TU WIEN Informatics

Schemagestütztes RDF-zu-Prolog Mapping für das AISA Projekt

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Business Informatics

eingereicht von

Marlene Hartmann, BSc.
Matrikelnummer 01426911

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung:
Univ.-Prof. Mag. Dr. Manuel Wimmer
Mag. Dr. Bernd Neumayr

Wien, 1. Juli 2022

Marlene Hartmann	Manuel Wimmer

TU WIEN Informatics

Schema-Aware RDF-to-Prolog Mapping for the AISA Project

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Marlene Hartmann, BSc.
Registration Number 01426911

to the Faculty of Informatics

at the TU Wien

Advisor:
Univ.-Prof. Mag. Dr. Manuel Wimmer
Mag. Dr. Bernd Neumayr

Vienna, 1st July, 2022

Marlene Hartmann | Manuel Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Marlene Hartmann, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juli 2022

Marlene Hartmann

Danksagung

Zuallererst möchte ich o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl dafür danken, dass er es mir ermöglicht hat beim AISA Projekt mitzuarbeiten. Ich danke auch Univ.-Prof. Mag. Dr. Manuel Wimmer die Betreuung seitens der TU Wien. Ein weiteres Dankeschön geht an Mag. Dr. Bernd Neumayr, der mich vom Anfang bis zum Ende des Projekts und meiner Thesis unterstützt hat. Dankbar bin ich vor allem für all die Freundschaften, die ich durch mein Studium gewonnen habe, und die allgegenwärtige gegenseitige Unterstützung während des ganzen Studiums. Besonders möchte ich meinen Freunden und meiner Familie danken, vor allem meiner Mutter Sabine Hartmann und meiner Großmutter Gertrude Muhr, die mir regelmäßig den Rücken gestärkt haben.

Acknowledgements

First of all I would like to thank o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl for giving me the opportunity to work on the AISA project. I also thank Univ.-Prof. Mag. Dr. Manuel Wimmer for supervising my thesis at the Vienna University of Technology. Another thank you goes to Mag. Dr. Bernd Neumayr, who supported me from the beginning to the end of the project and my thesis. Above all, I am grateful for all the friendships I have gained through my studies and the ubiquitous mutual support throughout my studies. I would especially like to thank my friends and family, especially my mother Sabine Hartmann and my grandmother Gertrude Muhr, who have consistently supported me.

Kurzfassung

Die Resource Description Framework (RDF)-Datenbank von SWI-Prolog ermöglicht einen schemaunabhängigen Ansatz für das Knowledge Graph (KG)-Prolog-Mapping, der für den einfachen KG-Zugriff und für das Schreiben von Fakten in den KG aus Prolog gut geeignet ist. Allerdings ist dieser Ansatz für Prolog-Programmierer unhandlich, wenn es darum geht, komplexe KG-Daten zu lesen. Das Ziel dieser Arbeit ist es, verschiedene Optionen zu untersuchen, um ein RDF-Prolog Mapping und einen Datenaustausch für das "AI Situational Awareness Foundation for Advancing Automation" (AISA) Projekt bereitzustellen, die die Inhalte des KG in einer für Prolog-Programmierer zugänglichen Form und entsprechend dem KG Schema bereitstellt. Dies bedeutet, dass im Vergleich zur schemalosen Abbildung viel weniger Fakten zur Beschreibung des Inhalts benötigt werden. Wir haben drei Varianten des schemagestützten RDF-Prolog Mappers implementiert und die Performance verglichen. Schließlich, zeigen wir eine Integration von der Prolog-Engine und dem AISA KG-System für den schemalosen Ansatz zusammen mit einer Variante des schemagestützten Ansatzes.

Abstract

The Resource Description Framework (RDF) database of SWI-Prolog facilitates a schema-oblivious approach to Knowledge Graph (KG)-Prolog mapping, which is well-suited for simple KG access and for writing facts to the KG from Prolog. However, this approach is unwieldly to use for Prolog programmers when it comes to reading complex KG data. The aim of this thesis is to investigate different options for providing an RDF-Prolog mapping and data interchange for the "AI Situational Awareness Foundation for Advancing Automation" (AISA) project that provides the contents of the KG in a form amenable to Prolog programmers and according to the KG schema. This means that significantly fewer facts are needed to describe content in comparison to the schema-oblivious mapping. We implemented three variants of the schema-aware RDF-Prolog mapper and compared the performance results. We provide a full integration of Prolog engine and AISA KG system for the schema-oblivious approach together with one variant of the schema-aware approach.

Foreword

This master thesis as been conducted in the context of the research of the EU Horizon 2020 project AISA ("Artificial Intelligence Situational Awareness"). My participation in this research project from April 2021 until September 2021 put me in an international software development team with partners from Germany, Hungary, Croatia, Spain, and Austria in the area of aeronautical information management. Not only the novel software for data mappings and interchanges based on semantic technology was created during this time, but also several parts and figures of this thesis were already published in the respective Deliverable 4.2[1] [NH21] of the AISA project (these parts are not marked separately in this thesis as I was the main author of this deliverable). This prototype is integrated into the AISA KG system, which was published in the Deliverable 4.1[2] [Neu21]. The code is published on Zenodo [HN23].

[1] https://aisa-project.eu/downloads/AISA%20D4.2.pdf
[2] https://aisa-project.eu/downloads/AISA_4.1.pdf

Contents

CHAPTER 1

Introduction

Section 1.1 of this chapter describes the context of this thesis and how it is embedded into the AISA project, followed by the problem statement in Section 1.2. Section 1.3 summarizes the contribution of this work. Section 1.4 presents the methodology and Section 1.5 the structure of this thesis.

1.1 Context

The objective of this thesis is to improve accessing the Knowledge Graph (KG) by creating different variants of a schema-aware RDF-Prolog mapper (also referred to as "KG-Prolog Mapper") for the AISA project. The EU Horizon 2020 Project AISA ("AI Situational Awareness Foundation for Advancing Automation") developed concepts and techniques such that "AI will enable the automated system to reach the same conclusions as air traffic control officers when confronted with the same problem" [ais]. Knowledge about aeronautical information such as flight plans, weather data, aircraft positions, flight trajectory predictions continuously gained by several sources including machine learning modules are maintained in an RDF-based KG, the schema of which is specified in RDFS/SHACL. For each part of the Knowledge Graph, named graphs are used for meta-data in order to capture when and from which component it was written. RDF stands for Resource Description Framework and is a data model and family of data serialization formats for information modeling and information exchange on the web [CHWL07a]. RDFS stands for Resource Description Framework Schema and is a semantic extension of RDF, providing a data-modelling vocabulary for RDF data [CHWL07b]. The SHACL Shapes Constraint Language is a language for specifying integrity constraints over RDF graphs and can be used, among others, to constrain the number of values that a property may have, the type of such values, numeric ranges, string matching patterns, and logical combinations of such constraints [KK17]. A Knowledge Graph is a graph-structured data set representing factual knowledge [Jam92]. SPARQL ("SPARQL Protocol and

RDF Query Language") is an RDF query language [PHS13]. Situational awareness is achieved through repeated SPARQL queries of the Knowledge Graph at short intervals. For queries that require more complex reasoning than SPARQL can conveniently provide, the AISA architecture foresees the use of Prolog. There are, for example, tasks defined in Deliverable 4.4[1] [TRT21], such as checking that an aircraft is climbing/descending towards cleared flight level, that require more than one SPARQL query since the previous and the current flight level is required for reasoning.

Figure 1.1 shows the architecture of the AISA project. Aeronautical data in the form of XML and JSON, which conform to their conceptual schema building on the Aeronautical Information Exchange Model (AIXM) [aix] and the Flight Information Exchange Model (FIXM) [fix] defined in UML, is mapped to RDFS/SHACL. The blue marked part of the figure shows the subject of this thesis and is integrated into the AISA KG system, which was published in Deliverable 4.1[2] [Neu21] of the AISA projects. This part consists of generating SPARQL based Prolog predicates from SHACL shapes and reading from and writing to the KG via predicates associated with SPARQL queries and updates.

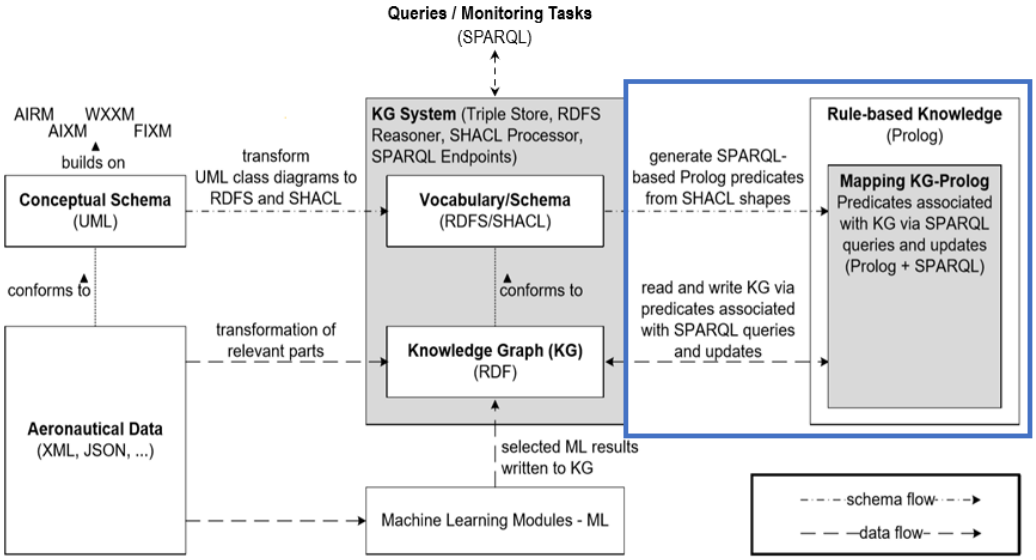


Figure 1.1: A schematic diagram showing the architecture of the AISA System (adapted from "WP 4 - AI Situational Awareness Syste" by B. Neumayr, June 2022, unpublished slide deck, presented as part of the AISA Final Review Meeting).

The AISA KG, which was already created by project partners, is a RDF dataset holding all the static and dynamic data and metadata relevant for AI situational awareness. The AISA KG is stored on a KG server and queried and updated via SPARQL. The

[1] https://aisa-project.eu/downloads/AISA_4.4.pdf
[2] https://aisa-project.eu/downloads/AISA_4.1.pdf

KG schema is specified in RDF Schema and SHACL. Data and metadata are added dynamically to the KG and processed and queried via application-specific engines mainly implemented in Java. A central control component implemented in Java is responsible for recurring invocation of the different engines. Advanced reasoning tasks over the KG are to be realized based on rule-based knowledge represented in Prolog.

1.2 Problem Statement

Existing solutions for mapping RDF to Prolog include schema-oblivious mapping such as provided from SWI-Prolog [swi]. The schema-oblivious approach can be realized easily but is unwieldy for Prolog programmers when it comes to reading complex KG data. In contrast, the intended solution is schema-aware, because the KG is highly structured with structural schemata in the form of SHACL shape graphs for all parts of the Knowledge Graph. Furthermore, it is assumed by the AISA project partners that preserving these schemata in the mapped Prolog facts is facilitating the development and maintenance of Prolog programmers.

The basic idea of the schema-aware mapping to be developed and investigated in this thesis is that each RDF node is mapped with its properties exactly into one Prolog fact. This naturally preserves in Prolog the schema to which this RDF node is subject to. Every RDFS class and corresponding SHACL node shape becomes a Prolog predicate. Every single-valued property becomes a single-valued argument of the Prolog predicate, potentially a null value. Every multi-valued property becomes a list-valued argument of the Prolog predicate, potentially empty. Building on this basic idea different mapping options also considering sub-classing and complex values in the KG will be explored.

The problem statement can be divided into 2 design problems:

The first specific design problem is to facilitate reading complex schema-conformant data in a KG from Prolog by designing a schema-aware KG-Prolog mapper that provides the contents of the KG in a form amenable to Prolog programmers and according to the KG schema. Thus, the purpose of this thesis is to improve accessing the KG from Prolog in the AISA project by designing a schema-aware RDF-Prolog mapper that takes care of data interchange and mapping between Prolog engine and KG, so that Prolog programmers can easily develop Prolog programs, which read from and write to the AISA KG.

The second specific design problem is how to integrate the schema-oblivious approach and the schema-aware approach with the AISA KG system and how to integrate Prolog programs into the KG manager. With its KG modules and central control component, advanced reasoning tasks in AISA that are invoked recurrently can easily be realized as Prolog programs, which read from and write to the AISA KG.

1.3 Contribution

The aim of the thesis is to investigate different options for providing an RDF-Prolog mapping and data interchange for AISA. As a result of our analysis of the possibilities of SWI-Prolog, we now see several realization variants for schema-aware read access of the KG which we will discuss in Chapter 3. For this purpose, a prototypical implementation of the schema-aware RDF-Prolog mapping is created. We implement the schema-aware approach in three different variants and conduct preliminary performance studies for comparison. This prototype is used by project partners in the AISA project for simulation experiments validating the concept of Artificial Intelligence Situational Awareness in the context of air traffic control. Based on the continuous feedback from actual use in the project, the prototypical implementation is subject to improvements. We provide a full integration of Prolog engine and AISA KG system for the schema-oblivious approach together with one variant of the schema-aware approach. The prototype is instrumental in reaching the following knowledge goal and answering the following knowledge question:

The knowledge goal is to understand the performance characteristics of different alternative approaches to mapping execution and data exchange. These variants do not affect the logical mapping and should not make changes necessary, apart from changing configuration options, to the Prolog programs developed by Prolog programmers in the AISA project. The corresponding knowledge question is: How do the performance characteristics of the different approaches to mapping execution and data exchange differ depending on varying data input sizes?

The RDF database of SWI-Prolog facilitates a schema-oblivious approach to KG-Prolog mapping which is well-suited for simple KG access and for writing facts to the KG from Prolog. For reading complex and structured data from the KG we investigate the schema-aware approach to KG-Prolog mapping.

We see the role of the schema-aware mapping primarily for reading the schema-aware parts of the KG, rather than as the sole means of accessing the KG. Especially for writing to the KG from Prolog, the schema-oblivious approach via RDF DB seems simpler and more flexible. We therefore focus our work on the schema-aware approach on read access of the KG.

1.4 Methodology

Wieringa [Wie14] describes design science as designing and investigating an artifact within a certain context with the goal to improve something in that context. Figure 1.2 shows a framework for design science. The social context consists of stakeholders affecting or affected by the project, possible users, operators, maintainers, instructors and sponsors. In our case, the stakeholders are our project partners from AISA, consisting of the "Faculty of Transport and Traffic Sciences at University of Zagreb" [unia], the "Johannes Kepler University Linz / Institute of Business Informatics" [jku], "Slot Consulting Ltd." [slo], the "Technische Universität Braunschweig, Institute of Flight Guidance" [tub],

the “Universidad Politécnica de Madrid” [unib], “Zurich University of Applied Sciences (ZHAW), School of Engineering” [unic] and “Skyguide Swiss Air Navigation Services Ltd” [sky], and the sponsor of the project, which is “SESAR Joint Undertaking” [ses]. The artefact of this thesis is the KG-Prolog mapper. Design and investigation corresponds to design problems and knowledge questions. Design problems ask for a change in the real world and knowledge questions ask for the knowledge about the world as it is. Our design problems are facilitating reading of complex schema-conformant data in a KG from Prolog by designing a schema-aware KG-Prolog mapper and its integration into the AISA KG system. Our knowledge question deals with the differences of the mapping approaches regarding performance and scalability. Design problems are treated by the design cycle, which iterates over three tasks: problem investigation, treatment design and treatment validation [Wie14]. Knowledge questions are treated by the empirical cycle, which iterates over the following tasks: Research problem analysis, research design and inference design, validation of research and inference design, research execution and data analysis [Wie14]. The knowledge context includes existing theories, knowledge, designs and products from science and engineering. This knowledge is used to answer knowledge questions and create new designs or improve existing solutions.

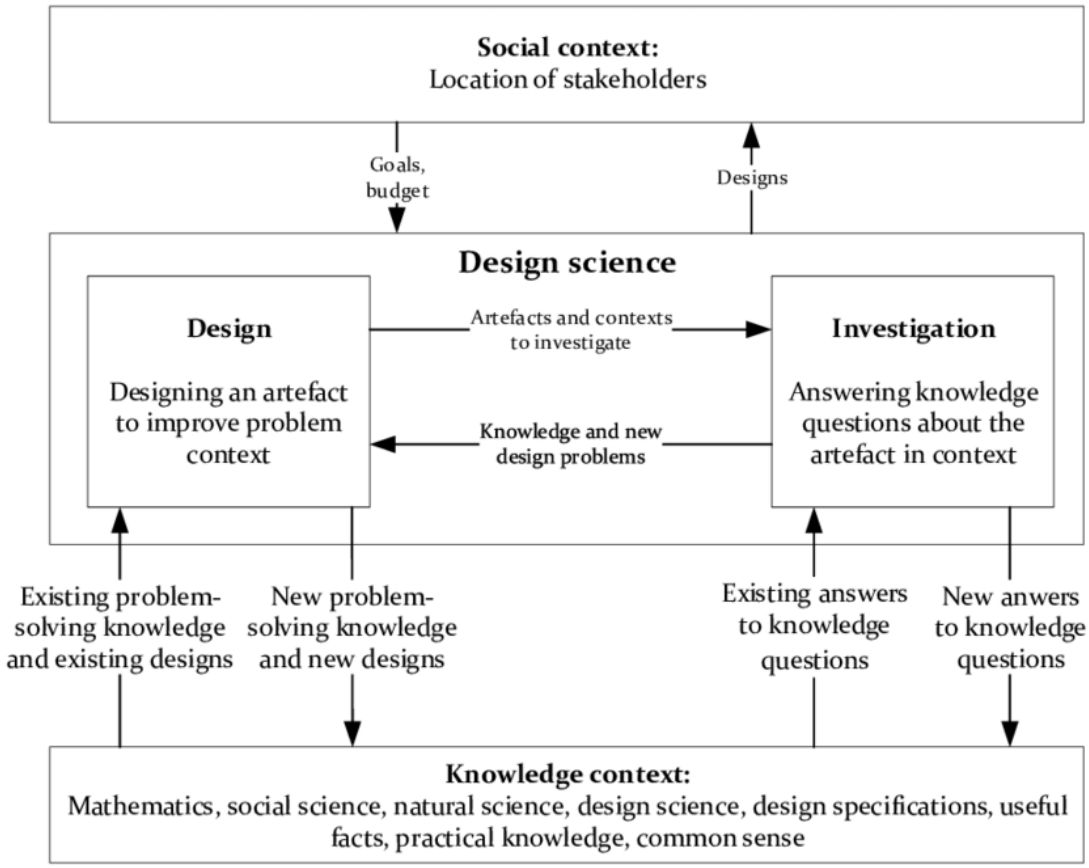


Figure 1.2: A framework for design science [Wie14]

The methodological approach of this thesis consists of the following steps:

1. Requirements Analysis
   First of all, we had to collect requirements given by the AISA project. Next to functional requirements that are fundamental for the functionality, which is mainly described in Chapter 3, the most important non-functional requirement for this project is the performance. As we want to come to the same conclusion as air traffic controllers when given flight data, the result must be provided within seconds.

2. Literature Review
   In the course of this thesis a literature review of the underlying technologies and similar problems was conducted to understand if and why a new approach for the project is necessary.

3. Implementation
   The next step is implementing prototypes for experimenting, which requires a design and draft phase for detailed specification. For this purpose, we implemented 3 different approaches:

   A) The first option consists of a Java program, which generates with the help of SPARQL a set of Prolog predicates from the RDFS/SHACL schema and a SPARQL query for each predicate. The result of executing the SPARQL query gives the facts for the respective predicate. These facts are written to a file and that file is loaded into Prolog.

   B) The Java program generates a Prolog module from the RDFS/SHACL schema with the predicates that are linked to the respective SPARQL queries by means of Prolog rules. The SPARQL queries for filling the predicates are only executed from Prolog at runtime.

   C) The relevant part of the Knowledge Graph is replicated in the main-memory RDF database of SWI-Prolog. The actual mapping is then formulated as Prolog rules with the RDF quadruples in the RDF database as input.

