JOHANNES KEPLER
UNIVERSITY LINZ

Author

Victoria Ines Kaar, BSc.

Submission

Institute of Business Informatics – Data & Knowledge Engineering

MAPPING AIXM SCHEMA AND INSTANCE DATA TO RDF(S)

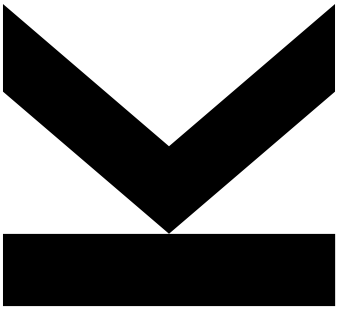
Thesis Supervisor

Mag. Dr. Christoph Schütz

Co-Supervisor

Mag. Dr. Bernd Neumayr

September 2021



Master's Thesis

to confer the academic degree of

Master of Science

in the Master's Program

Business Informatics

JOHANNES KEPLER

UNIVERSITY LINZ

Altenberger Straße 69

SWORN DECLARATION

I hereby declare under oath that the submitted Master's Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Place, Date

Signature

Abstract

Air traffic in Europe is currently increasing and with it the workload of the air traffic control operators. Furthermore, the number of aircraft and flights in air traffic for a given sector is constrained by the available resources of the air traffic control operators, which means that the workload of the air traffic control operators grows when air traffic is increasing. Air traffic control operators can lose situational awareness if a certain level of air traffic is exceeded. This, in turn, leads to unsafe decisions. In order to prevent the aforementioned risks, a situationally aware system is needed. Therefore, the collaborative research project AISA investigates the use of artificial intelligence in air traffic management; this master's thesis started as a preliminary study for the AISA project. The AISA project aims to build a situationally aware system to support air traffic control operators in their work. The input that is needed for this situationally aware system are the messages that are exchanged within air traffic management using the Aeronautical Information Exchange Model (AIXM), among other exchange models. In order to enable logical reasoning within a situationally aware system, it is beneficial to map AIXM input to RDF(S), which is a foundational technology for the Semantic Web and symbolic artificial intelligence.

The goal of this thesis is the development of a transformation procedure for the mapping of the AIXM schema to RDFS as well as the development of mapping rules to transform AIXM instance data into RDF statements. These procedures serve as a guideline for performing the mapping from AIXM to RDF(S). Additionally, the thesis also explains the automation process of these procedures that are introduced within this thesis. Therefore, the contribution of this thesis is not only the mapping procedures but also tools that automate those procedures and a description of the adaptions that are necessary to these tools to transform other exchange models in air traffic management in the future.

Kurzfassung

Der Luftverkehr in Europa nimmt derzeit zu und somit auch die Arbeitsbelastung der Luftverkehrskontrolle. Darüber hinaus wird die Anzahl der Flugzeuge und Flüge im Luftverkehr für einen bestimmten Sektor durch die verfügbaren Ressourcen der Luftverkehrskontrolle begrenzt, was bedeutet, dass die Arbeitsbelastung der Luftverkehrskontrolle mit dem steigenden Flugverkehr zunimmt. Bei Überschreitung eines bestimmten Verkehrsaufkommens kann die Luftverkehrskontrolle das Situationsbewusstsein verlieren. Dies wiederrum führt zu unsicheren Entscheidungen. Um die genannten Risiken zu vermeiden, wird ein situationsbewusstes System benötigt. Das Verbundforschungsprojekt AISA untersucht daher den Einsatz von künstlicher Intelligenz im Luftverkehrsmanagement; diese Masterarbeit ist als Vorstudie für das AISA-Projekt entstanden. Ziel des AISA-Projekts ist es, ein situationsbewusstes System zu entwickeln, dass die Luftverkehrskontrolle bei ihrer Arbeit unterstütz. Der Input, welcher für dieses situationsbewusste System benötigt wird, sind die Nachrichten, die innerhalb des Flugverkehrsmanagements unter Verwendung des Aeronautical Information Exchange Model (AIXM) und anderer Austauschmodelle ausgetauscht werden. Um logische Schlussfolgerungen innerhalb eines situationsbewussten Systems zu ermöglichen, ist es von Vorteil, den AIXM-Input auf RDF(S) abzubilden, einer grundlegenden Technologie für das Semantic Web und symbolische künstliche Intelligenz.

Ziel dieser Arbeit ist die Entwicklung eines Transformationsverfahrens für die Abbildung des AIXM-Schemas auf RDFS sowie die Entwicklung von Mapping-Regeln zur Transformation von AIXM-Instanzdaten in RDF-Anweisungen. Diese Verfahren dienen als Leitfaden für die Durchführung des Mappings von AIXM auf RDF(S). Darüber hinaus wird in dieser Arbeit auch der Automatisierungsprozess dieser Verfahren, die in dieser Arbeit vorgestellt werden, erläutert. Daher besteht der Beitrag dieser Masterarbeit nicht nur in den Transformationsverfahren, sondern auch in den Werkzeugen, welche diese Prozeduren automatisieren, sowie in der Beschreibung der Anpassungen, welche an diesen Werkzeugen notwendig sind, um andere Austauschmodelle im Luftverkehrsmanagement in Zukunft zu transformieren.

Table of Contents

Nomenclature

AI	Artificial intelligence
AIS	Aeronautical Information Service
AIXM	Aeronautical Information Exchange Model
ATC	Air Traffic Control
ATCO	Air Traffic Controller
ATM	Air Traffic Management
ICAO	International Civil Aviation Organization
IFR	Instrument Flight Rules
MOF	Meta Object Facility
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RML	RDF Mapping Language
UML	Unified Modelling Language
XML	Extensible Markup Language

1 Introduction

The objective of this master thesis is the mapping of the AIXM 5.1.1 UML diagrams to RDF Schema and the mapping of AIXM XML instance data to RDF. This chapter provides an overview of the thesis in general, describing the problem statement, the contribution of this thesis and its outline.

1.1 Problem Statement

Air traffic management (ATM) is a necessary service to enable air transport. In ATM, air traffic control (ATC) ensures that aircraft fly safely in airspace. More specifically, ATM is a navigation service that coordinates air traffic primarily through air traffic control (ATC), air traffic flow control (ATFM) and airspace management. For additional information on ATM, the thesis refers to [1] and [2].

ATC is a main function of ATM, handling flights during all phases, from taxiing, take-off, and landing to cruising. The services of ATC are provided by licensed air traffic control operators (ATCOs) to prevent collisions between aircraft in the air and on the ground [2].

The number of aircraft and flights in air traffic for a given sector is constrained by the workload of the ATCOs. As air traffic in Europe grows, the increasing number of aircraft causes the ATCOs' workload to rise to a level where they can lose situational awareness [3]. When that awareness is lost, the controller may make unsafe decisions. To minimize the probability of capacity problems and, thus, the problem of unsafe decision making, a situationally aware system is needed. The input for this system are messages that are exchanged within air traffic management in the Aeronautical Information Exchange Model standard (AIXM) [4][5][6], among other information exchange models. This master's thesis started as a preliminary study for the collaborative research project AISA [7] under the Single European Sky ATM Research Joint Undertaking within the EU's research program Horizon 2020. The project AISA investigates the use of artificial intelligence (AI) in air traffic management. This research project aims to build a situationally aware system to support the air traffic controllers (ATCOs) in their work. To enable reasoning through the situationally aware system in order to verify the plausibility of machine learning results, it is necessary to map AIXM input to RDF(S), which is the topic of this thesis.

The goal of this thesis is to provide a transformation procedure for the mapping of the AIXM schema to RDF(S) and a procedure for the creation of the mapping rules to map AIXM instance data to RDF. These procedures are intended to serve as a guideline for performing the mapping from AIXM to RDF(S). The thesis also introduces tools that help with the mapping of the schema and the instance data. Although AIXM is used to show the procedures and tools, there is also an explanation on how to use the procedures and tools in a generic way with other exchange models.

1.2 Contribution

The contribution of the master's thesis is to define and describe (i) how to transform AIXM UML class diagrams into RDF Schema (RDFS) and (ii) how to transform AIXM XML instance data into RDF statements.

Regarding the transformation from AIXM UML class diagrams into RDFS (Schema Mapping), this thesis defines mapping rules, a procedure for applying the mapping rules, and XQuery modules for automatic transformation of the AIXM UML class diagrams into RDFS. The mapping rules for this transformation process declare how the individual elements of an UML class diagram should be represented in RDFS.

Regarding the transformation from AIXM XML to RDF (Instance Data Mapping), this thesis defines the procedure to transform AIXM XML instance data into RDF triple statements using the tool RMLMapper [8], the procedure for creating the RML mapping rules that are processed with the RMLMapper to generate RDF triple statements from XML instance data, and XQuery modules for semi-automatic creation of the RML mapping rules.

As mentioned above, this thesis formulates two procedures: (i) the procedure for applying the mapping rules for the schema, and (ii) the procedure for the creation of the mapping rules for the instance data. The results of those procedures are, in the former case, the complete schema in RDFS and, in the latter case, the RML mapping rules that can be processed with the RMLMapper, which creates the RDF statements. The instance data that result from the transformation must meet the requirement that the mapped RDF statements must be compatible with the generated RDFS. This means that not only the mapped RDFS must be valid, but also that the mapped instance data must correspond to the generated RDFS.

Besides providing the procedures, this thesis also shows how to automate those procedures. As part of this thesis, an XQuery module was implemented to automatically generate the RDFS, using the AIXM UML class diagram as input. The Apache Jena framework [9] was used to validate the generated RDFS. Another XQuery module serves to automatically generate the mapping rules that are needed to generate RDF out of the AIXM instance data. The created RML mapping rules can be applied using the tool RMLMapper to create valid RDF instance data. Both XQuery modules are available online[1].

All the procedures and implementations were designed and tested using the AIXM schema version 5.1.1 and with the AIXM XML "Donlon" sample data. The presented mappings do not consider the AIXM Temporality Model [10] and the concept of time slices. This thesis uses UML class diagrams as input for the mapping because the AIXM schema is modelled using UML class diagrams.

1.3 Outline

The outline of the master's thesis is the following. The thesis is divided into six main chapters. Chapter 2 provides an overview of the state of the art. Chapter 3 describes AIXM Schema Mapping. Chapter 4 describes the AXIM instance mapping. Chapter 5 describes the implementation and automation of the mapping procedures. Chapter 6 concludes this thesis.

[1] https://github.com/jku-win-dke/mt2105-aixm-mapping

2 State of the Art

This chapter provides an overview of the state of the art of the technologies and procedures used within this thesis. It contains explanations of AIXM, UML, RDF, RML and XML and XQuery.

2.1 AIXM

The Aeronautical Information Exchange Model (AIXM) has the goal to provide the aeronautical information in a digital format. Aviation has an increasing dependency on timely, consistent, and high-quality aeronautical information. Therefore, an exchange standard is needed and AIXM was created. AIXM provides a model for aeronautical data which is a standard exchange format that is used for improving aeronautical information services systems as well as a common language to express aeronautical information for human and computer interpretation. AIXM also contains all the necessities for information processing for the aeronautical domain and additionally also military aeronautical information. For additional information on that topic the thesis refers to [4] and [5].

The AIXM standard covers data about the following concepts, among others [5]:

- Aerodrome/Heliport
    - Information about the airports and heliports.
    - Airports and heliports are defined areas either on land or on water and are intended to be used wholly or in part for the arrival, departure and the surface movement of aircrafts and helicopters.
    - These defined areas also include any buildings, installations, and equipment of the airport/heliport.
- Navigation Aids
    - Navigation aids in the aviation domain can be a navigation aid system, a radar system, or a special navigation system.
    - Navigation aid system provide guidance information of position to efficient and safely operate the aircrafts.
    - Radar services can be for example a Precision Approach Radar for landing an aircraft or Airport Surveillance Radar that provides short-range radar coverage in the vicinity of the airports.
- Terminal procedures
    - Procedures in the aviation domain are a collection of manoeuvres that are predetermined and include specified protection from obstacles.
    - For example, the “Instrument approach procedure” contains predetermined procedures that are referenced by flight instruments, which includes protection from obstacles, while the aircraft approaches the airport/heliport.
- En Route structures
    - These data describe the details of the routes to be flown, which is necessary to provide air traffic services from the end of the take-off and the initial climb phase to the beginning of the approach and the landing phase.
- Airspace boundaries
    - An airspace is defined by a three-dimensional portion (airspace boundaries) of space that is relevant for air traffic.

- Air Traffic Control and NOTAM services
    - The services of the Air Traffic Control supervise the airspace itself and the airspace of airports or heliports.
    - NOTAMs are Notices for Airman, which are text messages that are read by pilots, controllers and other operational personnel involved in flight operations. Digital NOTAMs are NOTAMs that correspond to the AIXM standard.
- Traffic restrictions
    - Contains all the restrictions on a single flight or a traffic flow, for example, a restriction that flights arriving at JFK can only use Airway A14.
    - Which means that those restrictions regulate the use of the route networks.

The AIXM is defined both in terms of XML Schema and UML. For the UML model of AIXM, an XMI representation is also available (see Section 2.1.2). AIXM instance data are usually exchanged in XML format. AIXM builds on the Geography Markup Language (GML) [11], which itself was influenced by the Resource Description Framework (RDF) [12].

The following section illustrates the AIXM standard using an excerpt of the AIXM schema as an example in order to show what kind of information is represented using AIXM. The example is the class `AirportHeliport` with three associations (see Figure 1).

The `AirportHeliport` class describes information related to airports and heliports, for example [6]:

- If the airport or heliport is *abandoned* or not.
- If the airport or heliport is *certified* according to the International Civil Aviation Organization (ICAO)
- The primary organization type (*controlType*) which controls the airport or heliport, which can be either civil or military.
- The *designator* of an airport or heliport.
- The International Civil Aviation Organization (ICAO) *location indicator* of the port.
- The primary official *name* of the port designated by appropriate authority.
- The indication if an airport or heliport is for *private use* only, which means that the airport or heliport is only for the use of the owners.
- The airport or heliport *type*, which can be either an aerodrome only, combined aerodrome/heliport or a simple landing site.

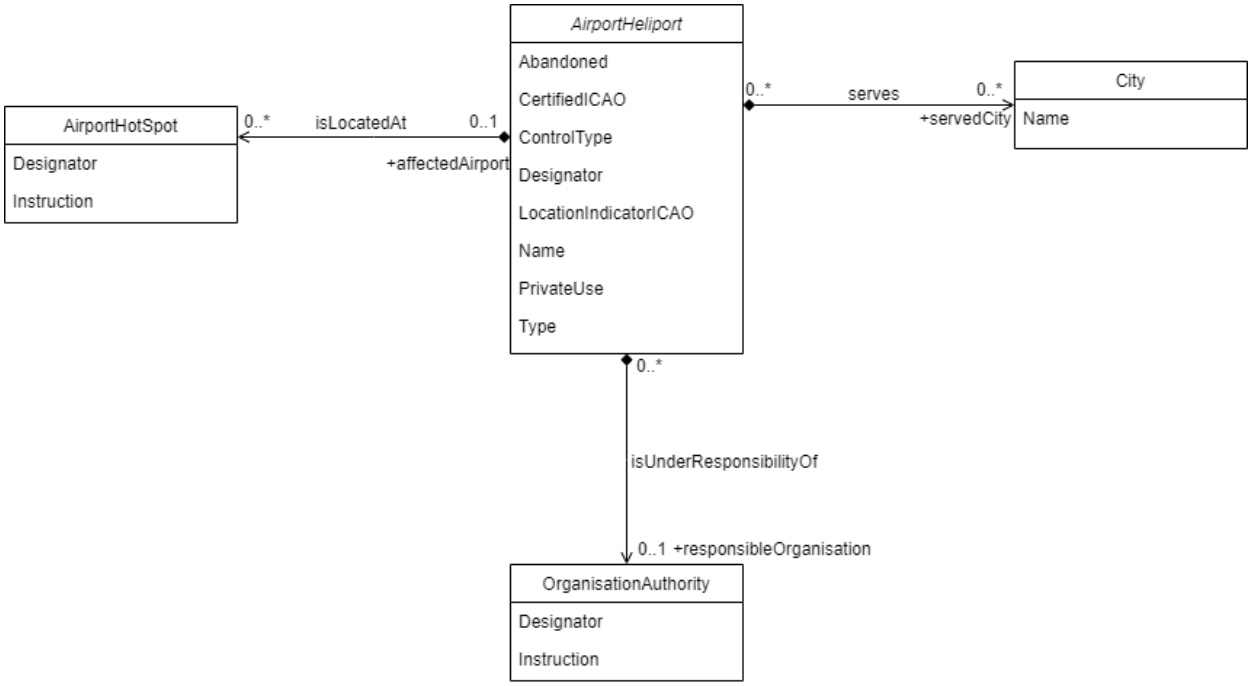


Figure 1. Simplified AirportHeliport class from the AIXM standard [6]

As it can be seen in Figure 1 the `AirportHeliport` class also has associations with other classes [6]:

- From the `AirportHeliport` class to
  - `OrganizationAuthority`, authority/organisation/state that is responsible for the air- or heliport.
  - `City`, that is served by this airport.
- To the `Airport/Heliport` class from
  - `AirportHotSpot`, defines the airport where the hot spot has been identified. A hot spot is a location with a history or potential risk of collision or runway incursion and therefore a heightened attention by the pilots is necessary.

