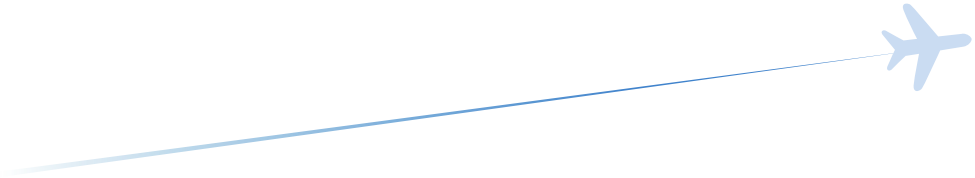


Proof-of-concept KG system

Deliverable ID:	**D4.1**
Dissemination Level:	**PU**
Project Acronym:	**AISA**
Grant:	**892618**
Call:	**H2020-SESAR-2019-2**
Topic:	**SESAR-ER4-01-2019**
Consortium Coordinator:	**FTTS**
Edition date:	**28 February 2021**
Edition:	**00.01.00**
Template Edition:	**02.00.02**

Founding Members

EUROPEAN UNION EUROCONTROL

SESAR JOINT UNDERTAKING

Authoring & Approval

Authors of the document

Name/Beneficiary	Position/Title	Date
Bernd Neumayr/JKU	University Assistant (Senior PostDoc)	12 March 2021

Reviewers internal to the project

Name/Beneficiary	Position/Title	Date
Michael Schrefl/JKU	Full Professor	12 March 2021
Tomislav Radišić/FTTS	Assistant Professor	13 March 2021

Approved for submission to the SJU By - Representatives of beneficiaries involved in the project

Name/Beneficiary	Position/Title	Date
Tomislav Radišić/FTTS	Project Coordinator	

Rejected By - Representatives of beneficiaries involved in the project

Name/Beneficiary	Position/Title	Date

Document History

Edition	Date	Status	Author	Justification
00.00.01	28/02/2021	First version	Bernd Neumayr	New document
00.00.02	11/03/2021	First full version	Bernd Neumayr	Completed Sect. 3
00.00.03	11/03/2021	First revision	Bernd Neumayr	Updated Figures
00.00.04	12/03/2021	Final draft	Bernd Neumayr	Internal Rev. Comments
00.01.00	13/03/2021	First Issue	Bernd Neumayr	First Issue

Copyright Statement

AISA

AI SITUATIONAL AWARENESS FOUNDATION FOR ADVANCING AUTOMATION

This deliverable is part of a project that has received funding from the SESAR Joint Undertaking under grant agreement No 892618 under European Union's Horizon 2020 research and innovation programme.

Abstract

The AISA Project-level Concept of Operations set out in Chapter 5 of Deliverable 2.1 has at its center the ATC Knowledge Graph (KG). In this deliverable (D 4.1) we describe an architecture for the data and metadata in such a KG and for the software components for incrementally processing and querying the data and metadata in the KG. The *Proof-of-Concept KG System* exemplifies this architecture and has the purpose of guiding further developments in WP 4 and WP 5. The proposed architecture of a KG system facilitates *SPARQL Queries Capturing Monitoring Tasks* based on traffic/airspace data converted to RDF. It further accomodates the integration with other components such as the Reasoning Engine in Prolog developed in Task 4.2 and Machine Learning Modules developed in WP3. The main goal is to develop and assess the concept of the artificial SA based on a KG from a functional perspective rather than to consider requirements of a real-time life system. In this deliverable we also describe the *UML-to-RDFS/SHACL mapper* which facilitates the transformation from information exchange models such as AIXM and FIXM modeled in UML to KG schemas in RDFS and SHACL.

1 Table of Contents

List of Tables

List of Figures

Executive Summary

This document describes deliverable D4.1 which is specified as a **Demonstrator**. The actual deliverable consists of two parts delivered as **GitHub** repositories which will be made available open source. The first part is the UML-to-RDFS/SHACL mapper (also referred to as AISA XMI mapper). The second part is the Proof-of-Concept KG system (also referred to as AISA KG system).

The UML-to-RDFS/SHACL mapper has already been used intensively in Task 4.3 to generate major parts of the KG schema from existing information exchange models. The generated KG schema consists of a vocabulary in RDF Schema (RDFS) and a set of structural constraints in the Shapes Constraint Language (SHACL). The generated SHACL constraints have been used, in turn, to validate the RDF data created manually in Task 4.3. The UML-to-RDFS/SHACL mapper is implemented in XQuery, it takes as input an UML class diagram represented in XMI (XML Model Interchange format) and a configuration file specifying which parts of the UML class diagram should be mapped, and produces as output an RDFS/SHACL document in RDF/XML format. To accommodate specifities of different information exchange models (e.g., different sets of UML stereotypes) the mapper comes with a plug-in architecture and currently has a FIXM plug-in and an AIXM plug-in.

Before discussing the proof-of-concept KG system let us recall the core purpose of the ATC knowledge graph in the project-level concept of operations, which is to facilitate *SPARQL Queries Capturing Monitoring Tasks*. In principle, for this purpose, it would suffice to transform and load all the relevant data into the KG and provide a SPARQL endpoint to pose the queries. This would, however, result in a nightmare regarding development and maintenance of the queries (with most queries being extremely complex and sharing large common parts) and regarding performance of query execution (with every query executed from scratch). Similar challenges exist for traditional databases where they are solved by materialized views (making queries and their results reuseable for other query executions), and incremental view maintenance (avoiding the need to recalculate query results once new data arrives). These proposed architecture will apply these concepts to KGs. From the perspective of Situational Awareness (SA) assessment, the KG will serve as memory of a *history sensitive self-aware system*, and, integrating with ML Modules, as memory of a *predictive system*. From this perspective it is clear that the data and metadata in the KG have to be fully versioned, that is, all historical states of the KG have to be queryable or at least reconstructible.

The Proof-of-Concept KG system demonstrates the KG system architecture described in this document. By KG system we refer to (1) the *KG* (in AISA: an RDF dataset), (2) *the application-independent software components* for storing, processing and querying the KG (in AISA: Apache Jena including Fuseki and TDB), (3) a set of *application-specific engines* (in AISA: a set of Java programs making heavy use of SPARQL which are responsible for loading, querying, inserting, processing, importing and exporting data and metadata in the KG) and (4) a *control component* (in AISA: a Java program which invokes the different engines and provides a command-line interface to allow users to invoke engines and to facilitate the scripting of experiments). Engines are invoked via the central control component in a serialized/synchronous manner. Components external to the KG system, like ML modules, are loosely coupled via file export and file import and run asynchronously; they are integrated into the KG system in a serialized/synchronous manner each by a specific engine which takes care, by separate invocations, of (1) the export from the KG of the input data for the external module and (2) the import to the KG of the output data from the external module. Rule-based reasoning in Prolog (to be discussed in D 4.2) will be integrated seamlessly into the system by Prolog-based engines, i.e., Java programs that call SWI-Prolog via the Java/Prolog interface JPL.

Intended Audience

This document is intended for use by those employed within SESAR Joint Undertaking and by the experts from the ATM community, other professionals working on research and development in the fields of data and knowledge engineering and information management, those employed in EUROCONTROL and the ANSPs who might take advantage of the proposed methods. The components described in this document should act as central components of the AI Situational Awareness System developed in the project and act as technical basis for further developments at a later stage of the AISA project. In particular, this document will be useful to partners involved in the project as a basis for further development in WP4 and WP5.

1 Introduction

In Task 4.1 we have developed (1) the UML-to-RDFS/SHACL mapper and (2) a KG system architecture and proof-of-concept prototype KG system geared towards the AISA project-level Concept of Operations. This document describes these developments.

1.1 Definitions

In the technical scope of this deliverable we give very specific technical meaning to otherwise broad terms.

Knowledge Graph (KG). A knowledge graph is a persistent RDF dataset, that is, a set of named RDF graphs comprising data and metadata.

KG System. By KG system we refer to (1) the *KG*, (2) *the application-independent software components* for storing, processing and querying the KG, (3) a set of *application-specific engines* which are responsible for loading, querying, inserting, processing, importing and exporting data and metadata in the KG and (4) a *control component* which invokes the different engines.

The complete list of acronyms and definitions of the terms mentioned in this paper can be found at the end of the document in the Appendix A – Glossary.

1.2 Purpose of the document

The purpose of this document is to describe the work undertaken in Task 4.1 to develop building blocks for the implementation of a system that serves to assess the concept of AI situational awareness. The current version sets the way for the forthcoming developments of WP 4 as well as Task 5.1 in the AISA project. The KG system architecture proposed and the software described in this document may evolve together with evolving requirements in the remainder of WP4 and Task 5.1.

1.3 Structure and methodology

This document describes the software developed in Task 4.1 (UML to RDFS/SHACL Mapper and Proof-of-Concept KG system). Section 2 describes the UML-to-RDFS/SHACL mapper and how it is applied. Section 3 proposes a KG system architecture for SA systems, describes a small Proof-of-Concept KG system together with a compact Java library, i.e., set of (abstract) classes and interfaces, facilitating the development of a KG system following the proposed architecture.

Appendices at the end of this document contain a glossary of all terms and acronyms.