4. Evaluation
   A Prototype is created, which provides different approaches to mapping execution and data interchange. For the purpose of testing the implementations, test data sets including real flight data from Opensky Network[3] [ope] is created and adapted to cover all requirements like for example the handling of data types and missing values. Via small Prolog tests, it is ensured that all 3 implementation variants deliver the same results. Subsequently, performance benchmarking and evaluation of the various implementation options is conducted, especially regarding run time and scalability. Based on the results of these performance measurements, one implementation variant is chosen for integration with the Proof-of-Concept KG

[3] https://opensky-network.org/

system. The evaluation for the various mapping execution and data interchange options is only based on usage scenarios and on performance tests of test data rather than on real usage scenarios which are developed by project partners. The later use of the experimental prototypes by other project partners will confirm or disconfirm the results derived from the evaluation. Single-thread use of the Knowledge Graph and Prolog can be assumed for the experimental prototype. Data interchange need not be live for time-varying data (e.g. updated aircraft positions or trajectory predictions) but can be invoked at fixed or varying time intervals.

1.5 Structure of this Thesis

This thesis is divided into 8 chapters. Chapter 2 introduces the used languages and general definitions regarding mapping and model transformations, explains used technology and software and describes the differences between schema-oblivious and schema-aware mapping. Chapter 3 explains the realization of the schema-aware approach, how the Prolog schema is generated from RDFS/SHACL and presents the 3 mapping variants in detail. Chapter 4 describes how data types and missing values are handled. Chapter 5 provides a technical documentation and a more detailed look at the implementation. Chapter 6 presents the results of the performance studies, the integration of one of the variants into the KG system and the overall results. Chapter 7 gives an overview of existing research in the field of schema-aware and schema-oblivious mapping and transformation approaches. Finally, Chapter 8 shows the findings of this thesis and discusses future work.

CHAPTER 2

Background

This chapter discusses the basics of Prolog and RDF/RDFS/SHACL in Section 2.1. Section 2.2 shows the general definition of mapping and model transformation. Section 2.3 presents relevant concepts of Prolog, Prolog software packages and additional technologies, which are used in this project. Section 2.4 discusses the two generic approaches for accessing the KG from Prolog, the schema-oblivious and the schema-aware mapping approach.

2.1 Languages

The aim of this thesis is to investigate different options for providing an RDF-Prolog mapper. This section discusses the basics of Prolog, RDF/RDFS, and SHACL. The usage of each of the basics is shown by a trivial example.

RDF is a World Wide Web Consortium (W3C) standard used for the description and exchange of graph data, providing an abstract syntax with two key data structures: RDF graphs, which are a set of subject-predicate-object triples that may consist of elements, which are IRIs, blank nodes and datatyped literals, and RDF datasets, which are collection of RDF graphs comprising default graphs and zero or more named graphs [CHWL07a]. When we talk about RDF quadruples, we refer RDF terms with a subject, a predicate, an object and a graph label.

The following is an example of RDF:

```
% Subject, Predicate, Object
<http://www.dbpedia.org/resource/Austria>
↪  <http://www.example.org/commonBorder>
↪  <http://www.dbpedia.org/resource/Switzerland> .
```

Turtle[1] is a textual syntax for RDF used to describe it in a compact and natural text form [DBC14].

The following is an example of RDF in Turtle:

```
@prefix ex:  <http://example.com/ns#> .
@prefix dbp: <http://www.dbpedia.org/resource/> .

% Austria has a common border to Switzerland
dbp:austria
    ex:commonBorder dbp:switzerland .
```

RDFS is a W3C standard that provides a data-modelling vocabulary for RDF data [CHWL07b]. RDFS can describe classes, properties and the relationship between those classes and properties. With `rdfs:subClassOf`, for example, it can be stated that one class is the subclass of another. In contrast, `rdfs:subPropertyOf` is used to state that a property is a subproperty of another. However, RDFS does not cover integrity constraints like cardinality, unlike SHACL, which is described in the next paragraph.

The following is an example of RDFS:

```
% Example namespace and namespace for rdfs
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex:  <http://example.com/ns#> .

% A federal state is a subClass of a country
ex:FederalState
    a rdfs:Class ;
    rdfs:subClassOf ex:Country .
```

SHACL is a W3C standard and a language for validating RDF graphs against SHACL shapes graphs, which consist of SHACL shapes that provide a set of conditions, defined in RDF [KK17]. These conditions are integrity constraints, such as cardinality and string based constraints, which validate the string representation of value nodes.

The following is an example of RDFS/SHACL:

```
% Example namespace
@prefix sh:    <http://www.w3.org/ns/shacl#> .
@prefix ex:  <http://example.com/ns#> .

% Country is a class with a property commonBorder of type Country
ex:Country
    a rdfs:Class n sh:NodeShape ;
    sh:property [
                  sh:class  ex:Country
```

[1] https://www.w3.org/TR/turtle/

```
    sh:path    ex:commonBorder
  ] .
```

SPARQL[2] [PS07] is a semantic query language for retrieving and manipulating RDF.

The following is an example of a SPARQL query:

```
# Example namespace
PREFIX  ex: <http://example.org/>

# Query all countries and their common borders
SELECT ?country
       ?commonBorder
WHERE
  { ?country a ex:country .
    ?country ex:commonBorder ?commonBorder .
  }
```

Prolog is a logic programming language [Bra13]. The program logic is expressed as predicates and represented as facts and rules. Computation is achieved through running queries over relations.

A **fact** we refer to a variable-free statement in Prolog consisting of a predicate name and a list of arguments conforming to the schema of the predicate. We distinguish: a *static fact* is hard-coded in a Prolog program and compiled and loaded into the Prolog engine, a *dynamically asserted fact* is added to the Prolog database at run-time, a *virtual fact* is derived by Prolog rules and is not asserted in the DB.

A **predicate** is a relation maintained and made available by the Prolog engine. The schema of a predicate is specified by its name (also referred to as predicate symbol) and its arity, i.e., the number of its arguments, and possibly as comments further information about the type of arguments. The *extension of a predicate* is given by a set of facts. We distinguish: a *static predicate* only has static facts, a *dynamic predicate* may also have dynamically asserted facts, a *derived predicate* additionally has virtual facts. By *method*, we refer to a predicate that is defined by a rule with side-effects.

A **rule** is a clause consisting of none, one or multiple variables, a head, neck and body. The head is the first part of a Prolog rule, which holds the by comma-separated list of parameters. The neck symbol *:-* separates the head from the body. The body has the form of a comma-separated list of predicate calls, which are also called rule's goals. Those rule goals can be followed by a comma-separated list of arguments, in parentheses.

The following is an example of a Prolog program:

[2] https://www.w3.org/TR/rdf-sparql-query/

```
% Rule
% Austria has a common border with Country
sharesBorder(austria,Country) :-
    sharesBorderWithAustria(Country) .

% Facts
% Austria has a common border with Switzerland
sharesBorder(austria,switzerland) .
% Austria has a commong border with Germany
sharesBorderWithAustria(germany) .

% Queries
% Do Austria and Switzerland have a common border?
?- sharesBorder(austria,switzerland).
true.

% Do Austria and China have a common border?
?- sharesBorder(austria,china).
false.

% Which countries have a common border?
?- sharesBorderWithAustria(X).
X = germany ;
X = switzerland .
```

The following shows an example of checking if an RDF triple exists in a certain graph using the library(semweb(rdf11))[3] and rdf/4[4].

```
% Import module for using rdf/4
:- use_module(library(semweb/rdf11)).

% Queries using rdf/4
% rdf/4: rdf(?Subject, ?Predicate, ?Object, ?Graph)
% To which countries does Austria have a common border?
?- rdf(austria, sharesBorder, Y, graph1).
Y = austria ;
Y = switzerland.

% Does Austria have a common border with China?
?- rdf(austria, sharesBorder, china, graph1)
false.

% Does Austria have a common border with Switzerland?
?- rdf(austria, sharesBorder, switzerland, graph1)
true.
```

[3] https://www.swi-prolog.org/pldoc/man?section=semweb-rdf11
[4] https://www.swi-prolog.org/pldoc/man?predicate=rdf/4

2.2 Definitions

Subsection 2.2.1 explains the mathematical concept of a mapping and defines schema mapping and data mapping, which are the theoretical basics for our mappings. Subsection 2.2.2 describes models, metamodels and model transformations.

2.2.1 Mapping

The mathematical concept of a mapping is defined as follows:

> "If S is a subset of $D(F)$, we say that F is defined on S. In this case, the set of $F(x)$ such that $x \in S$ is called the image of S under F and is denoted by $F(S)$. If T is any set which contains $F(S)$, then F is also called a **mapping** from S to T. This is often denoted by writing
>
> $$F : S \to T \tag{2.1}$$
>
> If $F(S) = T$, the mapping is said to be onto T." [Apo74]

In the course of this thesis we perform three types of mappings:

Schema Mapping describes how data structured according to the so called source schema is to be transformed into data structured according to the target schema [Fag06]. Defining schema mapping as a triple $M=(S_1, S_2, \Sigma)$, S_1 is the source schema, S_2 the target schema and Σ, which is called mapping expression, is a set of constraints over S_1 and S_2 [RO08]. $\langle s_1, s_2 \rangle$ describes an instance of mapping M, which satisfies all the constraints Σ, s_1 is an instance of S_1 and s_2 is an instance of S_2 [FKPT05]. We perform schema mapping by transforming the KG schema defined in RDFS into a structurally preserved Prolog schema.

Data Mapping specifies how to translate data from one source representation to a target representation [FBJV05]. We perform data mapping by translating the KG data defined in RDF into structurally preserved Prolog facts.

Metaschema Mapping is defined as $M=(MS_1, MS_2, \Sigma)$, whereas MS_1 is the source metaschema, MS_2 is the target metaschema and Σ is a set of constraints on MS_1 and MS_2 [RO08]. $\langle S_1, S_2 \rangle$ describes an instance of mapping M, which satisfies all the constraints Σ, S_1 is an instance of MS_1 and S_2 is an instance of MS_2. We perform metaschema mapping as the KG schema and the Prolog schema conform to a predefined source and target metaschema. This means that not all possible structures of RDFS are accepted and the target consists of only a subset of all existing Prolog schema structures. Consequently, we adapted the metaschemas to the need of the AISA project.

2.2.2 Model Transformation

Before we look at the definition of a model transformation, we need to define what a model and a metamodel is.

Model. Didonet Del Fabro et al. [FBJV05] describe a model as a directed graph:

$$G = (V, A) \tag{2.2}$$

V is the set of vertices denoting the model elements and the set of labeled edges A denotes the association between model elements. A model element has an identifier, which may be implemented as URIs, and a value of any data type, such as integer, strings, classes.

The philosophical concept of a model is defined as follows:

> "In many senses, also considering that it is recognized that observer and observations alter the reality itself, at a philosophical level one can agree that "everything is a model," since nothing can be processed by the human mind without being "modeled."" [BCW17]

This means that even a transformation itself could be seen as a model. A descriptive transformation is called transformation model and provides uniformity between the model and the transformation regarding the description language. This uniformity allows to conduct higher order transformations, which are transformations on transformations [BBG+06].

Metamodel. Didonet Del Fabro et al. [FBJV05] define a metamodel as a model, which specifies the structure of a model. The structure of this model conforms to the metamodel. Given a model

$$M = (V, A) \tag{2.3}$$

and a metamodel

$$MM = (V', A') \tag{2.4}$$

There is an outgoing edge for each model element $e \in V$ to an element $me \in V'$, which is called a meta-edge and denoted by *Meta (e,me)*.

Transformation. A transformation is a functional mapping constraint like a query, for example [BM07]. To define a mapping, correspondences between two schemas must be identified. Then the correspondences need to be translated into mapping constraints [MHH00]. If the mapping constraints are not already transformations, then they need to be translated into transformations [BM07].

Model Transformation. A model transformation is an operation in which one set of models serves as input and another set of models is produced as output [FBJV05]. Given a set of input models M_1, ..., M_n and a set of output models OM_1, ..., OM_n, a transformation model can be denoted by *Mt*. The transformation model conforms to a transformation metamodel.

2.3 Software Technologies

Assuming basic knowledge of Prolog, in this section we only discuss some specifically relevant concepts of Prolog as well as Prolog software packages and additional technologies used in this project.

Apache Jena[5] [jena] is a free and open source Java framework for building Semantic Web and Linked Data applications. It provides an API to read from and write to RDF graphs. Data can be stored in models and those models can be queried through SPARQL.

Apache Jena Fuseki[6] [jenb] is a SPARQL server, which can run as a standalone server.

SWI-Prolog[7] [swi] is a free implementation of Prolog and very well suited as a starting point for a KG-Prolog mapper as it provides an in-memory RDF database [WSW03] as well as a SPARQL client library. The core system of SWI-Prolog is often used as a tool for building research prototypes, primarily for knowledge intensive and interactive systems. Wielemaker et al. [WSTL10] describes SWI-Prolog as "an integrating tooling, supporting a wide range of ideas developed in the Prolog community and acting as a clue between foreign resources".

By **Prolog engine**, we refer to an instance of SWI-Prolog running as a process. Client applications communicate with the Prolog engine via *queries*. Prolog *programs* are compiled and loaded into the Prolog engine. The Prolog engine makes predicates defined in these programs available for querying. Predicates may be defined by asserted facts and/or by rules. Prolog programs loaded to a Prolog engine define a shared *database* (see below). Queries and rules may refer to built-in predicates with side-effects such as updating the database.

Prolog programs may be defined as **Prolog modules**, each with a unique module name. Prolog modules act as namespace for the predicates defined by the program and facilitate separation of programs running within the same Prolog engine.

JPL[8] [jpl] is a software library that provides a bidirectional interface between Java and Prolog. JPL facilitates to run a Prolog engine embedded within the Java virtual machine. With the use of JPL, Java programs can control the Prolog engine by loading Prolog programs, asserting facts, posing queries, and invoking built-in predicates with side-effects. JPL is not a pure Java implementation of Prolog, but makes extensive use of native implementations of Prolog and only works with SWI-Prolog. JPL enables hybrid Prolog+Java applications to be designed and implemented so as to take best advantage of both language systems.

By **database** we refer to the set of predicates including their schemas and extensions maintained in-memory and made available for querying in the Prolog engine. We may

[5] https://jena.apache.org/
[6] https://jena.apache.org/documentation/fuseki2/
[7] https://www.swi-prolog.org/
[8] https://jpl7.org/

distinguish: the *extensional database* comprising static facts and dynamically asserted facts and the *intensional database* comprising virtual facts.

The in-memory **RDF database (RDF DB)** [WSW03] of SWI-Prolog [swi] maintains within a Prolog engine an RDF dataset and makes it available for querying via dynamic predicate `rdf/4` to the programs run by the Prolog engine. RDF quadruples can be added dynamically to the RDF database via method `rdf_assert/4` or by loading RDF files (typically one file per named graph) via method `rdf_load/2`. The contents of the RDF database can be saved as RDF files (typically one file per named graph) to the file system via method `rdf_save/2`.

The **SPARQL client library** of SWI-Prolog facilitates to execute SPARQL queries on an HTTP SPARQL endpoint, as provided by the AISA KG server, from Prolog.

2.4 KG Access from Prolog

In this subsection, we introduce the problem of accessing the AISA KG from Prolog. We discuss two generic approaches for accessing KGs from Prolog, the schema-oblivious and the schema-aware approach.

2.4.1 Schema-oblivious approach

Following the schema-oblivious approach for accessing the KG from Prolog, every RDF quadruple is represented as a Prolog fact. Figure 2.1 shows the conceptual architecture of a lightweight implementation of the schema-oblivious approach based on SWI-Prolog's RDF database. The system takes care of (partial) data replication between KG and in-memory RDF DB and the Prolog program reads from the RDF DB and writes to the RDF DB.

Figure 2.1: Conceptual architecture of the schema-oblivious approach

Figure 2.2 shows an example of schema-oblivious mapping. The top left of the figure shows the KG schema, which is defined in RDFS + SHACL. Below the schema is the KG data in RDF. KG schema and KG data can be read the following way: `:D-AIP` is an `:Aircraft` and is linked to 2 `:Flight` instances `:DLH28W` and `:DLH99W`. A `:Flight` has 2 properties, `:origin` and `:destination`. A `:Flight` has the property `:wingspan`, which consists of a `:value` and a `:unit` or a `:NilReason`. On the right, next to the KG schema and the KG data, the respective Prolog facts are shown. The library function `rdf/4` of SWI-Prolog consists of a Subject, Predicate, Object and a Graph. That means that each link is represented in one fact and can be interpreted the following way: `:D-AIP` is of `rdf:type :Aircraft` in Graph `:g1`. `:DAIP` has a

`:wingspan` `_:b1` in Graph `:g1`. `_:b1` has `:value` '35.8' in Graph `:g1` and a `:unit` 'm' in Graph `:g1`.

The schema-oblivious approach based on Prolog's RDF database is well suited for writing results from Prolog programs to the KG and also provides a highly-flexible approach for querying the KG from Prolog. The schema-oblivious approach is, however, unwieldy for Prolog programmers when it comes to reading KG data that has a complex structure, because knowledge about one object is distributed over many facts.

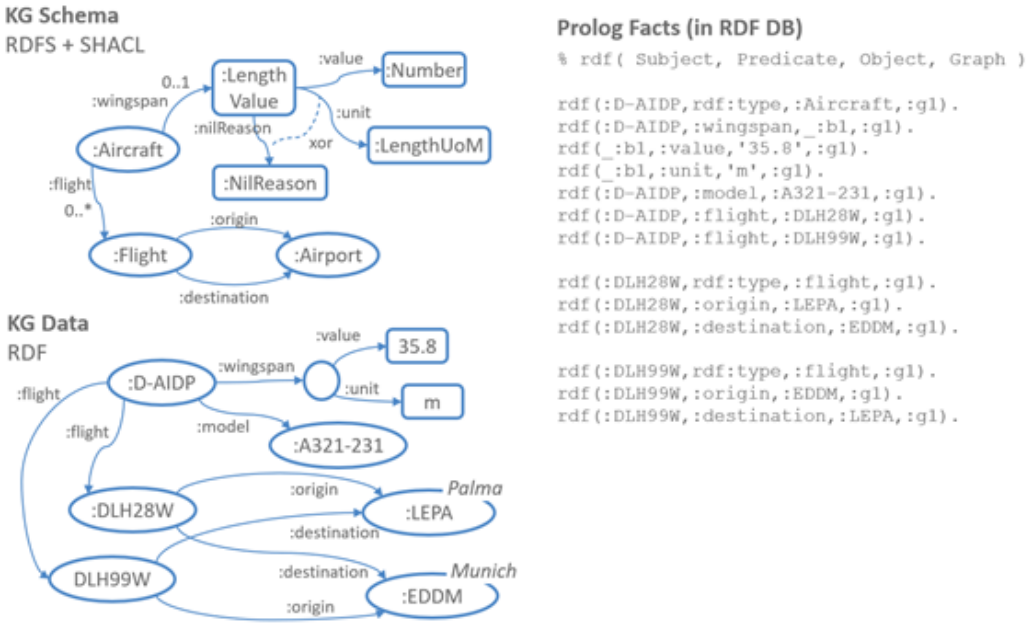


Figure 2.2: Schema-oblivious mapping example

2.4.2 Schema-aware approach

In order to overcome the shortcomings of the schema-oblivious approach when it comes to reading structured data from the KG we develop the schema-aware approach to KG-Prolog mapping. Figure 2.3 shows the conceptual architecture of the schema-aware approach. When accessing the KG from Prolog via a schema-aware approach, facts about one object in one named graph are combined in a single fact according to the KG schema.

This KG schema directly represents the conceptual schema. The implementation of a schema-aware approach additionally has components for mapping generation at design/compile time and mapping execution. This approach is more convenient for Prolog programmers, as knowledge about one object is already collected in schema-conforming facts.