AIXM also contains a Temporality Model [10], as the information and notifications are usually provided in advance of their effective dates, which means that the aeronautical systems must store both, the data of the current situation and the future changes. Furthermore, many changes indicated by AIXM notifications modify the baseline only temporarily. As the Temporality Model is not considered in the mapping procedures it is not explained further within this thesis.

2.2 UML

The Unified Modelling Language (UML) is a standard for modelling software, systems, and data that is administered by the Object Management Group (OMG). The primary usage of UML is to specify, visualize, construct and document software systems artifacts, which are items that are created or collected during the development of the software. Those artifacts can include requirements, design, use cases, etc.

With UML, a standard procedure was created to write blueprints for software and systems. Those blueprints not only contain technical artefacts but also conceptual things (system functions, business processes) as well as concrete things (programming language, database schema, reusable software components). But UML is not purely restricted to software modelling only, it can also be used for modelling hardware, business processes, system engineering and organizational structures. The intention behind UML is to provide a common way of expressing and capturing behaviours, relationships, and high-level ideas in a notation that is not only easy to learn but also efficient to write.

When specifying and designing a system, different viewpoints should be considered to achieve a broader and more comprehensive understanding of the software that should be built. UML allows to look at the system from different viewpoints, with the different diagram types the language provides. For further information on UML this thesis refers to [13] and [14].

As it was already mentioned in the introduction (Chapter 1), this thesis only considers class diagrams. Class diagrams belong to the “static structural viewpoint” of UML. Models that are described with that viewpoint in mind picture the structural aspects of a system. Therefore, the class diagram belongs to the main UML diagram type “structure diagrams”. Those diagrams help with the visualization, specification, construction, and documentation of the systems static aspects.

2.2.1 Class Diagram

As it was already mentioned above the class diagram is part of the structure diagrams of UML. This type of diagram is the most used and common diagram that is found in the object-oriented systems. The class diagram is used for illustrating the static viewpoint of a software system by using a set of classes, interfaces, and their relationships (see Figure 2). It includes the following elements [13]:

- **Class**
  - A class is a template to create the objects and provides a specification of the object attributes and the operations (function or procedures) that a class instance can execute.
- **Interface**
  - An interface is a contract consisting of a public set of operations and attributes for a specific class.
  - Every instance of that specific class, that also realizes the interface must fulfil the contract of that interface.
  - Interfaces are not instantiable, instead they are implemented by a class instance.
- **Relationship**
  - Relationships specifies the connection of elements with each other.

- An element can reference one or more related elements and this relation can either be logical or physical.
- UML defines many different relationship types:
  - *Generalization* relationship connects generalized classes to specialized classes. (For example, the superclass-subclass relation)
  - *Association* is a structural relationship between classes. Associations map the physical structure of things.
  - *Dependency* relationship states that an entity uses services and information of another class.

- **Enumerator**
  - An enumerator is used when a definite set of values need to be specified.
  - Example: A Boolean can be specified with the definite set of the values “true” and “false”.

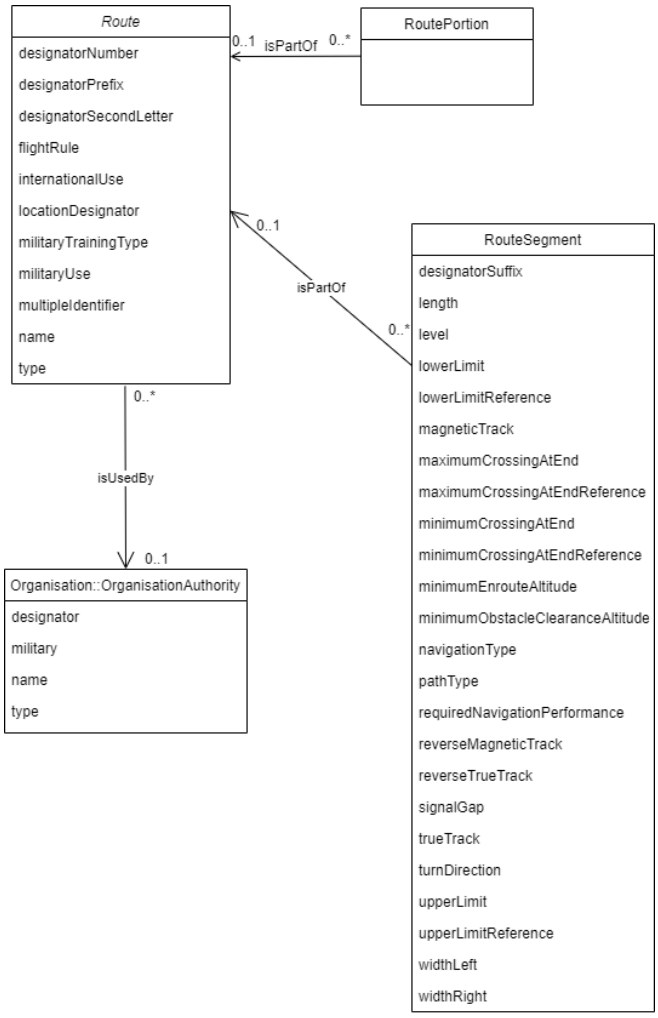


Figure 2 - UML class diagram of AIXM routes [6]

2.2.2 XML Metadata Interchange

XML Metadata Interchange (XMI) is a machine-readable representation for models. To use that format the model must be defined in terms of the Meta Object Facility standard (MOF). Since the UML metamodel is defined as a MOF metamodel, this format can be used as a model interchange format for UML. For more information on XMI the thesis refers to [15]. Based on Figure 2 this section explains the representation of the UML diagram in XMI.

The class `Route` in the AIXM model is represented with the XML tag `ownedMember` and the attributes `xmi:type`, `xmi:id` and `name` (see Figure 3). The attribute `xmi:type` declares that this element is an UML class and should be created and displayed as an UML class, when the XMI file is viewed in a Modelling-Software such as Enterprise Architect. The attribute `xmi:id` and `name` define the ID of the element and the name. The attributes of the class `Route` are represented with the XML tag `ownedAttribute` with the same attributes, that were already explained above.

```
<ownedMember xmi:type="uml:Class" xmi:id="EAID_0C8EA976_CB4B_45ee_B996_EF792EB84707" name="Route">
        <ownedAttribute xmi:type="uml:Property"
                xmi:id="EAID_0D2C42F6_6313_4a68_A995_8F42D631E242" name="designatorPrefix">
                <type xmi:idref="EAID_749966FF_E8B9_44d5_9350_22CEA11FD029"/>
                <lowerValue xmi:type="uml:LiteralInteger"
                        xmi:id="EAID_LI009709_6313_4a68_A995_8F42D631E242" value="1"/>
                <upperValue xmi:type="uml:LiteralUnlimitedNatural"
                        xmi:id="EAID_LI009710_6313_4a68_A995_8F42D631E242" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:type="uml:Property"
                xmi:id="EAID_2D4D01F6_0DC1_4a17_ADB3_FA23C7547ADA" name="designatorSecondLetter">
                <type xmi:idref="EAID_B5D960C3_72AF_45bf_90B5_DF6E00DEEE0C"/>
                <lowerValue xmi:type="uml:LiteralInteger"
                        xmi:id="EAID_LI009711_0DC1_4a17_ADB3_FA23C7547ADA" value="1"/>
                <upperValue xmi:type="uml:LiteralUnlimitedNatural"
                        xmi:id="EAID_LI009712_0DC1_4a17_ADB3_FA23C7547ADA" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:type="uml:Property"
                xmi:id="EAID_E2F26706_577E_46b7_AC5D_CEBA7233758E" name="designatorNumber">
                <type xmi:idref="EAID_D6B1C23F_9293_461d_8F34_50B2A50FAB9F"/>
                <lowerValue xmi:type="uml:LiteralInteger"
                        xmi:id="EAID_LI009713_577E_46b7_AC5D_CEBA7233758E" value="1"/>
                <upperValue xmi:type="uml:LiteralUnlimitedNatural"
                        xmi:id="EAID_LI009714_577E_46b7_AC5D_CEBA7233758E" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:type="uml:Property"
                xmi:id="EAID_485843B9_3D0F_47b0_998F_013F302EE126" name="name">
                <type xmi:idref="EAID_DC9949C7_7F85_4139_A973_4003104727A3"/>
                <lowerValue xmi:type="uml:LiteralInteger"
                        xmi:id="EAID_LI009719_3D0F_47b0_998F_013F302EE126" value="1"/>
                <upperValue xmi:type="uml:LiteralUnlimitedNatural"
                        xmi:id="EAID_LI009720_3D0F_47b0_998F_013F302EE126" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:type="uml:Property"
                xmi:id="EAID_8933ED13_5FBF_4f77_86F6_3B83A67A9043" name="type">
                <type xmi:idref="EAID_16CB04BC_49D4_4eda_A553_D54C0CBE0A00"/>
                <lowerValue xmi:type="uml:LiteralInteger"
                        xmi:id="EAID_LI009721_5FBF_4f77_86F6_3B83A67A9043" value="1"/>
                <upperValue xmi:type="uml:LiteralUnlimitedNatural"
                        xmi:id="EAID_LI009722_5FBF_4f77_86F6_3B83A67A9043" value="1"/>
        </ownedAttribute>
        ...
</ownedMember>
```

Figure 3 - Definition of class "`Route`" in XMI (Excerpt)

The association between two classes is represented with the XML element `connector`. The connector contains two sub elements `source` and `target`, which represent the two classes that are associated with each other. For example, Figure 4 shows the association between `Route (source)` and `RoutePortion (target)`. The direction of the association can be found in the XML element `properties` in the attribute `direction`.

```
<connector xmi:idref="EAID_66442DED_3FE9_4794_B4E2_9A657A5F0AE5">
        <source xmi:idref="EAID_0C8EA976_CB4B_45ee_B996_EF792EB84707">
                <model ea_localid="844" type="Class" name="Route"/>
                <role name="referencedRoute" visibility="Public" targetScope="instance"/>
                <type multiplicity="0..1" aggregation="none"/>
                <constraints/>
                <modifiers isOrdered="false" isNavigable="false"/>
                <style value="Navigable=Navigable;Owned=0;"/>
                <documentation value="The route referenced by the route portion."/>
                <xrefs/>
                <tags/>
        </source>
        <target xmi:idref="EAID_BFBC4EBC_A2E5_4e75_880C_6E6AC1E37C79">
                <model ea_localid="848" type="Class" name="RoutePortion"/>
                <role visibility="Public" targetScope="instance"/>
                <type multiplicity="0..*" aggregation="none"/>
                <constraints/>
                <modifiers isOrdered="false" isNavigable="false"/>
                <style value="Navigable=Unspecified;Owned=0;"/>
                <documentation/>
                <xrefs/>
                <tags/>
        </target>
        <model ea_localid="147"/>
        <properties ea_type="Association" direction="Destination -&gt; Source"/>
        <modifiers isRoot="false" isLeaf="false"/>
        <parameterSubstitutions/>
        <documentation/>
        <appearance linemode="3" linecolor="0" linewidth="0"
                seqno="0" headStyle="0" lineStyle="0"/>
        <labels lb="0..1" lt="+referencedRoute" mt="isPartOf" rb="0..*"/>
        <extendedProperties virtualInheritance="0"/>
        <style/>
        <xrefs/>
        <tags/>
</connector>
```

Figure 4 - Definition of association between `Route` and `RoutePortion` in XMI

The XMI representation of the AIXM model serves as the basis for the transformation of AIXM UML into RDFS (see Chapter 3).

2.3 RDF

The Resource Description Framework (RDF) provides a uniform and integrated access to information services and sources. The original purpose of the RDF language was for representing metadata for Web resources and objects which can be identified on the Web. RDF can also be used to integrate data from different systems. RDF has since become a general framework to exchange and integrate data. Furthermore, RDF is a widely used representation format for knowledge-based systems. For additional information on RDF the thesis refers to [16] and [17].

In order to work with RDF data, the Apache Jena framework [9] can be used. Apache Jena is an open-source framework for Java, to help with building Linked Data and Semantic Web applications. The framework supports SPARQL for querying and updating RDF models and its server which can present RDF data and process SPARQL queries. It also contains the RDF API which can be used to read, write, and validate RDF data. In this thesis, Apache Jena was used for validating the generated RDF

2.3.1 Vocabulary and Statements

The data model of RDF is based on the concept of RDF statements, which makes the data model very simple and flexible. RDF Statements are built in the triple form, which means it consists of a subject, a predicate, and an object. The predicate represents the binary relationship between the subject and the object. While literals are allowed as objects of an RDF Statement, they may not be used as a subject or predicate. There is a distinction in the data model between resources and literals:

- Resources are object identifiers represented by URIs.
- Literals are strings.

The RDF vocabulary is very synonymous with XML namespaces, because it consists of a set of URI. Although the vocabulary is a set of URI, the URI must be unique within the vocabulary. Table 1 contains two RDF statements. Both statements have the same subject because the URI of the subject identifies a certain document. The first statement can be read as following. The resource `http://www.cat.com/airportHeliport#R20301` has a predicate `servedCity` (which is described in `http://purl.org/dc/elements/1.1/servedCity`). The value of the predicate is the resource `http://www.cat.com/city#ID110` (object).

Table 1: RDF statements examples [16]

Statement	Element	Value (URIref or iteral)
1	Subject	`http://www.cat.com/airportHeliport#R20301`
	Predicate	`http://purl.org/dc/elements/1.1/servedCity`
	Object	`http://www.cat.com/city#ID110`
2	Subject	`http://www.cat.com/airportHeliport#R20301`
	Predicate	`http://purl.org/dc/elements/1.1/name`
	Object	`John F. Kennedy International Airport`

2.3.2 Triples and Graphs

A basic serialization format of RDF statements can be the following:

- *<Subject> <Predicate> <Object>.* → This form is used if *Object* is a relative or absolute URIref.
- *<Subject> <Predicate> "Object".* → This form is used if *Object* is a literal.

