manager already supports exposing parametric queries as API using the Exploration API assets, and therefore the same functionalities described above can be implemented in an easier way without the need for the deployment of additional components. As shown in Figure 17, the conversion pipeline remains unchanged with respect to Figure 15, but in this case the Asset Manager directly supports the features previously exposed via the Converter and Mappings Resolvers. The F-Rel version of the SPRINT Interoperability Framework will then support an enhanced Chimera framework with the aforementioned new blocks, and new Exploration APIs to support the dynamic behavior of new generic Converters.

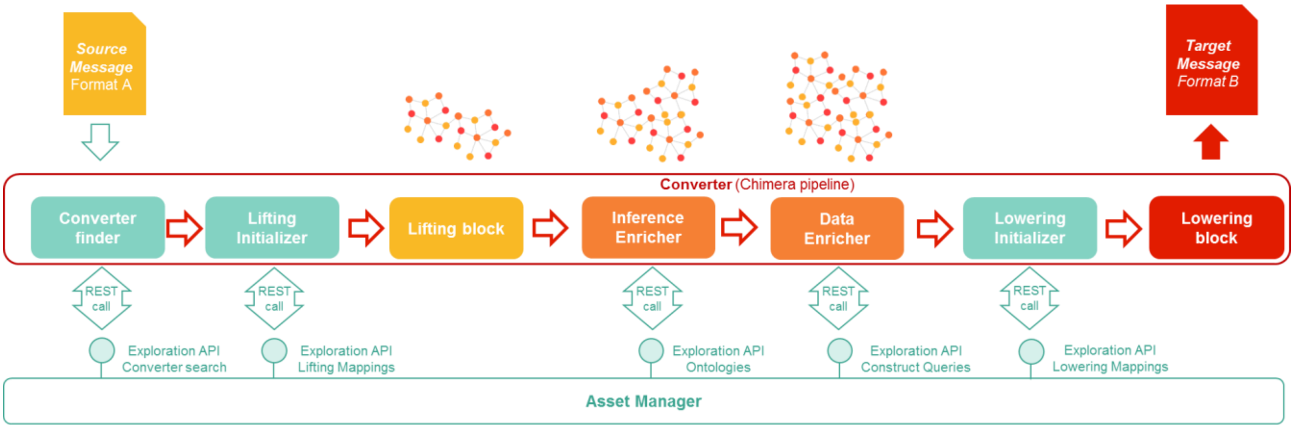


Figure 17 Simplified interaction between the Asset Manager and the Converter using Exploration API

Discovering the mappings which allow efficiently transforming a message from its original format to another one, conforming to a different standard or set of specifications, is not a trivial task. The critical factor is represented by the granularity of the information published inside the Asset Manager. To describe such issue, we will take as an example the case already explored by the ST4RT project, namely the conversion of a specific “Reservation request” message from TAP/TSI 918 into its corresponding “Pre-booking request” in FSM. Both 918 and FSM are very large specifications and comprise a lot of message types. If we publish a mapping naming it “FSM to IT2Rail ontology”, we are basically stating that all of FSM is covered, but this could not be true. We can either trust the Technical Management Board, or anyway the responsible for the publication approval, to check the coverage of a mapping wrt. to a specification and a target ontology, or we can rely on automated testing. In the context of SPRINT F-Rel, we will assume that the mappings published inside the Asset Manager are “trusted” and accurate, and we will focus on how a generic/universal Converter can exploit such information to execute whatever mapping is correctly defined inside the Asset Manager.

3.4 ASSET MANAGER AS A COMPANION OF NATIONAL ACCESS POINTS

SPRINT deliverable D2.3 performed an analysis of the current landscape of the National Access Points, reporting about the differences in the publication processes, in the authorization policies and in the required data and metadata formats. As such deliverable pointed out, the automation features of the Asset Manager can be exploited to obtain a NAP companion, a tool which is able to interact with different NAPs to import and export asset descriptions and their related datasets. For the F-Rel, we will focus on how to use the Asset Manager to access and aggregate metadata from multiple National Access Points. Publishing new assets in a National Access Point requires a registration process which is restricted to Transport Operators only. Since the Italian NAP for journey planning has not yet been established, we could not use Trenitalia/FSTech as “testing Trasport Operator” and therefore we have no technical means to test any IF-based solution for contributing to a NAP. We will anyway provide an initial design of a solution to let the Asset Manager hide the complexities of publishing an asset to a NAP, as the features of the Asset Manager (such as describing asset lifecycle management processes via BPMN and automating the post-publication phase via Jenkins) support implementing such scenario.