The basic idea of the schema-aware mapping is that each RDF node is mapped with its properties exactly into one Prolog fact. This naturally preserves the KG schema in

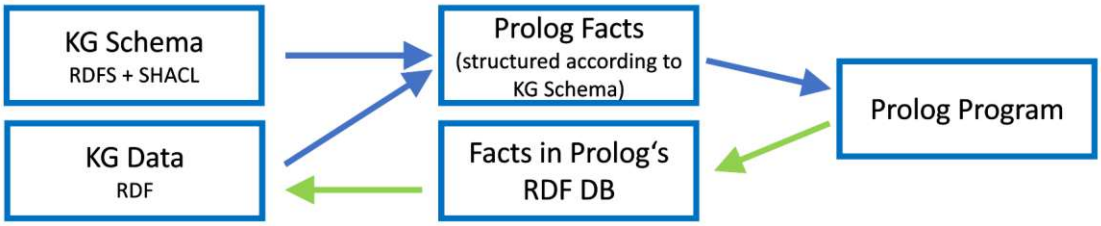


Figure 2.3: Conceptual architecture of the schema-aware approach

Prolog. Every RDFS class and corresponding SHACL node shape becomes a Prolog predicate. Every single-valued property becomes a single-valued argument of the Prolog predicate, potentially a null value. Every multi-valued property becomes a list-valued argument of the Prolog predicate, potentially empty.

The AISA KG is highly structured with structural schemata in the form of SHACL shape graphs for all parts of the KG. In comparison to the schema-oblivious approach, the schema-aware approach preserves this structure and, thus, facilitates the development and maintenance of Prolog programs. The design goal of this approach is to improve the access to the AISA KG from Prolog.

Figure 2.4 shows an example for schema-aware mapping. The KG schema and the KG data is the same as in Figure 2.2, which shows the resulting Prolog facts following a schema-oblivious approach. On the right of this figure, the Prolog schema, which is represented as comment, is shown. Below the Prolog schema, there are the Prolog facts, which are built up according to the Prolog schema. The Prolog schema shows that `:Aircraft` has 2 optional properties and one multi-valued property: an instance of `:Aircraft` can have zero or one instance of `:wingspan` and `:model`, and zero, one or multiple instances of `:flight`. The Prolog schema and facts can be interpreted the following way: `:Aircraft` from Graph `:g1` with the id `:D-AIDP` has a `:wingspan` val('35.8','m'), a `:model` `:A321-231` and a list of `:Flight` instances `[:DLH28W, :DLH99W]`. The `:Aircraft` from Graph `:g1` with the id `:D-AIDP` has a `:Flight` `:DLH28W` with the `:origin` `:LEPA` and the `:destination` `:EDDM`. The `:Flight` from Graph `:g1` with the id `:DLH99W` has the `:origin` `:EDDM` and the `:destination` `:LEPA`.

In comparison to the schema-oblivious mapping, only 3 facts instead of 13 facts are required in order to represent the example shown in the figure. It is assumed that this approach is more readable and intuitive to use for Prolog programmers, because facts about one object in one named graph are combined into a single fact according to the KG schema, which directly represents the conceptual schema.

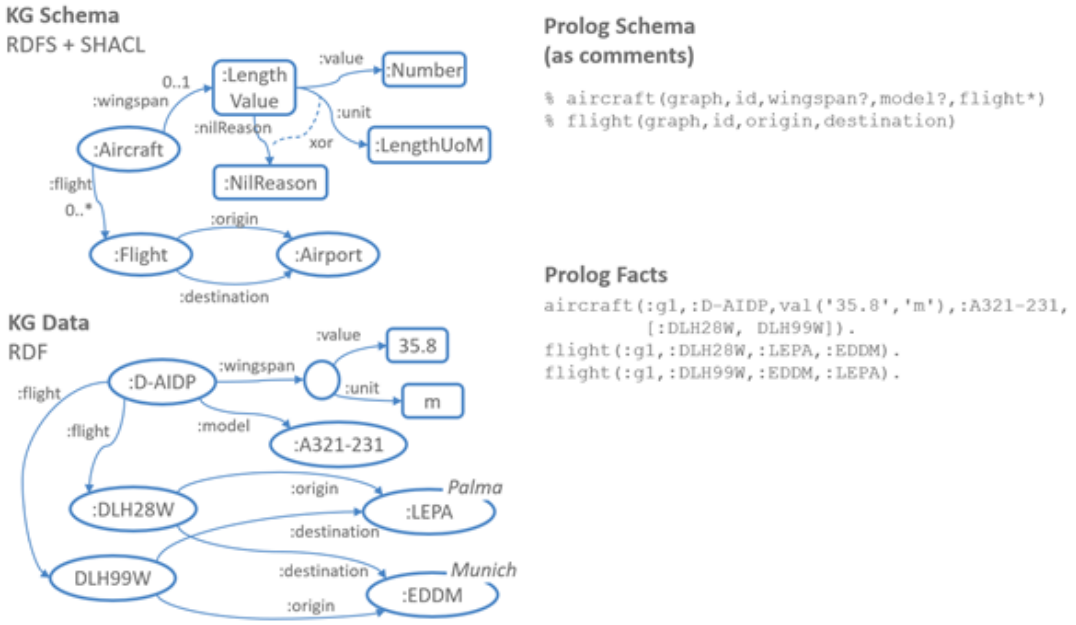


Figure 2.4: Schema-aware mapping example

CHAPTER 3

Realization of the Schema-aware RDF-Prolog Mapper

In this chapter, we discuss the realization of the schema-aware approach. Section 3.1 gives an overview of the realization of the schema-aware KG-Prolog mapper and of the three realization variants. Section 3.2 discusses the generation of the Prolog schema from RDFS/SHACL which is independent of the realization variant. Section 3.3 discusses schema-aware mapping with SPARQL queries executed in Java (Realization variant A). Section 3.4 goes into detail about schema-aware mapping SPARQL queries executed in Prolog (Realization variant B). Section 3.5 discusses realization variant C which is based on schema-aware mapping rules in Prolog on top of the RDF DB.

Results of preliminary performance studies for these three variants are discussed in Chapter 6.1 and details about the mapping of data types and missing values common to all mapping variants are discussed in Chapter 4.

3.1 Overview

Figure 3.1 gives an overview of the realization of the schema-aware approach. The **mapping generator** takes as input the **KG schema** represented in RDF schema and SHACL and produces, first, a **Prolog schema**, and, second, depending on the realization variant, **schema-specific mapping rules/queries**. Depending on the realization variant, see Chapter 3.3 for realization variant A, Chapter 3.4 for realization variant B and Chapter 3.5 for realization variant C, the schema-specific mapping rules or queries are executed, in Java or Prolog to produce the mapped **input Prolog facts** from the **KG data** represented in RDF. It is assumed that the KG data conforms to the KG schema. This can be ensured by validating the KG against the KG schema using a SHACL validator in order to detect incorrect data prior to mapping. Depending on

the realization variant, the input Prolog facts are asserted in the Prolog database or made available only as virtual facts by Prolog rules. The **Prolog schema** comprises a description of the structure of mapped input facts as well as inheritance rules which represent subclass hierarchies from the KG schema. The Prolog schema is independent of the realization variant, because each mapping variant creates the same result. The Prolog schema of input facts and inheritance rules are written as comments in the result files. This schema will be inspected by the Prolog programmer when writing a **Prolog program** which accesses the input Prolog facts.

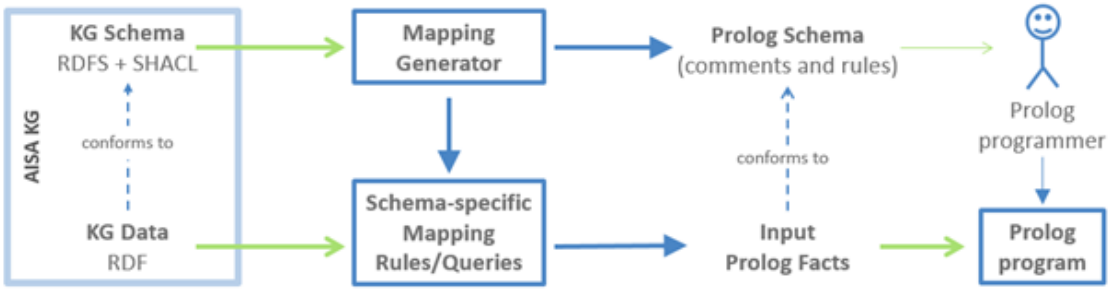


Figure 3.1: Conceptual architecture of the schema-aware approach

We have investigated and developed three different realization variants. Figure 3.2 gives an overview of these variants. **Variant A** takes the KG data as input and generates schema-specific SPARQL queries in Java. Each SPARQL query corresponds to one Prolog predicate and each row in its result set is transformed using Java into one Prolog fact and written into a Prolog file, which can be loaded into Prolog. **Variant B** takes the KG schema as input and generates schema-specific SPARQL queries embedded into Prolog rules. These Prolog rules with embedded SPARQL queries can then be loaded into Prolog and queried for the input Prolog facts. Variant B comes in two subvariants, in the original **subvariant B1** the input facts remain virtual and SPARQL queries are executed as part of the Prolog program. In **subvariant B2** the SPARQL queries are executed separately and assert the resulting input facts in the Prolog database. **Variant C** replicates the RDF quadruples from the KG in Prolog's RDF DB and generates schema-specific mapping rules in Prolog. The mapping rules can then be loaded into Prolog and used for querying input Prolog facts.

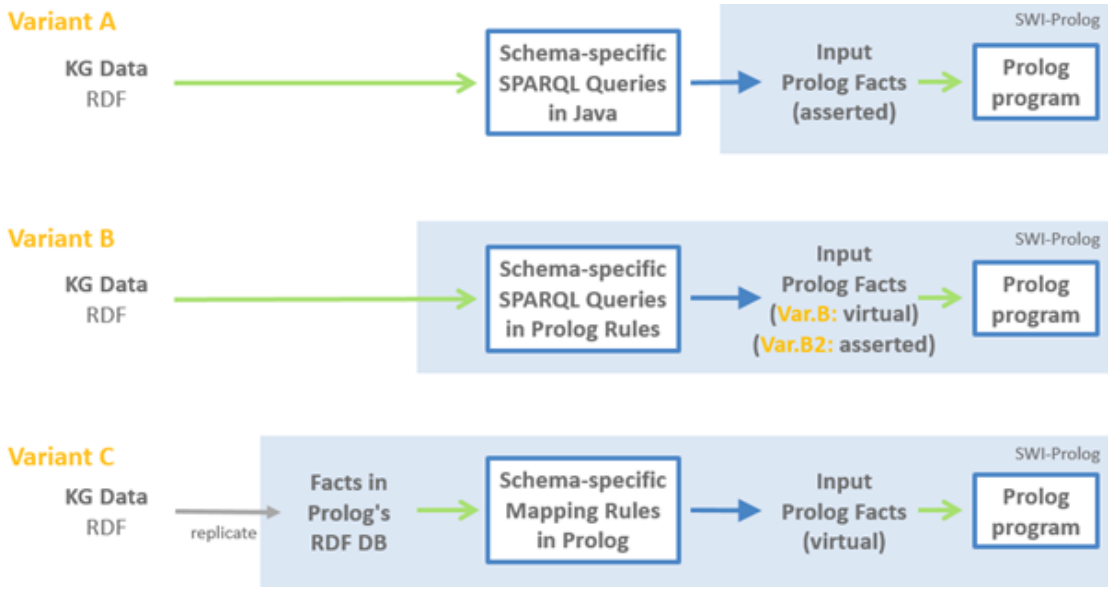


Figure 3.2: Realization variants for schema-specific mapping rules or queries in Prolog or Java

3.2 Generating the Prolog schema from RDFS/SHACL

The Prolog schema produced by the mapping generator consists of a description of the structure of mapped input facts together with rules that realized the subclass hierarchies from the KG schema. The Prolog schema is independent of the three realization variants. The mapping generator is integrated into the mappers of each variant and can be found on GitHub[1].

The SHACL Shapes Constraint Language is a language for specifying integrity constraints over RDF graphs and can be used, among others, to constrain the number of values that a property may have, the type of such values, numeric ranges, string matching patterns, and logical combinations of such constraints. Also inheritance can be specified using SHACL. Following a schema-aware approach for mapping RDFS/SHACL to Prolog, the SHACL schema predetermines how the final facts look like. For each target shape in SHACL, there is one predicate. A target shape is a SHACL shape, which is defined as an `rdfs:Class`, as well as a `sh:NodeShape`. For each target shape, a schema comment is generated. Node shapes that are not defined as `rdfs:Class` are not considered for the mapping.

Listing 1 shows the predicate schema for `aixm:organisationAuthority-Association` with the properties `aixm:type`, `aixm:annotation` and `aixm:the-OrganisationAuthority`. The order of the arguments in the predicate schema is defined by `sh:order` in the property shapes of the SHACL schema. The question mark next

[1] https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper

```
% aixm_OrganisationAuthorityAssociation(Graph,
↪  OrganisationAuthorityAssociation, Type?, Annotation*,
↪  TheOrganisationAuthority)
```

Listing 1: Generated Prolog schema of `aixm_OrganisationAuthorityAssociation` as a comment

to `Type` in the listing means that the value is optional, which is defined in the SHACL schema by an absent `sh:minCount` (or `sh:minCount=0` and a `sh:maxCount=1`). The "*" sign next to `Annotation` denotes the multiplicity and marks that this property is a list of zero, one or multiple values, which is defined in the SHACL schema by a missing `sh:minCount` (or `sh:minCount=0` and an absent `sh:maxCount` (or `sh:maxCount>1`). Regarding `TheOrganisationAuthority`, there is no question mark or multiplicity sign "*" after the argument name in the predicate schema. This means that there is only one value and this value is mandatory.

Each property shape of a target shape in SHACL is represented as an argument of the fact.

For each inheritance defined in the SHACL schema, an inheritance rule is printed to the fact file. At first, for each `rdfs:Class` with a subclass and for each subclass, i.e. an `rdfs:Class` with the attribute `rdfs:subClassOf`, a fact is generated. Then the inheritance rule, which is a combination of those 2 facts, is generated. An inheritance rule consists of the super class, the respective subclass and the name of the predicate ends with the suffix "Combined". Listing 2 shows the inheritance rule `aixm_AirportHeliportResponsibilityOrganisation_Combined`. This inheritance rule consists of `aixm_PropertiesWithSchedule` and its sub class `aixm_AirportHeliportResponsibilityOrganisation`.

```
aixm_AirportHeliportResponsibilityOrganisation_Combined(Graph,
↪  AirportHeliportResponsibilityOrganisation, AnnotationList,
↪  SpecialDateAuthorityList, TimeIntervalList, Role,
↪  TheOrganisationAuthority) :-
  aixm_AirportHeliportResponsibilityOrganisation(Graph,
  ↪  AirportHeliportResponsibilityOrganisation, Role,
  ↪  TheOrganisationAuthority),
  aixm_PropertiesWithSchedule(Graph,
  ↪  AirportHeliportResponsibilityOrganisation, AnnotationList,
  ↪  SpecialDateAuthorityList, TimeIntervalList) .
```

Listing 2: Inheritance rule `aixm_AirportHeliportResponsibilityOrganisation_Combined`

3.3 Mapping Variant A - SPARQL queries in Java

With this variant, we implemented a Java program, which generates with the help of SPARQL a set of Prolog predicates from the RDFS/SHACL schema and a SPARQL query for each predicate. The result of executing the SPARQL query is translated into the facts for the respective predicate. These facts are written to a file and that file is loaded into Prolog.

The Mapping Generator produces a target Prolog schema (i.e., a set of target predicates) and schema-specific SPARQL Select Queries (one query per target predicate). The Schema-aware Runtime System executes for each target predicate the associated query to produce a set of Prolog facts (one fact for each query solution) to populate the target predicate. The system may assert the generated facts directly via JPL[2] or write them to a Prolog fact file and invoke the loading of this fact file together with invoking the Prolog program.

Mapping variant A[3] is open source and available for experimentation. Before execution of the mapping, Jena Fuseki (version 3.16.0) server should be started. The starting point and main method of the mapping can be found in the `Shacl2PrologLauncher.java` file. When executing the Shacl2PrologLauncher, first, a connection to the Jena Fuseki server is established and the input files are uploaded. After uploading the input files, the schema is fetched from Jena Fuseki and the SHACL shapes are parsed. The parsed SHACL shapes serve as input for the instantiation of `Mapper.java`. At initialization time, the mapper creates an instantiation of `KnowledgeGraphClass.java` for each target shape and a linked instantiation of `KnowledgeGraphProperty.java` for each of its properties. At creation time of those classes and its properties, the dedicated SPARQL query, the facts schema and further information, which is used multiple times during mapping, is saved to variables in order to avoid processing the same data repeatedly. After the instantiation, the mapper iterates over the list of knowledge graph classes and generates the respective queries, which are saved to `/output/queries.sparql` by a PrintWriter. After the generation of the query file, the mapper executes the queries separately and processes the result sets to have the correct data types processable by Prolog. At this point, the facts are generated and saved to the `/output/facts.pl` file. Next to the facts, static content like prefix registration, `rdf_meta`[4] and the use of modules in Prolog are also printed to the facts file. Furthermore, inheritance rules for `rdfs:subClassOf` relations between node shapes, as explained in the previous section, are generated and printed to the facts file. After the creation of the facts file, a short Prolog program `/output/program.pl` is consulted and executed. This program loads the facts file into Prolog and demonstrates in a short example how output can be saved in `/output/output.ttl`, which will be load to Fuseki at next. Finally, the performance results (see Section 6.1.2) are saved to `/output/performance_results.csv`.

[2] https://jpl7.org/
[3] https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper/tree/main/at.jku.dke.aisa.mapperA
[4] https://www.swi-prolog.org/pldoc/man?predicate=rdf_meta/1

3.4 Mapping Variant B – SPARQL queries in Prolog

Mapping variant B also consists of a Java program, which generates a Prolog module from the RDFS/SHACL schema with the predicates that are linked to the respective SPARQL queries by means of Prolog rules. The SPARQL queries for filling the predicates are only executed from Prolog at runtime. The Mapping Generator produces a target Prolog schema (i.e., a set of target predicates), schema-specific SPARQL Select Queries (one query per target predicate) and Prolog rules (one per target predicate) in which the queries are embedded. The Schema-aware Runtime System invokes the Prolog program which in turn calls the Prolog rules with the embedded SPARQL queries. We have developed and investigated two subvariants of variant B. In the original variant, SPARQL queries are embedded in Prolog rules that are used to derive input Prolog facts which remain virtual (i.e., the input facts are not asserted in the Prolog database). In subvariant B2, each SPARQL query is executed only once and the resulting input facts are asserted in the Prolog database.

Realization variant B1[5] and its subvariant B2[6] are available open source for experimentation. Analogous to mapping variant A, as described in Section 3.3, the Jena Fuseki server need to be started, the SHACL shapes are parsed and the knowledge graph classes and properties are created. At creation time of those classes and its properties, the dedicated SPARQL query, the facts schema and further information, which is used multiple times during mapping, is saved to variables in order to avoid processing the same data over and over again. At next, the `/output/facts.pl` file is generated and the content printed. The content of the facts file consists of static content like the use of Prolog libraries and modules and convert methods to handle data types and lists correctly. Furthermore, the inheritance rules and the prolog modules with embedded SPARQL queries are generated and printed to the facts file. Finally, the performance results (see Section 6.2) are saved and the facts file is consulted and executed analogous to mapping variant A, as described in Section 3.3.

3.5 Mapping Variant C – Mapping Rules in Prolog

Mapping variant C also consists of a Java program. The relevant part of the Knowledge Graph is replicated in the main-memory RDF database of SWI-Prolog. The actual mapping is then formulated as Prolog rules with the RDF quadruples in the RDF database as input. The Mapping Generator procures a target Prolog schema (i.e., a set of target predicates) and Mapping Rules in Prolog to map from rdf/4 to target predicates. The Schema-aware Runtime System takes care of replicating data from the KG into Prolog's RDF DB (this is already implemented by the schema-oblivious Runtime System) and invokes the Prolog program which in turn calls the Mapping Rules.

[5] https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper/tree/main/at.jku.dke.aisa.mapperB
[6] https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper/tree/main/at.jku.dke.aisa.mapperB2

Mapping variant C[7] is available open source for experimentation. Analogous to mapping variant A, as described in Section 3.3, and mapping variant B, as described in Section 3.4, the Jena Fuseki server need to be started and the SHACL shapes are parsed. The data is written to `/output/dataset.trig`. The parsed SHACL shapes serve as input for the instantiation of `Mapper.java`. At initialization time, the mapper creates an instantiation of `KnowledgeGraphClass.java` for each target shape and a linked instantiation of `KnowledgeGraphProperty.java` for each of its properties. At next, the `/output/facts.pl` file is generated. Static content like the use of Prolog libraries and modules, prefix registrations, load data set and `rdfs:subClassOf` relations are printed to the facts file. Next to the static context, inheritance rules and the facts rules are generated and printed. Finally, the performance results (see Section 6.4) are saved and the facts file is consulted and executed analogous to mapping variant A, as described in Section 3.3.