1.4 Relations to other documents

The document is linked to project deliverable:

- AISA D2.1: Concept of Operations for AI Situational Awareness
- AISA D2.2: Requirements for Automation of Monitoring Tasks via AI SA

2 UML to RDFS/SHACL Mapper

2.1 Introduction

The UML to RDFS/SHACL Mapper (alternatively referred to as AISA-XMI-Mapper) maps selected classes of UML class diagrams to RDF Schema (RDFS) and Shape Constraint Language (SHACL) documents. RDFS defines the vocabulary of the domain which is described by the UML class diagrams, i.e. classes and class hierarchies. SHACL defines structural constraints of the domain. The mapper is created with the aim of mapping aeronautical UML models (AIXM 5.1.1., FIXM 3.0.1. SESAR) which adhere to a specific modelling style. Therefore, models provided to the mapper must fulfill certain semantic and syntactic requirements.

2.1.1 Semantic Requirements

The semantic requirements capture the modeling style followed by the aeronautical UML models. Other UML models mapped must also adhere to these.

- Requirement 1: Class names must be unique within a model (AIXM, FIXM, ...). There can be a UML class called "Route" in an AIXM based model and an FIXM based model but there must not be two different UML classes called "Route" in one model even if they are in different packages.
- Requirement 2: Models contain only directed associations.
- Requirement 3: Role names (at the target) of associations with the same source class must be unique within the source class.
- Requirement 4: Role names must exist, if there is more than one association between a source and a target class. If there is only one association and no role name provided, the role name is constructed using the name of the target class.

Requirement 1 is validated by the mapper and, if violated, throws an error. Requirements 2-4 are assumed to be UML model requirements and are not validated by the mapper.

2.1.2 Syntactic Requirements

- Models to-be mapped must be exported to a single XMI file (version 2.1) by the Enterprise Architect (version 14.1).

2.1.3 Architecture

The architecture of the mapper is shown in the figure below. A configuration file refers to XMI files and keeps lists of selected UML classes. A single configuration file is provided as input to the mapper. Based on the configuration file, selected subsets of models are extracted by the extractor module. Extracted subsets of models are mapped by model-specific plugins to RDFS/SHACL documents provided as RDF/XML files.

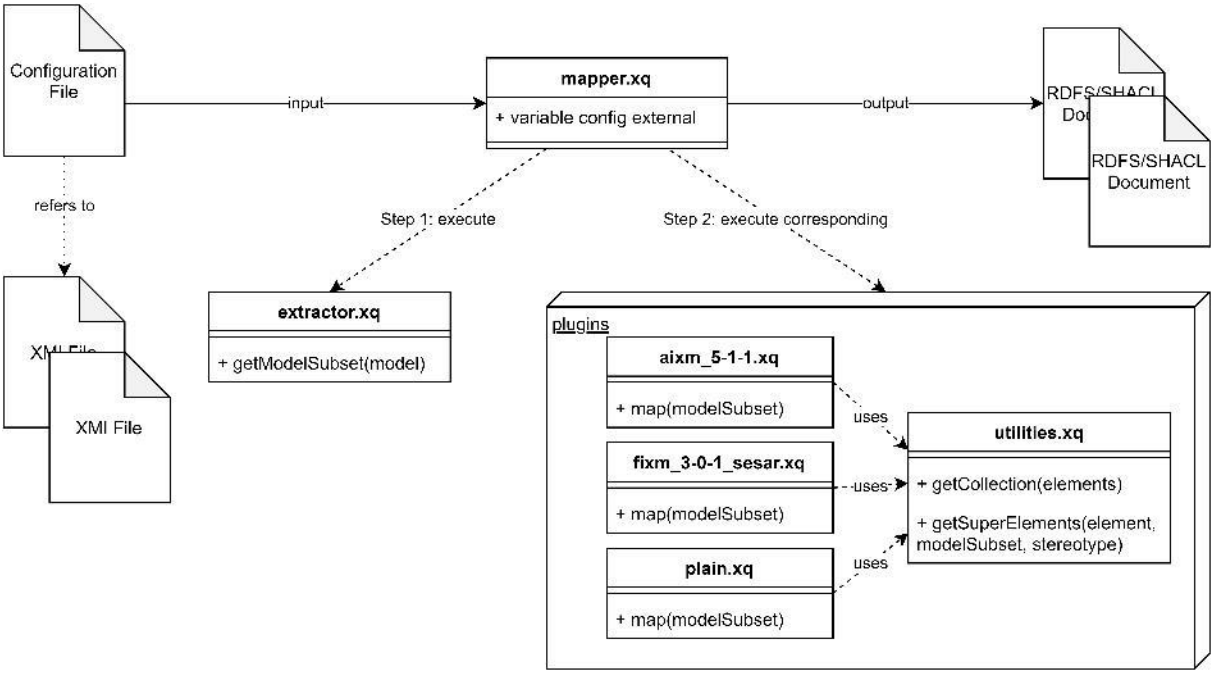


Figure 1 Architecture of the UML to RDFS/SHACL Mapper

2.1.4 How-to: Running the Mapper

There are different ways to run the mapper. Among them the following two ways have been applied in the project so far:

- Approach 1: Install a W3C compliant XQuery processor (e.g. BaseX) and run the file mapper.xq:
    - Using the BaseX command line tool:
      `basex -b$config="<locationOfTheConfigurationFile.xml>" mapper.xq`
    - Or using the BaseX GUI and manually binding the location of the configuration file to the config variable.
- Approach 2: Write a Java program which runs mapper.xq. This approach is demonstrated by RunMapper.java and has been applied in Task 4.3.

2.1.5 How-to: Transforming generated RDFS/SHACL documents to Turtle format

The generated RDFS/SHACL documents are in RDF/XML format which makes them rather difficult to read for humans. The Turtle RDF syntax is much easier to read and it is easy to transform from RDF/XML to Turtle. One approach is to use functionality from Apache Jena to do this transformation from Java. This approach is demonstrated in TransformXML2TTL.java and has been applied in Task 4.3.

2.1.6 How-to: Validating data graphs

There are a different ways to validate data with generated RDFS/SHACL documents. One approach is to invoke functionality provided by Apache Jena for RDFS reasoning and SHACL validation from Java. This approach is demonstrated in ValidationWithSHACL.java .

When validating RDF data against generated SHACL documents one needs to be careful to make sure that in the data the same namespaces are used as in the generated RDFS/SHACL document. One needs to be aware that the generated RDFS/SHACL schemas use namespace http://www.aisa-project.eu/vocabulary/aixm_5-1-1# for AIXM, http://www.aisa-project.eu/vocabulary/fixm_3-0-1_sesar# for FIXM, and http://www.aisa-project.eu/xquery/plain# for plain models.

2.1.7 Performance

Performance of the AISA XMI Mapper is not important in AISA because the schemas are typically only mapped once in the beginning. Hence, the XQuery code was not written to optimize performance. We have nonetheless conducted some preliminary performance studies to get a feeling for the mapper's performance characteristics. The performance studies have been conducted with a Lenovo Thinkpad T470p using the provided configuration files (see https://github.com/jku-win-dke/AISA-XMI-Mapper/blob/main/configurations/) and running mapper.xq with the BaseX GUI.

Execution	AIXM_DONLON.xml	AIXM_COCESNA.xml	FIXM_EDDF-VHHH.xml
1	32 821 ms	102 323 ms	12 555 ms
2	34 069 ms	102 758 ms	12 336 ms
3	33 137 ms	103 382 ms	12 524 ms
4	33 875 ms	103 906 ms	12 333 ms
5	34 443 ms	104 194 ms	12 335 ms
Average	33 669ms	103 313 ms	12 417 ms

Table 1 Results of Preliminary Performance Studies for the AISA XMI Mapper

2.2 Configuration File

2.2.1 Structure of the configuration file

In configuration files a set of UML classes of different models to-be mapped can be specified. The following attributes (or parameters) must be provided:

- **input**: The path to the model's XMI file.
- **type**: The type of the model determines the plugin used for mapping, i.e. type can be "aixm_5-1-1", "fixm_3-0-1_sesar" or "plain".
- **output**: The path to the to-be generated RDFS/SHACL document.
- **connectorLevel**: The number *i* of the connector level indicates that UML classes reachable by at most i connection step from a specified UML class are also to be included in the mapping.

The connectorLevel can be "1", "2", ..., "n". Value *n* stands for possibly infinite number of traversals. It is recommended to use "n" to include not visible classes (especially from stereotype "choice" in AIXM and FIXM) of a data graph. However, using connectorLevel "n" decreases performance and increases the size of the schema eventually including classes which are not required. If "n" is not used, then the connector level should be choosen in a way that it resolves necessary datatypes, e.g. in AIXM a minimum of connector level 4 is recommended.

The example below shows that the the classes "AirportHeliport" and "City" of the model at "input/AIXM_5.1.1.xmi" should be mapped by the plugin with the name "aixm_5-1-1" and using a connector level of "n".

```
<configuration>
    <selection>
            <models>
            <model      input="input/AIXM_5.1.1.xmi"      type="aixm_5-1-1"
            output="output/AIXM_example.xml">
                  <classes connectorLevel="n">
                        <class>AirportHeliport</class>
                        <class>City</class>
                  </classes>
            </model>
            <model ... >
                  ...
            </model>
            </models>
    </selection>
</configuration>
```

2.2.2 How-to: Writing a configuration file

In order to determine the UML classes to be selected, only consider UML classes from the namespace of the model. In addition, TimeSlice classes in AIXM cannot be selected because they are not part of the AIXM UML class diagrams, instead they are generated by the mapper if the parent feature (i.e. the feature class to which the feature timeslice class belongs) is selected. As an example, see the decisions for the configuration of the Donlon airport example.

Additional configuration files can be maintained without changing existing ones. However, the mapper can only consider one configuration file at a time. Make sure that the reference to the to-be used configuration file is correctly set in the mapper.xq (variable $config).