3.4.1 Automating metadata aggregation from multiple NAPs

The Asset Manager plays the role of a master data management inside the ecosystem, being the single source of truth for all the other components. As the National Access Points play the same role in the EU-wide scenario, providing a unique place to look for transportation-oriented datasets, the Asset Manager can harvest metadata from such remote service and provide a NAP-aware user interface. Integrating the Asset Manager with the NAPs is not a quick task, since each NAP uses a different metadata schema and a different API to let users access its metadata. Reading data and metadata from NAPs is anyway always possible, and we can therefore design an automated solution which leverages on the Asset Manager and the Converter framework to:

- connect to each NAP,
- fetch the metadata of its assets,
- convert such metadata into an RDF format to be easily queried via SPARQL,
- store the resulting triples inside the RDF repositories
- show that the Asset Manager can show both local and remote assets.

Since we will just collect metadata, dataset will still reside inside NAPs, and the users will be redirected to the owning NAP whenever they'll need to download it. The process of collecting remote metadata will be implemented as a periodic job which will run once a day (performing such job multiple times a day to collect metadata of remote Journey planning assets could be useless as they don't change so frequently).

Metadata mapping

The first step in aggregating metadata coming from multiple NAPs is to decide a metadata schema to be used to store converted metadata. As described in D2.3, there is already an effort, named Coordinated Metadata Catalogue, to harmonize metadata in National Access Points, led by the German Federal Highway Research Institute (BASt) together with the Austrian and the Dutch governments. The specifications of the Coordinated Metadata Catalogue include a mapping to DCAT-AP [6], which has been also described in D2.3. DCAT-AP is also at the core of the Asset Manager RDF metadata. Performing a conversion from the source NAP metadata structure to the Coordinated Metadata Catalogue format using DCAT-AP terms therefore enables direct integration between the RDF representation of local metadata and the RDF representation of remote data.

In F-Rel we will therefore use the RDF representation of the Coordinated Metadata Catalogue (using DCAT-AP) as the target for our metadata conversion, which will be implemented using the same Chimera framework that we're also using to build Converters.

---

[6] Coordinated metadata catalogue to DCAT-AP mappings: https://eip.its-platform.eu/sites/default/files/EU%20EIP_Coord.%20Metadata%20Catalogue_Annex%20II_v2.0_191115.xlsx

Harvesting metadata from NAPs

Fetching metadata from a NAP and converting it as RDF instances of a metadata schema based on DCAT-AP and the Coordinated Metadata Catalogue format is a specific conversion process, which can be implemented using the Chimera framework in the same configuration explained in Section 3.3. The conversion pipeline, as shown in Figure 18, performs just the identification of the lifting mappings and the lifting process. The output of the lifting block will then be stored as a graph inside the RDF repository, to be later accessed via SPARQL.