Figure 5 shows an example of RDF triples from the RDF statements in Table 1.

```
<http://www.cat.com/airportHeliport#R20301>
        <http://purl.org/dc/elements/1.1/servedCity>
                <http://www.cat.com/city#ID110> .

<http://www.cat.com/airportHeliport#R20301>
        <http://purl.org/dc/elements/1.1/name>
                "John F. Kennedy International Airport" .
```

Figure 5 - Code example for RDF triples

A set of RDF triples can be considered a graph. Within such a graph the subject and the object of RDF triples are represented as nodes. The predicates of the RDF triples are represented by arcs. If nodes are labelled with literals, the node has the form of a box. If nodes are labelled with URIrefs, the node has the form of an ellipse. Only absolute URIrefs are allowed for the nodes and arc labels. Figure 6 shows an example of a visualized RDF graph, from the RDF statements in Table 1. For further information on this topic this thesis refers to [16] and [18].

Other RDF notations like RDF/XML or RDF/Turtle will be explained in chapter 2.3.3 and 2.3.4.

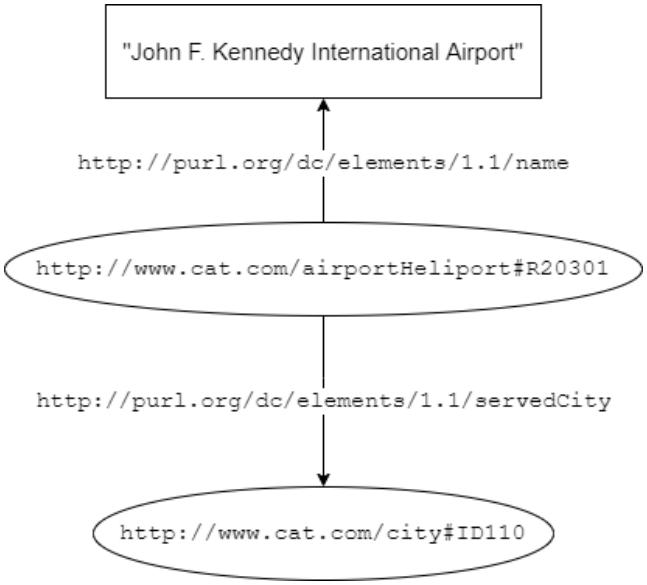


Figure 6 - Example of RDF graph [16]

2.3.3 RDF/XML

RDF/XML is another notation for RDF statements and is the preferred notation in the domain of semantic web. The RDF vocabulary serves as the basis for RDF/XML (see Table 2). Figure 7 gives and overview of the RDF/XML notation:

```
<?xml version="1.0"?>
<rdf:RDF
xml:base="http://www.cat.com/airportHeliport"
xmlns:cs="http://www.cat.com/schema/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dc="http://purl.org/dc/elements/1.1/">
        <cs:AirportHeliport rdf:about="#R20301">
                <dc:servedCity rdf:resource="http://www.cat.com/city#ID110"/>
                <dc:name>John F. Kennedy International Airport</dc:name>
        </cs:AirportHeliport>
</rdf:RDF>
```

Figure 7 - Code example for RDF/XML [16]

In the `rdf:RDF` tag the vocabularies of the schema (`cs`), of RDF (`rdf`) and of the Dublin Core (`dc`) are set. The tag `cs:AirportHeliport` defines the RDF statement from Table 1. It indicates that the `AirportHeliport` is associated with the city the airport serves. With the `dc:city` tag the city predicate of the subject "`#R20301`" is defined. The attribute `rdf:resource` indicates the object of that specific subject. The same goes for the tag `dc:name`. It is another predicate of the subject ""`#R20301`" and has the literal value "John F. Kennedy International Airport". The meaning of `rdf:about` and `rdf:resource` are explained in Table 2. The thesis refers to [16] for more detailed information.

2.3.4 RDF/Turtle

Another notation for RDF statements is the textual notation RDF/Turtle. With this notation it is possible to write an RDF statement in a natural and compact text form. The following example gives an overview of a RDF statement in the RDF/Turtle notation [19]:

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix city: <http://www.perceive.net/schemas/servedCity/> .
<http://example.org/#jfk-airport>
        <http://www.perceive.net/schemas/relationship/servedCity>
                <http://example.org/#new-york> .
    city:servedCity <#new-york> ;
    a foaf:AirportHeliport ;
    foaf:name "John F. Kennedy International Airport" .

<#new-york>
    port:airportHeliport <#jfk-airport> ;
    a foaf:City ;
    foaf:name "New York" .
```

Figure 8 - Code example for RDF/Turtle

As it is shown in Figure 8, an RDF/Turtle statement consists of sequences of subjects, predicates, and objects. The statement is terminated with a dot (“.”) and the subject, predicate and object are separated by a whitespace. A simple triple statement of this RDF/Turtle model would be the following:

Figure 9 - Code example of a triple statement in RDF/Turtle

In some cases, the same subject will be referenced by one or more predicates. The series of those predicates with their object value are separated by a semicolon (“;”). In RDF/Turtle a predicate list is modelled as follows:

```
<http://example.org/#jfk-airport>
        <http://www.perceive.net/schemas/relationship/servedCity>
                <http://example.org/#new-york> ;
        <http://xmlns.com/foaf/0.1/name> "John F. Kennedy International Airport" .
```

Figure 10 - Code example of predicate list in RDF/Turtle

2.3.5 RDF Schema

RDF in general is very flexible, but it offers no way to define application-specific properties and classes. Therefore, RDF Schema was introduced and with it an extensible type system to RDF. The schema offers ways to model hierarchies of properties and classes, which partly cured the lack of expressiveness of RDF. More precisely, with the RDF Schema class and subclass hierarchies as well was property domains and ranges can be defined. The thesis refers to [16] and [17] for further details.

Class

A class in RDF Schema is a resource that has an `rdf:type` property with the value `rdfs:Class`. `rdfs:Class` is part of the RDF Schema vocabulary. A class can also be defined as a subclass of another class. This can be done with the property `rdfs:subClassOf`. Figure 11 shows how a class, and a subclass can be defined, while Figure 12 shows the same example in the RDF/XML notation.

```
cs:Airfield     rdf:type            rdfs:Class .
cs:Airport      rdf:type            rdfs:Class .
cs:Airport      rdfs:subClassof     cs:Airfield .
```

Figure 11 - Definition of an RDF Class and Subclass

```
<?xml version=?1.0"?>
<rdf:RDF
        xml:base=?http://www.cat.com/schema/"
        xmlns:rdf=?http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:dc=?http://purl.org/dc/elements/1.1/">

        <rdfs:Class rdf:about=Airfield"/>

        <rdfs:Class rdf:about=Airport">
                <rdfs:subClassOf rdf:resource=Airfield"/>
        </rdfs:Class>
</rdf:RDF>
```

Figure 12 - Code example for RDF/XML class notation

Property

A property in RDF Schema is an instance of the class `rdf:Property`. `rdfs:domain` indicates that a certain property applies to a certain class. `rdfs:range` indicates that the value of a certain property is an instance of a certain class or of a certain XML Schema datatype. The example below shows the usage of an `rdf:Property` with a XML Schema datatype as range:

```
<?xml version=?1.0"?>
<rdf:RDF
        xml:base=?http://www.cat.com/schema/"
        xmlns:rdf=?http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:dc=?http://purl.org/dc/elements/1.1/">

        <rdfs:Class rdf:about=Airfield"/>

        <rdfs:Class rdf:about=Airport">
                <rdfs:subClassOf rdf:resource=Airfield"/>
        </rdfs:Class>

        <rdf:Property rdf:about="airport-name">
                <rdfs:domain rdf:resource="Airport"/>
                <rdfs:range rdf:resource="&xsd;string"/>
        </rdf:Property>
</rdf:RDF
```

Figure 13 - Code example for RDF/XML property notation

2.3.6 Summary of Important RDF(S)Vocabulary

Table 2 states the RDF and RDF Schema vocabulary that was used in this thesis.

Table 2: RDF/RDF Schema vocabulary [16]

Basic RDF Vocabulary	
Term	**Description**
rdf:RDF	Indicates an RDF/XML document.
rdf:about	Indicates the subject of an RDF statement.
rdf:resource	Indicates the object of an RDF statement.
Basic RDF Schema Vocabulary	
Term	**Description**
rdfs:Class	Resource indicating the class of all classes.
rdfs:subClassOf	Property that defines that a class is a subclass of another class.
rdf:Property	Resource indicating the class of all properties.
rdfs:domain	Property that is used to indicate that a particular property applies to a designated class (= domain of that property).
rdfs:range	Property that is used to indicate that the values of a property are instances of a designated class (= range of that property).

2.4 RDF Mapping Language

The RDF Mapping language (RML) [20] [21] is based on R2RML [20] [21] and is a superset of R2RML. With RML, customized mapping rules can be expressed to map heterogeneous data structures and serializations into the RDF data model. R2RML, on the other hand, is the W3C standard mapping language for mapping relational databases into RDF. RML aims to broaden the scope of R2RML to provide a mapping for multiple heterogeneous sources, which leads to higher integrated datasets and also a better, richer interlinking among the resources.

2.4.1 R2RML

With R2RML, customized mappings can be expressed to convert data from a relational database into an RDF dataset. One or more **triples maps** are the basis of an R2RML mapping, which consists of three parts:

- one **logical table** (`rr:LogicalTable`)
- one **subject map** (`rr:SubjectMap`)
- zero or more **predicate-object maps** (`rr:predicateObjectMap`).

Thus, a **triples map** is a rule which defines how the tuples of the logical table will be mapped to RDF Triples. The mapping of the data is carried out using a **logical table** with an iteration over the rows of the table. As a **logical table** can either be used a relational table, an SQL view or an SQL select query. The rule that is used to generate the URIs for the resources is defined in the **subject map**. All RDF triples that are generated from a triples map use the same subject map to obtain the subject of the triple. The predicate-object map also consists of two parts:

- Predicate Maps
- Object Maps

While the **predicate map** contains the rules that generate the predicates of the RDF triples, the **object map** contains the rules that generate the objects of the RDF triples. Together, the **subject map**, the **predicate map** and the **object map** form the **term map**. This map is used to generate the RDF terms, which means in the domain of R2RML a RDF term corresponds to the term RDF statement. This term map can either be a constant-valued Term Map or a column-valued Term Map. A constant-valued term map generates the same RDF term, while a column-valued term map is a string template that is valid and can contain referenced columns.

A join condition (`rr:joinCondition`) is needed, when the triples maps refer to different logical tables. The R2RML join condition works exactly like the join in SQL:

- It consists of the reference to a column name which exists in the logical table of triples map.
- The triples map contains
    - the Referencing-Object-Map (`rr:child`) and
    - the reference to a column name which exists in the logical table of the triples map of the Referencing Object Map (`rr:parent`)

Figure 14 summarizes all the above explained concepts of R2RML graphically.

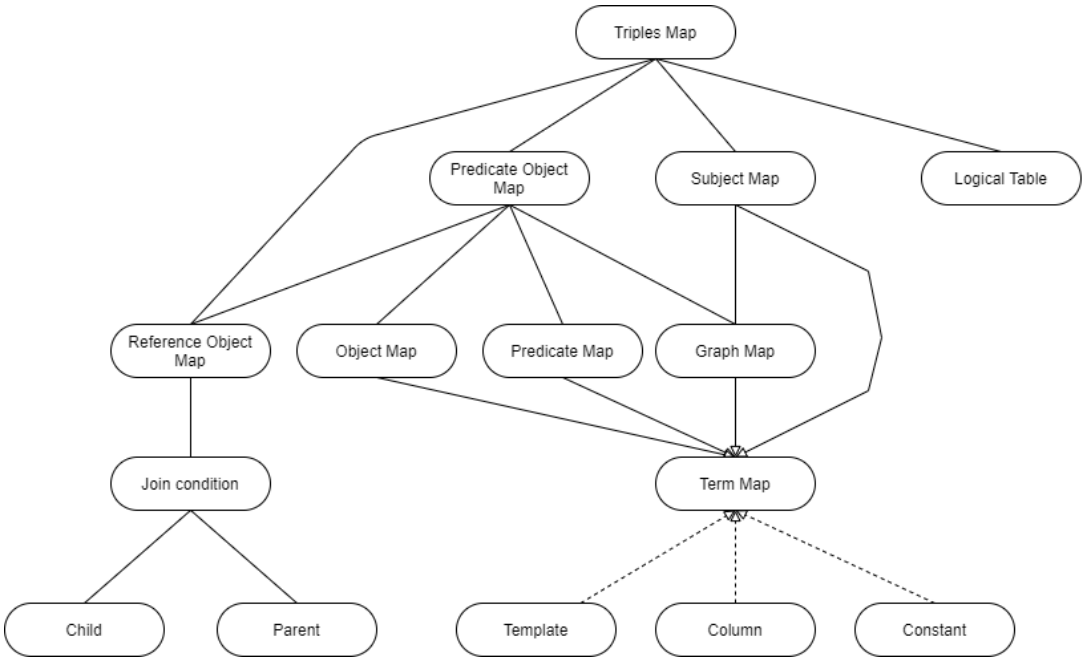


Figure 14 - R2RML concepts [21]

2.4.2 RML and its Differences to R2RML

RML retains the mapping definitions of R2RML but excludes all the references specific to relational databases from the model. In comparison to R2RML, the potential input in RML is not limited to a relational database and it also can be a variation of one or more input sources. RML also defines the mappings in a more generic way, so they are easier to cover other references of other data structures. But in general the RML mapping follows the same syntax as R2RML. Table 3 summarises the differences between R2RML and RML.

Table 3: Summary of differences between R2RML and RML [20]

	RML	R2RML
Input reference	Table name	Source
Value reference	Column	Reference
Iteration model	Per row (implicit)	Defined
Source expression	SQL (implicit)	Reference formulation

The following section will address and explain the concepts of R2RML (Section 2.4.1) in the frame of RML. Concepts like the definition and explanation of the triples map, subject map and predicate-object map will not be made, as they do not differ between R2RML and RML.

Logical Source

RML's logical source is an extension of R2RML's logical table. It is used to define the input source with its data, that should be mapped to RDF triples. While in R2RML the logical table defines the database table that should be mapped, in RML the logical source can define any input source. Therefore, the generic concept of the logical source was introduced in RML.

Reference Formulation

As RML offers a broader variety of input sources, it also needs to deal with different data serialisations and therefore, with different ways to refer to the elements or objects. While R2RML uses the column names for this purpose, in RML any reference to the logical source needs to be defined in a way that is relevant to the input source. RML uses the reference formulation to indicate the formulation that is used to refer to the input source's data, for example if an XML file is used XPath was defined as the reference formulation.

Iterator

In R2RML the definition of the iterator is implicit because it is already known that a per-row iteration is necessary. As the logical source in RML can refer to any data source, the iteration pattern cannot be implicitly assumed, so it needs to be defined. Therefore, the iterator was introduced in RML to define the iteration pattern over the input source. The iterator also specifies the data extract that should be mapped with every iteration. That means, the iterator specifies or defines how the data should be accessed. If the input data source is for example an XML document, the iterator can be defined as an XPath expression.

Logical Reference

R2RML uses a column-valued term map to determine the column names of the database. With RML a more generic way is needed and therefore the logical reference was introduced. The value of the logical reference must be a valid reference to the input dataset's data. It should be a valid reference according to:

- the already defined reference formulation,
- the string template that is used to define the template-valued term map and
- the value of the iterator.

Referencing Object Map

The last concept that is extended by RML is the referencing object map. In RML the join condition works as follows:

- The child reference (`rr:child`) defines the reference to the data value of the logical source which contains the referencing object map.
- The parent reference (`rr:parent`) defines the reference to the data extract of the parent triples map of the referencing object map. The reference of the parent is defined with a reference formulation that is defined in the logical source of the parent triples map (`rr:parentTriplesMap`).

Figure 15 shows the above-mentioned explanation of the join condition in a code example.