[7] https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper/tree/main/at.jku.dke.aisa.mapperC

CHAPTER 4

Handling of Data Types and Missing Values

Important for the implementation is the correct handling of data types, lists, and missing values. Our goal was to process the data types into a Prolog readable format and to get exactly the same results from each mapping variant. Generally, the mappers only consider data types relevant for the project: `xsd:string`, `xsd:integer`, `xsd:decimal`, `xsd:unsignedInt` and `xsd:dateTime`. Listings in this chapter only contain relevant parts of the code, some parts of the code are therefore omitted and replaced by "...".

In this chapter, we first discuss in Section 4.1, how reading and processing values and data types is solved differently for each realization. The remaining sections show examples of mapping nilReason (Section 4.2), values without unit of measurement (Section 4.3), values with unit of measurement (Section 4.4), indeterminate positions and dateTimes (Section 4.5), missing values (Section 4.6), and, finally, the mapping of multi-valued properties to Prolog lists (Section 4.7). These examples apply for all realization variants since only the way the data is queried and processed is different, but the final results are equal. Finally, Section 4.8 summarizes this chapter.

4.1 Value handling in the different mapping realization variants

Mapping variant A puts the value and the data type together within the SPARQL query. The content of the results set are processed in the right format for Prolog in the next step in Java.

Given the data shown in Listing 3 and the example SPARQL query for `aixm:OrganizationAuthorityAssociation` showed in Listing 4, the binding for variable `?role` in the result set of the SPARQL query will be:

`val:OPERATE:http://www.w3.org/2001/XMLSchema#string`

Mapping variant A processes this result in Java and the final result is:

`val("OPERATE"^^xsd:'string')`

This result of data mapping is then written to the fact:

```
aixm_AirportHeliportResponsibilityOrganisation(
graph:'902_donlon-data.ttl', s1:'A-a72cfd3a',
val("OPERATE"^^xsd:'string'),
uuid:'74efb6ba-a52a-46c0-a16b-03860d356882' ).
```

```
s1:A-a72cfd3a
  a aixm:AirportHeliportResponsibilityOrganisation;
  aixm:role [rdf:value "OPERATE"];
  aixm:theOrganisationAuthority <uuid:74efb6ba-a52a-46c0-a16b-03860d356882>;
  aixm:annotation s1:n002.
```

Listing 3: Example data `aixm:AirportHeliportResponsibilityOrganisation`

Mapping variant B also puts the value and the data type together within the SPARQL query, which can be seen in Listing 5, just like in variant A. However, the processing of the result set happens in Prolog and is called within the Prolog module.

With `convVal(Value,ValueVal)` and `convert(Value,ValueList)` in line 33 and 34 of Listing 5 the Prolog data type mapping methods (see Listing 6) are called, which process the SPARQL query results in the right format:

`convert(ConcatenatedString,ListOfAtoms)`, see line 1 in Listing 6, is called to concatenate values to a list and converting the values of the list in the right format.

`convert(Null,[])`, see line 6 in Listing 6, is called to handle empty lists.

`convVal(String,Value)`, see line 11 in Listing 6, is called to handle and map data types correctly. Detailed examples of how input data looks after mapping are shown in the next subsections of this chapter.

Mapping variant C does the processing of values, data types and missing values within the generated mapping rules. Further handling of the value and data type in Java or Prolog is not necessary for this mapping variant. Listing 7 shows an example for such a mapping rule.

```
1  SELECT ?graph ?airportHeliportResponsibilityOrganisation ?role
   ↪  ?theOrganisationAuthority
2  WHERE
3    { GRAPH ?graph
4      {
5        ?airportHeliportResponsibilityOrganisation rdf:type
         ↪  aixm:AirportHeliportResponsibilityOrganisation .
6        OPTIONAL { ?airportHeliportResponsibilityOrganisation aixm:role
         ↪  ?_role .
7          {
8            {
9              ?_role rdf:value ?roleValue .
10             FILTER ( NOT EXISTS {?_role (aixm:uom | fixm:uom |
               ↪  plain:uom) ?roleUoM})
11             BIND(
               ↪  concat('val:',STR(?roleValue),':',STR(DATATYPE(?roleValue)))
               ↪  AS ?role)
12           }
13             UNION
14           {
15             ?_role
16               rdf:value ?roleValue ;
17               (aixm:uom | fixm:uom | plain:uom) ?roleUoM .
18             BIND(
               ↪  concat('xval:',STR(?roleValue),':',STR(DATATYPE(?roleValue))
               ↪  ,':',?roleUoM) AS ?role)
19           }
20             UNION
21           {
22             ?_role  aixm:nilReason ?roleNilReason .
23             BIND(concat('nil:',?roleNilReason) AS ?role)
24           }
25           UNION
26           {
27             ?_role  gml:indeterminatePosition ?indeterminatePosition .
28             BIND(concat('indeterminate:',?indeterminatePosition) AS
               ↪  ?role)
29           }
30         }
31       }
32       ?airportHeliportResponsibilityOrganisation
         ↪  aixm:theOrganisationAuthority ?theOrganisationAuthority .
33     }
34   }
```

Listing 4: SPARQL query of `aixm:AirportHeliportResponsibilityOrganisation` from mapping variant A

```
1  aixm_ConditionCombination(Graph, ConditionCombination,
   ↪ LogicalOperatorVal, FlightList, AircraftList,
2  WeatherList, SubConditionList) :-
3    sparql_query(
4        '
5  ...
6  PREFIX aixm: <http://www.aisa-project.eu/vocabulary/aixm_5-1-1#>
7  ...
8
9  SELECT ?graph ?conditionCombination ?logicalOperator
10 (GROUP_CONCAT(DISTINCT ?flight;SEPARATOR=",") AS ?flightConcat)
11 (GROUP_CONCAT(DISTINCT ?aircraft;SEPARATOR=",") AS ?aircraftConcat)
12 (GROUP_CONCAT(DISTINCT ?weather;SEPARATOR=",") AS ?weatherConcat)
13 (GROUP_CONCAT(DISTINCT ?subCondition;SEPARATOR=",") AS
↪  ?subConditionConcat)
14 WHERE
15   { GRAPH ?graph
16     {
17       ?conditionCombination rdf:type aixm:ConditionCombination .
18       OPTIONAL { ?conditionCombination aixm:logicalOperator
↪  ?_logicalOperator .
19         {
20           {
21             ?_logicalOperator rdf:value ?logicalOperatorValue .
22             FILTER ( NOT EXISTS {?_logicalOperator (aixm:uom | fixm:uom
↪  | plain:uom) ?logicalOperatorUoM})
23
↪  BIND(concat(\'val:/:\',STR(?logicalOperatorValue),\':/:\',STR(DATATYPE(
↪  ?logicalOperatorValue))) AS ?logicalOperator)
24           }
25 ...
26       OPTIONAL {?conditionCombination aixm:subCondition ?subCondition .}
27     }
28   }
29 GROUP BY ?graph ?conditionCombination ?logicalOperator
30
31       '
32 ,row(Graph,ConditionCombination,LogicalOperator,FlightConcat,
   ↪ AircraftConcat,WeatherConcat,SubConditionConcat),[]),
33 convVal(LogicalOperator,LogicalOperatorVal),
   ↪ convert(FlightConcat,FlightList),
   ↪ convert(AircraftConcat,AircraftList),
34 convert(WeatherConcat,WeatherList),
   ↪ convert(SubConditionConcat,SubConditionList).
35
```

Listing 5: Prolog module with embedded SPARQL query of `aixm:ConditionCombination` from mapping variant B

```
convert(ConcatenatedString,ListOfAtoms) :-
  ConcatenatedString = literal(XConcatenatedString),
  split_string(XConcatenatedString, ',', '', ListOfStrings),
  maplist(convVal, ListOfStrings, ListOfAtoms).

convert(Null,[]) :-
  Null = '$null$'.

string_atom(X,Y) :- atom_string(Y,X).

convVal(String,Value) :-
  String \= "$null$",
  ( String = literal(XString) ; ( (\+ String = literal(_)), XString =
  ↪  String ) ),
  re_split(":/:",XString,List),
  ( ( List = [X], string_atom(X,Value) ) ;
    ( List = ["nil",_,NilReason], Value =
    ↪  nil(^^(NilReason,'http://www.w3.org/2001/XMLSchema#string')) ) ;
    ( List = ["indeterminate",_,Indeterminate], Value =
    ↪  indeterminate(^^(Indeterminate,
    ↪  'http://www.w3.org/2001/XMLSchema#string')) ) ;
    (
      (
        ( List = ["val",_,Val,_,TypeS], Value = val(^^(CastVal,Type)) )
        ↪  ;
        ( List = ["xval",_,Val,_,TypeS,_,Uom], Value =
        ↪  xval(^^(CastVal,Type),
        ↪  ^^(Uom,'http://www.w3.org/2001/XMLSchema#string') ) )
      ) ,
      string_atom(TypeS,Type) ,
    (
    % Numeric Type
      ( ( Type = 'http://www.w3.org/2001/XMLSchema#integer';
          Type = 'http://www.w3.org/2001/XMLSchema#decimal';
          Type = 'http://www.w3.org/2001/XMLSchema#unsignedInt' ),
        number_string(CastVal,Val)
      );
      ( Type = 'http://www.w3.org/2001/XMLSchema#dateTime',
        CastVal = date_time(Year,Month,Day,Hour,Minute,Second,0),
        sub_string(Val, 0, 4, _, YearString),
        ↪  number_string(Year,YearString),
        sub_string(Val, 5, 2, _, MonthString),
        ↪  number_string(Month,MonthString),
        sub_string(Val, 8, 2, _, DayString),
        ↪  number_string(Day,DayString),
        sub_string(Val, 11, 2, _, HourString),
        ↪  number_string(Hour,HourString),
        sub_string(Val, 14, 2, _, MinuteString),
        ↪  number_string(Minute,MinuteString),
        sub_string(Val, 17, 2, _, SecondString),
        ↪  number_string(Second,SecondString)
      );
      % ELSE (e.g. TYPE = String)
      ( Type \= 'http://www.w3.org/2001/XMLSchema#integer',
        Type \= 'http://www.w3.org/2001/XMLSchema#decimal',
        Type \= 'http://www.w3.org/2001/XMLSchema#unsignedInt',
        Type \= 'http://www.w3.org/2001/XMLSchema#dateTime',
        CastVal = Val ) ) ) ).
```

Listing 6: Prolog methods for data type handling of mapping variant B

```
1  aixm_AirportHeliportResponsibilityOrganisation(Graph,
   ↪ AirportHeliportResponsibilityOrganisation, Role,
   ↪ TheOrganisationAuthority) :-
2    rdf(AirportHeliportResponsibilityOrganisation,rdf:type,
     ↪ aixm:'AirportHeliportResponsibilityOrganisation',Graph)
3    ,(
4      ( Role='$null$',
5        \+ rdf( AirportHeliportResponsibilityOrganisation,
         ↪ aixm:'role',_Role,Graph )
6      );
7    ( rdf( AirportHeliportResponsibilityOrganisation,
     ↪ aixm:'role',RoleNode,Graph )),
8        (
9          (
10           rdf(RoleNode,rdf:value,RoleValue,Graph),
11          \+ ( rdf( RoleNode, aixm:uom, _RoleUOM, Graph );
12          rdf( RoleNode, fixm:uom, _RoleUOM, Graph );
13          rdf( RoleNode, plain:uom, _RoleUOM, Graph ) ),
14           Role=val(RoleValue)
15         );
16         (
17           rdf( RoleNode,rdf:value,RoleValue,Graph ),
18           ( rdf( RoleNode, aixm:uom, RoleUOM, Graph );
19           rdf( RoleNode, fixm:uom, RoleUOM, Graph );
20           rdf( RoleNode, plain:uom, RoleUOM, Graph ) ),
21           Role=xval(RoleValue,RoleUOM)
22         );
23         (
24           rdf( RoleNode,aixm:nilReason, RoleNilReason, Graph ),
25           Role=nil(RoleNilReason)
26         );
27         (
28           rdf( RoleNode,gml:indeterminatePosition, RoleIndeterminate,
             ↪ Graph ),
29           Role=indeterminate(RoleIndeterminate)
30         )
31       )
32   )
33   ,rdf(AirportHeliportResponsibilityOrganisation,
     ↪ aixm:'theOrganisationAuthority',TheOrganisationAuthority,Graph) .
```

Listing 7: Mapping rule of mapping variant C

4.2 NilReasons

If the data is of type `aixm:NilReason`, then the results are mapped to format **nil(NilReason)**.

Example: As shown in Listing 8, `aixm:AirportHeliportTimeSlice` has a property `aixm:designatorIATA`, which is defined as a `DataTypeNodeShape` `aixm:CodeIATA-Type`.

```
1  aixm:AirportHeliportTimeSlice
2          a             sh:NodeShape , aixm:FeatureNodeShape , rdfs:Class ;
3          sh:and        ( aixm:AIXMTimeSlice ) ;
4  ...
5          sh:property   [ sh:maxCount  1 ;
6                          sh:node      aixm:CodeIATAType ;
7                          sh:order     4 ;
8                          sh:path      aixm:designatorIATA
9                        ] ;
10 ... .
11 aixm:CodeIATAType  a  aixm:DataTypeNodeShape , sh:NodeShape ;
12         sh:and        ( aixm:CodeIATABaseType ) ;
13         sh:property   [ sh:maxCount  1 ;
14                         sh:node      aixm:NilReasonEnumeration ;
15                         sh:order     1 ;
16                         sh:path      aixm:nilReason
17                       ] ;
18         sh:property   [ sh:maxCount  1 ;
19                         sh:order     1 ;
20                         sh:path      rdf:value
21                       ] ;
22         sh:xone       ( [ sh:property  [ sh:minCount  1 ;
23                                          sh:order     1 ;
24                                          sh:path      rdf:value
25                                        ]
26                         ]
27                         [ sh:property  [ sh:minCount  1 ;
28                                          sh:order     1 ;
29                                          sh:path      aixm:nilReason
30                                        ]
31                         ]
32                       ) .
```

Listing 8: Example SHACL shapes of `aixm:AirportHeliportTimeSlice` and `aixm:CodeIATAType`

The `DataTypeNodeShape` `aixm:CodeIATAType` shows that `aixm:designatorIATA` of `aixm:AirportHeliportTimeSlice` is either represented as a value `rdf:value` or as a nilReason `aixm:NilReason`.

Given the data from Listing 9, the mapping result is produced as in Listing 10.

```
  s1:AHP_EADH
    a aixm:AirportHeliportTimeSlice;
...
    aixm:designatorIATA [ aixm:nilReason "unknown"];
....
```

Listing 9: Example data `aixm:AirportHeliportTimeSlice` with an `aixm:designatorIATA` `aixm:nilReason` value

```
% aixm_AirportHeliportTimeSlice(Graph, AirportHeliportTimeSlice,
... DesignatorIATA?, ...)
aixm_AirportHeliportTimeSlice(graph:'149_donlon-data.ttl', s1:'AHP_EADH',
... nil("unknown"^^xsd:'string'), ...).
```

Listing 10: Mapping result of `aixm:AirportHeliportTimeSlice` with an `aixm:designatorIATA` `aixm:nilReason` value

4.3 Values without Unit of Measurement

If no unit of measurement exists for a value when querying or posing the mapping rule, then the result is mapped in the format **val(Value)**.

Example: Listing 13 shows a snippet of the SHACL shapes from `aixm:AirportHeliportTimeSlice`. This shape has a property `aixm:designator`, which is defined as a `DataTypeNodeShape` `aixm:CodeAirportHeliportDesignatorType`.

The `DataTypeNodeShape` `aixm:CodeAirportHeliportDesignatorType` shows that `aixm:designator` of `aixm:AirportHeliportTimeSlice` can either be represented as a value `rdf:value` or as a NilReason `aixm:NilReason`.

Given the data for `aixm:AirportHeliportTimeSlice` from Listing 11, the mapping result is produced as in Listing 12.

```
  s1:AHP_EADH
    a aixm:AirportHeliportTimeSlice;
...
    aixm:designator [rdf:value "EADH"];
....
```

Listing 11: Snippet of data from `donlon-data.ttl`: `aixm:AirportHeliportTimeSlice` with a value property

```
% aixm_AirportHeliportTimeSlice(Graph, AirportHeliportTimeSlice,
↪  Designator?, ...)
aixm_AirportHeliportTimeSlice(graph:'149_donlon-data.ttl', s1:'AHP_EADH',
val("EADH"^^xsd:'string'), ...).
```

Listing 12: Data handling and mapping result: `aixm:AirportHeliportTimeSlice` with a value without an unit of measurement

```
1  aixm:AirportHeliportTimeSlice
2        a             sh:NodeShape , aixm:FeatureNodeShape , rdfs:Class ;
3        sh:and        ( aixm:AIXMTimeSlice ) ;
4  ...
5        sh:property   [ sh:maxCount  1 ;
6                        sh:node      aixm:CodeAirportHeliportDesignatorType ;
7                        sh:order     1 ;
8                        sh:path      aixm:designator
9                      ] ;
10 ....
11 aixm:CodeAirportHeliportDesignatorType
12       a             aixm:DataTypeNodeShape , sh:NodeShape ;
13       sh:and        ( aixm:CodeAirportHeliportDesignatorBaseType ) ;
14       sh:property   [ sh:maxCount  1 ;
15                       sh:node      aixm:NilReasonEnumeration ;
16                       sh:order     1 ;
17                       sh:path      aixm:nilReason
18                     ] ;
19       sh:property   [ sh:maxCount  1 ;
20                       sh:order     1 ;
21                       sh:path      rdf:value
22                     ] ;
23       sh:xone       ( [ sh:property  [ sh:minCount  1 ;
24                                        sh:order     1 ;
25                                        sh:path      rdf:value
26                                      ]
27                       ]
28                       [ sh:property  [ sh:minCount  1 ;
29                                        sh:order     1 ;
30                                        sh:path      aixm:nilReason
31                                      ]
32                       ]
33                     ) .
```

Listing 13: Snippet of SHACL shapes from `donlon-shacl.ttl`: `aixm:AirportHeliportTimeSlice` and `aixm:CodeAirportHeliportDesignatorType` with value properties

4.4 Values with Unit of Measurement

If a unit of measurement exists for a value when querying or posing the mapping rule, then the result are mapped in the format **xval(Value,UoM)**.

Example: As shown in Listing 16, `aixm:AirportHeliportTimeSlice` has a property `aixm:fieldElevation`, which is defined as a `DataTypeNodeShape` `aixm:ValDistanceVerticalType`.

The `DataTypeNodeShape` `aixm:ValDistanceVerticalType` shows that `aixm:fieldElevation` of `aixm:AirportHeliportTimeSlice` can either be represented as a value `rdf:value` with a unit of measurement `aixm:uom` or as an `aixm:nilReason`.