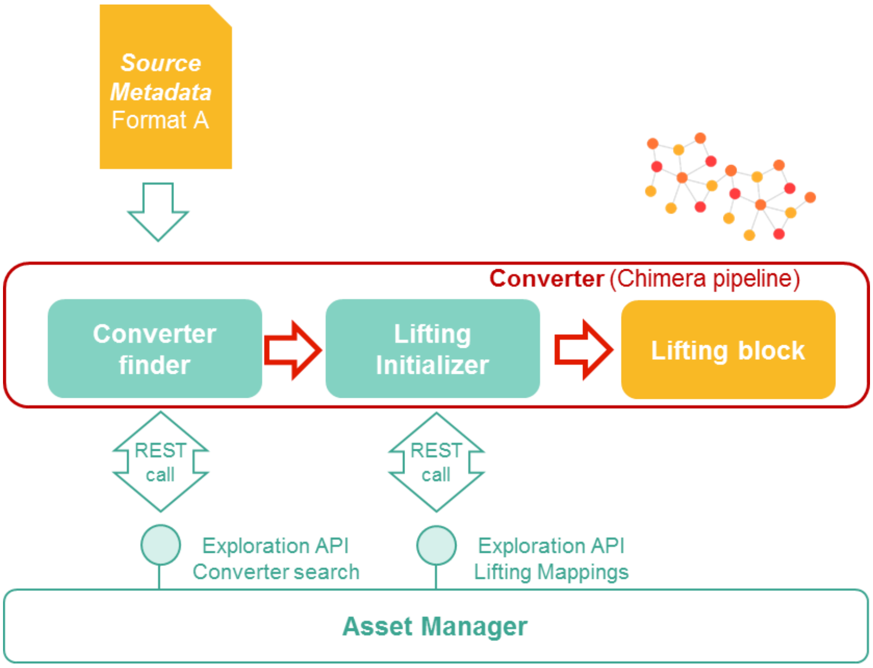


Figure 18 Chimera pipeline used to convert metadata from a specific NAP and store it in the RDF repository

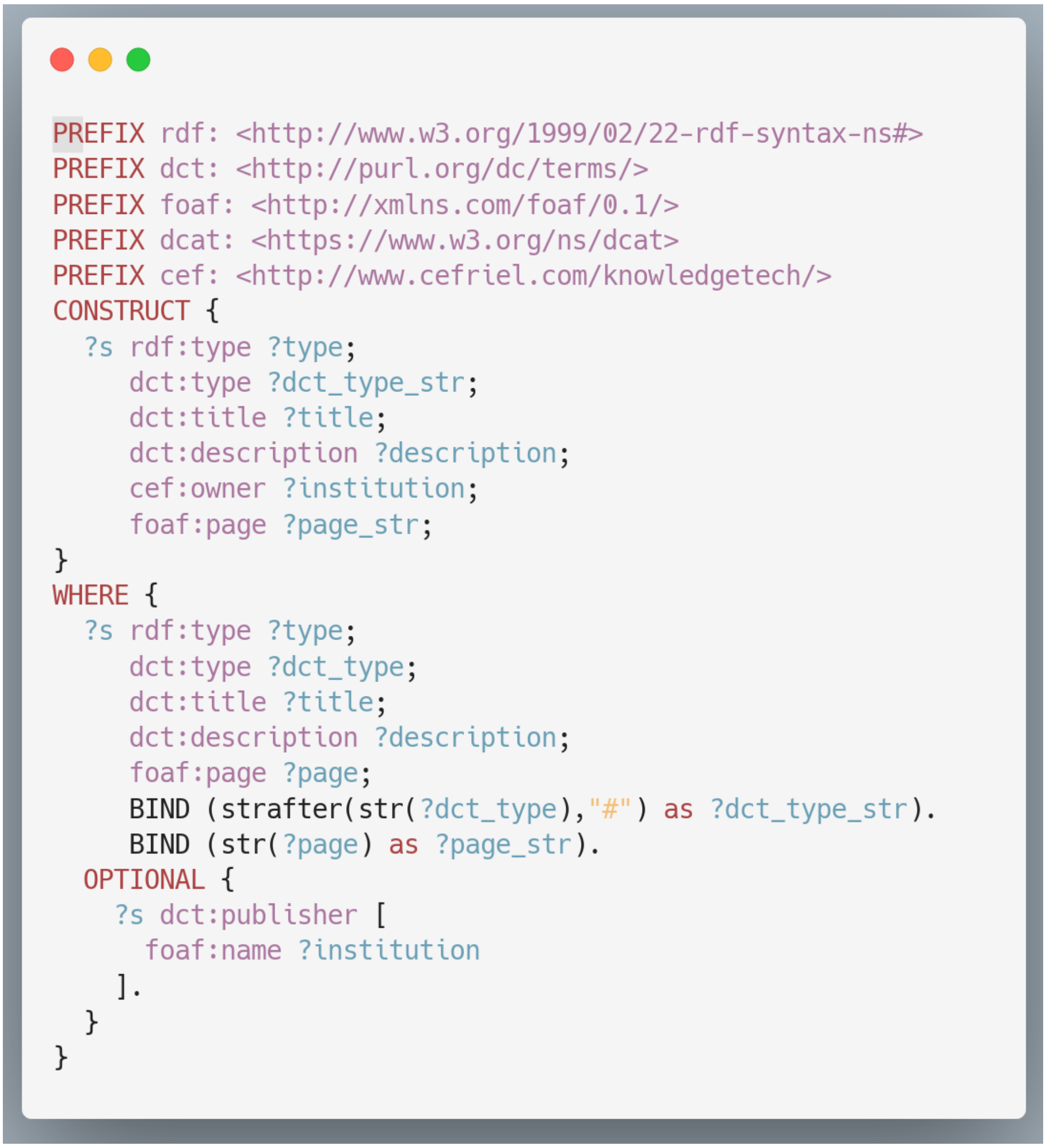
In the following list we report how to retrieve the complete set of metadata from some of the available National Access Points. For each of the NAPs that will be demonstrated according to the scenarios in D5.4, an RML mapping will be developed to extract RDF triples from the incoming metadata.