2.2.3 How-to: Extensions

In case the mapper should be further configurable by attributes or connections of classes which should only be mapped or which should not be mapped, this information should be provided as an inclusion or exclusion list in the configuration file. As an example:

```
...
        <classes>
                <class name="AirportHeliport">
                        <attributes>
                                <attribute>name</attribute>
                        </attributes>
                        <connectors>
                                <connector>serves</connector>
                        </connectors>
                </class>
                ...
        </classes>
...
```

The extractor module needs to be adapted accordingly. Furthermore, one must consider this configuration in the mapping plugins. Simply check while mapping attributes or connectors of an UML class if this attribute or connector is part of the list in the configuration file.

2.3 Mapper

2.3.1 mapper.xq

The mapper.xq is the main module of the mapper. The variable $config refering to the location of the configuration file needs to be set externally. For each model specified in the configuration file, it delegates the extraction process to the extractor.xq. After the extraction the mapper delegates the mapping process to the corresponding plugin, and finally writes the result to a file.

2.3.2 extractor.xq

The extractor.xq extracts a subset of UML classes and connections from an XMI file based on the configuration file. The following steps are performed:

1. Extracting the selected UML classes
2. Extracting of corresponding UML classes and connections (recursive):
    1. Extract outgoing connections from the set of selected and extracted UML classes
    2. Extract UML classes with an ingoing connection from 2.1.
    3. Extract UML classes which are association classes of connections from 2.1.
    4. Extract UML classes which are the range of attributes of selected and extracted UML classes
    5. If connectorLevel="n":
        1. If the extracted model subset increased in size, then add another cycle of extraction.
        2. Otherwise, return the extracted model subset.
    6. Otherwise:
        1. If the extracted model subset increased in size and connectorLevel > 1, then add another cycle of extraction and reduce the connectorLevel by 1.

2. Otherwise, return the extracted model subset.

In the end, the extracted model subset is returned to the mapper.xq.

2.3.3 Plugins

Plugins are implementations of different models' mapping semantics. Each plugin is a XQuery module with the task to map a given model subset to an RDFS/SHACL document. We use different plugins for different models because there is no one fits all mapping approach. For example, stereotypes or attributes may have different meanings or may be used differently in different models. By default, the following plugins are available:

1. utilities.xq provides basic functionality for plugins
2. aixm_5-1-1.xq for AIXM 5.1.1
3. fixm_3-0-1_sesar.xq for FIXM 3.0.1 SESAR
4. plain.xq for plain UML models (no consideration of stereotypes)

The mapper can simply be extended by adding new plugins as XQuery modules to the plugin folder and by adding them to the delegation of the mapping process in the mapper.xq (variable $mappedModel). A new plugin may be useful, if a model needs to be mapped that uses stereotypes differently than in previous models. In addition, a new plugin may also be useful, if an existing plugin needs to be adapated, e.g. different namespaces or updating the meaning of a stereotype.

2.3.3.1 utilities.xq

The utilities.xq provides basic functionality used in the plugins. It provides two functions:

1. Transform a sequence of elements to an RDF/XML list
2. Find super classes of an class in a given model subset with two options:
    1. Super elements are not from a certain stereotype
    2. Call this function recursively to find all super classes of a class

2.3.3.2 plain.xq

The plain.xq targets models which are not based AIXM and FIXM and do not use stereotypes.

2.3.3.2.1 Mapping of UML classes

For each UML class from the extracted subset, a SHACL shape / RDFS class is generated. Super classes of a UML class are mapped into rdfs:subClassOf and sh:and. Attributes are mapped into optional property shapes, while connections are mapped into property shapes with the cardinality of the relationship being represented in the sh:minCount and sh:maxCount. If a UML class is an association class, connections are resolved such that the source class has a property shape which targets the association class, while the association class has a property shape which targets the target class.

2.3.3.3 aixm_5-1-1.xq

The aixm_5-1-1.xq targets models which are based on AIXM 5.1.1. First, basic elements are added and then, element by element of the extracted model subset is mapped.

2.3.3.3.1 Basic Elements for AIXM features

If the extracted model subset contains an element with stereotype "feature", the following basic classes are added to the result:

An empty SHACL shape named "aixm:AIXMFeature" which could be extended by general AIXM feature properties. This shape represents the abstract AIXMFeature class. Its identifier attribute is not mapped into a property shape because the identifier of features is used as resource identifier (IRI).

```
aixm:AIXMFeature a sh:NodeShape .
```

A SHACL shape named "aixm:AIXMTimeSlice" which keeps general and mandatory attributes of feature time slices, i.e. gml:validTime, aixm:interpretation, aixm:sequenceNumber, aixm:correctionNumber.

```
aixm:AIXMTimeSlice
 a sh:NodeShape ;
 sh:property [
        sh:maxCount 1 ;
        sh:minCount 1 ;
        sh:node aixm:NoNumberType ;
        sh:path aixm:correctionNumber
 ] ;
        sh:property [
                sh:maxCount 1 ;
                sh:minCount 1 ;
                sh:node aixm:NoNumberType ;
                sh:path aixm:sequenceNumber
        ] ;
        sh:property [
                sh:maxCount 1 ;
                sh:minCount 1 ;
                sh:node aixm:TimeSliceInterpretationType ;
                sh:path aixm:interpretation
        ] ;
        sh:property [
                sh:class gml:TimePeriod ;
                sh:maxCount 1 ;
                sh:minCount 1 ;
                sh:path gml:validTime
        ] .
```

A SHACL shape and RDFS class named "gml:TimePeriod" (type of gml:validTime) which keeps a gml:beginPosition and a gml:endPosition.

```
 gml:TimePeriod
        a rdfs:Class , sh:NodeShape ;
        sh:property [
                sh:maxCount 1 ;
                sh:minCount 1 ;
                sh:node gml:TimePrimitive ;
```

```
                sh:path gml:endPosition
        ] ;
        sh:property [
                sh:maxCount 1 ;
                sh:minCount 1 ;
                sh:node gml:TimePrimitive ;
                sh:path gml:beginPosition
        ] .
```

A SHACL shape named "gml:TimePrimitive" (type of gml:beginPosition and gml:endPosition) which can have xsd:dateTime as rdf:value or can be a gml:indeterminatePosition.

```
gml:TimePrimitive
        a sh:NodeShape ;
        sh:property [
                sh:datatype xsd:string ;
                sh:maxCount 1 ;
                sh:path gml:indeterminatePosition ;
                sh:in ( "after" "before" "now" "unknown" )
        ] ;
        sh:property [
                sh:datatype xsd:dateTime ;
                sh:maxCount 1 ;
                sh:path rdf:value
        ] ;
        sh:xone (
                [
                        sh:property [
                                sh:minCount 1 ;
                                sh:path rdf:value
                        ]
                ]
                [
                        sh:property [
                                sh:minCount 1 ;
                                sh:path gml:indeterminatePosition
                        ]
                ]
        ) .
```

A SHACL shape named "aixm:TimeSliceInterpretationType" (type of aixm:interpretation) which can have the rdf:value "BASELINE" or "TEMPDELTA".

```
aixm:TimeSliceInterpretationType
        a sh:NodeShape ;
        sh:property [
                sh:in ( "BASELINE" "TEMPDELTA" ) ;
                sh:maxCount 1 ;
                sh:minCount 1 ;
                sh:path rdf:value
        ] .
```

A SHACL shape named "aixm:NoNumberType" (type of aixm:sequenceNumber and aixm:correctionNumber) which has an xsd:unsignedInt as rdf:value.

```
77.   aixm:NoNumberType
78.           a sh:NodeShape ;
79.           sh:property [
80.                   sh:datatype xsd:unsignedInt ;
81.                   sh:maxCount 1 ;
82.                   sh:minCount 1 ;
83.                   sh:path rdf:value
84.           ] .
```

These basic elements are mandatory for AIXM features and not modelled accordingly in the AIXM 5.1.1 UML class diagrams, therefore, they are added manually. Other GML constructs like gml:pos inherited through gml:Point are also not part of the AIXM 5.1.1 UML class diagrams. A generated AIXM RDFS/SHACL document could be combined with a GML RDFS/SHACL document for a complete validation of the data.