```
<#Airport-Mapping> a rr:TriplesMap ;
    rml:logicalSource [
        rml:iterator              "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice";
        rml:referenceFormulation  <http://semweb.mmlab.be/ns/ql#XPath>;
        rml:source                "airport-heliport.xml";
    ];
    rr:subjectMap [
        rr:template  "http://example.org/AirportHeliport/{@id}"
    ];
    rr:predicateObjectMap [
        rr:predicate  <http://www.aixm.aero/schema/5.1.1/servedCity>;
        rr:objectMap [
            rr:parentTriplesMap  <#Location-Mapping>
            rr:joinCondition [
                rr:child   "servedCity/City/@id";
                rr:parent  "@id"
            ] ;
        ];
    ].

<#Location-Mapping> a rr:TriplesMap;
    rml:logicalSource [
        rml:iterator              "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice//City";
        rml:referenceFormulation  <http://semweb.mmlab.be/ns/ql#XPath>;
        rml:source                "airport-heliport.xml";
    ];
    rr:subjectMap [
        rr:template  "http://example.org/AirportHeliport/City/{@id}"
    ];
    rr:predicateObjectMap [
        rr:predicate  <http://www.aixm.aero/schema/5.1.1/cityName>
        rr:objectMap [
            rml:reference  "cityName"
        ];
    ].
```

Figure 15 - Code example for a join condition in RML [20]

Which means the ID that is referred to in the `rr:child` reference is the ID of the city that is contained in the airport-heliport triples map, while the ID that is referred to in the `rr:parent` reference is the ID of the city of the `<#Location-Mapping>` triples map that is stated in the `rr:parentTriplesMap`.

Figure 16 summarises the above explained concepts of RML graphically. The green-coloured boxes are the concepts that are extended by RML, while the boxes in orange are the concepts that are specific RML concepts.

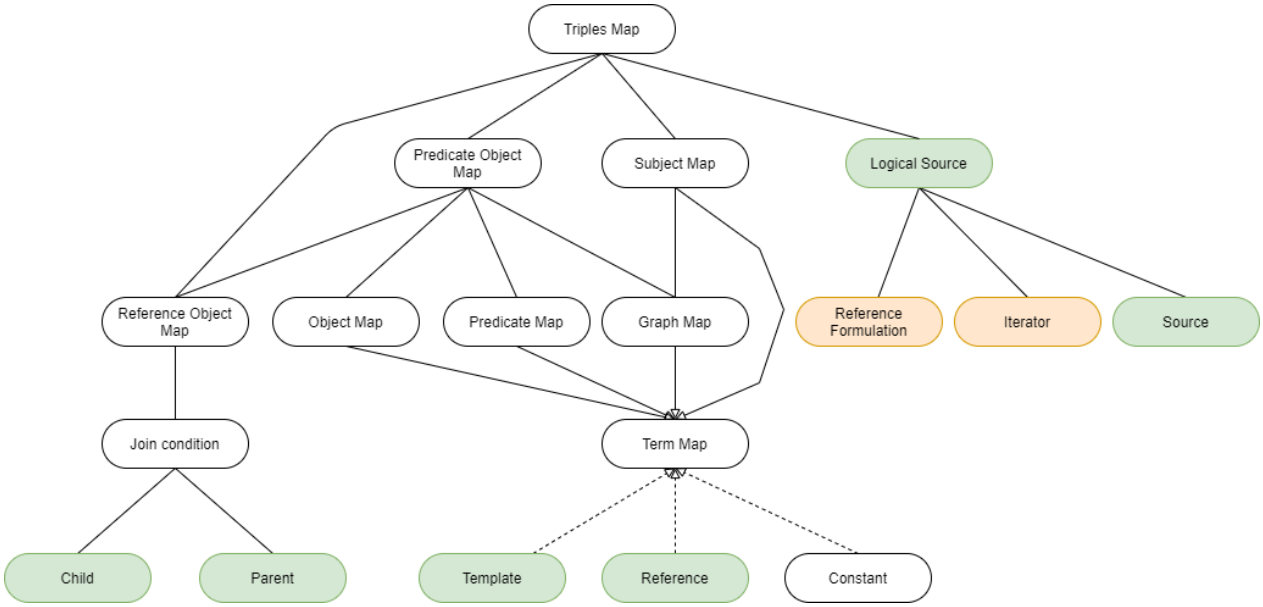


Figure 16 - RML concepts [21]

To execute the RML rules that were written to map the data into RDF, the tool RMLMapper was created by RML.io. It is a command line tool, where the RML rule file can be selected and executed. After a successful execution a RDF file will be created with valid RDF data. [8]

2.5 XML and XQuery

The Extensible Markup Language (XML) is a mark-up language that was designed for describing structured documents. Compared to HTML, XML is a flexible hierarchical data model consisting of elements. In XML the tags also do not have semantics that indicates how the documents should be presented through a Web browser. By default, an XML document (see Figure 17) consists of plain text and markup (tags), that can be interpreted by applications. Usually, an XML document starts with an optional instruction tag that contains the encoding declaration (UTF-8, UTF-16, etc.) as well as the version of the XML document. For further information on XML the thesis refers to [16] and [22].

```
<?xml version =”1.0” encoding=”UTF-8” ?>
<airportHeliports>
        <airportHeliport docid=”www.cat.com/airportHeliport#R1”>
                <name>John F. Kennedy International Airport</name>
                <ICAO>KJFK</ICAO>
                <type>airport</type>
        </airportHeliport >
        <airportHeliport docid=” www.cat.com/airportHeliport#R2”>
                <name>Anduki Airfield</name>
                <ICAO/>
                <type>heliport</type>
        </airportHeliport>
</airportHeliport>
```

Figure 17 - Code example for XML [16]

In an XML document, a start tag and an end tag mark an element, the contents of the element contained between those tags. When an XML tag has no content, an empty-element tag can be used for this purpose. An example for an empty-element tag can be seen in the second `airportHeliport` element with the `ICAO` tag. XML elements can be nested in other XML elements. This is used to define a structure for the document. Usually there is one root element (`airportHeliports`) and within this root element, other elements of the same type are nested (`airportHeliport`).

Element tags (start-tag and empty-element tag) can contain zero ore more attributes. An attribute consists of a name that is followed by an “=” which is followed by the value of the attribute. The value of the attribute is usually enclosed in double quotes. The start-tag `airportHeliport` for example has the attribute `docid=”www.cat.com/airportHeliport#R1”`.

XML also has the concepts of URI, URI reference, Namespace, and qualified name. These concepts are fundamental to structure the Semantic Web as a federated, distributed information space, because these concepts provide a scheme for addressing which is distributed, effective and stable.

Table 4 gives an explanation and an example to the XML concepts that were mentioned above.

Table 4: XML concepts [16]

Term	Explanation	Example
Resource	Anything with an identity, be it a retrievable digital entity or a physical entity	Digital entity: Image, electronic document, etc. Physical Entity: Book, etc.
Uniform Resource Identifier (URI)	Identifies a resource in the Web with a character string	URI following the HTTP scheme: http://www.mysite.com/pub/foobar.html
URI Reference (URIref)	URI with an optional fragment identifier. The fragment identifier is preceded by the character "#".	URIref identifying an individual. http://www.w3.org/People/EM/contact#me
Namespace	Collection of names, identified by a URIref	RDF namespace: http://www.w3.org/1999/02/22-rdf-syntax-ns# With the prefix: rdf
Qualified Name (QName)	Identifies a name in a namespace. Syntax: n:p n = namespace prefix p = local part	QName rdf:description: Namespace = rdf Local part = description. It expands to the URIref: http://www.w3.org/1999/02/22-rdf-syntax-ns#description

To process and query those XML data sources, the XML query language XQuery [23] was invented. XQuery is designed in a way, that queries are concise and can be easily understood. XQuery is a functional programming language [24] and it can also be used to transform XML documents.

3 Schema Mapping

The Unified Modelling Language (UML) is a widely used modelling language for the representation of data sources. For an introduction to UML, the thesis refers to Section 2.2. For further information on UML the thesis refers to the UML standard [25]. UML is also used to represent the AIXM standard. This chapter first presents the general rules for mapping UML to RDFS. It then exemplifies how those rules can be used to transform the AIXM schema into RDFS.

3.1 Rules

This master's thesis considers UML classes and their attributes, associations between classes, and generalization/specialization. Table 5 gives an overview of the mapping rules, which will be explained in the following. In general, these rules help to transform the XMI representation of the UML class diagram to the RDFS representation.

Table 5: Overview of the mapping rules for UML to RDFS according to [26]

UML	RDFS
Class	rdfs:Class
Attribute	rdf:Property rdfs:domain rdfs:range
Association	rdf:Property rdfs:domain rdfs:range
Generalization	rdfs:subClassOf

The rules defined in the following section are used to produce valid RDFS from UML class diagrams. Rules 1 to 3, 5 and 6 were taken as-is from Tong et al. [26]. Rule 4 was modified with respect to Tong et al. to ensure that the associations are still navigable after the transformation from UML to RDFS is conducted.

- **Rule 1: Identifier mapping**
  Each class diagram has a set of identifiers. For each identifier in this set, a corresponding international resource identifier (IRI) must be created. If the UML class diagram contains two identifiers with the same name, one of the identifiers must be renamed during the transformation process. Within the thesis the URI consists of the URL "http://aixm.aero/" and the name of the UML class, the attributes of the UML class or the associations of the UML class. If for example a name of an attribute of an UML class appears more than once, the class name was added to the attribute name.
- **Rule 2: Class mapping**
  For each class in the class diagram an RDFS class must be created.

```
<rdfs:Class rdf:ID ="http://aixm.aero/#airfield"/>
```

Figure 18 - Code example for RDFS class

- **Rule 3: Generalization mapping**
  If the class `airfield` in the class diagram is a generalization of another class (`airport`), then the class `airfield` is the superclass of `airport`, which in turn is referred to as the subclass of `airfield`. In RDFS, the property `subClassOf` establishes the generalization relationship.

```
<rdfs:Class rdf:ID ="http://aixm.aero/#airfield"/>
<rdfs:Class rdf:ID ="http://aixm.aero/#airport">
        <rdfs:subClassOf rdf:resource ="http://aixm.aero/#airfield"/>
</rdfs:Class>
```

Figure 19 - Code example for the generalization mapping

- **Rule 4: Association mapping**
  For each association in the UML class diagram an RDFS class must be created. In order to make the associations navigable, the following approach was chosen within the master's thesis. For each association in the UML class diagram, an RDFS property must be created. For each property, the domain and range is defined. The domain contains the source of the association and the range contains the target of the association.

```
<rdfs:Class rdf:ID ="http://aixm.aero/#airport"/>
<rdfs:Class rdf:ID ="http://aixm.aero/#runway"/>
<rdf:Property rdfs:ID = "http://aixm.aero/#located_at">
        <rdfs:domain rdf:resource = "http://aixm.aero/#runway"/>
        <rdfs:range rdf:resource = "http://aixm.aero/#airport"/>
</rdf:Property>
```

Figure 20 - Code example for association mapping

- **Rule 5: Attributes mapping**
  Each class consists of one or more attributes and for each attribute of the class, an RDFS property must be created. Each property consists of a domain and a range. The domain refers to the class to which the attribute belongs. The range specifies the datatype of the attribute.

```
<rdfs:Class rdf:ID ="http://aixm.aero/#airfield"/>
<rdf:Property rdfs:ID = "http://aixm.aero/#name - airfield">
        <rdfs:domain rdf:resource = "http://aixm.aero/#airfield"/>
        <rdfs:range rdf:resource = "http://aixm.aero/#xsd:string"/>
</rdf:Property>
<rdf:Property rdfs:ID = "http://aixm.aero/#ICAO - airfield">
        <rdfs:domain rdf:resource = "http://aixm.aero/#airfield"/>
        <rdfs:range rdf:resource = "http://aixm.aero/#xsd:string"/>
</rdf:Property>
```

Figure 21 - Code example for attributes mapping

- **Rule 6: Datatype mapping**
  For each datatype in the UML class diagram a corresponding RDFS datatype must be defined. In general, RDFS uses XML Schema datatypes for the equivalent UML data types as shown in Table 6.

Table 6: Mapping of UML datatype to RDFS [26]

UML	RDFS
string	xsd:string
smallint	xsd:short
integer	xsd:integer
decimal	xsd:decimal
float	xsd:float
time	xsd:Time
date	xsd:Date

3.2 Transformation Procedure

In the following, this section illustrates how to work with the general mapping rules on an example from the AIXM standard. This section thus explains how to create valid RDFS from UML class diagrams using the presented mapping rules. In detail that means this section shows the transformation of the UML XMI representation to the XML representation of RDFS. Figure 22 shows a simplified extract of the UML representation of information about airports and runways from the AIXM specification.

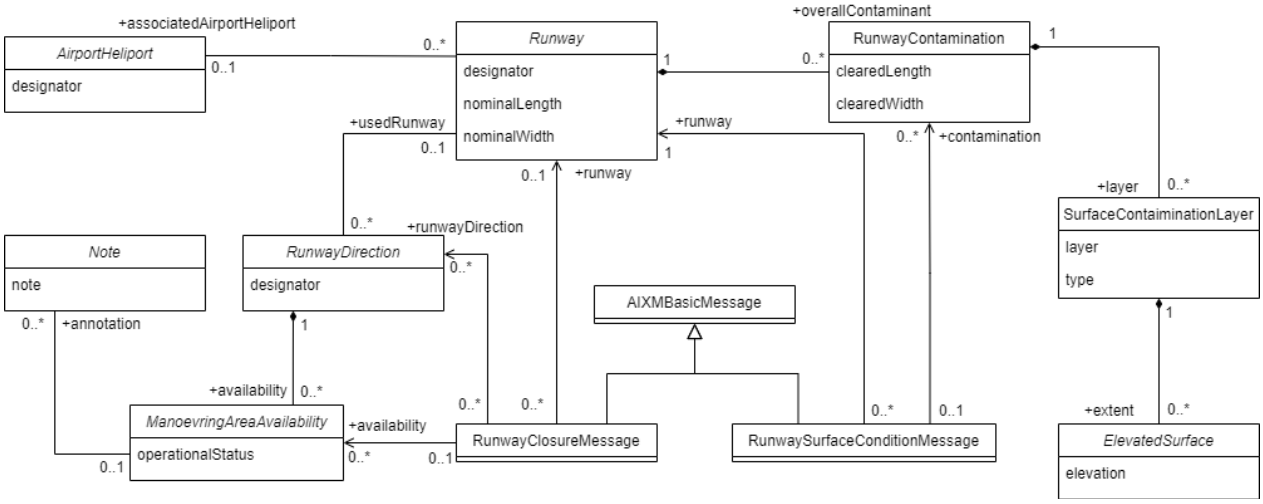


Figure 22 - Simplified AIXM UML diagram

3.2.1 Step 1: Examination of the UML Diagram

In order to get started the first step is to fully identify classes, attributes, association and their direction and generalization/specialization relationships between classes of the UML class diagram that should be mapped.

Classes and Attributes

The simplified AIXM diagram contains eleven classes, e.g., `Runway`, `AirportHeliport` and `Note`. Each class contains attributes, for example the class `Runway` has the attributes `designator`, `nominalLength` and `nominalWidth`.

Associations

The class `Runway` has several associations. Now the source and target classes of the associations must be identified. For example, the association between `Runway` and `RunwayClosureMessage`.

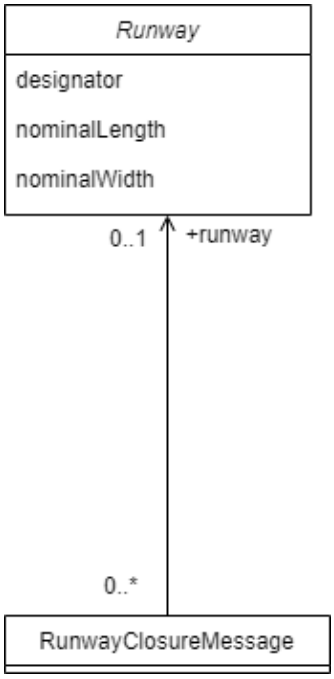


Figure 23 - Association between `Runway` and `RunwayClosureMessage`

In this association the class `RunwayClosureMessage` is the source and the class `Runway` is the target. While one runway can have zero or many closure messages, one closure message can only belong to a maximum of one runway. The runway to which the closure message belongs is defined in the closure message. This is the reason, why the closure message is the source of this association.