- **Belgium**
    - API endpoint: https://www.transportdata.be/api/3
    - Metadata endpoint: https://www.transportdata.be/api/3/action/package_search
    - Format: JSON
- **Finland**
    - API endpoint: https://finap.fi/ote
    - Metadata endpoint: https://finap.fi/ote/service-search?response_format=json
    - Format: JSON

- **Denmark**
    - API endpoint: None
    - Metadata endpoint: https://nap.vd.dk/rdf
    - Format: RDF
- **Netherlands**
    - API endpoint: None
    - Metadata endpoint: https://nt.ndw.nu/services-spoa/rest/v1/ui/multimodaal
    - Format: JSON
- **Greece**
    - API endpoint: http://data.nap.imet.gr/api/3
    - Metadata endpoint: http://data.nap.imet.gr/api/3/action/package_search
    - Format: JSON
- **France**
    - API endpoint: https://transport.data.gouv.fr/swaggerui
    - Metadata endpoint: https://transport.data.gouv.fr/api/datasets
    - Format: JSON
- **Czech Republic**
    - API endpoint: None
    - Metadata endpoint: https://data.gov.cz/sparql
    - Format: SPARQL results

Accessing metadata from the Asset Manager

Since metadata coming from multiple NAPs will be aggregated using the Coordinated Metadata Catalogue format, which in turn is aligned with DCAT-AP, accessing remote assets from the Asset Manager will use a specific SPARQL query which will be exposed as an Exploration API. A similar query/API is already deployed in the Asset Manager (as shown in Figure 19) to extract results for the "Explore" page in the Publisher and the "Search" page in the Store, therefore the only adaptation that will be done to such query will be to output whether an asset will be local or remote, as that will be the basis to show graphically different results in the UI.



```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dcat: <https://www.w3.org/ns/dcat>
PREFIX cef: <http://www.cefriel.com/knowledgetech/>
CONSTRUCT {
  ?s rdf:type ?type;
     dct:type ?dct_type_str;
     dct:title ?title;
     dct:description ?description;
     cef:owner ?institution;
     foaf:page ?page_str;
}
WHERE {
  ?s rdf:type ?type;
     dct:type ?dct_type;
     dct:title ?title;
     dct:description ?description;
     foaf:page ?page;
     BIND (strafter(str(?dct_type),"#") as ?dct_type_str).
     BIND (str(?page) as ?page_str).
  OPTIONAL {
    ?s dct:publisher [
      foaf:name ?institution
    ].
  }
}
```

Figure 19 SPARQL query for the “get assets” Exploration API

The modified Exploration API will be then reused internally by both the Store and the Publisher to show the results distinguishing between local and remote assets. Remote assets will be also public, since metadata coming from NAPs is public, and therefore all the pages representing remote assets will be free from the “Request access” button.

3.4.2 Automating contributions to NAPs

The analysis of the current landscape of the National Access Points showed that:

- each of them accepts a different set of metadata;
- each of them has a different publication process;

- even if Transmodel-based formats should be mandatory, no existing NAP requires it mandatorily;
- no NAP provide out-of-the-box support for automatic data conversion.

Developing a solution to contribute to NAPs using the IF technologies developed in SPRINT therefore essentially means developing different specific and dedicated publication workflows, and deploy them in the Asset Manager as part of the lifecycle management. Such workflows would then use ad-hoc Converters, to turn asset metadata from the Asset Manager-specific format to the one accepted by the target NAP. As an advanced feature, an IF-based NAP companion could make use of data Converters to convert datasets from the format supplied by the Contributor into NeTEx or any other Transmodel-based format. The resulting publication workflow, using the Enterprise Integration Patterns notation, is shown in Figure 20, while a possible NAP-aware lifecycle management process is shown in Figure 21.