2.3.3.3.2 Basic Mapping Methods

Some mapping methods in AIXM are used in multiple cases, including mapping of attributes, connectors and association classes:

Attributes of a UML class are mapped into optional (i.e. sh:minCount 0) property shapes with the attribute type being the target node. The name of the attribute is used as sh:path. Example attribute aixm:name of aixm:AirportHeliport:

```
1.  aixm:AirportHeliportTimeSlice
2.   sh:property [
3.           sh:path aixm:name ;
4.           sh:node aixm:TextNameType ;
5.           sh:maxCount 1 ;
6.   ] .
```

Connections to other UML classes are mapped into property shapes with the sh:minCount and sh:maxCount representing the cardinality of the relationship. The target class is specified by the sh:class constraint. If a role name is provided, this name is used for sh:path. Otherwise, the sh:path is the combination of "the" plus the target class name. There is an exception of mapping connections: association classes. If an association class for a connection exists, the property of the source UML class targets the association class. Furthermore, the association class has a property shape which targets the connection's target class. Example of aixm:AirportHeliport with a connection to the class aixm:City and a connection to the class aixm:OrganisationAuthority with an association class with the association class aixm:AirportHeliportResponsibilityOrganisation:

```
7.  aixm:AirportHeliportTimeSlice
8.   sh:property [
9.           sh:class aixm:City ;
10.                  sh:path aixm:servedCity
11.          ] ;
12.          sh:property  [
13.                  sh:class
   aixm:AirportHeliportResponsibilityOrganisation ;
14.                  sh:maxCount 1 ;
15.                  sh:path aixm:responsibleOrganisation
16.          ] .
```

A UML class can be an **association class** for a connection between two other classes. As already explained above, a property shape is added to an association class targeting the connection's target class. The sh:path is always the combination of "the" plus the target class name since the role name is already used by the source class. Example of the connection between aixm:AirportHeliport and aixm:OrganisationAuthority with aixm:AirportHeliportResponsibilityOrganisation as assocation class:

```
aixm:AirportHeliportResponsibilityOrganisation
        sh:property [
                sh:class aixm:OrganisationAuthority ;
                sh:maxCount 1 ;
                sh:minCount 1 ;
                sh:path aixm:theOrganisationAuthority
                   ] .
```

2.3.3.3.3 Mapping of UML classes

UML classes of AIXM 5.1.1 are mapped based on their stereotype:

Stereotype **"feature"**: For each UML class with stereotype "feature" two SHACL shapes / RDFS classes are generated. The first SHACL shape / RDFS class extends the aixm:AIXMFeature shape and has only one property named aixm:timeSlice. The second SHACL shape / RDFS class extends the aixm:AIXMTimeSlice shape and is named like the UML class with the phrase "TimeSlice" added at the end. For each super class of the feature, a rdfs:subClassOf and sh:and statement are added to the corresponding TimeSlice. Furthermore, the TimeSlice holds all attributes and connections of the corresponding feature as property shapes. Therefore, the three basic methods explained above are used. Example feature aixm:AirportHeliport with aixm:AirportHeliportTimeSlice:

```
aixm:AirportHeliport
 a rdfs:Class , sh:NodeShape ;
 sh:and ( aixm:AIXMFeature ) ;
 sh:property [
        sh:path aixm:timeSlice ;
        sh:class aixm:AirportHeliportTimeSlice ;
 ] .
aixm:AirportHeliportTimeSlice
 a rdfs:Class , sh:NodeShape ;
        sh:and ( aixm:AIXMTimeSlice ) ;
        sh:property [
                sh:path aixm:name ;
                sh:node aixm:TextNameType ;
                sh:maxCount 1 ;
        ] ... .
```

Stereotype **"object"**: For each UML class with stereotype "object" a SHACL shape / RDFS class is generated. Super classes and the three basic mapping methods are used exactly in the same way as in UML classes with stereotype "feature". The only difference between features and objects is that there are no added TimeSlice classes for objects. Example aixm:AirportHeliportUsage:

```
aixm:AirportHeliportUsage
        a rdfs:Class , sh:NodeShape ;
        rdfs:subClassOf aixm:UsageCondition ;
        sh:and ( aixm:UsageCondition ) ;
```

```
20.          sh:property [
21.                  sh:maxCount 1 ;
22.                  sh:node aixm:CodeOperationAirportHeliportType ;
23.                  sh:path aixm:operation
24.          ] .
```

Stereotype **"CodeList"**: For each UML class with stereotype "CodeList" a SHACL shape is generated. Its attribute names are allowed values and therefore mapped as a SHACL list into sh:in. If a super class with stereotype "XSDsimpleType" exists, a SHACL datatype statement is added. Example aixm:NilReasonEnumeration and aixm:UomDistanceVerticalType:

```
25.   aixm:NilReasonEnumeration
26.          a sh:NodeShape ;
27.          sh:datatype xsd:string ;
28.          sh:in ( "inapplicable" "missing" "template" "unknown"
   "withheld" "other" ) .
29.   aixm:UomDistanceVerticalType
30.          a sh:NodeShape ;
31.          sh:datatype  xsd:string ;
32.          sh:in ( "FT" "M" "FL" "SM" "OTHER" ) .
```

Stereotype **"DataType"**: For each UML class with stereotype "DataType" a SHACL shape is generated. For each super class with stereotype "DataType", a sh:and statement is added. The property shape with sh:path rdf:value is always added to classes with stereotype "DataType". If a super class with stereotype "XSDsimpleType" exists, a sh:datatype constraint is added for the rdf:value property shape. If a super class with stereotype "CodeList" exists, a sh:node constraint is added for the property shape of rdf:value. If an attribute with stereotype "XSDfacet" exists, it is added as corresponding SHACL constraint (e.g. minLength) for the rdf:value property shape. If a super class with stereotype "XSDsimpleType" exists, a SHACL datatype constraint is added for the rdf:value property shape. All other attributes with stereotype not being "XSDfacet" are mapped according to the basic mapping of attributes. If an attribute from type "NilReasonEnumeration" exists, a SHACL exactly one (sh:xone) constraint is added, specifying that either a aixm:nilReason can occur or all other properties and rdf:value. Classes with stereotype "DataType" are typically used in attributes and not in connections, thus, the basic mapping methods 2 and 3 are not used. Example aixm:ValDistanceVerticalType and its super class aixm:ValDistanceVerticalBaseType:

```
33.   aixm:ValDistanceVerticalType
34.          a sh:NodeShape ;
35.          sh:and ( aixm:ValDistanceVerticalBaseType ) ;
36.          sh:property [
37.                  sh:maxCount 1 ;
38.                  sh:node aixm:NilReasonEnumeration ;
39.                  sh:path aixm:nilReason
40.          ] ;
41.          sh:property [
42.                  sh:maxCount 1 ;
43.                  sh:node aixm:UomDistanceVerticalType ;
44.                  sh:path aixm:uom
45.          ] ;
46.          sh:property [
47.                  sh:maxCount 1 ;
48.                  sh:path rdf:value
```

```
49.          ] ;
50.          sh:xone (
51.                  [
52.                          sh:property [
53.                                  sh:minCount 1 ;
54.                                  sh:path rdf:value
55.                          ] ;
56.                          sh:property [
57.                                  sh:path aixm:uom
58.                          ]
59.                  ]
60.                  [
61.                          sh:property [
62.                                  sh:minCount 1 ;
63.                                  sh:path aixm:nilReason
64.                          ]
65.                  ]
66.          ) .
67.   aixm:ValDistanceVerticalBaseType
68.          a sh:NodeShape ;
69.          sh:property [
70.                  sh:datatype xsd:string ;
71.                  sh:maxCount 1 ;
72.                  sh:path rdf:value ;
73.                  sh:pattern "^((\\+|\\-){0,1}[0-9]{1,8}(\\.[0-
   9]{1,4}){0,1})|UNL|GND|FLOOR|CEILING$"
74.          ] .
```

Stereotype **"choice"**: For each UML class with stereotype "choice" a SHACL shape is generated. The generated SHACL shape is only a link between a UML class and a choice between allowed classes. Therefore, the SHACL shape of the choice class only contains the connections in a sh:xone (only one connection is allowed). Example aixm:SignficantPoint:

```
75.   aixm:SignificantPoint
76.   a sh:NodeShape ;
77.   sh:xone (
78.          [ sh:class aixm:AirportHeliport ]
79.          [ sh:class aixm:TouchDownLiftOff ]
80.          [ sh:class aixm:RunwayCentrelinePoint ]
81.          [ sh:class aixm:Point ]
82.          [ sh:class aixm:Navaid ]
83.          [ sh:class aixm:DesignatedPoint ]
84.   ) .
```

Stereotype **"XSDsimpleType"**: No mapping. Super classes with this stereotype are used to derive sh:datatype constraints in sub classes (with stereotype "DataType" or "CodeList").

Stereotype **"XSDcomplexType"**: No mapping.

No stereotype: UML classes with no stereotypes are mapped the same as UML classes with stereotype "object". Example gml:Point:

```
85.   gml:Point a rdfs:Class , sh:NodeShape .
```

2.3.3.4 fixm_3-0-1_sesar.xq

The fixm_3-0-1_sesar.xq target models which are based on FIXM 3.0.1 SESAR.

2.3.3.4.1 Mapping of UML classes

UML classes of FIXM 3.0.1 SESAR are mapped based on their stereotype:

Stereotype **"enumeration"**: For each UML class with stereotype "enumeration" a SHACL shape is generated. It has a single mandatory (sh:minCount 1) property with the sh:path being fixm:uom or rdf:value. In case the name of the UML class contains "Measure" the sh:path is fixm:uom, otherwise it is rdf:value. The attribute names of the UML class are allowed values and therefore mapped as a SHACL list into sh:in. Example fixm:AbrogationReasonCode and fixm:TemperatureMeasure:

```
fixm:AbrogationReasonCode
 a sh:NodeShape ;
 sh:property [
        sh:in ( "TFL" "ROUTE" "CANCELLATION" "DELAY" "HOLD" ) ;
        sh:minCount 1 ;
        sh:path rdf:value
 ] .
fixm:TemperatureMeasure
 a sh:NodeShape ;
        sh:property [
                sh:in ( "FARENHEIT" "CELSIUS" "KELVIN" ) ;
                sh:minCount 1 ;
                sh:path fixm:uom
        ] .
```