Given the data from Listing 14, the mapping result is produced as in Listing 15.

```
  s1:AHP_EADH
    a aixm:AirportHeliportTimeSlice;
...
    aixm:fieldElevation [aixm:uom "M"; rdf:value "18"];
...
    .
```

Listing 14: Example data from `donlon-data.ttl`: `aixm:AirportHeliportTimeSlice` with a property having an unit of measurement

```
% aixm_AirportHeliportTimeSlice(Graph, AirportHeliportTimeSlice,
... FieldElevation?, ...)
aixm_AirportHeliportTimeSlice(graph:'149_donlon-data.ttl', s1:'AHP_EADH',
... xval("18"^^xsd:'string',"M"^^xsd:'string'), ...).
```

Listing 15: Example data handling of values with an unit of measurement

```
aixm:AirportHeliportTimeSlice
        a             sh:NodeShape , aixm:FeatureNodeShape , rdfs:Class ;
        sh:and        ( aixm:AIXMTimeSlice ) ;
...
        sh:property   [ sh:maxCount  1 ;
                        sh:node      aixm:ValDistanceVerticalType ;
                        sh:order     9 ;
                        sh:path      aixm:fieldElevation
                      ] ;
...
aixm:ValDistanceVerticalType
        a             aixm:DataTypeNodeShape , sh:NodeShape ;
        sh:and        ( aixm:ValDistanceVerticalBaseType ) ;
        sh:property   [ sh:maxCount  1 ;
                        sh:node      aixm:NilReasonEnumeration ;
                        sh:order     2 ;
                        sh:path      aixm:nilReason
                      ] ;
        sh:property   [ sh:maxCount  1 ;
                        sh:node      aixm:UomDistanceVerticalType ;
                        sh:order     1 ;
                        sh:path      aixm:uom
                      ] ;
        sh:property   [ sh:maxCount  1 ;
                        sh:order     1 ;
                        sh:path      rdf:value
                      ] ;
        sh:xone       ( [ sh:property  [ sh:minCount  1 ;
                                         sh:order     1 ;
                                         sh:path      rdf:value
                                       ] ;
                          sh:property  [ sh:order  2 ;
                                         sh:path   aixm:uom
                                       ]
                        ]
                        [ sh:property  [ sh:minCount  1 ;
                                         sh:order     1 ;
                                         sh:path      aixm:nilReason
                                       ]
                        ]
                      ) .
```

Listing 16: Snippet of SHACL shapes from `donlon-shacl.ttl`: `aixm:AirportHeliportTimeSlice` and `aixm:ValDistanceVerticalType` with a property having an unit of measurement

4.5 Indeterminate Position and DateTime

gml:TimePrimitive is defined as a BasicElementNodeShape with the properties gml:indeterminatePosition (which is an enumeration) and rdf:value (which is in this case a xsd:dateTime). If data with a gml:indeterminatePosition is mapped, the results will have the format **indeterminate(Value)**. If data with a xsd:dateTime is mapped, the results will have the format:

```
val(date_time(Year,Month,Day,Hour,Minutes,Seconds,Milliseconds)^^xsd:'dateTime').
```

Example: As shown in Listing 17, gml:TimePeriod has 2 properties gml:beginPosition and gml:endPosition, which are of type gml:TimePrimitive.

```
gml:TimePeriod  a     rdfs:Class , aixm:BasicElementNodeShape , sh:NodeShape ;
        sh:property   [ sh:maxCount   1 ;
                        sh:minCount   1 ;
                        sh:node       gml:TimePrimitive ;
                        sh:order      2 ;
                        sh:path       gml:endPosition
                      ] ;
        sh:property   [ sh:maxCount   1 ;
                        sh:minCount   1 ;
                        sh:node       gml:TimePrimitive ;
                        sh:order      1 ;
                        sh:path       gml:beginPosition
                      ] .
```

Listing 17: SHACL shape of gml:TimePeriod from donlon-shacl.ttl

Given the data shown in Listing 18, the resulting fact and mapping of the xsd:dateTime and gml:indeterminatePosition will look as shown in Listing 19:

```
s1:vtnull0
  a gml:TimePeriod;
  gml:beginPosition [rdf:value "2009-01-01T00:00:00Z"^^xsd:dateTime];
  gml:endPosition [gml:indeterminatePosition "unknown"].
```

Listing 18: Example data from donlon-data.ttl of gml:TimePeriod with an xsd:dateTime and a gml:indeterminatePosition

```
% gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition)
gml_TimePeriod(graph:'444_donlon-data.ttl', s1:'vtnull0',
val(date_time(2009, 1, 1, 0, 0, 0)^^xsd:'dateTime'),
indeterminate("unknown"^^xsd:'string')).
```

Listing 19: Data handling and mapping result of a date time and an indeterminate position

4.6 Missing Values

SHACL properties with no `sh:minCount` or a `sh:minCount<1`, are optional. This means that it is not mandatory to define this property in the data. If such a property is missing, we decided to fallback to the atom **'$null$'**, which is also used for variables that are unbound in SPARQL, when using the SWI-Prolog predicate `sparql_query`[1].

Example: As shown in Listing 20, `aixm:AirportHeliportTimeSlice` has an optional property `aixm:name`.

```
aixm:AirportHeliportTimeSlice
        a             sh:NodeShape , aixm:FeatureNodeShape , rdfs:Class ;
        sh:and        ( aixm:AIXMTimeSlice ) ;
...
        sh:property   [ sh:maxCount  1 ;
                        sh:node      aixm:TextNameType ;
                        sh:order     2 ;
                        sh:path      aixm:name
                      ] ;
...
```

Listing 20: Snippet of a SHACL shape from `donlon-shacl.ttl`: `aixm:AirportHeliportTimeSlice` with an optional value `aixm:name`

Listing 22 shows example data from `donlon-data.ttl` where this property is missing.

The missing value is represented by '$null$' and the final fact after mapping is showed in Listing 21.

[1] https://www.swi-prolog.org/pldoc/doc_for?object=sparql_query/3

```
% aixm_AirportHeliportTimeSlice(Graph, AirportHeliportTimeSlice,
... Name?, ...)
aixm_AirportHeliportTimeSlice(graph:'799_donlon-data.ttl', s2:'ID_ACT_11',
... '$null$', ...).
```

Listing 21: Data handling result: `aixm:AirportHeliportTimeSlice` with the missing value “Name?”

```
s2:ID_ACT_11
  a aixm:AirportHeliportTimeSlice;
  gml:validTime s2:ID_ACT_12;
  aixm:interpretation [rdf:value "TEMPDELTA"];
  aixm:sequenceNumber [rdf:value "1"^^xsd:unsignedInt];
  aixm:correctionNumber [rdf:value "0"^^xsd:unsignedInt];
  aixm:availability s2:ID_ACT_13;
  aixm:extension s2:ID_ACT_211;
  aixm:designatorIATA [rdf:value "ysdf"];
  aixm:servedCity s2:city1, s2:city2.
```

Listing 22: Example data from `donlon-data.ttl`: `aixm:AirportHeliportTimeSlice` without the property name

4.7 Lists

SHACL properties with having no `sh:maxCount=1` or having a `sh:maxCount` above 1 are concatenated and handled as lists in our mapping variants.

Example: `aixm:AirportHeliportTimeSlice`, which can be seen in Listing 23, has a property `aixm:servedCity`. This property has no `sh:maxCount`, therefore multiple values are permitted.

```
aixm:AirportHeliportTimeSlice
        a            sh:NodeShape , aixm:FeatureNodeShape , rdfs:Class ;
        sh:and       ( aixm:AIXMTimeSlice ) ;
...
        sh:property  [ sh:class  aixm:City ;
                       sh:order  166 ;
                       sh:path   aixm:servedCity
                     ] ;
...
```

Listing 23: Example SHACL shape from `donlon-shacl.ttl` with a list as a property

Listing 24 is a snippet from donlon-data.ttl and shows an `aixm:AirportHeliportTimeSlice` with 2 cities.

```
  s2:ID_ACT_11
    a aixm:AirportHeliportTimeSlice;
...
    aixm:servedCity s2:city1, s2:city2.
```

Listing 24: Example data from `donlon-data.ttl`: `aixm:AirportHeliportTimeSlice` with two cities.

The final fact after mapping can be seen in Listing 25.

```
% aixm_AirportHeliportTimeSlice(Graph, AirportHeliportTimeSlice,
... ServedCity*, ...)
aixm_AirportHeliportTimeSlice(graph:'799_donlon-data.ttl', s2:'ID_ACT_11',
... [s2:'city1', s2:'city2'], ...).
```

Listing 25: Data handling result: `aixm:AirportHeliportTimeSlice` with a list of cities.

As shown in Listing 26, an empty list is represented by "[]" if no data for a list is available.

```
% aixm_AirportHeliportTimeSlice(Graph, AirportHeliportTimeSlice,
... Contaminant*, ServedCity*, ...)
aixm_AirportHeliportTimeSlice(graph:'799_donlon-data.ttl', s2:'ID_ACT_11',
... [], [s2:'city1', s2:'city2'], ...).
```

Listing 26: Data handling result: `aixm:AirportHeliportTimeSlice` with an empty list of contaminants.

Other previously mentioned data type mapping rules are also considered within lists. Listing 27 shows an example for a list of values without an unit of measurement.

```
[val("D1"^^xsd:'string'), val("L"^^xsd:'string'), val("B1"^^xsd:'string')]
```

Listing 27: Data handling result: list of values without an unit of measurement

4.8 Summary

In this chapter, we discussed how data handling is achieved differently depending on the mapping variant. Mapping variant A handles the data types in the SPARQL queries and processes the results in Java. Mapping variant B also handles the data types in the SPARQL query, but in comparison to mapping variant A also takes care of the processing

within the Prolog modules. Mapping variant C takes care of the data handling and processing within the generated Prolog rules. For this variant, no further processing in Prolog or Java is required.

We also discussed how specific data types, such as `aixm:NilReason`, values with and without unit of measurement, `gml:indeterminatePosition`, `gml:dateTime`, missing values and lists, are handled.

CHAPTER 5

Prototypical Implementation

This chapter focuses on the structure and execution of the prototypical implementation. Section 5.1 serves as a technical documentation, giving an overview over the GitHub repository, explaining which files are used and how to execute performance tests and tests to assure the correct functionality of all three mapping variants. Section 5.2 dives deeper into the code and shows the structure of the prototypical implementation.

5.1 Technical Documentation

This section serves as a technical documentation. Subsection 5.1.1 gives an overview over the GitHub repository. Subsection 5.1.2 explains how to run the preliminary performance studies. Subsection 5.1.3 describes, which input files are used for mapping and Subsection 5.1.4 shows the output files, which depend on the mapping variant. Subsection 5.1.5 describes what the performance results look like and Subsection 5.1.6 how to run the performance tests. Finally, Subsection 5.1.7 explains how it is assured via tests that all three mapping variants deliver the same result.

5.1.1 Overview of GitHub repository

Java code, Prolog programs, example data and schemas (RDF, RDFS and SHACL) are available in a GitHub repository[1] [git].

The repository consists of the following projects and dependencies:

- at.jku.dke.aisa.kg.sample.adsb
  - depends on: at.jku.dke.aisa.kg

[1] https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper/

  - This bundle provides a sample KG system and is part of the AISA Proof-of-Concept KG system.

- at.jku.dke.aisa.kg.sample.prolog
  - depends on: at.jku.dke.aisa.kg
  - This bundle is part of the AISA Proof-of-Concept KG system. It initializes all modules registered with the KG system.

- at.jku.dke.aisa.kg
  - depends on: at.jku.dke.aisa.mapperC
  - This is the AISA Proof-of-Concept KG system which makes use of mapping variant C for reading the KG and the schema-oblivious mapping for writing to the KG. It builds on Apache Jena Fuseki and facilitates the modularization of the management of a KG with static and dynamic data.

- at.jku.dke.aisa.mapperA
  - This bundle contains mapping variant A,which generates with the help of SPARQL queries a set of Prolog predicates from the RDFS/SHACL schema and a SPARQL query for each predicate. The result of executing the SPARQL query gives the facts for the respective predicate. These facts are written to a file and that file can be loaded into Prolog.

- at.jku.dke.aisa.mapperB
  - This bundle contains mapping variant B1. The Java program generates a Prolog module from the RDFS/SHACL schema with the predicates that are linked to the respective SPARQL queries by means of Prolog rules. The SPARQL queries for filling the predicates are only executed from Prolog at runtime.

- at.jku.dke.aisa.mapperB2
  - This bundle contains the improved mapping variant B2. In comparison to mapping variant B1, each SPARQL query is only executed once, which increases the performance significantly.

- at.jku.dke.aisa.mapperC
  - This bundle contains the mapping variant C. The relevant part of the Knowledge Graph is replicated in the main-memory RDF database of SWI-Prolog. The actual mapping is then formulated as Prolog rules with the RDF quadruples in the RDF database as input.

Additionally, the installation of SWIPL [swi] is required, which can be downloaded from `https://www.swi-prolog.org/download/stable`. The default installation path "`C:\Program Files\swipl`" should be chosen. After installation, the system environment variable must be set for the bin folder of SWIPL (Variable "Path" with value "`C:\Program Files\swipl\bin`" for Windows).

In order to run the sample KG system with Prolog integration:

1. Start Jena Fuseki with any of the given configurations (e.g.: `AISA-fuseki-server-mem.bat`) or by configuring the Jena Fuseki server in Eclipse.
2. Run at.jku.dke.aisa.kg.sample.prolog.KGSystem
3. The output can be found in the fileoutput folder of the project.

Note: Jena Fuseki 3.17 does not work, rather use the given version 3.16 or try with the latest.

5.1.2 Running the preliminary performance studies

Before starting, make sure that swipl is installed and the system environment Path variable points to "`C:\Program Files\swipl\bin`".

1. Add all prefixes, data and SHACL files, which should be mapped, to the input folder.
2. Start Jena Fuseki with any of the given configurations (e.g.: `AISA-fuseki-server-mem.bat`) or by configuring the Jena Fuseki server in Eclipse.
3. Run Shacl2PrologLauncher (of the preferred mapping variant) in Eclipse.
4. The output can be found in the output folder.

Note: In case you want to restart the mapping, make sure to close the Jena Fuseki server first if started manually.

5.1.3 Input Files

There are 3 input files, which are used for mapping: the SHACL schema, the data, and the prefixes. All of those input files can be found in the input folder of each mapping bundle (e.g.: `/at.jku.dke.aisa.mapperA/input/`).

The KG-Schema can be defined in multiple RDFS/SHACL files, which are unioned into the Schema-Named-Graph in the KG. There must not be an overlap between these files, i.e., every SHACL shape must be defined in exactly one RDFS/SHACL file, otherwise definitions in blank nodes get duplicated.

SHACL schema

The SHACL schema is uploaded to the Jena Fuseki server at runtime. The SHACL schema is used to create a SHACL graph and parse SHACL shapes. The graph and the shapes are needed for the creation of the SPARQL queries and the Prolog facts.

All SHACL schema files, which are in the `/input/schema/` folder are uploaded to the Jena Fuseki server and be used for mapping.

Example file: `/input/schema/donlon-shacl.ttl`

Data

The data is uploaded to the Jena Fuseki server at runtime. The data is later retrieved by the SPARQL queries and the results processed to Prolog facts.

All data files which are in the `/input/data/` folder are uploaded to the Jena Fuseki server and be used for mapping.

Example file: `/input/data/donlon-data.ttl`

Namespace Prefixes

The prefix files should contain all prefixes which are used in the SHACL schema and in the data file. These prefixes are mainly required to abbreviate the URIs in the resulting fact files.

Example file: `/input/prefixes.ttl`

5.1.4 Output Files

Depending on which variant is executed, the mapper outputs 1 or 2 files, which can be found in the output folder: the SPARQL queries and the Prolog facts. Mapping variant A is the only mapper, which outputs the SPARQL queries and the Prolog facts at runtime in separate files. Mapping variants B and C only output Prolog rules in the facts file, which can later be loaded into Prolog.

SPARQL Queries

The SPARQL queries are generated at runtime, saved to a file and executed. The results of the query execution are used for Prolog facts generation.

Example file: `/output/queries.sparql`

Prolog facts and rules

Mapping variant A: The Prolog facts are generated and saved to a file at runtime by processing the results of the SPARQL queries and using the prefix mapping defined in the prefix file.

Mapping variants B and C: The Prolog rules are generated and saved to a file at runtime by using the given SHACL schema and the prefix mapping defined in the prefix file.

Example file: `/output/facts.pl`

5.1.5 Performance results

At the end of the execution of the mapping, the execution time is saved to a file. Next to the overall execution time and the execution time of separate parts of the mapping, also the number of data copies and the mapping variant is saved to the performance result. The number of data copies can be changed in the Shacl2PrologLauncher if required. (Default value=1)

Example file: `/output/performance_results.csv`

The content of a performance result, excluding the current time in milliseconds, looks alike the following:

Current time milliseconds	...	...	...	...	...	...
Mapping Variant	C	C	C	C	C	C
Number of data copies	1	10	100	200	300	1000
Jena Fuseki connection establishment	3945	3872	3944	3918	4130	3903
Loading shacl schema files	1753	2018	1718	2026	1745	1774
Loading data files	72	519	3091	6207	7732	25902
Fetching shacl schema	834	836	883	940	902	1084
Fetching and writing data set	2042	2356	4947	6251	9118	24250
Creating KnowledgeGraphClasses and KnowledgeGraphProperties	97	80	109	78	68	78
Creating facts file	82	74	119	64	77	62
Consult Program	611	717	1033	1682	2555	13699
Invoke run/0 in Prolog	6	20	150	345	524	2723
Invoke save/0 in Prolog	74	75	85	98	150	235
Load saved results to Fuseki	26	18	9	10	17	24
Execution time in milliseconds	9542	10585	16088	21619	27018	73734

5.1.6 How to start the performance tests

1. Go to Eclipse File->Export->Runnable Jar File.
2. Chose the main class of the mapping variant and the dedicated name (MapperA.jar|MapperB.jar|MapperC.jar) and select library handling “Package required libraries into generated JAR”.
3. Export the jar file into the project folder of the respective mapping variant. (e.g.: `/AISA-KG-Prolog-Mapper/at.jku.dke.aisa.mapperA/MapperA.jar`)
4. Make sure that Jena Fuseki server inclusive the right configuration files are in place (e.g.: `/AISA-KG-Prolog-Mapper/at.jku.dke.aisa.mapperA/apache-jena-fuseki-3.16.0/AISA-fuseki-server-mem.bat`)
5. Start `start_performance_test.bat`.
6. The results can be found in the output folder of the dedicated mapping variant.

5.1.7 Testing mapping variants by comparing resulting input facts

To check the proper functioning of the three mapping variants, we wrote a short test script in Prolog that checks wheter the different mapping variants produce the same facts. The test script text.pl can be found in the root directory of the repository and covers all kind of data types and known edge cases. Before the facts.pl files, which were generated by the three mapping variants, can be used for the tests, a small modification is necessary. One line has to be added to the beginning of each file so that each Prolog

program is loaded into a separate Prolog module:

`:- module(a,[]).` to factsA.pl

`:- module(b,[]).` to factsB.pl

`:- module(c,[]).` to factsC.pl

The modified files factsA.pl, factsB.pl, and factsC.pl can also be found in the root directory of the repository. At the beginning of the test script, all facts files (factsA.pl, factsB.pl and factsC.pl) are loaded. Then Prolog methods are called, which will show up differences between the results of the files. Listing 28 shows an example of how the results of the 3 facts files are compared within the test.pl file.

```
mismatch(inA_notinB,gml_TimePeriod(Graph, TimePeriod, BeginPosition,
↪  EndPosition)) :-
  a:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition),
  \+ b:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition).

mismatch(inB_notinA,gml_TimePeriod(Graph, TimePeriod, BeginPosition,
↪  EndPosition)) :-
  b:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition),
  \+ a:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition).

mismatch(inA_notinC,gml_TimePeriod(Graph, TimePeriod, BeginPosition,
↪  EndPosition)) :-
  a:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition),
  \+ c:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition).

mismatch(inC_notinA,gml_TimePeriod(Graph, TimePeriod, BeginPosition,
↪  EndPosition)) :-
  c:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition),
  \+ a:gml_TimePeriod(Graph, TimePeriod, BeginPosition, EndPosition).
```

Listing 28: Example from `test.pl`

If the test succeeds, which means that all 3 facts files show the same results, the output looks as follows:

```
?- mismatch(X,Y).
false.
```

If the test returns “true”, it means that there are differences in the 3 facts files. Those differences are also output in the console and can be checked for adaption of the KG-Prolog mapper.

5.2 Implementation of the KG-Prolog-Mapper

This section goes more into detail about the implementation of the KG-Prolog-Mapper. Generally, all mapping variants are built up equally. They consist of Java classes named Shacl2PrologLauncher.java, Mapper.java, ShaclUtil.java, AISAUtil.java, KnowledgeGraphClass.java and KnowledgeGraphProperty.java, but the implementation differs dependent of the mapping variant. Only mapping variant A has an additional Java class QueryExecutor.java for the execution of the generated queries. Figure 5.1 shows a class diagram of the KG-Prolog mapper including the QueryExecutor class from variant A. Java classes with only static methods, namely AISAUtil.java and ShaclUtil.java are not displayed on the class diagram.