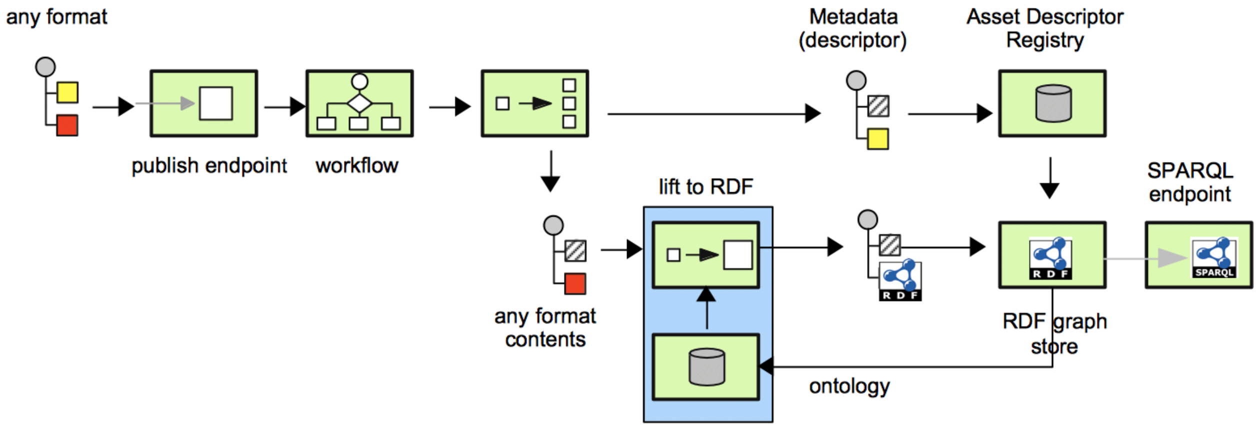


Figure 20 EIP-based description of an IF-based publication flow

In the lifecycle management process example, it becomes clear how contributing to different NAPs implies building many metadata Converters and asset uploaders. Before doing that, it will be also important to identify which will be the destination NAP. That decision could be automated using the asset metadata, or delegated to the user via a BPMN Human task.