Stereotype **"choice"**: For each UML class with stereotype "choice" a SHACL shape is generated. There are two different cases: (1) a choice class is used as attribute or (2) a choice class is used via connections. In case (1) the generated SHACL shape is only a link between a UML class and a choice between allowed attributes or connected classes. Therefore, the SHACL shape of the choice class only contains the attributes and connections in a sh:xone (only one attribute or connection is allowed). In case (2) the generated SHACL shape is also an RDFS class. It also provides the choice between attributes and connections in a sh:xone but including their paths and maxCount constraint. Example fixm:PersonOrOrganization (case 1) and fixm:AircraftType (case 2):

```
 fixm:PersonOrOrganization
        a sh:NodeShape ;
        sh:xone (
                [ sh:class  fixm:Organization ]
                [ sh:class  fixm:Person ]
        ) .
 fixm:AircraftType
        a rdfs:Class , sh:NodeShape ;
                sh:xone (
                [
                        sh:property [
                                sh:maxCount 1 ;
                                sh:minCount 1 ;
                                sh:node fixm:IcaoAircraftIdentifier ;
                                sh:path fixm:icaoModelIdentifier
                        ]
```

```
                ]
                [
                        sh:property [
                                sh:maxCount 1 ;
                                sh:minCount 1 ;
                                sh:node fixm:FreeText ;
                                sh:path fixm:otherModelData
                        ]
                ]
        ) .
```

No stereotype: For each UML class with no stereotype a SHACL shape is generated. If a UML class or one of its super classes are not based on an XSD datatype, it is also an RDFS class with its super classes as rdfs:subClassOf Triple added. In every case, super classes are added as sh:and statements. In case, there is an attribute called "uom", an sh:and statements needs to include the SHACL shape of that attribute. If the UML class (or one of its super classes) is connected to an XSD datatype, a SHACL property shape with sh:path rdf:value is added (together with its constraints and datatype). Attributes of classes are mapped into optional property shapes. In case the type of an attribute is one of a few possible XSD datatypes, the attribute's property shape targets a blank node shape with a single property shape that has the rdf:value as sh:path. The blank node shape is necessary to keep the structure of instance data consistent. In all other cases, attributes are simply mapped into optional property shapes. Connections of a UML class are also mapped into property shapes. Example attribute fixm:topOfClimb with an XSD datatype in fixm:TrajectoryPointRole, and attribute fixm:aircraftColours as well as connection fixm:aircraftType in fixm:Aircraft:

```
fixm:TrajectoryPointRole
        a rdfs:Class , sh:NodeShape ;
        sh:property [
                sh:maxCount 1 ;
                sh:node [
                        a sh:NodeShape ;
                        sh:property [
                                sh:datatype xsd:boolean ;
                                sh:path rdf:value
                        ]
                ] ;
                sh:path fixm:topOfClimb
        ] ... .
fixm:Aircraft
        a rdfs:Class , sh:NodeShape ;
        rdfs:subClassOf fixm:Feature ;
        sh:and ( fixm:Feature ) ;
        sh:property [
                sh:node fixm:FreeText ;
                sh:maxCount 1 ;
                sh:path fixm:aircraftColours
        ] ;
        sh:property [
                sh:class fixm:AircraftType ;
                sh:maxCount 1 ;
                sh:path fixm:aircraftType
        ] ... .
```

2.4 RDFS/SHACL Document

The resulting document combines RDFS and SHACL because in AISA both formats are generated from the same source and used together. The combination of RDFS and SHACL is very similar to UML class diagrams.

Example aixm:AirportHeliport:

```
aixm:AirportHeliport
        a rdfs:Class ;          # This is RDFS!
        a sh:NodeShape ;        # This is SHACL!
        ... .
```

3 Proof-of-Concept KG System

We developed the **architecture** for the AISA KG system, a compact **Java library** (delivered as Java package in GitHub repository https://github.com/bneumayr/aisa-kg-system/) supporting this architecture, and a **small proof-of-concept KG system** (also delivered as a Java package in the GitHub repository) exemplifying the architecture as well as the usage of the Java library.

The AISA Project-level Concept of Operations set out in Chapter 5 of Deliverable 2.1 has at its center the ATC Knowledge Graph (KG). In this section we describe an architecture for the data and metadata in such a KG and for the software components for incrementally processing and querying the data in the KG. The *Proof-of-Concept KG System* exemplifies this architecture and has the purpose of guiding further developments in WP 4 and WP 5. The proposed architecture of a KG system facilitates *SPARQL Queries Capturing Monitoring Tasks* based on traffic/airspace data converted to RDF. It further accomodates the integration with other components such as the Prolog Reasoning Engine in Prolog developed in Task 4.2 and Machine Learning Modules developed in WP3. The main goal is to develop and assess the concept of the artificial SA based on a KG from a functional perspective rather than to consider requirements of a real-time life system.

Air traffic data comes at rather high frequency, for example new flight states every 10 seconds, and the SPARQL queries capturing monitoring task need to consider the newly incoming data but also consider how the new flight states compare to previous flight states. Monitoring queries should not be formulated only against the 'raw' input data but the input data should be processed beforehand, calculations made, common parts of queries should be executed once and materialized so that many queries can reuse the results. We also have to consider that some parts of monitoring queries are difficult or impossible to formulate in SPARQL, especially spatio-temporal calculations such as, as a simple example, the distance between two flights. With new incoming data the intermediate results have to be updated as well, for this purpose the KG system needs to support incremental processing – avoiding the need to recalculate and query everything from scratch once new data arrives. Furthermore, an artificial SA is history sensitive and the KG thus needs to keep track of all its previous states.

3.1 Requirements and Setting

Based on the project-level concept of operations and discussions we collected the following requirements for a KG system architecture suitable for AISA.

The architecture should accommodate

- Recurring SPARQL queries for monitoring the situation (situational awareness is 'translated' into SPARQL queries).
- Ad-hoc SPARQL queries for checking the KG's state during an experiment.
- RDF Schema reasoning to consider subClassOf and subPropertyOf hierarchies in the global schema when necessary
- SHACL Validation: checking conformance against global schema when loading data

- Simple Inference/Derivation rules are formulated in SPARQL update requests or construct queries (there is no need for SHACL rules which would serve the same purpose).
- Arithmetic and spatio-temporal processing will be coded in Java programs (e.g., calculate distance between two flights) and not in SPARQL.
- External modules (e.g., machine learning modules, data from KG as input for ML models and predictions from ML as input to KG) will be loosely coupled via file import and epxort and with asynchronous processing.
- Data loading, i.e., loading of asserted data, can be done incrementally (data loaded into the KG as they arrive over time) or at once (data with different timestampls loaded into the KG at once
- The experiments conducted in Task 5.1 to assess AI SA will be conducted with one software component which should have central control over the experiment.
- History sensitiveness and traceability: every piece of data or metadata in the KG comes with the following metadata: when was the piece of data inserted into the KG and by which module. Further, in case of derived informaton, it must be traceable based on which state of the KG the data was derived.
- Distinguishing logical time (or simulation time) and physical system time: to rerun experiments or to run experiments in 'slow-motion' the system needs to distinguish between logical system time and physical system time (the real time of the system for performance measurements). To make the difference clear: when rerunning an experiment the logical times remain the same while the physical times change. Logical time progress should be under the control of the person or system who runs an experiment.
- Transaction time (logical and/or physical system time) vs Observation/Occurrence time: The occurrence time is the time in the real world when an event happens or is predicted to happen, the transaction time (no matter if physical or logical) is the time when the information about the event is added to the KG.
- Incremental Processing/Querying: adding data to the KG should not make necessary to rerun from scratch all queries. Recurring queries should only consider the data added since their last invocation.

Requirements for technology choices

- The programming language of choice in AISA is Java. There is no need for GUIs or otherwise advanced user interfaces. Experiments will be automated in Java or via simple command line interfaces. It is more important to provide the functionality via Java or provide Java code snippets demonstrating how things should be done according to the architecture.
- The core components should be based on open-source technology so that the produced software can also be made available open source as to maximize impact

Keeping in mind the short project duration and limited resources one of the central design goals is to keep everything as simple as possible. To this end we specifiy deliberate limitations of the KG systems. These limitations help to simplify the implementation, maintenance, orchestration, and traceability of the experiments:

- Append-only: once added to the KG, data is not changed/updated anymore. Only outdated data that is not needed/queried anymore may be retracted from the KG (without consequences on the querying).

- Central control with serialized engine invocations: For the setup of the experiments we assume central control with synchronous (i.e., serialized) invocation of modules and, hence, serialized 'transactions' on the KG, avoiding the need for concurrency control which would add to the technical complexity of the KG system without clear benefits for the experiments conducted in WP 5. The only exception to this rule are ad-hoc queries (but by only querying 'committed' named-graphs and ignoring 'in-progress' named graphs these ad-hoc queries can also be considered 'serialized').
- Asynchronous processing (with external modules) is supported by two connected module invocations (two separate but connected KG transactions) akin to check-out and check-in in version control systems: first, an invocation to export data from the KG which is used as input for the external module and, second, an invocation to import the results of the external module to the KG. The external processing happens in-between these two invocations in an asynchronous manner without access to the KG.

RDF gives a lot of freedom. How to deal with this freedom and what basic structure we impose is the topic of this section. Named graphs are used to facilitate the management of the knowledge graph (such as for incremental/partial replication, incremental processing, purging of old data). The named graphs do not come with metadata that are of interest in user queries but only with administrative metadata that is needed to decide whether a named graph should be considered in a query or in some part of a query.

3.2 System Components and Deployment

All the basic functionality for working with RDF, RDFS, SPARQL, and SHACL in Java is provided by the open-source software library Apache Jena. As RDF graph store we use Jena TDB and as SPARQL server we use Apache Jena Fuseki which comprises Jena TDB.