```
<connector xmi:idref="EAID_82E4591C_1B24_46b6_A502_7782D7F4CF8E">
        <source
           xmi:idref="EAID_82AD95DB_E21D_401f_9794_FE3FE1720197">
                <model ea_localid="32" type="Class" name="RunwayClosureMessage"/>
                <role visibility="Public" targetScope="instance"/>
                <type multiplicity="0..*" aggregation="none" containment="Unspecified"/>
                <constraints/>
                <modifiers isOrdered="false" changeable="none" isNavigable="false"/>
                <style
                   value="Union=0;Derived=0;AllowDuplicates=0;Owned=0;Navigable=Unspecified;"/>
                <documentation/>
                <xrefs/>
                <tags/>
        </source>
        <target
           xmi:idref="EAID_7488621C_F90E_46e5_9D90_E9095F8F21BB">
                <model ea_localid="28" type="Class" name="Runway"/>
                <role name="runway" visibility="Public" targetScope="instance"/>
                <type multiplicity="0..1" aggregation="none" containment="Unspecified"/>
                <constraints/>
                <modifiers isOrdered="false" changeable="none" isNavigable="true"/>
                <style
                   value="Union=0;AllowDuplicates=0;Owned=0;Navigable=Navigable;Derived=1;"/>
                <documentation/>
                <xrefs/>
                <tags/>
        </target>
        <model ea_localid="72"/>
        <properties ea_type="Association" direction="Source -&gt; Destination"/>
        <modifiers isRoot="false" isLeaf="false"/>
        <parameterSubstitutions/>
        <documentation/>
        <appearance
           linemode="3" linecolor="-1" linewidth="0" seqno="0" headStyle="0" lineStyle="0"/>
        <labels lb="0..*" rb="0..1" rt="+/runway"/>
        <extendedProperties/>
        <style/>
        <xrefs/>
        <tags/>
</connector>
```

Figure 24 - XMI representation of the association between `Runway` and `RunwayClosureMessage`

Figure 24 shows the XML representation of the association between `Runway` and `RunwayClosureMessage` (Figure 23). While source and target always define the classes that are linked through the association, the actual direction can be found in the element `properties` in the attribute `direction` (see Table 7 for details). In case of Figure 24, the source is greater than ("-&gt") the target, which means the association flows from source to target. The type of the connection can be found in the element `properties` in the attribute `ea_type`. Consequently, when mapping the associations, only the connectors that have the attribute `ea_type` with the value `Association` have to be considered.

Table 7: Direction of association

Attribute content	Association flow
"Source -> Destination"	Source → Target
"Destination -> Source"	Target → Source

Generalization

The class `AIXMBasicMessage` is the superclass of `RunwayClosureMessage` and `RunwaySurfaceConditionMessage` (see Figure 25).

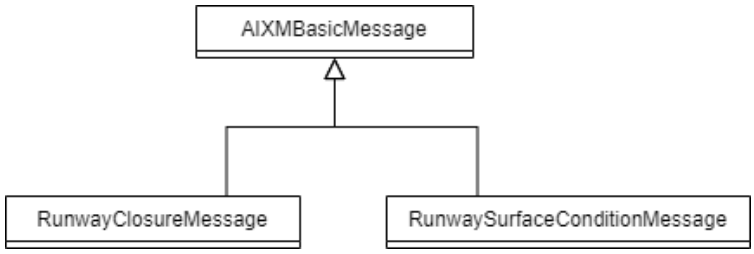


Figure 25 - Generalization example of the simplified AIXM UML diagram

Figure 26 shows the XMI representation of the generalization from Figure 25. The tags relevant in this figure are the `source`, `target` and `properties` tags. The remaining tags only serve to graphically represent the association between the classes in the UML diagram.

```
<connector xmi:idref="EAID_17AA4E84_4B7B_4b5d_8630_2CB3109EE927">
        <source xmi:idref="EAID_82AD95DB_E21D_401f_9794_FE3FE1720197">
                <model ea_localid="32" type="Class" name="RunwayClosureMessage"/>
                <role visibility="Public" targetScope="instance"/>
                <type aggregation="none" containment="Unspecified"/>
                <constraints/>
                <modifiers isOrdered="false" changeable="none" isNavigable="false"/>
                <style value="Union=0;Derived=0;AllowDuplicates=0;"/>
                <documentation/>
                <xrefs/>
                <tags/>
        </source>
        <target xmi:idref="EAID_3EC62C41_5192_4854_8413_1D31518D410A">
                <model ea_localid="30" type="Class" name="AIXMBasicMessage"/>
                <role visibility="Public" targetScope="instance"/>
                <type aggregation="none" containment="Unspecified"/>
                <constraints/>
                <modifiers isOrdered="false" changeable="none" isNavigable="true"/>
                <style value="Union=0;Derived=0;AllowDuplicates=0;"/>
                <documentation/>
                <xrefs/>
                <tags/>
        </target>
        <model ea_localid="15"/>
        <properties ea_type="Generalization" direction="Source -&gt; Destination"/>
        <parameterSubstitutions/>
        <documentation/>
        <appearance
           linemode="3" linecolor="-1" linewidth="0" seqno="0" headStyle="0" lineStyle="0"/>
        <labels/>
        <extendedProperties virtualInheritance="0"/>
        <style/>
        <xrefs/>
        <tags/>
</connector>
```

Figure 26 - XMI representation of generalization relationship between `AIXMBasicMessage` and `RunwayClosureMessage`

In the example in Figure 26 the target of this connection defines the superclass (`AIXMBasicMessage`), while the source defines the subclass (`RunwayClosureMessage`). The type of the connection can again be found in the element `properties` in the attribute `ea_type`. That means, when mapping the associations only the connectors that have the attribute `ea_type` with the value `Generalization` have to be considered.

3.2.2 Step 2: Transforming Classes and Generalizations

Step 2 of the procedure is to create an `rdfs:Class` element for each class in the UML class diagram. When looking at Figure 27, the UML classes are defined with the element `packagedElement` and the property `xmi:type` with the value `uml:Class`. Thus, for each UML class defined in the XMI representation (see Figure 27) an `rdfs:Class` element must be defined.

```
<packagedElement xmi:type="uml:Class"
   xmi:id="EAID_7488621C_F90E_46e5_9D90_E9095F8F21BB" name="Runway" visibility="public">
        <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_4D0ECC80_FEAB_420c_897A_5E14E4A51062"
           name="designator" visibility="public" isStatic="false" isReadOnly="false"
           isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
                <lowerValue xmi:type="uml:LiteralInteger"
                   xmi:id="EAID_LI000145_FEAB_420c_897A_5E14E4A51062" value="1"/>
                <upperValue xmi:type="uml:LiteralInteger"
                   xmi:id="EAID_LI000146_FEAB_420c_897A_5E14E4A51062" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_F7301A9B_792B_456b_866E_924DE8827CF6"
           name="nominalLength" visibility="public" isStatic="false" isReadOnly="false"
           isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
                <lowerValue xmi:type="uml:LiteralInteger"
                   xmi:id="EAID_LI000147_792B_456b_866E_924DE8827CF6" value="1"/>
                <upperValue xmi:type="uml:LiteralInteger"
                   xmi:id="EAID_LI000148_792B_456b_866E_924DE8827CF6" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_B18A1DAA_5D5C_4a85_9E13_706B35ACAFD9"
           name="nominalWidth" visibility="public" isStatic="false" isReadOnly="false"
           isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
                <lowerValue xmi:type="uml:LiteralInteger"
                   xmi:id="EAID_LI000149_5D5C_4a85_9E13_706B35ACAFD9" value="1"/>
                <upperValue xmi:type="uml:LiteralInteger"
                   xmi:id="EAID_LI000150_5D5C_4a85_9E13_706B35ACAFD9" value="1"/>
        </ownedAttribute>
</packagedElement>
```

Figure 27 - XMI representation of the UML class `Runway`

Figure 28 shows the XML representation of the resulting RDFS class for the UML class `Runway`.

```
<rdfs:Class xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#Runway"/>
```

Figure 28 - RDFS class `Runway`

If the UML class has a generalization connector, an `rdfs:subClassOf` element must be defined. For the generalization mapping, only the connectors with the property `ea_type` and the value `Generalization` must be considered. Figure 26 shows an example of a generalization connector.

As it can be seen in Figure 25 the AIXM class `RunwayClosureMessage` is a subclass of the AIXM class `AIXMBasicMessage`. Figure 29 shows the XML representation of the RDFS class for the UML class `RunwayClosureMessage`.

```
<rdfs:Class xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#RunwayClosureMessage">
        <rdfs:subClassOf rdf:resource="aixm.aero/#AIXMBasicMessage"/>
</rdfs:Class>
```

Figure 29 - RDFS mapping result `RunwayClosureMessage`

3.2.3 Step 3: Transforming the Attributes

The third step of the procedure is the mapping of the attributes. For each attribute of a class an `rdfs:Property` element must be created. The attributes are declared with the XML element `ownedAttribute` and the property `xmi:type` with the value `uml:Property` (Figure 27). The domain of the property defines the class to which the attribute belongs to, while the range of the property defines the datatype of the attribute. As it can be seen in Figure 27, the class `Runway` has three attributes: `designator`, `nominalLength`, and `nominalWidth`. Figure 30 then shows the result in RDFS for the attributes of the class `Runway`.

```
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#designator">
        <rdfs:domain rdf:resource="http://aixm.aero/#Runway"/>
        <rdfs:range rdf:resource="http://aixm.aero/#xsd:string"/>
</rdfs:Property>
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#nominalLength">
        <rdfs:domain rdf:resource="http://aixm.aero/#Runway"/>
        <rdfs:range rdf:resource="http://aixm.aero/#xsd:decimal"/>
</rdfs:Property>
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#nominalWidth">
        <rdfs:domain rdf:resource="http://aixm.aero/#Runway"/>
        <rdfs:range rdf:resource="http://aixm.aero/#xsd:decimal"/>
</rdfs:Property>
```

Figure 30 - RDFS mapping result of the attributes of the class `Runway`

3.2.4 Step 4: Transforming the Associations

The last step of this procedure is the mapping of the associations. For each association an `rdfs:Property` element must be created. The domain of the property refers to the source of the association, while the range of the property refers to the target of the association. As it can be seen in Figure 22 the class `Runway` has five associations. An example of an association definition can be found in Figure 23. For the mapping of the associations only the connectors with the property `ea_type` that have the value `Association` have to be considered; the direction of the

association is also important (see Section 3.2.1 for details). Figure 31 shows the result in RDFS for the associations of the class `Runway`.

```
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#associatedAirportHeliport">
        <rdfs:domain rdf:resource="http://aixm.aero/#Runway"/>
        <rdfs:range rdf:resource="http://aixm.aero/#AirportHeliport"/>
</rdfs:Property>
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#runwayClosure">
        <rdfs:domain rdf:resource="http://aixm.aero/#RunwayClosureMessage"/>
        <rdfs:range rdf:resource="http://aixm.aero/#Runway"/>
</rdfs:Property>
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#runwaySurface">
        <rdfs:domain rdf:resource="http://aixm.aero/#RunwaySurfaceConditionMessage"/>
        <rdfs:range rdf:resource="http://aixm.aero/#Runway"/>
</rdfs:Property>
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#runwayDirection">
        <rdfs:domain rdf:resource="http://aixm.aero/#RunwayDirection"/>
        <rdfs:range rdf:resource="http://aixm.aero/#Runway"/>
</rdfs:Property>
<rdfs:Property xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   rdf:about="http://aixm.aero/#runwayContaminant">
        <rdfs:domain rdf:resource="http://aixm.aero/#RunwayContamination"/>
        <rdfs:range rdf:resource="http://aixm.aero/#Runway"/>
</rdfs:Property>
```

Figure 31 - RDFS mapping result of the associations of the class `Runway`

4 Instance Data Mapping

This chapter describes how to define the R2RML mapping rules that are needed to transfer the XML instance data for a particular ATM information model to RDF. In particular, the first section explains the elements of R2RML mapping rules in general, with an illustration from AIXM. are necessary to create RDF from AIXM instance data. In the second section the procedure on how to exactly define and create the mapping rules is explained.

4.1 Mapping Rules

In this Section, the thesis describes which RML mapping rules have to be defined to transform AIXM XML data into RDF. For more information, the thesis refers to Dimou et. al ([20]) and Kyzirakos et. al ([21]) as well as the tutorial “Generate RDF from an XML file” [8]. The thesis also refers to Section 2.4 for a quick introduction to RML mapping rules. In general, there are two basic rules behind the creation of the RML mapping rules:

1. Rules that describe the XML file.
    - Definition of the logical source.
2. Rules that define how the RDF terms (subject, predicate, and object) are created and how they are assembled into an RDF triple.
    - Definition of the subject map, predicate map and object map.

Both basic rules are combined to a triples map (Rule 1), as it groups all the rules. Rule 2 belongs to the first main rule and the Rules 3 to 5 belong to the second main rule.

- **Rule 1: Triples Maps**
  Looking at the XML file that contains the instance data, there is one main element in the XML file and for that element a triples map needs to be declared. Figure 32 shows the declaration of a triples map for the transformation of `AirportHeliport` elements into RDF.

```
<http://example.com/ns#TriplesMapAirportHeliport> a rr:TriplesMap ;
```

Figure 32 - Declaration of a triples map

  If the element `AirportHeliport` contains sub-elements that represent associations to other classes, another triples map needs to be declared (see Figure 33).

```
<http://example.com/ns#TriplesMapCity-ID_110>
```

Figure 33 - Declaration of a second triples map

- **Rule 2: Logical Source**
  For each triples map, a logical source definition is needed. The logical source contains the data source, the reference formulation, and the iterator for the data source. Figure 34 shows the logical source of the `AirportHeliport` triples map (see Figure 32).

```
rml:logicalSource [
        rml:iterator "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice" ;
        rml:referenceFormulation <http://semweb.mmlab.be/ns/ql#XPath> ;
        rml:source "airport-heliport.xml"
] ;
```

Figure 34 - Definition of the logical source

- **Rule 3: Subject Map**
  Each triples map needs one subject map, which contains the subject of the RDF Triple. Figure 35 shows the definition of the subject map.

```
rr:subjectMap [
    rr:template "http://example.org/AirportHeliport/{@id}"
];
```

Figure 35 - Definition of the subject map

- **Rule 4: Predicate-Object Maps**
  Each triples map can have one or many predicate-object maps, that define the predicate as well as the object of the RDF Triple. Figure 36 shows the definition of a predicate-object map.

```
rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant aixm:AirportHeliport
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate aixm:designator;
    rr:objectMap [
      rml:reference "designator"
    ]
  ];
```

Figure 36 - Definition of the predicate-object map

The first predicate-object map annotates the predicate and objects to the class and is only needed once. The second predicate-object map annotates the value of the designator (object) with the predicate `aixm:designator` and is needed again for all the elements that are nested within the `AirportHeliport` element.

- **Rule 5: Associations between Triples Maps**
  In order to use subjects of another triples map, a referencing object map is needed (see Figure 37). The predicate-object map refers to another triples map `#City` with a join condition which defines how the two triples maps should be joined. The second part of the figure shows the triples map the predicate-object map refers to.

```
rr:predicateObjectMap [
    rr:predicate aixm:servedCity;
    rr:objectMap [
      rr:parentTriplesMap <#City>;
      rr:joinCondition [
        rr:child "servedCity/City/@id";
        rr:parent "@id";
      ];
    ];
  ].