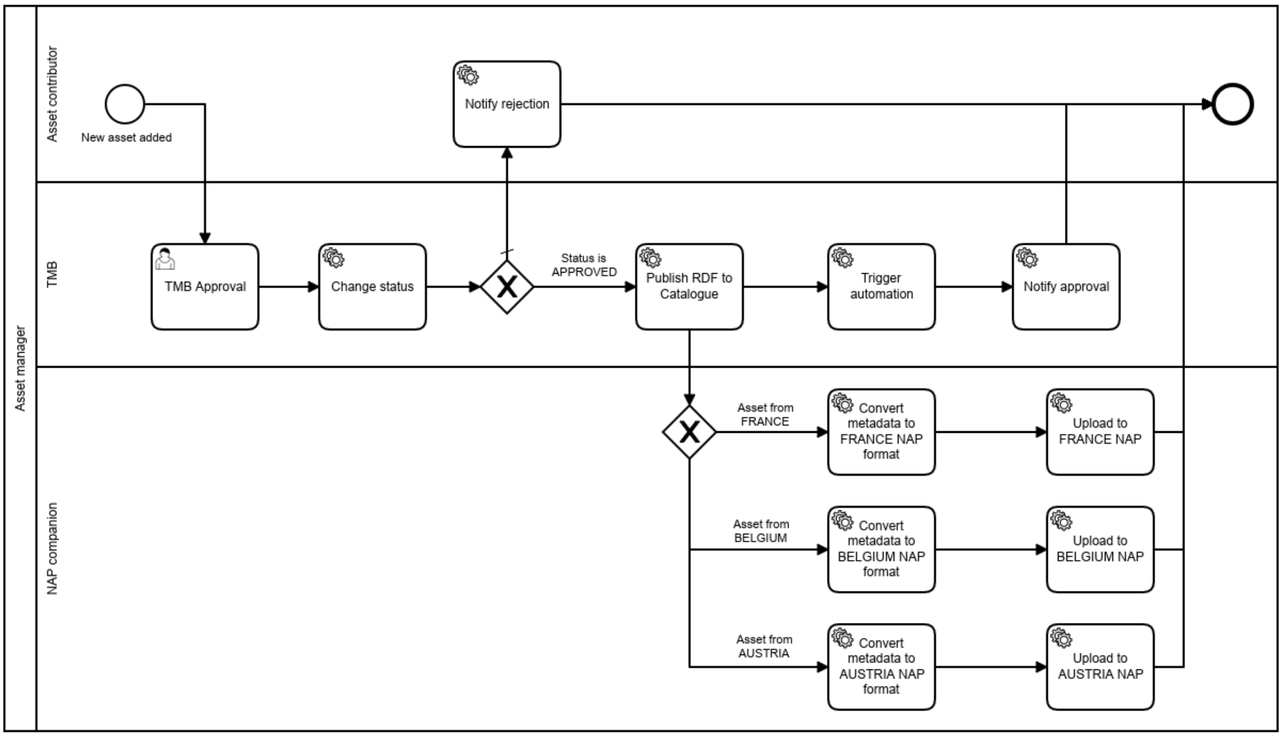


Figure 21 NAP-aware lifecycle process

4. SCALABLE/DISTRIBUTED RDF-BASED DATA ACCESS

Several tools for RDF generation are widely known in the literature. In S2R ecosystem, efficient tools are needed to access to a combination of Linked Data-enabled sites, SPARQL endpoints on top of native RDF stores, and virtual SPARQL endpoints on top of mapping-enabled data sources (CSVs, relational databases, REST APIs) that are available as IF assets.

Semantic Web query federation tools focus on gathering the maximum number of responses by distributing a SPARQL query over various RDF data sources. These tools can be classified into three different categories: index-only, index-free and hybrid (index+ASK) [2]. More specifically, the tasks to be performed by these tools are usually: selection of relevant sources for the query, query planning, generation of subqueries and query execution. ANAPSID [3] is a hybrid tool that adapts the execution plan of the queries at runtime, based on the availability and condition of each of the SPARQL endpoints on which the queries are to be executed. FedX [4] is an index-free tool that makes use of ASK queries for source selection and implements a heuristic-based query schedule. MULDER [5] is a query federator that focuses on the generation and use of source descriptors to perform source selection, query decomposition and subsequent optimization. Ontario [6] is a federated query processing approach for heterogeneous data sources. In its source selection step, Ontario uses source descriptions named RDF Molecule Templates which keep information on the sources. The system implements a virtualization approach of query answering techniques for efficient execution. Similarly, Squerall [7] is a system that takes as its inputs data and mappings and uses a middleware to aggregate the intermediate results in a distributed manner. SaGe [8] is a preemption-based SPARQL query engine for public endpoints. This system uses preemptable execution plans and performs time-sharing planning. SaGE focuses on the problem of RDF data availability for complex queries on public endpoints. Consequently, SaGe provides an alternative to the current practice of copying RDF data dump.

Also, there is a set of tools that perform query federation using Link Traversal. Basically, this approach focuses on retrieving information from the URIs of each resource in a distributed context. Examples of this kind of tools are SIHJoin [9], WODQA [10] or SQUIN [11].

Other strategies such as Triple Pattern Fragments (TPF) [12] are able to perform SPARQL queries on multiple data sources. Comunica is a meta tool for federated queries that besides allowing access through interfaces such as TPF, is capable of querying SPARQL endpoints, datasets in different RDF serializations or in HDT [13].

On the other hand, throughout the years, data integration systems [14] have evolved to the use of ontologies as a common model for data access [15], what is called Ontology-Based Data Access (OBDA). Most of the works proposed under this approach are focused on providing access to relational databases [15] [16] [17] and optimizations on the SPARQL-to-SQL translation process. There are multiple proposals focused on the translation of SPARQL queries to query data in their original format. The first proposal for translating SPARQL-to-SQL is defined in [18]. Multiple tools are proposed in the optimization of this process that take into account this mapping specification, such as Morph-RDB [16], Ontop [15], or Ultrawrap [17]. Additionally, there are specific studies on how SPARQL operators affect the translation of the query to SQL [19]. Beyond relational databases, MorphxR2RML [20] formally defines the translation from SPARQL to NoSQL databases. In addition, Morph-CSV [21] is a proposal to enhance the SPARQL-to-SQL process when the data source is a set of tabular data (i.e. CSV files). It exploits information from tabular metadata and mapping rules to explicitly enforce implicit constraints of the original datasets. MorphCSV follows an OBDA approach including the exploitation of additional information from mappings, tabular metadata and queries for tabular datasets.