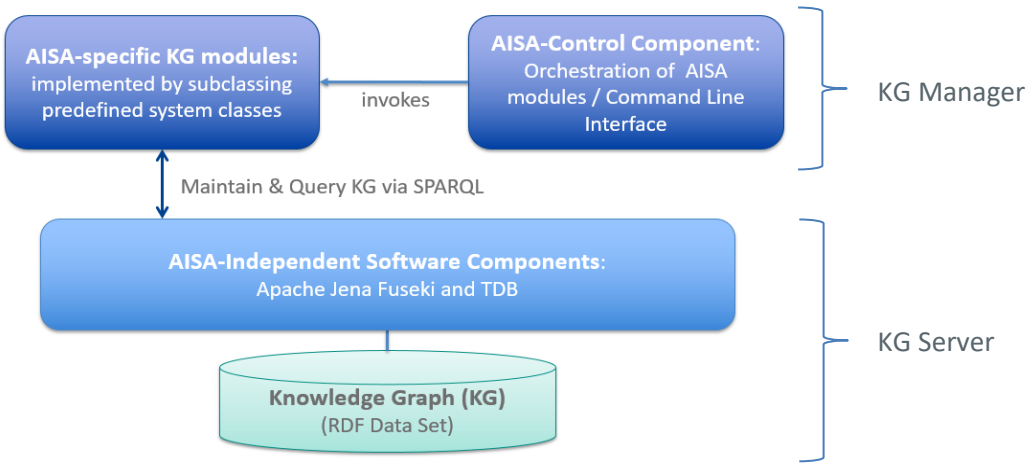


Figure 2 Components of the KG System

The KG system runs in two Java processes, one is the **KG Server**, which is an Apache Jena Fuseki instance, which acts as storage engine and SPARQL server, the other is the central control

component referred to as **KG Manager** which has registered a set of KG modules (implemented by subclassing Java class provided with our Java library) which interact with the KG Server via SPARQL. The loosely coupled integration via file export and file import of external software components (such as the ML modules developed in WP 3) is also accomplished by KG modules which run as part of the KG Manager.

3.2.1 Configuring Apache Jena Fuseki

Currently we are using version 3.16 of Apache Jena Fuseki 3.16 as a stand-alone server with UI. (Apparently there are problems with the current version 3.17 with creating and fetching datasets). Fuseki is configured with a single dataset (holding the AISA KG), made available via service http://localhost:3030/aisakg/. To simplify switching between different configurations (for performance studies) of the service available at /aisakg/, different configuration files may be added to Fuseki's main directory with different batch files to start Fuseki with one of these configurations.

The configuration files are contained in the github repository in folder fuseki-configs/. They can be added to Fuseki's installation directory and started with, for example:

```
> java -Xmx1200M -jar fuseki-server.jar --config=AISA-config-tdb.ttl
```

The Fuseki standalone server also serves a web application which can be used to pose queries and inspect the current state of the AISA KG. There is no need to use this web interface for adding datasets or for further configurations. We will mainly use it for ad-hoc queries to inspect/debug the state of the AISA KG during experiments which are controlled from Java programs. The web interface also provides an overview of the available services/endpoints (as configured by us) and some statistics.

3.3 KG System Architecture

From a conceptual/logical perspective, a KG system comprises a **KG**, which is an RDF Dataset, and a **KG Module System** (see Figure 3).

The KG which can be identified and located by its URL consists of a set of Named Graphs, i.e., RDF Graphs each identified by a URI. We distinguish Named Graphs into *Static Data Named Graphs,* which are created once and do not change afterwards, *Dynamed Data Named Graphs,* and *New Data Named Graphs*. A dynamic data named graph has a sequence of new data named graphs as components with new data named graphs appended over time. The URI of a new data named graph is constructed from the URI of the dynamic data named graph it belongs to and its sequence number. A new data named graph comes with temporal metadata indicating when it was inserted and committed to the KG (commitTime) and on which state of the KG it is based on (basedOnTime). A dynamic data named graph may materialize the union of its new data named graphs with metadata lastRefresh indicating when this union was refreshed the last time.

The KG Module System (which represents the central control component of the KG Manager process) comprises a set of KG Modules and controls the invocation of these modules and is also responsible for advancing the logical time of experiments. KG modules are distinguised into Single-run modules and multiple-run modules. A single-run module creates a static data named graph and a multiple-run

module creates and maintains a dynamic data named graph and creates the new data named graphs belonging to that dynamic data named graph. Every time a multiple-run module is invoked, it creates a new new data named graph. Multiple-run modules are further distinguished into *internal modules* and *external modules*. Internal modules are invoked by calling method run() and are executed in a serialized manner with the resulting new data named graph being inserted and committed immediately. External modules are invoked by calling method exportInput() together with a later call of method importResults(). The KG (a persistent RDF dataset on the KG Server) is fully managed by the KG Module System in that every named graph in that RDF dataset is created by a module.

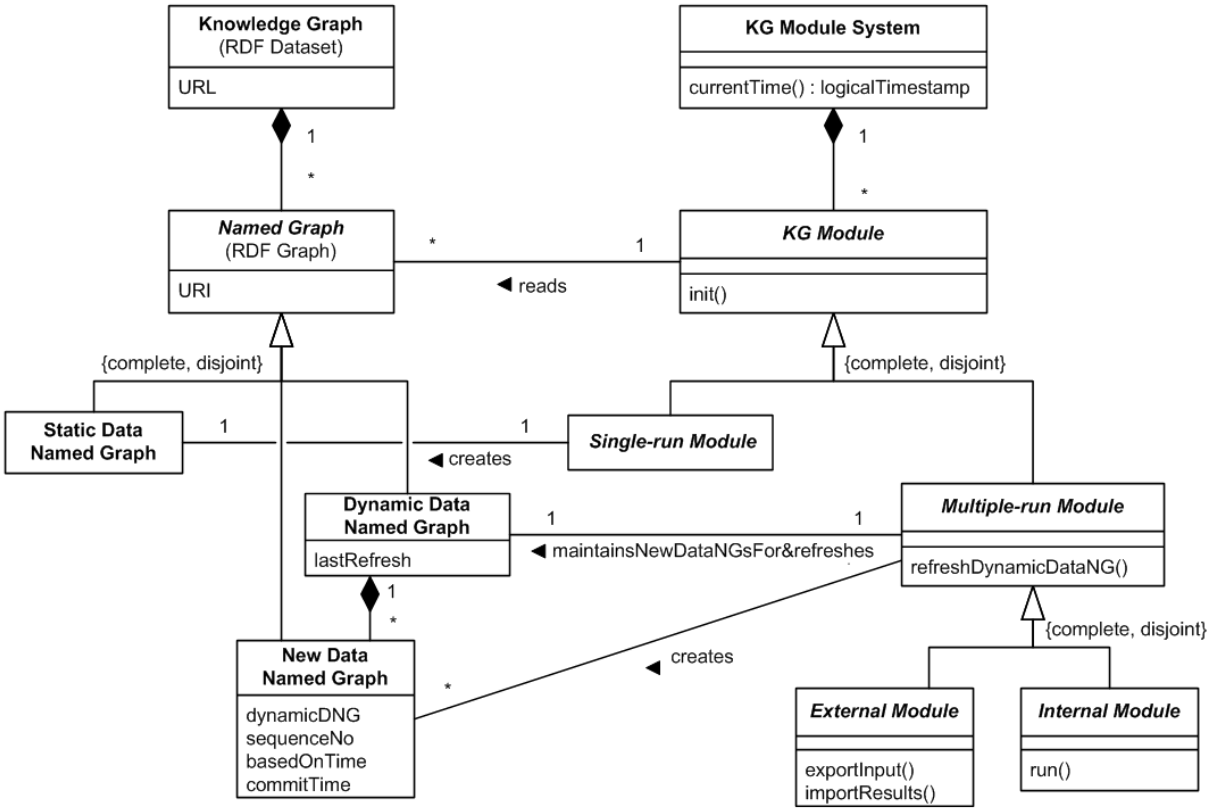


Figure 3 Conceptual structural model in UML of the KG and the KG Module System

3.3.1 Logical Time vs Physical Time

In order to rerun experiments with exactly the same input data (possibly also with the same ML predictions) but possibly with different monitoring queries or different rule-based knowledge we need to distinguish logical time (or simulated time) from the physical system time. The physical system time is only used for performance time measurements while the logical time is in full control of the KG manager.

3.3.2 Metadata Management

Each invocation of a module (via init(), run(), or exportInputs() followed by importResults()) produces a new named graph. What metadata is attached to such a named graph depends on the module, and it is hard-coded in the module. The module's named graph contains the module's static metadata. A new data named graph has at least the following generic metadata (which is managed by the system and needs not to be taken care of by the developers of a concrete KG system) :

- invocation time (a *logical* timestamp)
- commit time (a *logical* timestamp)
- the dynamic data named graph it belongs to and indirectly the module that inserted it
- sequence no (to simplify referring to previous turn, etc)

3.3.3 Selecting collections of Named Graphs of Modules

Objects are fully described within one graph, or, if the description is fragmented over multiple graphs, then this fragmentation is known at the schema level. Thus the developer can ask, based on this fragmentation schema knowledge, queries in the form "… WHERE { GRAPH ?g { ?s ?p ?o }; GRAPH ?g2 { ?s ?p2 ?o2 } } …" without the danger of missing query solutions. If objects are represented in multiple graphs then the developer knows which parts are present in which graph and can formulate the queries accordingly. Thus, there is no necessity to build union graphs in advance for querying; instead as part of the queries for each graph pattern the graphs to be considered are specified within the query. To hugely simplify this approach, the Java library described in the next section comes with functionality for SPARQL preprocessing as described in Section 3.5. Additionally one has the option to build materialized union graphs by implementing method refreshDynamicDataNG() by the module and by calling this method to refresh the materialized union graph.