<#City>
  rml:logicalSource [
    rml:source "airport-heliport.xml";
    rml:referenceFormulation ql:XPath;
    rml:iterator
        "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice//City"
  ];
  rr:subjectMap [
    rr:template "http://example.org/City/{@id}"
  ];
  rr:predicateObjectMap [
    rr:predicate aixm:cityName;
    rr:objectMap [
      rml:reference "cityName"
    ]
  ].
```

Figure 37 - Definition of associations between triples maps

A join condition is necessary when the triples maps are based on different logical sources. As both the triples map `City` and the triples map `AirportHeliport` have different logical sources respectively a different iterator a join condition is necessary. The triples map `City` is defined according to the rules described above.

4.2 Defining the Mapping Rules

This Section explains and exemplifies how to create RML mapping rules that can be processed with the RMLMapper ([8]) to create valid RDF data. To process the RML mapping rules with the RMLMapper, the notation in which the RML rules must be written is RDF/Turtle. To explain the rules in more detail the simplified AIXM XML instance data in Figure 38 is used as a running example throughout this.

```
<?xml version="1.0" encoding="UTF-8"?>

<AIXMBasicMessage >
        <hasMember>
                <AirportHeliport
                    id="uuid.dd062d88-3e64-4a5d-bebd-89476db9ebea">
                        <timeSlice>
                                <AirportHeliportTimeSlice id="AHP_EADH">
                                        <designator>EADH</designator>
                                        <name>DONLON/DOWNTOWN HELIPORT</name>
                                        <locationIndicatorICAO>
                                                EADH
                                        </locationIndicatorICAO>
                                        <servedCity>
                                                <City id="ID_110">
                                                        <cityName>
                                                                DONLON
                                                        </cityName>
                                                </City>
                                        </servedCity>
                                </AirportHeliportTimeSlice>
                        </timeSlice>
                </AirportHeliport>
        </hasMember>
        <hasMember>
                <AirportHeliport
                    id="uuid.1b54b2d6-a5ff-4e57-94c2-f4047a381c64">
                        <timeSlice>
                                <AirportHeliportTimeSlice id="AHP_EADD">
                                        <designator>EADD</designator>
                                        <name>DONLON/INTL.</name>
                                        <locationIndicatorICAO>
                                                EADD
                                        </locationIndicatorICAO>
                                        <servedCity>
                                                <City id="ID_109">
                                                        <cityName>
                                                                DONLON
                                                        </cityName>
                                                </City>
                                        </servedCity>
                                </AirportHeliportTimeSlice>
                        </timeSlice>
                </AirportHeliport>
        </hasMember>
</AIXMBasicMessage>
```

Figure 38 - XML instance data example

4.2.1 Step 1: Understanding the XML File

The first step of the procedure to create the RML mapping rules, is to understand the XML file and its structure. This includes the identification of

- the main element for the triples map
- the iterator for the logical source
- the associations and their own logical sources

At first, the main element, the subject of the RDF triples needs to be identified. In this example the main element would be the `AirportHeliport` element because this XML file contains the instance data about the Air- and Heliports. This means, that the triples map will be the `AirportHeliport` triples map (see Section 4.2.2).

The next part would be the identification of the iterator for the logical source. The iterator defines the pattern with which the XML file is to be iterated. As the element `AirportHeliportTimeSlice` is the element that contains all the sub-elements with the Air-/Heliport data, this element is thus the iterator. The definition of the iterator in this example would be "`//AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice`".

The last part of this step is the identification of associations and their own logical sources. When looking at the example (Figure 38), each `AirportHeliport` has an association (`servedCity`) with a `City`. As the `City` element is a super-element on its own, a second triples map is needed, as well as another iterator, because the `City` element ins more nested in the XML file. The iterator for the triples map `City` would be "`//AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice//City`".

4.2.2 Step 2: Definition of the Tiples Map and its Content

After the understanding process of the XML file, the triples maps and its content can be defined in the second and last step of this process.

Step 2.1: Definition of the Prefixes

The first step in the creation process is to define the needed prefixes for RML, R2RML, RDF and AIXM (Figure 39).

```
@prefix rml: <http://semweb.mmlab.be/ns/rml#> .
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix ql: <http://semweb.mmlab.be/ns/ql#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <http://example.org/rules/> .
@prefix aixm: <http://www.aixm.aero/schema/5.1.1/> .
@base <http://example.com/ns#>.
```

Figure 39 - Prefixes for the RML mapping rules

Step 2.2: Definition of the Triples Map and the Logical Source

After the prefix declaration, the triples map `AirportHeliport` with the logical source is defined (Figure 40). The value of `rml:source` is the XML file, the contents of which are shown in Figure 38, the value of `rml:iterator` is the iterator that was identified in Section 4.2.1 and the value of `rml:referenceFormulation` is XPath, as the file to be processed by the RMLMapper is a XML file.

```
rml:logicalSource [
    rml:source "airport-heliport-simplified.xml";
    rml:iterator
        "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice";
    rml:referenceFormulation ql:XPath;
  ];
```

Figure 40 - Definition of triples map and logical source

Step 2.3: Definition of the Subject Map

After the definition of the logical source the subject map is defined (Figure 41). The subject map states the RDF term “subject” for the resulting RDF triples. In this example the ID of the `AirportHeliport` is the subject of the RDF triples.

```
rr:subjectMap [
    rr:template "http://example.org/AirportHeliport/{@id}"
  ];
```

Figure 41 - Definition of subject map

Step 2.4: Definition of the Predicate-Object Maps and the Join Conditions

The last step in the creation procedure is the definition of all the predicate-object maps and the necessary join conditions (Figure 42). For each sub-element of the element `AirportHeliportTimeSlice` a predicate-object map needs to be defined. Within the predicate-object map, the value of `rr:predicate` is the name of the RDF term “predicate”, while the value of the `rr:objectMap` references to the element in the XML file, where the RMLMapper extracts the value for the RDF term “object”.

```
rr:predicateObjectMap [
    rr:predicate aixm:designator;
    rr:objectMap [
      rml:reference "designator"
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate aixm:name;
    rr:objectMap [
      rml:reference "name"
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate aixm:locationIndicatorICAO;
    rr:objectMap [
      rml:reference "locationIndicatorICAO"
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate aixm:servedCity;
    rr:objectMap [
      rr:parentTriplesMap <#City>;
      rr:joinCondition [
        rr:child "servedCity/City/@id";
        rr:parent "@id";
      ];
    ];
  ].

<#City>
  rml:logicalSource [
    rml:source "airport-heliport.xml";
    rml:referenceFormulation ql:XPath;
    rml:iterator
        "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice//City"
  ];
  rr:subjectMap [
    rr:template "http://example.org/City/{@id}"
  ];
  rr:predicateObjectMap [
    rr:predicate aixm:cityName;
    rr:objectMap [
      rml:reference "cityName"
    ]
  ].
```

Figure 42 - Definition of the predicate-object maps and the join conditions

The predicate-object map for the element `servedCity` contains the join condition to the triples map `City`. The value of the `rr:parentTriplesMap` is the name of the triples map `City`. The `rr:joinCondition` contains the values for `rr:child rr:parent`, which specify the key to be used to perform the join. The value of `rr:child` defines the key on the side of the triples map `AirportHeliport` while the value of `rr:parent` defines the key on the side of the triples map `City` and in this example both triples maps are joined via the ID of the `City`.

4.3 Transforming AXIM XML Instance Data to RDF Using RMLMapper

After the creation of the RML mapping rules in Section 4.2, this section shows and explain how to process the RML mapping rules with the tool RMLMapper ([8]) to receive valid RDF instance data.

Step 1: Compiling the Source Code

To use the RMLMapper, the project needs to be checked out from the RMLio GitHub page via Git or another version management software, like Sourcetree, which was used within this master's thesis.

After the check-out of the project the source code needs to be built, with either the command line or an integrated development environment (IDE), such as Eclipse or IntelliJ. Within this thesis the Eclipse IDE was used, therefore this section will explain the building process within Eclipse.

The first step is to import the project as a Maven project. After the import, the first step is to successfully build the Maven project. Once the project has been built, the project can now be installed. After the successful maven install the "target" folder within the project should now contain two JAR-Files.

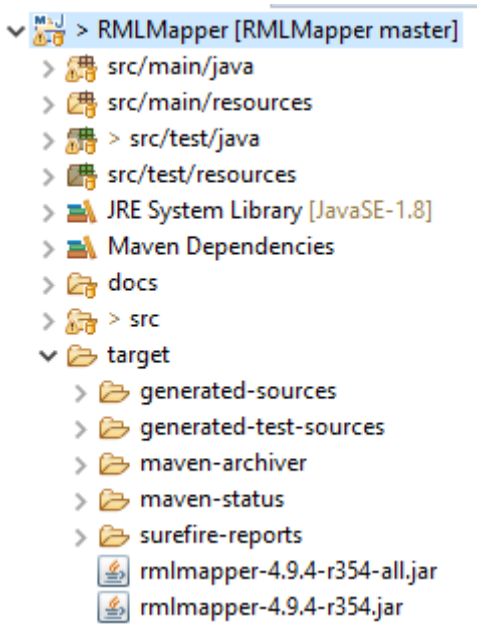


Figure 43 - `target` folder after the successful maven install

Step 2: Using the RMLMapper to Process the RML Rules

After the successful maven install, the second step can be carried out. Therefore, the XML-File needs to be copied into the folder “target”, right where the .jar-Files were filed after the install. Figure 44 shows the content of the “target” folder, after the insertion of the XML-File.

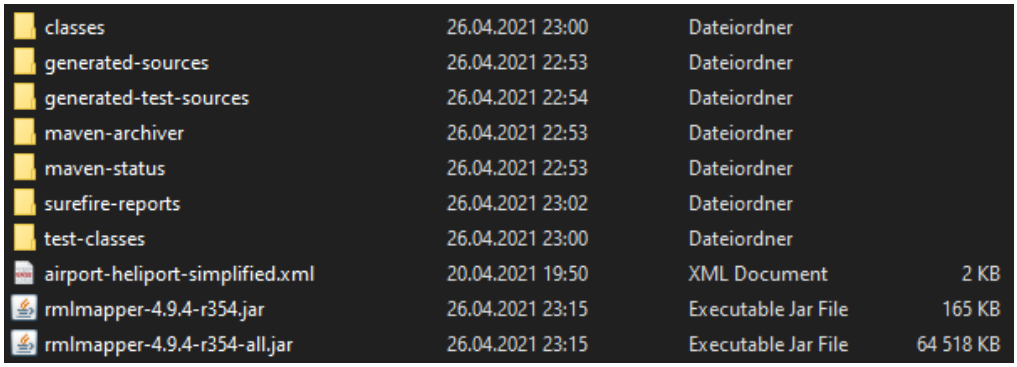


Figure 44 - Content of the RMLMapper `target` folder

Next, a command terminal is needed, which needs to be opened in the “target” folder. After opening the command terminal, the command in Figure 45 needs to be executed, to create the RDF instance data with the RML mapping rules.

```
java -jar rmlmapper-4.8.2-r308.jar
        -m "...\RML-Rules-airport.ttl"
```

Figure 45 - Command line command for running the RMLMapper

The first line of the command specifies that a Java JAR File is to be executed and which JAR-File. In the second line of the command the RML mapping rules file is stated. With `-m` one or more paths to the mapping files can be provided. After the execution of this command, the RDF instance data is displayed in the command line. Figure 46 shows the mapped RDF instance data.

```
<http://example.org/AirportHeliport/AHP_EADH>
        <http://www.aixm.aero/schema/5.1.1/servedCity>
                <http://example.org/City/ID_110>.
<http://example.org/AirportHeliport/AHP_EADH>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
                <http://www.aixm.aero/schema/5.1.1/AirportHeliport>.
<http://example.org/AirportHeliport/AHP_EADH>
        <http://www.aixm.aero/schema/5.1.1/designator>
                "EADH".
<http://example.org/AirportHeliport/AHP_EADH>
        <http://www.aixm.aero/schema/5.1.1/name>
                "DONLON/DOWNTOWN HELIPORT".
<http://example.org/AirportHeliport/AHP_EADH>
        <http://www.aixm.aero/schema/5.1.1/locationIndicatorICAO>
                "EADH".
<http://example.org/AirportHeliport/AHP_EADD>
        <http://www.aixm.aero/schema/5.1.1/servedCity>
                <http://example.org/City/ID_109>.
<http://example.org/AirportHeliport/AHP_EADD>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
                <http://www.aixm.aero/schema/5.1.1/AirportHeliport>.
<http://example.org/AirportHeliport/AHP_EADD>
        <http://www.aixm.aero/schema/5.1.1/designator>
                "EADD".
<http://example.org/AirportHeliport/AHP_EADD>
        <http://www.aixm.aero/schema/5.1.1/name>
                "DONLON/INTL.".
<http://example.org/AirportHeliport/AHP_EADD>
        <http://www.aixm.aero/schema/5.1.1/locationIndicatorICAO>
                "EADD".
<http://example.org/City/ID_110>
        <http://www.aixm.aero/schema/5.1.1/cityName>
                "DONLON".
<http://example.org/City/ID_109>
        <http://www.aixm.aero/schema/5.1.1/cityName>
                "DONLON".
```

Figure 46 - Mapped RDF instance data

In order to save the mapping result into an output file needs to be adapted, as it is shown in Figure 47. With `-o` the path to the output file can be provided. After the execution of this command, the RDF instance data no longer will be displayed in the command line but are saved into the defined output file.

```
java -jar rmlmapper-4.8.2-r308.jar
        -m "...\RML-Rules-airport.ttl"
        -o "...\Output-Files\RMLMapper-Output-Airport.rdf"
```

Figure 47 - Command line command for running the RMLMapper with defined output file

5 Automating the Mapping Procedures

This chapter describes the automation of the mapping procedures defined in Chapter 3 and Chapter 4. First, this chapter presents the tool that was created to automate the schema mapping process and the adaptions that are necessary to process other schemas with this tool. The second section explains the tool that was created for the creation of the RML mapping rules for the instance data mapping and the adaptions that are necessary to create RML mapping rules for other instance data.

All the implementations that are explained within this chapter were designed and tested using the AIXM schema version 5.1.1 and the AIXM XML “Donlon” sample data. The presented implementation do not consider the Temporality Model.

5.1 Schema Mapping

This section contains all the details that are necessary to understand, work with and adapt the tool that was created to automate the schema mapping. It explains the tool itself and the adaptions that need to be made to the tool, to map other schemas.

The tool to automate the schema mapping is an XQuery module which consists of two files: the file that defines the data source (AIXM Schema 5.1.1) and the module call as well as the file that contains the implemented mapping procedure. Since the AXM 5.1.1 UML schema is available in XMI, which is the XML format to exchange metadata about UML models via XML, a language for transforming XML documents is required; XQuery is one such language.

5.1.1 Tool for Automating the Schema Mapping Process

As described above the tool which automates the schema-mapping process consists of two files: the call script and the processing module. The content, and the workings of both files will be described in this section.

Call Script

Besides the module call the file also contains the declaration of the namespace. Figure 48 shows the namespaces and imports that are necessary for this tool. The first two imports define on one hand the path to the file that contains the functions to carry out the mapping to RDFS and on the other hand the File module which contains the necessary functions to write the resulting RDFS in a file. The path to the second file is an absolute path that points to the exact filing location of the second file. The remaining lines contain the namespaces for RDF, XMI, UML, RDFS and the File output.

```
import module namespace test="http://www.w3.org/2000/01/rdf-schema#"
        at "...\xml2rdfs-AIXM-Original.xqm";

import module namespace file = 'http://expath.org/ns/file';

declare namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#";
declare namespace xmi = "http://schema.omg.org/spec/XMI/2.1";
declare namespace uml = "http://schema.omg.org/spec/UML/2.1";
declare namespace rdfs="http://www.w3.org/2000/01/rdf-schema#";
declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
```

Figure 48 - Schema Mapping tool - Module call namespaces

After the definition of the imports and namespaces, the actual module call is carried out. This can be seen in Figure 49. The first line is the declaration of the variable that contains the path to the output file, which needs to be created beforehand. The second line contains the module call. With the function `file:write` the resulting RDFS is written into the declared output file. `rdfs:extractRDFSchema` is the actual module call where AIXM 5.1.1 is passed on to the module.