In OBDA, different mapping languages have been proposed for defining transformation rules between ontology representation languages and data sources in different formats; these include SQL and NoSQL databases, as well as data in plain text such as CSV, XML and JSON. The RDB2RDF W3C Working Group published two recommendations for transforming the content of relational databases into RDF: Direct Mapping [22] and R2RML [23]. After the recommendation was released, new needs and requirements arose in relation to supporting other formats beyond relational databases, and this resulted in the creation of new mapping languages such as RML [24] which considers data sources in CSV, JSON and XML formats, xR2RML [20] for MongoDB databases, KR2RML [25] that considers nested data, CSVW [26] to annotate CSV files on the Web, and D2RML [27] for XML, JSON and REST/SPARQL endpoints, among others. Non-declarative mapping languages have also been proposed, for example SPARQL-Generate [28] extends the SPARQL 1.1 by taking as input an RDF dataset and a set of documents in multiple formats, and generating RDF through the SPARQL CONSTRUCT clause.

To evaluate the performance and scalability of different tools capable of executing SPARQL queries on RDF data sets, several benchmarks have been proposed in the literature. Generally, they consist of a set of queries that meet various characteristics (e.g. operators, form of the query or number of joins), a scale data generator and a series of measures for evaluation such as the total time of execution or the amount of results obtained. The Berlin SPARQL Benchmark (BSBM) [29] proposes a benchmark for the comparison of native Triple Stores (e.g. Virtuoso) and query translation tools for when data is loaded in relational databases (SPARQL to SQL). Focused on the domain of online shopping in addition to a scale data generator, BSBM proposes a set of 25 SPARQL queries that emulate a potential consumer's search for a product. The metrics used in this benchmark are: number of times the set of queries is evaluated in one hour, number of evaluated queries per second and data loading time. Additionally, GTFS-Madrid-Bench [30], a benchmarking for virtual knowledge graph access in the transport domain. This benchmark uses the de-facto standard model for publishing open data on web, it scales up and distributes the original dataset in several formats and sizes.

5. CONCLUSIONS AND NEXT STEPS

The final release of the SPRINT Interoperability Framework will provide automation features which, in our opinion, will enable an easier establishment of an IP ecosystem. Collaborative ontology engineering will be surely needed to provide up-to-date documentation to the multilateral effort of building the IP4 ontology, and the automatic generation of ontologies will help understanding other specifications and their relationship with the IP4 ontology. Since a large part of the integration process between different TSPs will focus on converting data, a tool to suggest possible mappings will surely help to handle the burden of finding correspondences in complex data models, while the possibility to implement complex transformation with minimal coding via our Converter framework will help obtaining an efficient interoperability. Finally, our asset management solution provided ways to automate most of the management part of a complex ecosystem, and such a tool could become a key component in building the IP4 ecosystem.

6. REFERENCES

[1] D. Garijo, "WIDOCO: A wizard for documenting ontologies," in *International Semantic Web Conference*, 2017.

[2] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov and A.-C. Ngonga, "A fine-grained evaluation of SPARQL endpoint federation systems," *Semantic Web Journal,* p. 493 – 518, 2014.

[3] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo and E. Ruckhaus, "ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints," in *International Semantic Web Conference*, 2011.

[4] A. Schwarte, P. Haase, K. Hose, R. Schenkel and M. Schmidt, "FedX: Optimization Techniques for Federated Query Processing on Linked Data," in *International Semantic Web Conference*, 2011.

[5] K. Endris, M. Galkin, I. Lytra, M. Mami, M.-E. Vidal and S. Auer, "MULDER: Querying the linked data web by bridging RDF molecule templates," in *Database and expert systems applications, DEXA*, 2017.

[6] K. Endris, P. Rohde, M.-E. Vidal and S. Auer, "Ontario: Federated Query Processing Against a Semantic Data Lake," in *International Conference on Database and Expert Systems*, 2019.