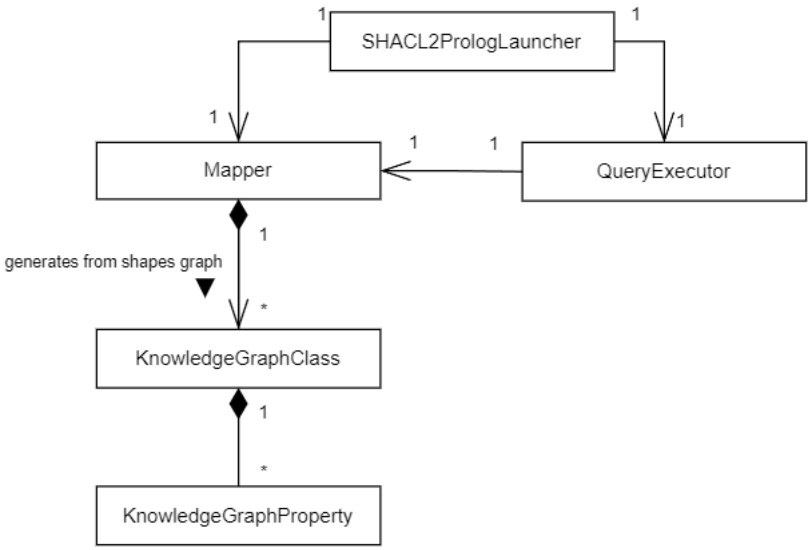


Figure 5.1: Class diagram of the KG-Prolog mapper including the `QueryExecutor.java` class, which is only part of variant A

The following subsections describe the implementation details for each Java class. Subsection 5.2.1 describes the main classes for starting the mapping variants. Subsection 5.2.2 explains the classes, which execute the mapping. Subsection 5.2.3 presents a utility class used to create SHACL resources and nodes for searching subjects, properties and objects in the shape graphs and Subsection 5.2.4 shows a utility class used to create AISA resources and nodes used for finding different types of `NodeShapes` in the shape graphs. Subsection 5.2.5 describes how a `NodeShape` of the SHACL graph is represented and Subsection 5.2.6 shows how a `PropertyShape` is represented in Java. Subsection 5.2.7 presents the query executor, which is only used in mapping variant A.

5.2.1 Shacl2PrologLauncher.java

The `Shacl2PrologLauncher.java` class is the main class to map SHACL shapes and data to SPARQL queries and Prolog facts. An additional prefix file is used for

the prefix mapping of the facts. This class takes SHACL shapes, data and prefixes as input files and produces SPARQL queries and Prolog facts as output files. This class also holds other parameters such as the number of data copies, which can be configured. When the main method of this class is executed, a connection to Jena Fuseki server is established and the input data and schema are uploaded to the server. Then the schema is fetched from the server and the SHACL shapes parsed, which are then forwarded to the `Mapper.java` class.

For mapping variant A, the next step is to create the SPARQL file, which is done with the help of the Mapper. Next, a file for the facts is created. The file is filled with content by the `Mapper.java` and the `QueryExecutor.java` classes. The Mapper prints all of the static content and inheritance rules and the QueryExecutor takes care of the facts.

For mapping variant B, the Prolog file with the embedded queries, static content the Prolog modules and rules as well as the inheritance rules is created and filled by the `Mapper.java` class.

For mapping variant C, the facts file containing all the static content, the facts rules and inheritance rules is filled by the `Mapper.java` class.

Independently of the mapping variant, the performance results are output to the console and saved to a file at the end of the execution.

5.2.2 Mapper.java

The main task of the Mapper is to create SPARQL queries (mapping variant A,B), Prolog facts (mapping variant A) and Prolog rules (mapping variant B,C). It takes SHACL graph, parsed SHACL shapes, prefixes and the graph name as input. For each target shape as input a `KnowledgeGraphClass.java` class is created and for each property shape of the node shape a `KnowledgeGraphProperty.java` class. `KnowledgeGraphClass.java` and `KnowledgeGraphProperty.java` are used for easier creation of SPARQL queries, Prolog rules and Prolog facts. The constructor of the Mapper creates all required variables for later use, such as the PrefixMapping, the super classes and all instances of KnowledgeGraphclass and KnowledgeGraphProperty. The PrefixMappings are ordered by the length of the URI in descending order. The Mapper has methods which print static content, which is necessary to execute the output file in Prolog. The content of this static content differs depending on the mapping variant.

The Mapper of mapping variant A generates the facts in the generateFact method, which takes the querySolution of the queryExecutor as input and creates Prolog facts out of it. For the generation, the dedicated SHACL schema is required to make use of information such as the subClasses.

The Mapper of mapping variant B generates and prints the Prolog rules for each instance of KnowledgeGraphClass.

The Mapper of mapping variant C prints the Prolog fact rule for each instance of KnowledgeGraphClass.

5.2.3 ShaclUtil.java

The `ShaclUtil.java` class is the same for each mapping variant. It is a utility class used to create SHACL resources and nodes for searching subjects, properties and objects in the shape graphs.

Listing 29 shows an example usage for retrieving the `sh:order` of a property.

```
public class ShaclUtil {
...
        public static Resource SHACL_ORDER_RESOURCE = //
            ResourceFactory.createResource("http://www.w3.org/ns/shacl#order");
...
        public static Node shaclOrderAsNode() {
                return SHACL_ORDER_RESOURCE.asNode();
        }
...
}

// returns all triples with the matching pattern
// in this example it returns the sh:order of a property
property.getShapeGraph()//
    .find(property.getShapeNode(), ShaclUtil.shaclOrderAsNode(), null);
```

Listing 29: Example usage for retrieving the `sh:order` of a property

5.2.4 AISAUtil.java

The `AISAUtil.java` class is the same for each mapping variant. It is a utility class used to create AISA resources and nodes used for finding different types of NodeShapes in the shape graphs.

Listing 30 shows an example usage for retrieving an `ObjectNodeShape` in the shape graphs.

5.2.5 KnowledgeGraphClass.java

The `KnowledgeGraphClass.java` class represents a `NodeShape` of the SHACL graph. For each PropertyShape of the given shape, a `KnowledgeGraphProperty.java` instance is created which is linked to this KnowledgeGraphClass. The respective SPARQL queries (for mapping variant A,B) and the Prolog schema (for all variants) are created at instantiation time of this class.

5.2.6 KnowledgeGraphProperty.java

An instance of `KnowledgeGraphProperty.java` represents a `PropertyShape`, which belongs to a KnowledgeGraphClass of the SHACL graph. At instantiation time,

```
public class AISAUtil {

        public static Resource OBJECT_NODE_SHAPE = //
            ResourceFactory.createResource(//
            "http://www.aisa-project.eu/vocabulary/aixm_5-1-1#ObjectNodeShape");

...
        public static Node objectNodeShapeAsNode() {
                return OBJECT_NODE_SHAPE.asNode();
        }
...
}

// returns all triples with the matching pattern
// in this example it returns the ObjectNodeShapes of the given shape graph
shaclGraph.find(null, null, AISAUtil.objectNodeShapeAsNode());
```

Listing 30: Example usage for retrieving an `ObjectNodeShape` in the SHACL shapes graphs

all required variables are created in the constructor.

5.2.7 QueryExecutor.java

The `QueryExecuter.java` class is only implemented for mapping variant A, as there is a SPARQL query generated for each predicate. The `Shacl2PrologLauncher.java` class creates the QueryExecutor object, which takes the SPARQL file as input and executes each query separately. After query execution, the resultSet is forwarded to the `Mapper.java` class to create the Prolog facts.

CHAPTER 6

Evaluation

This chapter presents the results of the performance studies in Section 6.1. Section 6.2 shows the integration of one of the variants into the Proof-Of-Concept KG system. Finally, Section 6.3 discusses the overall results.

6.1 Performance Studies

To get first insights into the performance characteristics of the different realization variants, we conducted initial performance measurements. These initial performance studies measure how the **execution time** for the **different mapping variants** scales with **increasing size of KG data** while the size of the KG schema remains constant.

The measured execution time include the loading of schema and data into the KG, the mapping generation from the schema, the execution of the generated mapping rules/queries on the KG data, and the execution of a Prolog program that accesses the mapped input facts and produces a named graph as result that is written back to the KG.

The KG schema is loaded from two files containing vocabulary and structural constraints represented in RDF schema and SHACL. The first schema file is based on a fragment of AIXM and specifies 203 SHACL node shapes of which 37 are also RDFS classes. The second schema file is based on a fragment of FIXM and specifies 267 SHACL node shapes of which 109 are also specified as RDFS classes. Based on this KG schema, the mapping generator produces the schema of 146 Prolog predicates, each with a mapping rule or query (depending on the mapping variant) to generate according input facts from the KG data.

The KG data is loaded from two RDF files. The first RDF file contains 231 RDF statements conforming with the AIXM schema fragment and the second RDF file contains 254 RDF statements conforming with the FIXM schema fragment. This corresponds to

the amount of *new* data we expect the AISA KG system has to deal with per execution round.

Parameter **number of data copies** specifies how many times each of these two files are to be loaded into the KG, with each copy being stored in a new named graph within the KG. For our preliminary performance studies, we scale the size of the KG data from 1 data copy (amounting to 485 RDF quadruples in two named graphs) to 1000 data copies (amounting to 485000 RDF quadruples in 2000 named graphs). We run the program for each realization variant with increasing number of data copies: with 1, 10, 100, 400, 700 and 1000 data copies.

The remainder of this chapter is structured as follows: Section 6.1.1 provides details on the setup of the performance studies. Sections 6.1.2, 6.1.3 and 6.1.4 discuss the measurements obtained for each of the realization variants described in the previous chapter. Section 6.1.5 summarizes the performance studies by comparing the total execution times of the different variants.

6.1.1 Setup of Performance Studies

The following performance measurements were generated by running start_performance_test.bat (which can be found in the bundles of the mappers on GitHub) for each variant on a computer with an Intel©Core™i3-2130 CPU @3.40 GHz 3.40 GHz, 2 kernels, 4 logical processors running with 8 GB of physical RAM, running Windows 10 Pro.

In order to test the performance of each mapping variant, we created a constant `NUMBER_OF_DATA_COPIES`, which is by default 1. This constant indicates how many data copies of the data are used for one execution of the mapper. In detail, there is an iteration around the code, which uploads the data of the input folder to Jena Fuseki server. If the number is above 1, the data is uploaded multiple times. Additionally, the number of data copies can be overwritten by adding the number as an argument when starting the SHACL2PrologLauncher. The use of this constant makes it easier to test the performance and scalability of the mapper with only a limited amount of defined data.

Our input consists of 2 SHACL schema and 2 data files. The AIXM schema, which can be found in `/input/schema/donlon-shacl.ttl`, consists of 203 `sh:Nodeshapes`. 37 of these `sh:NodeShapes` are `rdfs:Classes` and therefore target shapes relevant for mapping. The FIXM schema, which can be found in `/input/schema/FIXM_EDDF_VHHH.ttl`, consists of 267 `sh:NodeShapes`. 109 of these `sh:NodeShapes` are `rdfs:Classes`, therefore target shapes and also relevant for mapping. The AIXM data, which can be found in `/input/data/donlon-data.ttl`, consists of 33 resources. The FIXM data, which can be found in `/input/data/FIXM_EDDF_VHHH.ttl`, consists of 40 resources.

For the purpose of testing the mapper automatically with different number of data copies, we created performance test scripts and exported .JAR files for each mapping variant.

We created the JAR file using Eclipse: File -> Export... -> Runnable JAR file. As launch configuration, the Shacl2PrologLauncher of the desired mapping variant has to be chosen. The export destination should be the bundle of the selected mapper. For our test JARs, we selected as library handling to package required libraries into generated JAR.

The `start_performance_test.bat` file, which can be found in each mapping variant bundle, first starts the Jena Fuseki server, executes the mapper with a specific number of data copies and stops the Jena Fuseki server. This step is iterated with the number of data copies 1, 10, 100, 400, 700 and 1000. Restarting Jena Fuseki server before every execution run of a mapping variant is needed in order to get proper results.

One run of the mapper is called by the performance test script the following way:

```
java -jar MapperA.jar 100
```

As already mentioned, the default number of data copies is 1 and one data copy contains 485 RDF statements in two named graphs. In this example, the number of data copies is overwritten by the launch argument 100. That means that the data in the input folder is uploaded to Jena Fuseki 100 times and consequently, we use a hundred times more data than by using the default number of data copies, namely 485000 RDF quadruples in 200 named graphs.

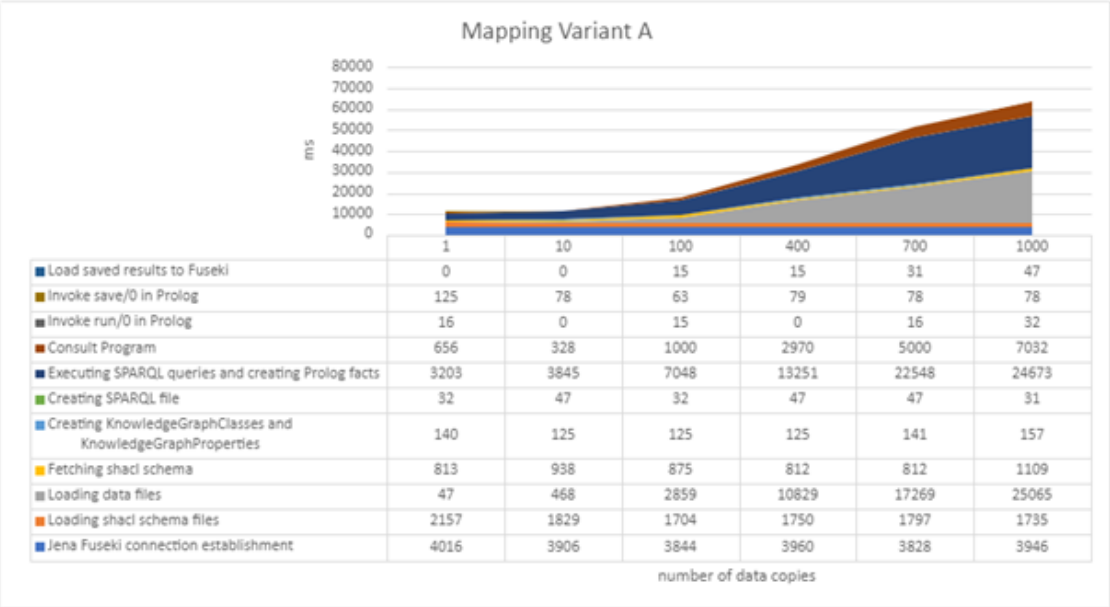
Besides the total execution time, which is interesting for comparing the mapping variants and the overall scalability, we divided the mapper in logical sections and measured the execution time of each part. Some of these parts are dependent on the mapping variant or the number of data copies, others are equally in terms of execution time for every variant. Mapping, mapping variant and data copy independent parts are “Jena Fuseki connection establishment” and “Loading shacl schema files”. “Loading data files” is only dependent on the data copies, but independent of the mapping and the specific mapping variant. The step “Fetching shacl schema” and “Creating KnowledgeGraphClasses and KnowledgeGraphProperties” are mapping specific, but independent from the number of data copies.

The performance of each variant dependent part is described more in detail in the following subsections.

6.1.2 Performance Studies of Mapping Variant A

In order to conduct preliminary performance studies, we ran mapping variant A with 1, 10, 100, 400, 700 and 1000 data copies. Overall, the performance of mapping variant A is decent and scales good with higher numbers of data copies. To get better insight, what exactly takes the most time, we measured the time of 11 separate parts of the program. Figure 6.1 shows the results of running the mapper with different numbers of data copies for each step in milliseconds:

- Jena Fuseki connection establishment: This step is data copy, mapping and variant independent. This means that the execution time of this step is on average equally for all mapping variants.
- Loading SHACL schema files: This step is data copy, mapping and variant independent. This means that the execution time of this step is on average equally for all mapping variants.
- Loading data files: This step is only dependent on the number of data copies, but independent from the mapping and mapping variant.
- Fetching SHACL schema: This step is mapping specific, but independent from the number of data copies.
- Creating KnowledgeGraphClasses and KnowledgeGraphProperties: This step is mapping-specific, but independent from the number of data copies.
- Creating SPARQL file: This step is dependent on the mapping variant. However, it does not take much time as the SPARQL queries are already generated at instantiation time of the mapper.
- Executing SPARQL queries and creating Prolog facts: This step is dependent on the mapping variant and on the number of data copies. For each target shape defined in /input/schema/, there is a SPARQL query, which will be executed in this step. Each row in the result set describes one resource of the input data from /input/data/ and will be processed into one fact. This means that the execution time of this step depends on the number of data copies (from 1 to 1000 data copies with 485 to 485000 RDF statements in 2 to 2000 named graphs) and consequently, the number of mapped input facts which will be generated (from 106 to 106000 Prolog facts).
- Consult Program: This step is dependent on the mapping variant. Consulting /output/program.pl reads program.pl as a Prolog resource and loads the previously generated facts.pl file into Prolog. The execution time of this step scales with the number of data copies, which increases the number of facts in the facts.pl file. The size of the generated Prolog program (facts.pl), which contains not only the generated facts, but also the predicate schema and the inheritance rules, ranges from 87 KB to 25.8 MB.
- Invoke run/0 in Prolog: This step is dependent on the mapping variant.
- Invoke save/0 in Prolog: This step is independent from the mapping variant. This Prolog method saves the content of the given graph to /output/output.ttl
- Load saved results to Fuseki: This step is independent from the mapping variant. This Prolog method loads the content of /output/output.ttl with the given graph to Fuseki.



	1	10	100	400	700	1000
Load saved results to Fuseki	0	0	15	15	31	47
Invoke save/0 in Prolog	125	78	63	79	78	78
Invoke run/0 in Prolog	16	0	15	0	16	32
Consult Program	656	328	1000	2970	5000	7032
Executing SPARQL queries and creating Prolog facts	3203	3845	7048	13251	22548	24673
Creating SPARQL file	32	47	32	47	47	31
Creating KnowledgeGraphClasses and KnowledgeGraphProperties	140	125	125	125	141	157
Fetching shacl schema	813	938	875	812	812	1109
Loading data files	47	468	2859	10829	17269	25065
Loading shacl schema files	2157	1829	1704	1750	1797	1735
Jena Fuseki connection establishment	4016	3906	3844	3960	3828	3946

Figure 6.1: Performance results of mapping variant A. KG data size scaled from 1 data copy (485 RDF quadruples) to 1000 data copies (485000 quadruples).

6.1.3 Performance Studies of Mapping Variant B

In order to conduct preliminary performance studies, we ran mapping variant B with 1, 10, 100, 400, 700 and 1000 data copies. Overall, the performance of mapping variant B scales very bad with the number of data copies. To get better insight, what exactly takes the most time, we measured the time of 10 separate parts of the program. Figure 6.2 shows the results of running the mapper with different number of data copies for each step in milliseconds:

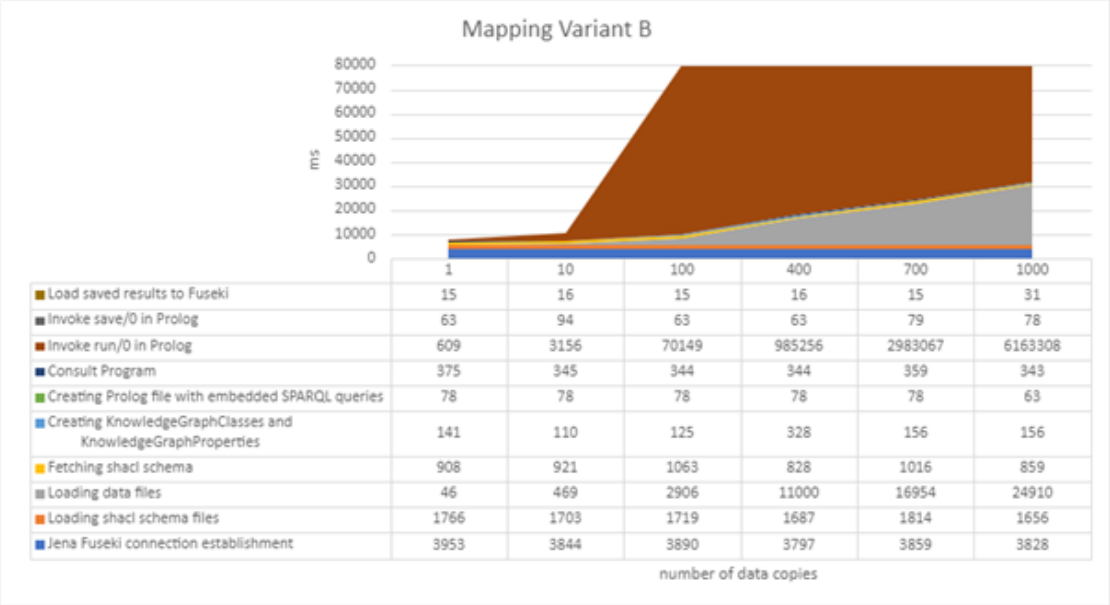
- Jena Fuseki connection establishment: (same as in variant A).
- Loading SHACL schema files: (same as in variant A).
- Loading data files: (same as in variant A).
- Fetching SHACL schema: (same as in variant A).
- Creating KnowledgeGraphClasses and KnowledgeGraphProperties: This step is mapping-specific, but independent from the number of data copies.
- Creating Prolog file with embedded SPARQL queries: This step is mapping-specific for variant B, but does not take much time in total, because information, which was already processed in the previous step, is combined and written out to /output/facts.pl.