```
declare variable $filePath := "...\RDF-Schema-FULL.rdf";

file:write($filePath, rdfs:extractRDFSchema(db:open("AIXM_5.1.1")/xmi:XMI))
```

Figure 49 - Schema Mapping tool - Module call

Processing Module

The processing module consists of five functions that will be explained in this section:

- `rdfs:extractRDFSchema`
- `rdfs:extractUMLElements`
- `rdfs:extractUMLAttributes`
- `rdfs:extractUMLAssociations`
- `rdfs:extractUMLDiagrams`
- `rdfs:extractUMLDiagramElements`

Before implementing the functions, however, namespaces must be also included in this file. Figure 50 shows the namespaces that must be included, in order for the module to work. The first line defines the namespace of the module, which in this case is the namespaces of RDFS. The remaining lines contain the namespaces for RDF, XMI and UML.

```
module namespace rdfs="http://www.w3.org/2000/01/rdf-schema#";

declare namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#";
declare namespace xmi = "http://schema.omg.org/spec/XMI/2.1";
declare namespace uml = "http://schema.omg.org/spec/UML/2.1";
```

Figure 50 - Schema mapping tool - Namespaces of Process Mapping

After the namespace declaration the first and main function `rdfs:extractRDFSchema` is implemented as it is shown in Figure 51. In the first part of the function body four variables were defined that are needed throughout the whole module, `allElements`, `allClassElements`, `allDiagrams` and `allGeneralizations`. In the second part of the function body the remaining functions are called, and the results are saved within variables. In the return string the contents of those variables are put together to generate the mapped RDFS.

```
declare function rdfs:extractRDFSchema(
  $rdfSchema as element()
) as element()+ {

  let $allElements := $rdfSchema//ownedMember
  let $allClassElements := $allElements[@xmi:type="uml:Class"]
  let $allDiagrams := $rdfSchema//diagram
  let $allGeneralizations := $rdfSchema//connector

  let $elements
        := rdfs:extractUMLElements($allGeneralizations, $allClassElements, $allElements)
  let $attributes
        := rdfs:extractUMLAttributes($allClassElements)
  let $associations
        := rdfs:extractUMLAssociations($allGeneralizations, $allClassElements)
  let $diagrams
        := rdfs:extractUMLDiagrams($allDiagrams)
  let $diagramElements
        := rdfs:extractUMLDiagramElements($allElements, $allDiagrams)
  return
    <rdf:RDF>
      {$elements}
      {$attributes}
      {$associations}
      {$diagrams}
      {$diagramElements}
    </rdf:RDF>
};
```

Figure 51 - Schema mapping tool - `rdfs:extractRDFSchema`

The second function within the processing module, `rdfs:extractUMLElements`, performs the mapping of all UML elements within the class diagram. Those UML elements are mapped to RDFS classes, as it is shown in Figure 52. The first part in the function body is the extraction of the element name, which becomes the class name. This function also includes the mapping of the generalizations. Which means, that if the element the function is currently looking at is a sub-class of another class, it is mapped as the `rdfs:subClassOf` the super class. The function selects from all associations those with the type `Generalization` and checks if the current element is among them. If yes, the tag `rdfs:subClassOf` is generated. If not, the generation of the tag `rdfs:subClassOf` is skipped.

```
declare function rdfs:extractUMLElements(
  $allGeneralizations as element()+,
  $allClassElements as element()+,
  $allElements as element()+
) as element()+{
  for $element in $allClassElements
  let $elementIRI := "http://aixm.aero/" || "#" ||$element/@name/string()
  return
    <rdfs:Class rdf:about="{$elementIRI}">
    {
      let $generalizations := $allGeneralizations
      for $generalization in $generalizations
      where $generalization/properties[@ea_type="Generalization"]
        and $generalization/source[@xmi:idref=$element/@xmi:id]
      let $generalizationName
        := $allElements[@xmi:id = $generalization/target/@xmi:idref]
      let $generalizationIRI
        := "http://aixm.aero/" || "#" || $generalizationName/@name/string()
      return
        <rdfs:subClassOf rdf:resource="{$generalizationIRI}"/>
    }
    </rdfs:Class>
};
```

Figure 52 - Schema mapping tool - `rdfs:extractUMLElements`

The function `rdfs:extractUMLAttributes` performs the mapping of the UML attributes (see Figure 53). The function iterates over each element and creates a `rdfs:Property` element for each attribute and for each attribute, the class the attribute belongs to and the datatype are determined. The class is mapped as the `rdfs:domain` and the datatype as the `rdfs:range`. This information is assembled within the return string.

```
declare function rdfs:extractUMLAttributes(
  $allClassElements as element()+
) as element()+{
  for $element in $allClassElements
  let $elementIRI := "http://aixm.aero/" || "#" ||$element/@name/string()
  return
      let $attributes := $element/ownedAttribute
      for $attribute in $attributes
      let $attributeIRI := "http://aixm.aero/" ||
                           "#" ||
                           $element/@name/string() ||
                           "-" ||
                           $attribute/@name/string()
      let $elementName
        := $allClassElements[@xmi:id = $attribute/type/@xmi:idref]
      let $rangeIRI
        := "http://aixm.aero/" || "#" || $elementName/@name/string()
      return
        <rdfs:Property rdf:about="{$attributeIRI}">
          <rdfs:domain rdf:resource="{$elementIRI}"/>
          <rdfs:range rdf:resource="{$rangeIRI}"/>
        </rdfs:Property>
};
```

Figure 53 - Schema mapping tool - `rdfs:extractUMLAttributes`

The next function, `rdfs:extractUMLAssociations`, maps the remaining associations (see Figure 54). As it was already described in Section 3.2.1 the associations are restricted to associations only and additionally, the direction of the association must be considered. After these checks and depending on the direction of the association, the return string is compiled.

```
declare function rdfs:extractUMLAssociations(
  $allGeneralizations as element()+,
  $allClassElements as element()+
) as element()+{
  for $element in $allClassElements
  let $elementIRI := "http://aixm.aero/" || "#" ||$element/@name/string()
  return
      let $associations := $allGeneralizations
      for $association in $associations
      where $association/properties[@ea_type="Association"]
        and ($association/target[@xmi:idref=$element/@xmi:id]
          or $association/source[@xmi:idref=$element/@xmi:id])
      let $domainIRI := "http://aixm.aero/" || "#" || $association/source/model/@name
      let $rangeIRI := "http://aixm.aero/" || "#" || $association/target/model/@name
      return
        if($association/properties[@direction="Source -&gt; Destination"]) then
          <rdfs:Property
            rdf:about="{"http://aixm.aero/" || "#" || $association/source/model/@name  || "-" || $association/labels/@mt}">
            <rdfs:domain rdf:resource="{$domainIRI}"/>
            <rdfs:range rdf:resource="{$rangeIRI}"/>
          </rdfs:Property>
        else
          <rdfs:Property
            rdf:about="{"http://aixm.aero/" || "#" || $association/target/model/@name || "-" || $association/labels/@mt}">
            <rdfs:domain rdf:resource="{$rangeIRI}"/>
            <rdfs:range rdf:resource="{$domainIRI}"/>
          </rdfs:Property>
};
```

Figure 54 - Schema mapping tool - `rdfs:extractUMLAssociations`

The last two functions – `rdfs:extractDiagrams` and `rdfs:extractDiagramElements` – are responsible for the mapping of the UML diagrams and the classes that belong to the different diagrams. The first function (see Figure 55) generates for each UML diagram an `rdfs:Class` tag, while the second function (see Figure 56) generates for each class within the diagram a `rdfs:Property` tag, that has the diagram as the domain and the class as the range.

```
declare function rdfs:extractUMLDiagrams(
  $allDiagrams as element()+
) as element()+{
  for $diagram in $allDiagrams
  let $diagramIRI := "http://aixm.aero/" ||
                     "#" ||
                     fn:replace($diagram/properties/@name/string(), " ", "")
  return
    <rdfs:Class rdf:about="{$diagramIRI}"/>
};
```

Figure 55 - Schema mapping tool - `rdfs:extractUMLDiagrams`

```
declare function rdfs:extractUMLDiagramElements(
  $allElements as element()+,
  $allDiagrams as element()+
) as element()+ {
  for $diagram in $allDiagrams
  let $diagramIRI
        := "http://aixm.aero/" || "#" ||fn:replace($diagram/properties/@name/string(), " ", "")
  return
      let $elements := $diagram/elements/element
      for $element in $elements
      let $elementName := $allElements[@xmi:id = $element/@subject]
      let $elementIRI := "http://aixm.aero/" || "#" || $elementName/@name/string()
      return
        <rdfs:Property rdf:about="https://aixm.aero/#hasElement">
          <rdfs:domain rdf:resource="{$diagramIRI}"/>
          <rdfs:range rdf:resource="{$elementIRI}"/>
        </rdfs:Property>
};
```

Figure 56 - Schema mapping tool - `rdfs:extractUMLDiagramElements`

5.1.2 Adaptation of the Tool to Map Other Schemas

Within the schema mapping tool, the following things need to be adapted in order to process or map other schemas, e.g., IWXXM or FIXM, with it.

Call Script

Within the call script only the value of the `db:open` parameter (see Figure 57 yellow marking) needs to be changed to the new database name.

```
declare variable $filePath := "...\RDF-Schema-FULL.rdf";

file:write($filePath, rdfs:extractRDFSchema(db:open("AIXM_5.1.1")/xmi:XMI))
```

Figure 57 - Schema Mapping tool - Adaptions within Module Call file

Processing Module

The processing module file needs to be adapted a little bit more, as this module contains lots of XPath expressions, which all need to be adapted to the new schema that should be processed. This section contains an example section from the processing module to show, what needs to be adapted. Figure 58 highlights the parts that need to be adapted in the main function of the processing module:

- `allElements` = The value of this variable needs to be changed to the new main element tag of the Schema file.
- `allDiagrams` = the value of this variable needs to be changed to the new main element tag that is about the UML class diagrams of the new Schema.
- `allGeneralizations` = The value of this variable needs to be changed to the new main element tag, that contains all the associations of the UML class diagram.

```
declare function rdfs:extractRDFSchema(
  $rdfSchema as element()
) as element()+ {

  let $allElements := $rdfSchema//ownedMember
  let $allClassElements := $allElements[@xmi:type="uml:Class"]
  let $allDiagrams := $rdfSchema//diagram
  let $allGeneralizations := $rdfSchema//connector

  let $elements
        := rdfs:extractUMLElements($allGeneralizations, $allClassElements, $allElements)
  let $attributes
        := rdfs:extractUMLAttributes($allClassElements)
  let $associations
        := rdfs:extractUMLAssociations($allGeneralizations, $allClassElements)
  let $diagrams
        := rdfs:extractUMLDiagrams($allDiagrams)
  let $diagramElements
        := rdfs:extractUMLDiagramElements($allElements, $allDiagrams)
  return
    <rdf:RDF>
      {$elements}
      {$attributes}
      {$associations}
      {$diagrams}
      {$diagramElements}
    </rdf:RDF>
};
```

Figure 58 - Schema Mapping tool - Adaptions within the Processing Module file

5.2 Instance Data Mapping

This section contains all the details that are necessary to understand, work with and adapt the tool that was created to automate the creation process of the RML mapping rules. First, it provides general information and describes the decisions that were made within this thesis regarding the automation tool. Second, it also explains the tool and the adaptions that need to be made to the tool to create RML mapping rules for other instance data.

The tool to automate the creation of the RML mapping rules is also an XQuery module, which too consists of two files: the file that defines the data source and the iterator for the XML-File that should be processed and the module call as well as the file that contains the implemented creation process of the RML mapping rules. Since the AIXM instance data was available in XML, a language for transforming XML documents is required; XQuery is one such language. Although the RMLMapper needs the RML mapping rules in the RDF/Turtle format, XQuery was chosen, because it was easier to generate the RML mapping rules in RDF/XML and convert them to RDF/Turtle, than implement the automated creation process in a different language. Based on the points made above the technology choice to implement the tool for automating the creation process of the RML mapping rules has been XQuery. The reason the tool RMLMapper was chosen within this thesis and not a complete self-implementation of the mapping process was that a functioning mapping tool for instance data already existed. Therefore, to save time and effort, the tool RMLMapper was used.

5.2.1 Tool for Automating the Creation of the RML Mapping Rules

As it was already described above the tool which automates the creation process of the RML mapping rules also consists of two files, the call script, and the processing module. The content, and the workings of both files will be described in this section.

Call Script

This file contains beside the module call, also the declaration of the namespaces. Figure 59 shows the namespaces and imports that are necessary for this tool. The first two imports define on one hand the path to the file that contains the functions to carry out the creations of the RML rules and on the other hand the module “File” which contains the necessary functions to write the resulting mapping rules into a file. The path to the second file is again an absolute path, which points to the exact filing location of the second file. The remaining lines contain the namespaces for RDF, XMI, UML, RDFS, File output, XML Schema and AIXM.

```
import module namespace test="http://www.w3.org/2000/01/rdf-schema#"
        at "...\rule-generator.xqm";

import module namespace file = 'http://expath.org/ns/file';

declare namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#";
declare namespace xmi = "http://schema.omg.org/spec/XMI/2.1";
declare namespace uml = "http://schema.omg.org/spec/UML/2.1";
declare namespace rdfs="http://www.w3.org/2000/01/rdf-schema#";
declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
declare namespace aixm = "http://www.aixm.aero/schema/5.1.1";
```

Figure 59 - RML mapping rules tool - Module call namespaces

After the definition of the imports and the namespaces, the actual module call is carried out. This can be seen in Figure 60. The first line is again the declaration of the variable that contains the path to the output file. The second variable contains the `iriPath`, which is needed for the subject map and the predicate-object maps of the RML mapping rules. The variables `inputFileName` and `iterator` are needed within the logical source of the RML mapping rules. The last line is again the module call within the `file:write` function, which writes the resulting RML mapping rules into the declared output-file.

```
declare variable $outputFilePath := "...\RDF-Rules.xml";
declare variable $iriPath := "http://example.org/AirportHeliport";
declare variable $inputFileName := "airport-heliport-simplified.xml";
declare variable $iterator
        := "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice";

file:write($outputFilePath,
        rdfs:generateRules(xquery:eval('db:open("airport-heliport-simplified")'||$iterator),
                                        $inputFileName, $iterator, $iriPath))
```

Figure 60 - RML mapping rules tool - Module call

Processing Module

The processing module consists of seven functions that will be explained in this section:

- `rdfs:generateTriplesMap`
- `rdfs:generateLogicalSource`
- `rdfs:generateSubjectMap`
- `rdfs:generatePredicateObjectMaps`
- `rdfs:generatePredicateObjectMapsRules`
- `rdfs:generatePredicateObjectMapsRelations`
- `rdfs:generateRecursion`

Before implementing the functions, however, namespaces must be also included in this file. Figure 61 shows the namespaces that must be included, in order for the module to work. The first line defines the namespace of the module, which in this case is the namespaces of RDFS. The remaining lines contain the namespaces for RDF, RR (R2RML), AIXM and RML.

```
module namespace rdfs="http://www.w3.org/2000/01/rdf-schema#";

declare namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#";
declare namespace rr="http://www.w3.org/ns/r2rml#";
declare namespace aixm = "http://www.aixm.aero/schema/5.1.1";
declare namespace rml ="http://semweb.mmlab.be/ns/rml#";
```

Figure 61 - RML mapping rules tool - Namespaces of Processing Module

After the namespace declaration the first and main function `rdfs:generateTriplesMap` is implemented as it is shown in Figure 62. Within this function body the remaining functions are called, and the results are saved within the defined variables. After the function calls, the return string is put together, which is composed of the name of the triples map, as well as the logical source, the subject map and the predicate-object maps.