[7] M. Mami, D. Graux, S. Scerri, H. Jabeen and S. Auer, "Squerall: virtual ontology-based access to heterogeneous," in *International SemanticWeb Conference*, 2019.

[8] T. Minier, H. Skaf-Molli and P. Molli, "SaGe: Web Preemption for Public SPARQL Query services," in *World Wide Web Conference*, 2019.

[9] T. Ladwig, "SIHJoin: Querying Remote and Local Linked Data," in *The Semantic Web: Research and Applications*, 2011.

[10] Z. Akar, T. Gökmen, E. Ekinci and O. Dikenelli, "Querying the Web of Interlinked Datasets using VOID Descriptions," in *Workshop on Linked Data on the Web*, 2012.

[11] O. Hartig, "SQUIN: A Traversal Based Query Execution System for the Web of Linked Data," in *ACM SIGMOD International Conference on Management of Data*, 2013.

[12] A. Meroño-Peñuela and R. Hoekstra, "grlc makes GitHub taste like linked data APIs," in *European Semantic Web Conference*, 2016.

[13] E. Daga, L. Panziera and C. Pedrinaci, "Basil: A cloud platform for sharing and reusing SPARQL queries as Web APIs," in *CEUR Workshop Proceedings*, 2015.

[14] M. Fernández-López, A. Gómez-Pérez and N. Juristo, "METHONTOLOGY: From Ontological Art Towards Ontological Engineering," in *AAAI*, 1997.

[15] S. Staab, R. Studer, H.-P. Schnurr and Y. Sure-Vetter, "Knowledge Processes and Ontologies," *IEEE Intelligent Systems,* vol. 16, no. 1, pp. 26-34, 2001.

[16] M. Suárez-Figueroa, A. Gómez-Pérez and M. Fernández-López, "The NeOn Methodology framework: A scenario-based methodology for ontology development," *Applied Ontology,* vol. 10, no. 2, pp. 107-145, 2015.

[17] A. Gomez-Perez, M. Fernández-López and O. Corcho, Ontological Engineering: With Examples from the Areas of Knowledge Management, E-Commerce and the Semantic Web, Springer Verlag, 2004.

[18] M. Suarez-Figueroa, Neon Methodology for Building Ontology Networks: Specification, Scheduling and Reuse, IOS Press, 2012.

[19] T. Tudorache, J. Vendetti and N. Noy, "Web-Protege: A Lightweight OWL Ontology Editor for the Web," in *Fifth OWLED Workshop on OWL: Experiences and Directions*, 2008.

[20] M. Dragoni, A. Bosca, M. Casu and A. Rexha, "Modeling, Managing, Exposing, and Linking Ontologies with a Wiki-based Tool," in *Proceedings of LREC*, 2014.

[21] L. Halilaj, N. Petersen, I. Grangel-González, C. Lange, S. Auer, G. Coskun and S. Lohmann, "VoCol: an integrated environment to support version-controlled vocabulary development," in *Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, 2016.

[22] A. Alobaid, D. Garijo, M. Poveda-Villalón, I. Santana-Perez, A. Fernández-Izquierdo and O. Corcho, "Automating ontology engineering support activities with OnToology," *Journal of Web Semantics,* 2018.

[23] M. Poveda-Villalón, A. Gómez-Pérez and M. Suárez-Figueroa, "OOPS! (OntOlogy Pitfall Scanner!): An on-line tool for ontology evaluation," *Int. J. Semant. Web Inf. Syst. (IJSWIS),* vol. 10, no. 2, pp. 7-34, 2014.

[24] J. Barnett, R. Akolkar, R. Auburn, B. M, B. D.C, C. J, M. S, L. T, M. Helbing, R. Hosn, R. T.V, R. K, N. Rosenthal and J. Roxendal, "State Chart XML (SCXML): State Machine Notation for Control Abstraction," W3C Recommendation, [Online]. Available: https://www.w3.org/TR/scxml/.

[25] V. L. Mikolov.T, "Exploiting Similarities among Languages for Machine Translation," 2013.

[26] S.Grayson, "Novel2Vec: Characterising 19th Century Fiction Via Word Embeddings," University College Dublin, Dublin, Ireland, 2016.