3.3.4 Deleting outdated new data named graphs

Data should be organized in a way that allows to delete/archive outdated new data named graphs without affecting query results. In our approach new data named graphs can only be deleted/archived as a whole. If one wants to persist selected parts of a named graph one has to copy these before. As a consequence of the deliberate simplification that every multiple-run module is associated with exactly one dynamic data named and graph and hence one sequence of new data named graphs it is not possible to implement (in a simple and traceable form) different archival strategies within one module. This small drawback is much outweight by the major simplifications due to the enforcement of a one-to-one relationship between multiple-run modules and dynamic data named graphs.

3.4 Java Library

The Java package supporting this approach is `at.jku.dke.aisa.kg` in the GitHub repository. Let us briefly explain how the conceptual model of Figure 3 is realized by this Java package. How to build a concrete KG system based on this package is demonstrated by package `at.jku.dke.aisa.kg.sample1` which is described in Section 3.8.

An instance of Java class *KGModuleSystem* has a connection *con* to an RDF dataset (typically stored on and served via Fuseki KG Server but the RDF dataset can also be in-memory and in the same Java process), defines namespace *prefixes* used by all modules, and manages the *logicalTime* 'clock' used for timestamping module versions.

UML class *KG Module* (see Figure 3) is realized by a Java interface KGModule and a Java class AbstractKGModule. The interface defines methods that are meant to be called from the KGModuleSystem to which the KGModule belongs.

- init() creates a static data or dynamic data named graph
- register(KGModuleSystem) registers the KGModuleSystem with the module, is the inverse of KGModuleSystem::register(KGModule) and called by the latter to establish a bidirectional relationships between the KG module system and its modules
- getName() returns the module's short name which is unique within the KGModuleSystem
- getModuleIri() returns the module's full name (IRI) which is constructed from the global graph namespace specified by the KGModuleSystem and the module's short name

The abstract class AbstractKGModule implements interface KGModule and specifies common functionality shared by all KG modules including the management of generic/administrative metadata. It implements the above methods with method init() calling an abstract method **doInit()** which is to be implemented by each concrete subclass of KGModule to load or create the module's named graph (either a static data named graph or a dynamic data named graph). Method doInit() is not included in the interface since it should be hidden from the KG manager, the latter should only call method init() which takes care of the generic metadata management which should not be reimplemented or overwritten by the concrete subclasses.

Most KG modules will have their own concrete subclass of KGModule. In this case the module's short name will be hard-coded in the constructor of the concrete subclass making the class a singleton class. The architecture also allows concrete subclasses of KGModule that act as engine of multiple modules, in this case the module's name will be fixed when calling the constructor.

UML class *Single-Run Module* is realized by Java interface SingleRunModule and absstract Java class AbstractSingleRunModule. Currently they are empty since single-run modules do not have generic functionality beyond what every module has. An example of a single-run module in KG system architecture is the schema module which creates the static data named graphs that contains the global RDFS vocabulary and the SHACL shapes. This global schema is written to the KG by invoking the module's init() method which in turn calls the doInit() method which is implemented by each concrete subclass of AbstractSingleRunModule.

UML class *Multiple-run Module* is realized by Java interface MultipleRunModule and abstract Java class AbstractMultipleRunModule. The interface adds method getTurn() which returns the current sequence number, which is incremented with each module invocation (also referred to as turn). The abstract class implements functionality common to internal and external modules such as getInputPath() and getOutputPath() which simplifies reading and writing from the file system based on generic module-specific file paths. Method initTurn() creates a new data named graph, queries the invocation time of the previous version and writes the first parts of the version specific metadata to the new data named graph including the logical invocationTime. Method commitTurn() commits a new data named graph by writing the logical commitTime to the new data named graph. Most importantly, the abstract class implements SPARQL preprocessing of special graph vars (see Section on SPARQL preprocessing) which is an integral part of the approach. It facilitates the selection of (1) all committed new data named graphs of a dynamic data named graph, (2) only those committed since the previous invocation of the current module or (3) those committed before the previous invocation.

UML class *External Module* is realized by Java interface ExternalModule and abstract Java class AbstractExternalModule. The interface adds methods exportInput() and importResults(). The abstract class implements these two methods which take care of generic metadata management. A new data named graph is created by exportInput() and committed by importResults(). The abstract class also specifies abstract methods doExportInput() and doImportResults() which have to be implemented by concrete subclasses and which take care of the actual export and import.

UML class *Internal Module* is realized by Java interface InternalModule and abstract Java class AbstractInternalModule. The interface adds method run() which is implemented by the abstract class. Calling run() creates a new data named graph and immediately commits it. As part of running the module it also measures the physical time duration and writes it to the new data named graph, this performance measurement stored in the KG can be used for monitoring and reporting the performance of modules. Method run() also calls an abstract method doRun() which is to be implemented by each concrete subclass to specify the actual queries, update requests, and further operations which are to be executed for deriving the contents of the new data named graph.

3.5 SPARQL Preprocessing

Java class *AbstractMultipleRunModule* implements SPARQL preprocessing which facilitates the formulation of compact SPARQL queries over the versioned and modularized KG. The preprocessing does not abstract away the versioning and modularization approach but it helps to avoid the writing of boilerplate code to select the named graphs from the KG (considering the logical invocation time of the current module invocation and of the previous module invocation) which should be considered for queries or update requests. The basic assumption here is that the description of complex objects/events is not distributed arbitrarily over named graphs, under this assumption we do not need to provide union graphs for querying.

A special graph variable has the form ?G<optional number>_<module_name>_<mode> where mode is one of "new", "old", or "all".

For example, any occurrence of

```
GRAPH ?G2_adsb_new {
```

is extended to (with 2021030117182300 being the previous invocation time and 2021030117184700 being the current invocation time).

```
GRAPH ?G2_adsb_new {
    ?G2_adsb_new
          aisa:module <http://aisa-project.eu/graphs/adsb>;
          aisa:commitTime ?time_G12_adsb_new.
    FILTER ( ?time_G2_adsb_new > 2021030117182300 )
    FILTER ( ?time_G2_adsb_new < 2021030117184700 )
```

Any occurrence of

```
GRAPH ?G2_adsb_old {
```

is extended to

```
GRAPH ?G2_adsb_old {
    ?G2_adsb_old
          aisa:module <http://aisa-project.eu/graphs/adsb>;
          aisa:commitTime ?time_G2_adsb_old.
    FILTER ( ?time_G2_adsb_old < 2021030117182300 )
```

Any occurrence of

```
GRAPH ?G2_adsb_all {
```

is extended to

```
GRAPH ?G2_adsb_ all {
    ?G2_adsb_ all
          aisa:module <http://aisa-project.eu/graphs/adsb>;
          aisa:commitTime ?time_G2_adsb_ all.
    FILTER ( ?time_G2_adsb_ all < 2021030117184700 )
```

The SPARQL preprocessor identifies special graph vars (e.g., ?G12_adsb_new) and includes query parts that select named graphs based on the module (e.g., adsb) they belong to and based on their commit time (e.g., after the invocation time of the previous module version). Class AbstractMultipleRunModule provides a method that takes as input a non-preprocessed SPARQL query or updata request and produces as output a preprocessed SPARQL query.

For example, the non-preprocessed SPARQL update request

```
INSERT { GRAPH ?TURN {
    [] aisa:state ?state;
        rdf:type aisa:LaggingState;
```

```
            aisa:lag ?lag. } }
    WHERE { GRAPH ?G12_adsb_new {
        ?state adsb:requestTime ?rtime;
            adsb:hasTimePosition ?ptime. }
        FILTER (?rtime > ?ptime)
        BIND ((?rtime - ?ptime) AS ?lag)  }
```

is expanded to

```
PREFIX graphs: <http://aisa-project.eu/graphs/>
PREFIX aisar: <http://aisa-project.eu/resources#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xs: <http://www.w3.org/2001/XMLSchema#>
PREFIX aisa: <http://aisa-project.eu/vocab#>
PREFIX adsb: <http://aisa-project.eu/adsb#>
INSERT { GRAPH graphs:adsbP1-5 {
   [] aisa:state ?state;
      rdf:type aisa:LaggingState;
      aisa:lag ?lag. } }
WHERE { GRAPH ?G12_adsb_new {
   ?G12_adsb_new
        aisa:module <http://aisa-project.eu/graphs/adsb>;
        aisa:commitTime ?time_G12_adsb_new.
   FILTER ( ?time_G12_adsb_new > 2021030117182300 )
     # graphs with commit time > invocation time of previous turn
   FILTER ( ?time_G12_adsb_new < 2021030117184700 )
     # graphs with commit time < invocation time of current turn
   ?state adsb:requestTime ?rtime;
      adsb:hasTimePosition ?ptime. }
   FILTER (?rtime > ?ptime)
   BIND ((?rtime - ?ptime) AS ?lag)  }
```

3.6 RDF Schema (RDFS) Reasoning

Supported reasoning tasks: transitive closure of taxonomies (rdfs:subClassOf and rdfs:subPropertyOf).