- Consult Program: This step is dependent on the mapping variant. Consulting /output/program.pl reads the file as a Prolog resource and loads the content of the previously generated facts.pl file. In comparison to mapping variant A, the facts.pl file does not contain facts, which scale on the number of data copies, but Prolog modules, which can be called to receive facts. This step scales on the size of the SHACL schemas, but it does not have a major impact as usually the schema does not consist of infinitely many SHACL shapes.

- Invoke run/0 in Prolog: This step is dependent on the mapping variant and takes the most time of variant B. For each iteration (each fact) of the Prolog method run, the respective Prolog rules with the embedded SPARQL queries for the facts have to be called. This means that queries are posed multiple times, which increases the exectution time heavily.

- Invoke save/0 in Prolog: (same as in variant A)

- Load saved results to Fuseki: (same as in variant A).

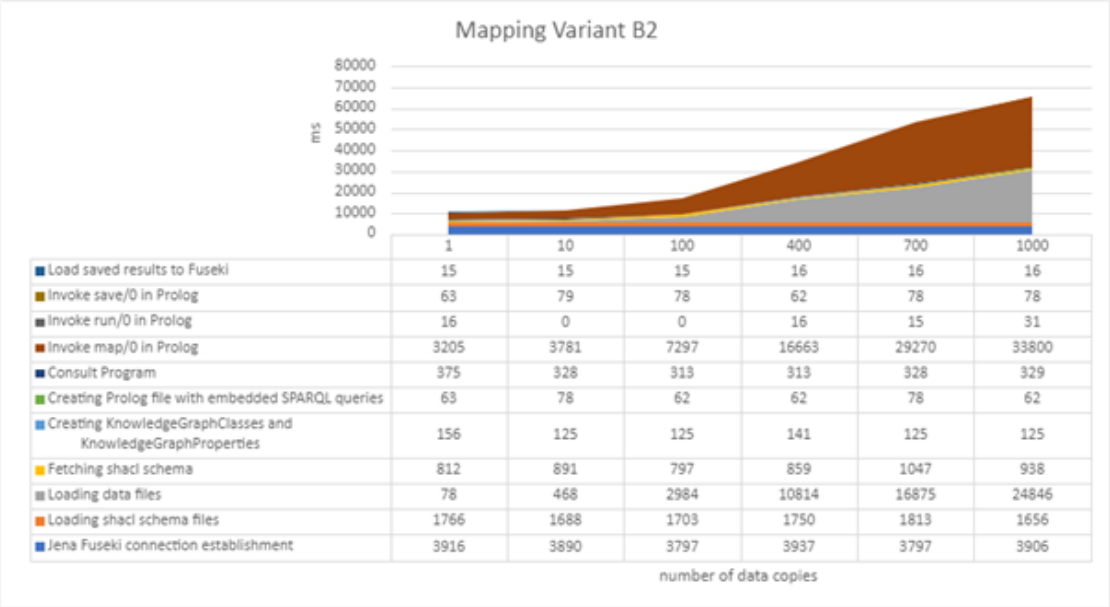


	1	10	100	400	700	1000
Load saved results to Fuseki	15	16	15	16	15	31
Invoke save/0 in Prolog	63	94	63	63	79	78
Invoke run/0 in Prolog	609	3156	70149	985256	2983067	6163308
Consult Program	375	345	344	344	359	343
Creating Prolog file with embedded SPARQL queries	78	78	78	78	78	63
Creating KnowledgeGraphClasses and KnowledgeGraphProperties	141	110	125	328	156	156
Fetching shacl schema	908	921	1063	828	1016	859
Loading data files	46	469	2906	11000	16954	24910
Loading shacl schema files	1766	1703	1719	1687	1814	1656
Jena Fuseki connection establishment	3953	3844	3890	3797	3859	3828

Figure 6.2: Performance results of mapping variant B1

In order to improve mapping variant B, we went for an approach to bypass the case that requests and queries are posed multiple times for the same facts. For this purpose, we created a map/0 method, which contains all SPARQL queries and asserts the facts into the database after querying. In comparison to the normal mapping variant B, the performance improves considerably and is now on average equally fast than mapping variant A and C. Figure 6.3 shows the performance of the improved mapping variant B2.

The most interesting part of this figure is “Invoke map/0 in Prolog”, which now takes around 31664 ms instead of 6227077 ms as for variant B1 for 1000 data copies.



	1	10	100	400	700	1000
Load saved results to Fuseki	15	15	15	16	16	16
Invoke save/0 in Prolog	63	79	78	62	78	78
Invoke run/0 in Prolog	16	0	0	16	15	31
Invoke map/0 in Prolog	3205	3781	7297	16663	29270	33800
Consult Program	375	328	313	313	328	329
Creating Prolog file with embedded SPARQL queries	63	78	62	62	78	62
Creating KnowledgeGraphClasses and KnowledgeGraphProperties	156	125	125	141	125	125
Fetching shacl schema	812	891	797	859	1047	938
Loading data files	78	468	2984	10814	16875	24846
Loading shacl schema files	1766	1688	1703	1750	1813	1656
Jena Fuseki connection establishment	3916	3890	3797	3937	3797	3906

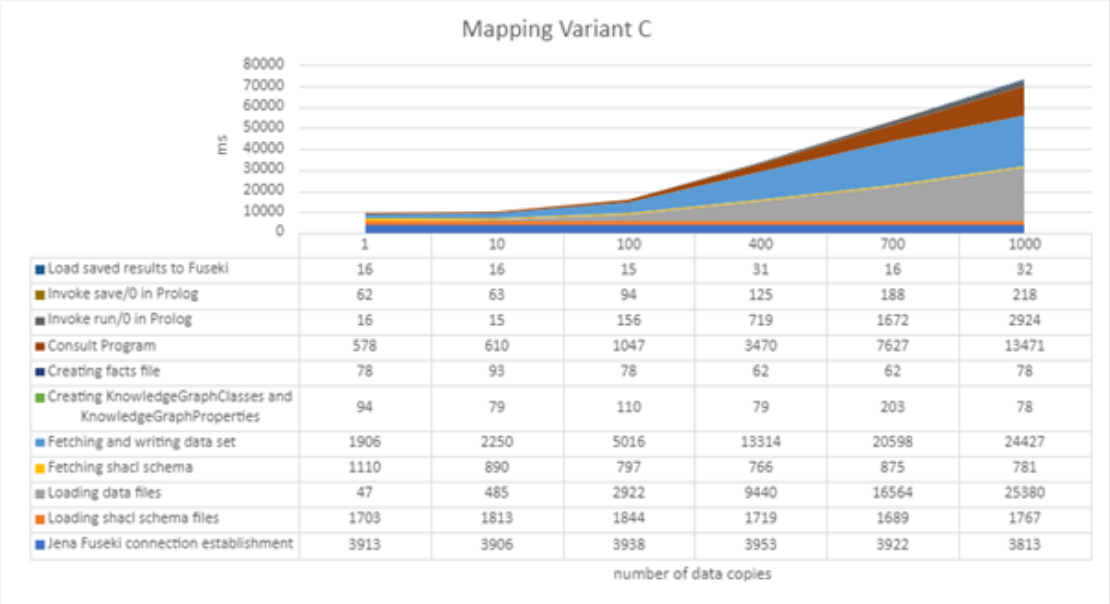
Figure 6.3: Performance results of mapping variant B2

6.1.4 Performance Studies of Mapping Variant C

In order to conduct preliminary performance studies, we ran mapping variant C with 1, 10, 100, 400, 700 and 1000 data copies. Overall, the performance of mapping variant C is decent and scales good with higher numbers of data copies. To get better insight, what exactly takes the most time, we measured the time of 11 separate parts of the program. Figure 6.4 shows the results of running the mapper with different number of data copies for each step in milliseconds:

- Jena Fuseki connection establishment: (same as in variant A).
- Loading SHACL schema files: (same as in variant A).
- Loading data files: (same as in variant A).
- Fetching SHACL schema: (same as in variant A).
- Fetching and writing data set: This step is mapping-specific for variant C and dependent from the number of data copies. The data is fetched from Fuseki and written to /output/dataset.trig.
- Creating KnowledgeGraphClasses and KnowledgeGraphProperties: This step is mapping-specific, but independent from the number of data copies as only the SHACL schemas are needed for the creation.

- Creating mapping rules: This step is dependent on the mapping variant. An import for dataset.trig and mapping rules are generated as Prolog rules and written to /output/facts.pl.
- Invoke run/0 in Prolog: This step is dependent on the mapping variant.
- Invoke save/0 in Prolog: (same as in variant A).
- Load saved results to Fuseki: (same as in variant A).



	1	10	100	400	700	1000
Load saved results to Fuseki	16	16	15	31	16	32
Invoke save/0 in Prolog	62	63	94	125	188	218
Invoke run/0 in Prolog	16	15	156	719	1672	2924
Consult Program	578	610	1047	3470	7627	13471
Creating facts file	78	93	78	62	62	78
Creating KnowledgeGraphClasses and KnowledgeGraphProperties	94	79	110	79	203	78
Fetching and writing data set	1906	2250	5016	13314	20598	24427
Fetching shacl schema	1110	890	797	766	875	781
Loading data files	47	485	2922	9440	16564	25380
Loading shacl schema files	1703	1813	1844	1719	1689	1767
Jena Fuseki connection establishment	3913	3906	3938	3953	3922	3813

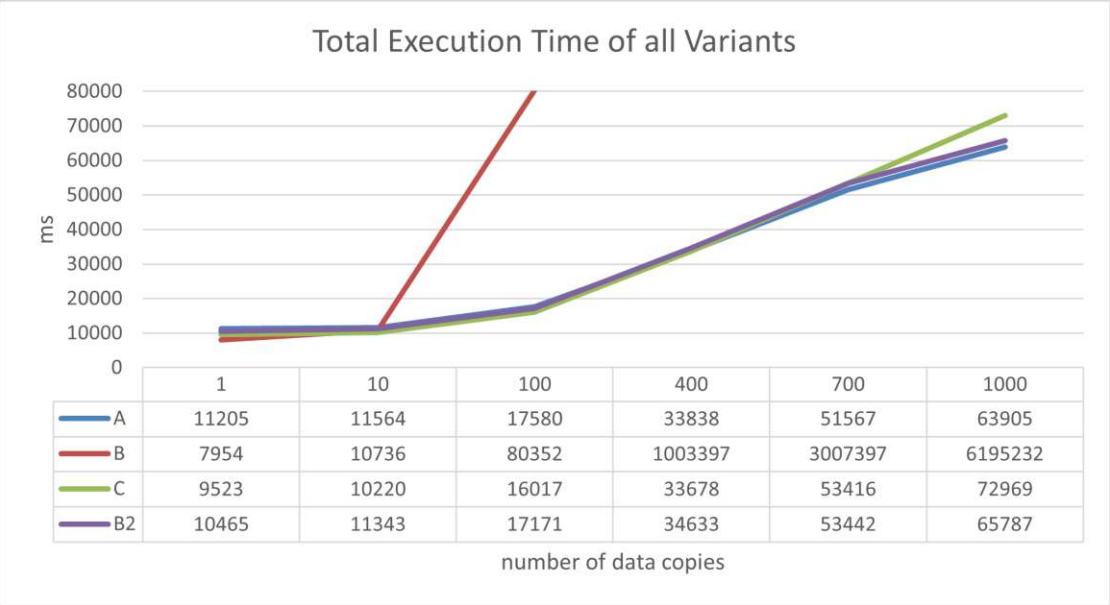
Figure 6.4: Performance results of mapping variant C

In variant C, the mapped input facts are not asserted in the Prolog DB but are made available by the mapping rules as virtual facts. For more complex programs, it will be advantageous to assert the mapped input facts in the Prolog database, as we have done with the improved variant B2.

6.1.5 Summary

We conducted performance studies regarding three variants and one subvariant of the schema-aware KG-Prolog mapper. Figure 6.5 shows the total execution time of these variants with regard to increasing number of data copies. Realization variants A, C and subvariant B2 have similar performance characteristics, only the original variant B has a significantly poorer performance. Since the different variants do not show significant differences in terms of performance, we can rely on other criteria when deciding which of the variants to integrate into the KG system. We did not test if using non-duplicated data of the same size makes a difference in terms of performance. However, we used real

data from the project as input and the performance of mapping data values are similar. Running the performance tests multiple times did not show significant differences in the results, therefore the total execution times are the results from one run instead of the mean of multiple runs, for example.



	1	10	100	400	700	1000
A	11205	11564	17580	33838	51567	63905
B	7954	10736	80352	1003397	3007397	6195232
C	9523	10220	16017	33678	53416	72969
B2	10465	11343	17171	34633	53442	65787

Figure 6.5: Total execution time of different mapping variants

6.2 Integration of Prolog with the Proof-of-Concept KG System

In Chapter 1 the AISA architecture and the small proof-of-concept KG system, which is visualized in Figure 1.1, has already been introduced.

In this section, we describe the integration of Prolog engine and KG-Prolog mapping in the proof-of-concept KG system, which is also shown in Figure 1.1. The integrated system realizes the schema-oblivious approach and variant C of the schema-aware approach. The integrated system builds on SWI-Prolog's internal, in-memory RDF database and an incremental full replication between the AISA KG (persisted via Jena TDB and accessed via Jena Fuseki) and SWI-Prolog's RDF DB.

Of the three realization variants described in the previous chapter we integrated one variant into the KG system. The rationale for choosing variant C is the following:

- The Prolog programmer has the full KG also in the form of RDF quadruples available and can flexibly choose between the schema-aware and the schema-oblivious approach.

- It is straightforward to cope with very frequent additions of new data to the KG with variant C and to incrementally make the new data available to Prolog.

Figure 6.6 shows the overall approach. The mapping generator receives the KG schema as input and produces as output the schema of Prolog predicates (documenting the schema of input Prolog facts for the Prolog programmer who will inspect this documentation when writing Prolog programs) and the schema-specific mapping rules in Prolog which take Prolog's RDF DB as input. The mapped predicates together with the mapping rules (one rule for each predicate) can be regarded as schema-aware view over the KG. The Prolog programs can access the KG directly via Prolog's RDF DB in a schema-oblivious manner, or schema-aware via the mapped predicates and the corresponding rules. Prolog programs write directly to the KG via Prolog's RDF DB.

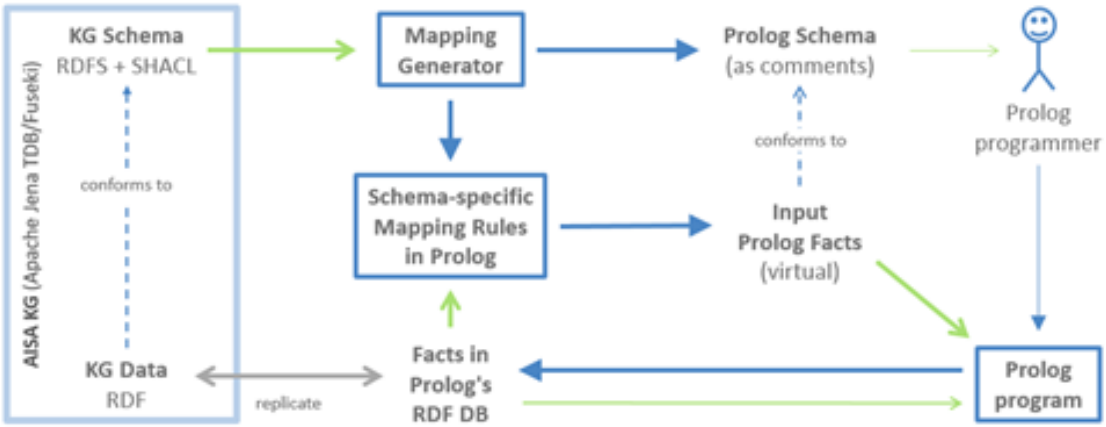


Figure 6.6: Conceptual architecture of the integrated KG-Prolog mapper

When working with the Java library for the AISA KG module system, the library takes care of replicating every named graph written to the AISA KG also in the RDF DB and, vice versa, data written to the RDF DB is replicated in the AISA KG.

The KG module system runs as a Java process that connects to the Fuseki KG Server (which runs in a separate process), and inserts data into the KG via its KG modules. There are mainly two types of KG modules: A single-run module (also referred to as static module) runs only once at start-up of the KG system and inserts a static data named graph into the KG. A multiple-run module (also referred to as dynamic module) runs multiple times, at start-up it adds a static data named graph and with every run it inserts a new data named graph.

The Prolog engine runs embedded in the KG module system's Java process. At start-up, the KG module system loads a Prolog program with generic code (global.pl, see Listing 31 and next paragraph) and invokes the single-run schema module (implemented by class SchemaLoader) which loads the RDFS/SHACL schema to the KG and executes the mapping generator, which generates and loads a Prolog program with the schema specific

mapping rules. At start-up, the KG module system also initializes the multiple-run modules. Initializing a Prolog-based KG module comes with loading a Prolog program that implements the Prolog parts of the Prolog-based module. That Prolog program is itself a Prolog module, with the Prolog module having the same name as the KG module from which it is loaded. Each such Prolog module implements a run method which is invoked each time the Prolog-based KG module runs.

```
1  :- use_module(library(semweb/rdf11)).
2  :- use_module(library(semweb/turtle)).
3  :- set_prolog_flag(toplevel_print_anon, false).
4
5  :- dynamic current_graph/1.
6
7  current_graph('http://www.ex.org/default').
8
9  :- rdf_meta
10       insert_rdf(t,t,t).
11
12 set_current_graph(Graph) :-
13   retractall(current_graph(_)),
14   asserta(current_graph(Graph)).
15
16 insert_rdf(Subject,Predicate,Object) :-
17   current_graph(Graph),
18   rdf_assert( Subject, Predicate, Object, Graph ).
```

Listing 31: Prolog program `global.pl`

The generic Prolog code (see Listing 31) loaded into the Prolog engine at start-up implements method insert_rdf/3 (Line 16), which is used by the Prolog modules to insert RDF statements into the new data named graph associated with the current run of the multiple-run module. Predicate current_graph/1 holds the IRI of the current new data named graph. At each run of a Prolog based module, Prolog method set_current_graph/1 (Line 12 of Listing 31) is called from Java (Line 36 in Listing 32) prior to calling the run/0 method (Line 37 from Listing 32).

```
36  new Query("set_current_graph('"+ getTurnIri() +"')").hasSolution();
37  new Query(getName()+":run").hasSolution();
```

Listing 32: Fragment from `PrologModule.java`

In the sample project (at.jku.dke.aisa.kg.sample.prolog) there are two Prolog-based modules called *prolog1* and *prolog2*. Listing 33 shows a sample Prolog module *prolog1*, the name of the Prolog module (set at Line 1) has to be the same as the name of the corresponding KG module. As an example of accessing the KG using the schema-aware approach it queries mapped predicate gml_TimePeriod/4 (Line 4) which in turn queries

the RDF DB. The run/0 method of module *prolog1* writes to the current new data named graph using method insert_rdf/3 (Lines 7 and 8). As an example of accessing the KG using the schema-oblivious approach, method run/0 also queries Prolog's RDF DB directly using predicate rdf/4 (see Line 8).

```
1  :- module(prolog1,[]).
2
3  testMapping(TimePeriod,Graph) :-
4          gml_TimePeriod(Graph, TimePeriod, _,_).
5
6  run :-
7    forall( testMapping(X,G), insert_rdf( X, 'http://ex.org/in', G ) ),
8    forall( rdf(_,_,_,G), insert_rdf( G, rdf:type, 'http://ex.org/Graph' ) ) .
```

Listing 33: Sample Prolog module prolog1

KG Data Replication. The approach is based on full incremental replication of the KG contents in SWI-Prolog's in-memory database. Incremental replication means that after the first replication of data, only changed content is replicated to the target, which avoids recalculation and querying data from scratch. After a named graph is written to the AISA KG it is fetched from the KG, written to a temporary RDF/XML file which is then loaded into RDF.