```
declare function rdfs:generateTriplesMap(
  $rdfSchema as item()+,
  $inputFileName as xs:string,
  $iterator as xs:string,
  $iriPath as xs:string
) as element()+ {
  let $logicalSource
        := rdfs:generateLogicalSource($inputFileName, $iterator)
  let $subjectMaps
        := rdfs:generateSubjectMap($inputFileName, $iriPath)
  let $predicateObjectMaps
        := rdfs:generatePredicateObjectMaps($inputFileName, $iterator, $iriPath)
  let $predicateObjectMapRules
        := rdfs:generatePredicateObjectMapsRules($rdfSchema)
  let $predicateObjectMapRelations
        := rdfs:generatePredicateObjectMapsRelations($rdfSchema, $inputFileName, $iriPath, $iterator)
  return
    <rdf:RDF>
      <rr:TriplesMap rdf:about="http://example.com/ns#TriplesMapAirportHeliport">
        {$logicalSource}
        {$subjectMaps}
        {$predicateObjectMaps}
        {$predicateObjectMapRules}
        {$predicateObjectMapRelations}
      </rr:TriplesMap>
    </rdf:RDF>
};
```

Figure 62 - RML mapping rules tool - `rdfs:generateTriplesMap`

The next function within the RML mapping rules creation tool is the function `rdfs:generateLogicalSource`, which generates the logical source of the triples map. The contents, as it was already explained in Section 2.4.2, are:

- iterator,
- reference formulation and
- data source

As it is shown in Figure 63, the function gets the variables `inputFileName` and `iterator` passed from `rdfs:generateTriplesMap`, which were already defined in the call script file. In the function body the return string is composed of the `data source` (instance data file), the `reference formulation` (XPath) and the `iterator`.

```
declare function rdfs:generateLogicalSource(
  $inputFileName as xs:string,
  $iterator as xs:string
) as element()+{
  let $i := $iterator
  return
    <rml:logicalSource rdf:parseType="Resource">
        <rml:source>{$inputFileName}</rml:source>
        <rml:referenceFormulation rdf:resource="http://semweb.mmlab.be/ns/ql#XPath"/>
        <rml:iterator>{$iterator}</rml:iterator>
    </rml:logicalSource>
```

Figure 63 - RML mapping rules tool - `rdfs:generateLogicalSource`

Within the function `rdfs:generateSubjectMap`, the triples map's subject map is generated, which corresponds to the subject of an RDF triple. As it is shown in Figure 64, the function body contains the return string which contains the IRI path and in this case the unique ID of the respective `AirportHeliport`.

```
declare function rdfs:generateSubjectMap(
  $inputFileName as xs:string,
  $iriPath as xs:string
) as element()+{
  let $path := $inputFileName
  return
    <rr:subjectMap rdf:parseType="Resource">
        <rr:template>
                {concat($iriPath, "/{@id}")}
        </rr:template>
    </rr:subjectMap>
};
```

Figure 64 - RML mapping rules tool - `rdfs:generateSubjectMap`

After the generation of the subject map, the predicate-object maps are created. This happens within the functions:

- `rdfs:generatePredicateObjectMap`
- `rdfs:generatePredicateObjectMapRules`
- `rdfs:generatePredicateObjectMapRelations`
- `rdfs:generateRecursion`

In the first function the predicate-object map is generated, which annotates the predicate and objects to the class (see Section 4.1). Figure 65 shows the function `rdfs:generatePredicateObjectMap` and the composed return string that refers to the class which should annotate the predicates and objects.

```
declare function rdfs:generatePredicateObjectMaps(
  $inputFileName as xs:string,
  $iterator as xs:string,
  $iriPath as xs:string
) as element()+{
  let $i := $inputFileName
  return
  <rr:predicateObjectMap rdf:parseType="Resource">
    <rr:predicate
        rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"/>
    <rr:objectMap rdf:parseType="Resource">
      <rr:constant
        rdf:resource="http://www.aixm.aero/schema/5.1.1/AirportHeliport"/>
    </rr:objectMap>
  </rr:predicateObjectMap>
};
```

Figure 65 - RML mapping rules tool - `rdfs:generatePredicateObjectMaps`

The function `rdfs:generatePredicateObjectMapRules` generates the predicates and objects for all the sub-elements within the main element. This function annotates the values of the objects with their predicates. As it can be seen in Figure 66 there is a check within the return string. The reason behind this check is, to filter out the elements with further sub-elements, because those represent the associations of that main element.

```
declare function rdfs:generatePredicateObjectMapsRules(
  $rdfSchema as item()+
) as element()+{
    let $source := $rdfSchema[1]
      let $labels := $source/*
      for $label in $labels
        let $children := $label/*
        let $resource := concat("http://www.aixm.aero/schema/5.1.1/",
                                $label/name())
        return
          if(fn:not(fn:exists($children))) then
          <rr:predicateObjectMap rdf:parseType="Resource">
            <rr:predicate rdf:resource="{$resource}"/>
            <rr:objectMap rdf:parseType="Resource">
              <rml:reference>{$label/name()}</rml:reference>
            </rr:objectMap>
          </rr:predicateObjectMap>
};
```

Figure 66 - RML mapping rules tool - `rdfs:generatePredicateObjectMapsRules`

In order to map associations, the predicate-object maps must be generated with a join. Within the tool, the generation of the RML mapping rules for the association happens within the functions `rdfs:generatePredicateObjectMapRelations` and `rdfs:generateRecursion`. Figure 67 shows the function call for the recursive function that creates the predicate-object maps for the associations. The recursive function is only called when the element the function is currently looking at has sub-elements.

```
declare function rdfs:generatePredicateObjectMapsRelations(
  $rdfSchema as item()+,
  $inputFileName as xs:string,
  $iriPath as xs:string,
  $iterator as xs:string
) as element ()+{
  let $source := $rdfSchema[1]/*
  let $labels := $source/*
  for $label in $labels
  return
    if (fn:exists($label)) then
      rdfs:generateRecursion($label, $iterator, $inputFileName, $iriPath)
};
```

Figure 67 - RML mapping rules tool - `rdfs:generatePredicateObjectMapsRelations`

Figure 68 shows the function `rdfs:generateRecursion`. The first part in the function body contains the recursive exit and thus the generation of the predicate-object map. The second part of the function body is the recursive descent, which means the function is called until the element the function is currently looking at has no more sub-elements.

```
declare function rdfs:generateRecursion(
  $label as element()+,
  $iterator as xs:string,
  $inputFileName as xs:string,
  $iriPath as xs:string
) as element()+{
  let $children := $label/*
  return
  (
    if(fn:not(fn:exists($children/*)))then
      let $resource := concat("http://www.aixm.aero/schema/5.1.1/", $label/parent::*/name())
      let $iteratorRelation := concat($iterator, "//", $label/name())
      let $parent := "@id"
      let $child := concat($label/parent::*/name(), "/", $label/name(), "/", "@id")
      return
        <rr:predicateObjectMap rdf:parseType="Resource">
          <rr:predicate rdf:resource="{$resource}"/>
          <rr:objectMap rdf:parseType="Resource">
            <rr:parentTriplesMap>
              <rdf:Description
                rdf:about="{concat("http://example.com/ns#TriplesMap", $label/name(), "-", $label/@id)}">
                <rml:logicalSource rdf:parseType="Resource">
                  <rml:iterator>{$iteratorRelation}</rml:iterator>
                  <rml:referenceFormulation rdf:resource="http://semweb.mmlab.be/ns/ql#XPath"/>
                  <rml:source>{$inputFileName}</rml:source>
                </rml:logicalSource>
                <rr:subjectMap rdf:parseType="Resource">
                  <rr:template>{concat($iriPath, "/", $label/name(), "/{@id}")}</rr:template>
                </rr:subjectMap>
                {
                  let $children := $label/*
                  for $child in $children
                    let $childResource := concat("http://www.aixm.aero/schema/5.1.1/", $child/name())
                    return
                     <rr:predicateObjectMap rdf:parseType="Resource">
                       <rr:predicate rdf:resource="{$childResource}"/>
                       <rr:objectMap rdf:parseType="Resource">
                         <rml:reference>{$child/name()}</rml:reference>
                       </rr:objectMap>
                     </rr:predicateObjectMap>
                }
              </rdf:Description>
            </rr:parentTriplesMap>
            <rr:joinCondition rdf:parseType="Resource">
              <rr:parent>{$parent}</rr:parent>
              <rr:child>{$child}</rr:child>
            </rr:joinCondition>
          </rr:objectMap>
        </rr:predicateObjectMap>
    else
      rdfs:generateRecursion($children, $iterator, $inputFileName, $iriPath)
    )
};
```

Figure 68 - RML mapping rules tool - `rdfs:generateRecursion`

5.2.2 Adaption of the Tool to Create RML Mapping Rules for Other Instance Data

Within the tool for the creation of the RML mapping rules, the following things need to be adapted, in order to process or map other schemas with it.

Call Script

To generate RML mapping rules for different instance data, the following parts need to be adapted within the module call file:

- `iriPath` = The last part of the value of this variable needs to be changed to the main element of the new instance data file (see Section 4.2.1 for details).
- `inputFileName` = The value of this variable needs to be changed to the file that contains the new instance data, for which the RML mapping rules are to be created.
- `iterator` = The value of this variable needs to be changed to the element which contains all the sub-elements with all the data to the main element (see Section 4.2.1 for details).
- `db:open` = The value of this function value needs to be changed to the new database name.

Figure 69 highlights the necessary adaptions that were described above.

```
declare variable $outputFilePath := "...\RDF-Rules.xml";
declare variable $iriPath := "http://example.org/AirportHeliport";
declare variable $inputFileName := "airport-heliport-simplified.xml";
declare variable $iterator
        := "/AIXMBasicMessage//AirportHeliport//AirportHeliportTimeSlice";

file:write($outputFilePath,
        rdfs:generateRules(xquery:eval('db:open("airport-heliport-simplified")'||$iterator),
                                        $inputFileName, $iterator, $iriPath))
```

Figure 69 - RML mapping rules - Adaptions within Module Call file

Processing Module

Within the processing module file nothing needs to be adapted, as this module already works generically.

6 Conclusion

This thesis may serve as the basis for further development regarding the transformation and mapping procedures for representing AIXM and other exchange models using RDF(S). The goal of the thesis was to map both AIXM schema and instance data to RDF(S). Therefore, two XQuery modules were created: (i) an XQuery module that transforms the XMI of the AIXM UML class diagram to RDFS, and (ii) an XQuery module that generates the RML mapping rules for a given set of AIXM instance data. Since little information was available in the literature on this subject, the automation of the procedures was done from scratch. Only for the transformation of the instance data, the RMLMapper was used to generate RDF statements from the automatically created RML mapping rules. In addition, the thesis also shows how to adapt the automation tools so that the procedures can be used for other types of aeronautical input data. The implementation of both tools is not generic but specific to AIXM. A goal of the implementation, however, was to design the tools in a way that allows easy adaptation to other aeronautical schemas or instance data. The mapping of the AIXM Temporality Model was not within the scope of this thesis. Future work may investigate the mapping of the AIXM Temporality Model to RDF(S), e.g., by using AIXM time slices as the basis for representation of contextualized knowledge graphs [27].

7 List of Figures

8 List of Tables

9 Bibliography

[1] M. Arblaster, ‘The Air Traffic Management Industry’, in *Air Traffic Management*, Elsevier Inc., 2018, pp. 1–8.

[2] M. Arblaster, ‘Operational and Technological Background on Air Traffic Management’, in *Air Traffic Management*, Elsevier Inc., 2018, pp. 11–38.

[3] N. A. Stanton and J. Piggott, ‘Situational awareness and safety’, vol. 7535, no. December 2001, pp. 189–204, 2017.

[4] AIXM, ‘AIXM’. [Online]. Available: http://www.aixm.aero/. [Accessed: 03-Jun-2021].

[5] B. Brunk and E. Prorsnicu, ‘Aeronautical Information Exchange Model (AIXM) Exchange Model goals, requirements and design’, p. 76, 2005.

[6] AIXM, ‘AIXM Model Documentation’. [Online]. Available: http://aixm.aero/sites/aixm.aero/files/imce/AIXM511HTML/index.html. [Accessed: 20-May-2021].

[7] A. I. S. Ituational, A. W. F. Oundation, F. O. R. A. Dvancing, and A. U. Aisa, ‘AISA Proposal – Part B’, vol. 1, 2020.

[8] RMLio, ‘RMLMapper’. [Online]. Available: https://rml.io/. [Accessed: 30-May-2021].

[9] Apache, ‘Apache Jena’. [Online]. Available: https://jena.apache.org/index.html. [Accessed: 30-May-2021].

[10] B. Murphy and E. Porosnico, ‘AIXM 5.1.1 Temporality Model’, pp. 1–30, 2019.

[11] OGC, ‘OGC® Geography Markup Language (GML) — Extended schemas and encoding rules’, *OpenGIS Recomm. Pap.*, p. 595, 2010.

[12] D. Burggraf, M. Trninic, R. Lake, and L. Rae, *Geography Mark-Up Language : Foundation for the Geo-Web*, 1st ed. Hoboken: John Wiley & Sons, 2004.

[13] J. Osis and U. Donins, ‘Unified Modeling Language: A Standard for Designing a Software’, in *Topological UML Modeling*, Elsevier, 2017, p. 335.

[14] D. Pilone and N. Pitman, *UML 2.0 in a Nutshell*. O’Reilly Media, Inc., 2005.

[15] Object Managment Group, ‘Information Technology --- XML Metadata Interchange (XMI)’, *XML Metadata Interchange (XMI) Specification*, vol. 2014, no. April. 2011.

[16] K. . Breitmann, M. A. Casanova, and W. Truskowski, *Semantic Web: Concepts, Technologies and Applications*. London: Springer-Verlag London Limited, 2007.

[17] S. Decker, P. Mitra, and S. Melnik, ‘Framework for the semantic web: an RDF tutorial’, *IEEE Internet Comput.*, vol. 4, no. 6, pp. 68–73, 2000.

[18] W3C, ‘RDF’. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/. [Accessed: 26-Apr-2021].

[19] W3C, ‘RDF 1.1 Turtle’. [Online]. Available: https://www.w3.org/TR/turtle/. [Accessed: 26-Apr-2021].

[20] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van De Walle, ‘RML: A generic language for integrated RDF mappings of heterogeneous data’, *CEUR Workshop Proc.*, vol. 1184, 2014.

[21] K. Kyzirakos *et al.*, ‘GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings’, *J. Web Semant.*, vol. 52–53, pp. 16–32, 2018.

[22] W3C, ‘XML’. [Online]. Available: https://www.w3.org/TR/xml/. [Accessed: 30-May-2021].

[23] W3C, ‘XQuery’. [Online]. Available: https://www.w3.org/TR/xquery-31/. [Accessed: 30-May-2021].

[24] G. Goos, J. Hartmanis, and J. Leeuwen, *Lecture Notes in Computer Science*, vol. 1716.

1999.

[25] J. L. Filho and J. L. Braga, ‘UML: Unified Modeling Language’, in *Encyclopedia of GIS*, no. December, 2017, pp. 2345–2346.

[26] Q. Tong, F. Zhang, and J. Cheng, ‘Construction of RDF(S) from UML class diagrams’, *J. Comput. Inf. Technol.*, vol. 22, no. 4, pp. 237–250, 2015.

[27] C. G. Schuetz, B. Neumayr, M. Schrefl, E. Gringinger, A. Vennesland, and S. Wilson, ‘The case for contextualized knowledge graphs in air traffic management’, *CEUR Workshop Proc.*, vol. 2317, 2018.