Location of RDFS statements: in one global schema

Execution of reasoning: at query time (possibly with query preprocessing)

The AISA KG will have one named graph (referred to as `graphs:schema`) containing the RDFS vocabulary and global SHACL schema. RDFS inferences based on the transitive closures of rdfs:subPropertyOf and rdfs:subClassOf can be used using property path expressions (namely ZeroOrMorePath expressions with operator `*`). Whenever a query should not only consider the direct instances of a class but also the indirect instances of a class with regard to the schema the query should be transformed from, e.g.,

```
WHERE {
    ?s  rdf:type  schema:Person.
}
```

to

```
WHERE {
    ?s  rdf:type  ?C_schema_Person.
    GRAPH graphs:schema {
        ?C_schema_Person  rdfs:subClassOf*  schema:Person.
    }
}
```

Similarly, triple patterns with properties for which also sub-properties should be matched should be transformed as follows. For example from

```
WHERE {
    ?s  schema:hasParent  ?o.
}
```

to

```
WHERE {
    ?s  ?P_schema_hasParent  ?o.
    GRAPH graphs:schema {
        ?P_schema_hasParent  rdfs:subPropertyOf*  schema:hasParent.
    }
}
```

The automation of these transformation patterns is rather straightforward and subject to future work. Then the developer only only needs to write

```
WHERE {
    ?s  rdf:type  ?C_schema_Person.
    ?s  ?P_schema_hasParent  ?o.
}
```

and the graph patterns over the schema (shown above in red) are added by a SPARQL preprocessor.

3.7 SHACL Validation

SHACL validation should be done prior to committing a new data named graph of a module as part of the doRun() or doImportResults() actions. How to invoke SHACL validation from Java is described in Section 2.1.6.

In some cases the to-be validated *data graph* will consist solely of a single new data named graph. in many cases the SHACL validation will need data from other named graphs in the KG. For this purpose it is up to the engine's developer to use a construct query to include these other named graphs into the data graph that is subject to validation.

In the basic approach all the SHACL shapes are in the KGs global schema. The global schema may be extended by modules-pecific SHACL shapes (we will investigate this as part of developing the KG-Prolog-Mapper in Task 4.2). In this case it will be up to the module's developer to specify the shapes graph accordingly as part of the implementation of the doRun() or doImportResults() actions.

It is further up to the module's developer to specify exception handling in case of a negative SHACL validation report. One approach is to just add the validation report to the new data named graph so that it can be queried and considered in further processing steps as needed.

3.8 Proof-of-Concept KG System

The proof-of-concept KG system (delivered as Java package at.jku.dke.aisa.kg.sample1 on the GitHub repository) implements

- a module **adsb** (class ADSBLoader) that incrementally loads ADS-B data into the KG. In the current implementation the ADBSLoader imports flight positions from a dataset contained in file input.trig (contained in folder fileinput/adsb/ associated with the module). File input.trig currently contains data generated from json-data retrieved from opensky-network.org every 10 seconds for a bounding box surrounding Austria on Feb 09 2021 from 19:14:00 to 19:24:50 and transformed to RDF based on code developed by students in a practical course under the supervision of Sebastian Gruber. The 59 named graphs are ordered by retrieval time with a distance of 10 seconds. The ADBSLoader imports one named graph per turn, hence, the real time distance between turns is 10 seconds. To make tests more interesting (especially with ADSBProcessor2) we have deleted one flight and state from graph:g0 and another one from graph:g1.

- a module **qadsb** (class QueryADSB) which incrementally queries the contents of dynamic data named graph *adsb* and outputs the query results to the console.

- a module **adsbP1** (class ADSBProcessor1) which incrementally identifies lagging flight states (i.e., flight states where the time position is older than the request time) and inserts these flights state classification into the KG and queries the newly inserted data to report them to the console

- a module **adsbP2** (class ADSBProcessor2) which incrementally identifies incoming flight and inserts these flight state classification into the KG and queries the identified incoming flights by 'joining' the newly created graph with existing **adsb** data

- a module **pairs** (class FlightPairs) which incrementally identifies pairs of flights and pairs of flight states in **adsb** and adds them to the KG. The module also calculates a distance (only based on latitute and longitude, not translating into miles/kilometres) between the two flights, demonstrating how such calculations can be done in Java by selecting the relevant data using SPARQL, doing the calculations in Java, creating the RDF model in Java and writing the model back into a new data named graph on the KG.

- a module **report** (class PerformanceReport) which queries the data and metadata (which also contains a runtime duration for every module invocation) of all new data named graphs to generate a performance report. The performance report is written to the console and also to the file system (to folder fileoutput/report/ associated with module report). The perfomance report is also written as CSV to fileoutput/report/aggregated_report_1.csv and in non-aggregated form to fileoutput/report/report_1.csv. Executing the report module multiple times would produce multiple version of these files, e.g., report_2.csv, report_3.csv, etc. With the number being the module's current sequence number. This also demonstrates how file imports and exports (which are central for integrating external modules, like ML modules, into the KG system, loosely coupled via the file system) are aligned with our versioning approach.

The modules are all implemented as concrete subclasses of AbstractInternalModule and implement interface InternalModule. The package also comprises concrete subclasses/implementations of AbstractSingleRunModule/SinglerunModule and AbstractExternalModule/ExternalModule but only as stubs without proper functionality just to check whether the metadata management works as expected also for these types of modules.

By running KGSystem1, which just consists of a main method, an instance of KGModuleSystem is created and the above modules are registered. Via the KGModuleSystem instance a connection with the KG Server is established and all the previous contents are deleted. Then the registered modules are executed in the order (adsb, qadsb, adsbP1, adsbP2, pairs) 50 times followed by one execution of module report.

3.8.1 Results of Initial Performance Measurements

To get first insights into the approach's performance characteristics we conducted initial performance measurements. The following performance report was generated by running KGSystem1 (as described above) as KG Manager and Fuseki with a TDB database as KG Server on the same machine (a HP EliteBook 850 G2 with an Intel® Core™ i7-5600U CPU @ 2.60 GHz, 2 kernels, 4 logical processors running with 16 GB of physical RAM, running Windows 10 Pro). The performance

report shows the execution time in ms per module invocation and the number of RDF triples in the new data named graph (minimum, maximum, and average of the 50 runs).

```
adsb,
259 ms (min), 825 ms (max), 375 ms (avg),
382 triples (min), 485 triples (max), 404 triples (avg).
qadsb,
114 ms (min), 505 ms (max), 152 ms (avg),
6 triples (min), 6 triples (max), 6 triples (avg).
adsbP1,
157 ms (min), 348 ms (max), 217 ms (avg),
6 triples (min), 72 triples (max), 34 triples (avg).
adsbP2,
127 ms (min), 355 ms (max), 161 ms (avg), 6 triples (min),
44 triples (max), 7 triples (avg).

pairs,
332 ms (min), 1239 ms (max), 449 ms (avg),
1077 triples (min), 1777 triples (max), 1234 triples (avg).
```

module	minTime	maxTime	avgTime	minCount	maxCount	avgCount
qadsb	114	505	152	6	6	6
adsbP1	157	348	217	6	72	34
adsbP2	127	355	161	6	44	7
adsb	259	825	375	382	485	404
pairs	332	1239	449	1077	1777	1234

3.8.2 Console Application for Interactive or Scripted KG Sessions

Class KGModuleSystem also provides a console application to facilitate interactive KGModuleSystem sessions where modules are invoked via textual commands. The following shows a sample interactive session (with user inputs in green – which could alternatively be provided by a text file).

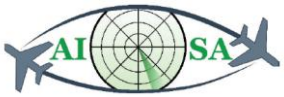
```
Your commands: > run adsb
adsb reads: http://aisa-project.eu/graphs/g0
adsb - Request time: 1612894440
> run adsb
adsb reads: http://aisa-project.eu/graphs/g1
adsb - Request time: 1612894460
> run pairs
pairs: 342 distances between flights inserted
> run adsb
adsb reads: http://aisa-project.eu/graphs/g2
adsb - Request time: 1612894470
> run pairs
pairs: 190 distances between flights inserted
> run pairs
```

```
pairs: 0 distances between flights inserted
> run report
adsb, 1, 545 ms, 396 triples.
adsb, 2, 303 ms, 396 triples.
adsb, 3, 290 ms, 417 triples.
pairs, 1, 582 ms, 2400 triples.
pairs, 2, 538 ms, 1336 triples.
pairs, 3, 245 ms, 6 triples.
adsb, 290 ms (min), 545 ms (max), 379 ms (avg), 396 triples (min),
417 triples (max), 403 triples (avg).
pairs, 245 ms (min), 582 ms (max), 455 ms (avg), 6 triples (min),
2400 triples (max), 1247 triples (avg).
> exit
Bye...
```

Appendix A Glossary

Abbreviation	Term
ADS-B	Automatic Dependent Surveillance-Broadcast
AI	Artificial Intelligence
AIXM	Aeronautical Information Exchange Model
ATC	Air Traffic Control
ATCO	Air Traffic Control Officer
ATM	Air Traffic Management
FIXM	Flight Information Exchange Model
ICAO	International Civil Aviation Organization
JPL	a Java/Prolog Interface
ML	Machine Learning
KG	Knowledge graph
PoC	Proof-of-Concept
RDF	Resource Description Framework
RDF/XML	a syntax to express an RDF as an XML document
RDFS	Resource Description Framework Schema
SA	Situational Awareness
SHACL	Shapes Constraint Language
SPARQL	SPARQL Protocol and RDF Query Language
SWIM	System-wide Information Management
Turtle	Terse RDF Triple Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XQuery	XML Query Language

Table 1 Table of acronyms


AI SA


SESAR
JOINT UNDERTAKING


University of Zagreb


Technische
Universität
Braunschweig


POLITÉCNICA
UNIVERSIDAD
POLITÉCNICA
DE MADRID


skyguide


zh
aw
School of
Engineering
ZAV Center for Aviation


JKU
JOHANNES KEPLER
UNIVERSITÄT LINZ


onsulting