For quickly inspecting the proper functioning of the system, the contents of the KG are also replicated on the file system in folder fileoutput/_NG_/ as Turtle files with the name of the named graph (i.e. the last part of the named graph's IRI) as file name. These files are deleted with the next start of the KG-System, together with all the other files in folder fileoutput/. All named graphs are collected in a dedicated folder (fileoutput/_NG_/) and not in module specific output folders (e.g., fileoutput/prolog1/). This is to, first, avoid cluttering the module-specific output folders which are dedicated to custom module-specific file output, and, second, to have the whole content of the KG at one place. Named graphs are written to the file system once they are finished and committed into the KG and replicated to Prolog. The current committed state of the KG can thus always be inspected by inspecting the files in folder fileoutput/_NG_/.

Summary. Prolog programs are fully integrated in the KG module system and the KG manager. A Prolog-based KG module is represented in Java by an instance of class PrologModule and in Prolog by a Prolog module represented by one Prolog program per module. Prolog modules are loaded into the Prolog engine during initialization of the KG module system. Each such Prolog module implements a method run/0, which executes queries over the RDF DB, which is an in-memory replica of the KG (using the predicates provided by the schema-aware mapping or directly following the schema-oblivious approach), performs reasoning, and writes results into the current named graph in the RDF DB, which is replicated to the KG.

6.3 Results

The design problem tackled by this thesis is to improve accessing the KG from Prolog by designing a KG-Prolog mapper that takes care of data interchange and mapping between Prolog engine and KG. We investigated schema-oblivious and schema-aware KG-Prolog mapping.

Following the schema-oblivious approach for accessing the KG from Prolog, every RDF quadruple is represented as a Prolog fact. This approach is well-suited for writing results from Prolog programs to the KG and is also highly flexible for querying the KG from Prolog. However, knowledge about one object is distributed over many facts, which builds a complex structure and is hard to read for Prolog programmers.

Following the schema-aware approach for accessing the KG from Prolog, facts about one object in one named graph are combined in a single fact according to the KG schema. This naturally preserves the KG schema in Prolog and is therefore more intuitive to read. Hence, it overcomes the shortcomings of the schema-oblivious approach by facilitating the development and maintenance of Prolog programs. Consequently, way less facts are required to describe a conceptual schema. One example already mentioned in Chapter 2.4 shows that the schema-oblivious approach pictured in Figure 2.2 consists of 13 facts in comparison to the schema-aware approach displayed in Figure 2.4 which consists of 3 facts.

To conclude, the schema-oblivious approach can be realized easily, but is unwieldy for Prolog programmers when it comes to reading complex KG data. The schema-aware approach provides the content of the KG in a more intuitive form which is easier understandable to Prolog programmers when it comes to reading complex KG data. For this purpose we created a prototypical implementation of three different schema-aware RDF-Prolog mapping variants. This prototype is instrumental to reach the knowledge goal and answer the knowledge question:

The knowledge goal defined is to understand the performance characteristics of different alternative approaches to mapping execution and data exchange. The corresponding knowledge question is: How do the performance characteristics of the different approaches to mapping execution and data exchange differ depending on varying data input sizes?

We conducted preliminary performance studies, which were explained in detail in Chapter 6.1, and noticed that the performance of mapping variant A and C is linear to the number of data copies. Only mapping variant B showed poor performance, because requests and queries were posed multiple times for the same facts. Hence, we bypassed this shortcoming by creating a method, which contains all SPARQL queries and asserts the facts into the database after querying. The improved mapping variant B is on average equally fast than mapping variant A and C. In conclusion, there is no significant difference in terms of execution time between mapping variant A, B and C.

Analysis of the capabilities of SWI-Prolog and our considerations for embedding Prolog into the AISA-KG system have called into question the purely schema-aware and SPARQL-

based approach. Based on these analyses and considerations, we opted for a combination between schema-aware and schema-oblivious approach, in order to get the flexibility of the schema-oblivious approach for write access and simple read access and to also get the convenience of the schema-aware approach for reading schema-conformant data.

Regarding the realization of the schema-aware approach, we have designed and implemented different variants and studied their performance. The three investigated variants are: (A) execute SPARQL queries and generate Prolog facts in Java and load generated input facts to Prolog; (B) executed SPARQL queries in Prolog and dynamically assert the generated input facts in Prolog; and (C) replicate KG to SWI-Prolog's in-memory RDF-database with input facts provided as virtual facts by mapping rules in Prolog. Our preliminary performance studies have not revealed significant differences among the final versions of these three variants.

The schema of the generated input facts generated from the KG schema, which is specified in RDFS and SHACL, is the same for all three variants. The only difference is how and in which system (in Java or Prolog) the mapping is executed at the instance level. The structure of predicates (arity and ordering of attributes) as well as the SPARQL queries or rules to populate these predicates are generated not only from validating SHACL properties but also from non-validating properties such as `sh:order`.

For the integration of Prolog into the KG system, we opted for variant C, because it facilitates incremental update of the generated input facts and because it inherently implements a combination of schema-aware and schema-oblivious approach. Prolog programs are fully integrated in the KG module system and can be invoked recurrently by the central control component to perform recurrent reasoning tasks over the AISA KG.

CHAPTER 7

Related Work

Previous research on mapping RDF to Prolog do not cover all issues this project encounters. There is no schema-aware mapping solution and especially not for the context of the AISA project yet. Furthermore, previous research on RDF-Prolog mapping focuses mainly on storing alternatives, but not on different options of mapping execution and data interchange. One of the goals of this thesis is to outline the results of experimenting with different alternatives. Another requirement not covered by previous research is the handling of missing data. In addition, the AISA project addresses aeronautical information, which does not only consist of static, but also of time-varying data (e.g., aircraft positions). In this chapter, we will give a short overview of related research on the mapping structure, the transformation approaches and on the used technology and software. The first section is about the mapping structure and includes already existing research on schema-oblivious and schema-aware mappings. The second section is about transformation approaches and goes further into detail about the use of model transformation tools vs. using dedicated transformation software.

7.1 Mapping Structures

Mapping strategies can be divided into schema-oblivious and schema-aware mapping [ADF04]. Theoharis et al. [TCK05] describe two different storing schemes for RDFS. They define schema-oblivious (also called generic or vertical) to consist of triples of the form subject-predicate-object. The *subject* attribute represents a resource that is the source of a property, whose name is specified in the *predicate* attribute and the *object* attribute represents a target resource or literal value for this property. Different properties of a specific resource are tied together using the same subject URI. For the schema-aware (also called specific or binary) mapping, they defined that one table is used per RDFS schema property or class. This definition can be applied analogously to Prolog, since the rdf/3 function consists also of subject-predicate-object. The main

difference to our work is that we did not create a predicate for each property, but one predicate per class with one argument per property of that class.

7.1.1 Schema-oblivious Mapping

Wielemaker et al. [WSW03] described the semantic web as "a promising application-area for the Prolog programming language for its non-determinism and pattern matching". They created "an infrastructure for loading and saving RDF, storing triples, elementary reasoning with triples and visualization, which aims at fast parsing, fast access and scalability". They also considered different requirements and storing alternatives depending on the ontology representation. Their first alternative was to build a secondary database following the RDFS datamodel. Their second approach was to build an external DBMS for the triple store. Another option was to use Predicate(Subject, Object) as database representation and store the inverse relation as well in InversePred(Object, Subject). Save/load using Prolog native syntax is twice as fast as parsing the RDF. In the end, they opted for an external module written in C to extend the functionality of Prolog, which turned out to reduce the memory requirements and to have the best access time.

7.1.2 Schema-aware Mapping

Störrle [Stö07] created a system called the Model Manipulation Toolkit (MoMaT). This system provides a Prolog-based model representation and query interface for models. In the first step, the input model is converted in a common format. This format is described by a meta-metamodel and is the least common denominator for many modeling languages. Secondly, each model element is interpreted as a Prolog fact of the form `me(type-id, [tag-value, ...])`, whereas type is the metaclass in the source metamodel, id is an unique identifier and tag-value is a property of the model element and its value. The model elements represented as Prolog facts can be queried based on selection critera like identifier, name or a combination of features.
Compared to the resulting Prolog facts from our schema-aware mapping, the name of the class defines the predicate name. In addition to the id, we also save the id of the graph in the fact. We do not save the element properties in a list, but they can represent lists of values or complex values.

Concerning schema-aware mapping, Kraska et al. [KR06] created mapping rules and a mapping tool called Genea, which maps Ontologies written in OWL into Relational Databases. They compared their solution with existing schema-oblivious solutions (Sesame) [BKvH02] and other schema-aware (DLDB) [PH03] approaches. It showed that Genea, with its schema-aware mapping approach, provides better performance and scalability than pure RDF triple stores [KR06].

Kejriwal et al. [KN09] created TRANS, which is another schema-aware mapping approach closely similar to Genea [KR06]. Comparisons with Genea showed that TRANS performed better in terms of query response time and space used to store the ontology instance data. One reason for the better performance of TRANS is that it does not map each

class or property to a separate table. That means that unlike to other schema-aware mapping approaches, there is no big increase in number of tables for an ontology of too many classes. Furthermore, TRANS also performs preprocessing over ontology hierarchy, supports OWL subsumption reasoning, instantiation, some consistency checking and also some intentional queries unlike Genea.

In Prolog, it is not only possible to represent simple relations as predicates, but also embedded lists and further complex structures, which are similar to object-relational databases. In comparison to the RDF-Prolog mapper of this thesis, present schema-aware RDF to relational database mappings do not support this functionality.

Theoharis et al. [TCK05] benchmarked and compared a schema-aware, a schema-oblivious and a hybrid of the schema-aware and schema-oblivious database representations of RDFS schemata and data. In comparison to our schema-aware approach, they map not only one table per RDFS class, but also per property. Concerning the schema-oblivious approach they have a single table with triples of the form subject-predicate-object and for the hybrid variant, they combine the functionality of both approaches and have one table per RDFS meta-class. Their main conclusion from their experiments of evaluating taxonomic queries is that the hybrid approach achieved the best performance, followed by the schema-aware representation, which achieved similar performance.

Documents and schemas defined in RDF/RDFS are machine-readable and used to describe data in the semantic web. Besides our RDF-to-Prolog mapping approach, there has already been research on different related approaches on how to optimize the use of RDF/RDFS.

Zongmin Ma et al. [MCY16] conducted a survey on different approaches to use relational databases and not-only SQL databases for RDF data management and classified them into vertical stores, horizontal stores, and type stores. In a *vertical store* (also referred to as triple store), all RDF triples are stored in a single table with columns subject, predicate, and object. In a *horizontal store* all information is stored in a wide table with one row for each subject and one column for each property. This leads to many null values and multiple values in one column of one row. To avoid null values and multi-valued columns, the single table can be partitioned into many tables, one table per predicate. In a *type store* the wide table is vertically partitioned into smaller tables with related predicates, typically one table per type with all properties of that type as columns. A table in a type store still has one row per subject and multi-valued columns. The number of null values is reduced. Our schema-aware approach to representing RDF in Prolog is akin to type stores; it adds dedicated support for different kinds of null values and complex values with units of measurement.

Teswanich et al. [TC07] transform RDF/RDFS into relational database format, so that data can be manipulated easily. They load the RDF/RDFS documents into the RDF Transformation Engine. The RDFS schema is converted to structure of tables and the RDF data is converted to records in the tables. One drawback of this approach is the waste of space due to duplication of the data. Also, the synchronisation of data between

the two stores is another issue to consider [RGK+09b].
In comparison to our approach, schema-aware mapping is achieved by creating a table for each property, which conceptually conforms to a Prolog predicate. In contrast, our approach combines all properties of a subject in a row.

Instead of putting effort, time and resources into development of visualization tools for new data storage paradigms, Ramanujam et al. [RGK+09c] focused on how to salvage existing, mature technologies such as relational database management systems for new data models such as RDF/RDFS. Therefore, Ramanujam et al. [RGK+09b] created a bridge between RDF stores and relational tools without creating an actual physical relational schema and duplicating data. With R2D (RDF-to-Database) they make relational tools available to RDF data stores by putting a JDBC wrapper around them that presents a relational view of the RDF store. Besides avoiding data replication, another advantage of R2D is that it can deal with sloppy data, in which schemas and constructs are incomplete. Based on a similarity threshold, the similarity of two tables and if they are considered to be the same is decided. Further, they extended R2D by including the ability to map blank nodes and to perform pattern matching and aggregation functions on data by enhancing the SQL-to-SPARQL transformation [RGK+09a].

The mapping approaches mentioned in the previous paragraphs produce a normalized output, which is equivalent to the input. These outputs might be atomic, non-structured and conforming to the First Normal Form (1NF). The 1NF is violated if a relation contains other relations or multi-valued attributes. Schek et al. [SS86] consider the 1NF is a bottleneck and extended the usual notion of schema, value, attribute and domain of 1NF to relations with relation-valued attributes by applying them recursively.
Preserving data types and the real structure of our input is important, therefore our mapping approach outputs results consisting of typed and complex values. However, compared to [SS86], our predicates are limited and not endlessly nested. Further details concerning data handling are described in Chapter 4.

7.2 Transformation Approaches

Model transformations can be achieved by using model transformation languages or by using general-purpose programming languages [HKT22]. Götz et al. [GTG21] conducted a systematic literature review with the goal of providing an overview of claims about the advantages and disadvantages of model transformation languages compared to general-purpose programming languages. They selected a total of 58 publications, divided the claims into 15 different categories, and designed a narrative to track claims and evidence in the literature. They concluded that existing literature claims many advantages of using model transformation languages, but that these claims are largely unverified and lack evidence for the stated advantages and disadvantages. The next two subsections go more into detail about model transformation languages and dedicated transformation software.

7.2.1 Using Model Transformation Tools

Regarding Model Driven Engineering (MDE), the focus of the work is on modeling rather than programming. MDE takes development to a higher level of abstraction with the goal to increase development productivity and quality [SK03]. Model transformations take one or more source models as input and produce one or more target models as output by following a set of transformation rules [CH03]. There are two main categories of model transformations: model-model transformations and model-code transformations (also called mode-text transformation). Model-code transformations only required a meta-model for the target programming language, hence they are a special case of model-model transformation [CH03].
Literature research showed no existing model-driven mapping approach between RDF and Prolog. Most already existing work uses Prolog for the definition of transformation rules as described in the following paragraphs.

Almendros-Jiménez et al. [AJI11] created a framework for specifying model transformations using Prolog. By using Prolog rules, an UML class diagram, which represents a database in form of an entity-relationship diagram, is transformed into an UML diagram that represents a relational database. A meta-model was created for both, the source and the target models to transform each instance of the source meta-model into an instance of the target meta-model of the target model. The framework uses SWI-Prolog for importing and exporting RDFS/OWL triples. First, the source model, which orginates from an OWL file is stored SWI-Prolog's database. Then a Prolog predicate `transform(+SourceModelFile,+TargetModelFile)` is called to transform the source model into the target model, which is also stored in an OWL file.
Almendros-Jiménez et al. [AILS16] introduced a transformation language called PTL (Prolog based Transformation Language) that uses Prolog as a transformation engine. PTL is a hybrid language consisting of ATL[1] [atl] (Atlas Transformation Language) style rules to define mappings from source to target model and Prolog used as a tool.

In recent years, a large number of model transformation languages and tools have been proposed that vary significantly in their capabilities, limitations and requirements, making it difficult to effectively use the most appropriate one for any given task. Kahani et al. [KBC+19] compared existing metamodel-based transformation tools and made the results publicly available on a website[2] [mde] that provides the ability to search for tools based on specific search criteria. Their study consisted of 60 tools and classifies, which are compared using 46 facets distributed over six categories: general, model-level, transformation style, user experience, collaboration support and runtime requirements. These parameters also serve as the search filter on their dedicated website[3] [mde]. Next to classification and comparison of existing model transformation tools, they contribute to the analysis of why some tools are discontinued and to the analysis of which facets

[1] https://www.eclipse.org/atl/
[2] http://www.mdetools.com/
[3] http://www.mdetools.com/

are supported or unsupported, as well as the relationship between the transformation approach and transformation language used.

Initially, we investigated a model-driven approach for mapping of RDFS to Prolog by using model-text transformations. Due to the overhead of the creation of transformation rules and also the implementation of the models we decided to not further investigate the usage of model transformation tools. Model-to-text transformation requires not only the creation of transformation rules, but also the models, which costs additional effort. The test data was created by project partners and the concrete desired output and requirements were not completely clear at the beginning of the project, but they were adjusted iteratively. Using input information from RDFS and SHACL, and creating three different mapping methods, requires redundant and complex use of information. Therefore, it was convenient to create objects of classes and properties, which process all the information needed for mapping coming from RDFS/SHACL at creation time. Developing the RDFS-to-Prolog mapper in pure Java and integrating it into the already existing KG system appeared to be more straightforward. Hence, we agreed on not using model transformation languages.

7.2.2 Dedicated Transformation Software

Model transformation languages have not been adopted widely in the industry yet [BCG19]. One reason for that might be that there is hardly evidence that model transformation languages are substantially better. This is the reason why often transformations are still written in general-purpose languages like Java. Burgueño et al. [BCG19] questioned the future of model transformation languages by creating an online survey and talking about the results in an open community discussion. The main reasons according to their research for not using model transformation languages are that they are not that easy to understand and learn, there is hardly any documentation, there is no real debugging, it is hard to hire staff that uses model transformation languages, general-purpose languages can be used for complex transformations, but some model transformation languages such as ATL are only useful for simple transformations and they have a complicated setup and IDE-specific dependencies.
They concluded that model transformation languages are becoming less popular, but will stay viable "in niches where their benefits can be more easily demonstrated" [BCG19]. Especially in cases where good tracing is required, model transformations are clearly advantageous. However, the reasons for negative results on model transformation languages are caused due to a lack of social and tool aspects, as mentioned above, and due to the improvements of general-purpose languages themselves, as they nowadays are capable of programming constructs, which were only presented in model transformation languages years ago. Höppner et al. [HKT22] compared the complexity and size of model transformations written in ATL, Java SE5 and Java SE14. They concluded that newer Java versions require less code to create transformations than older Java versions. Using newer Java versions for model transformations makes development easier, because less work is required for the set up of the transformation. They claim that general-purpose

languages are best for transformations where only little or no tracing is necessary.

CHAPTER 8

Conclusion and Future Work

The following sections conclude the work of this thesis and discuss future work.

8.1 Conclusion

The goal of this thesis was to create a schema-aware RDF-to-Prolog mapper for the AISA project. We investigate schema-oblivious and schema-aware mapping approaches. The schema-oblivious approach can be realized easily, but is very unwieldy to use for Prolog programmers when it comes to reading complex KG data. Therefore we implemented three variants of a schema-aware mapper to improve the access to the KG from Prolog. The schema-aware approaches preserve the structure of the KG schema and consequently facilitate the development and maintenance of Prolog programs.

Our knowledge goal is to understand the performance characteristics of different alternative approaches to mapping execution and data exchange, which leads us to the following knowledge question: How do the performance characteristics of the different approaches to mapping execution and data exchange differ depending on varying data input sizes? To answer this question we conducted preliminary performance studies for comparison of the three variants and one sub-variant of the schema-aware KG-Prolog mapper. Results of those performance measurements showed that realization variants A, C and sub-variant B2 have similar performance characteristics and only the original variant B has a significantly poorer performance.

We provide a full integration of Prolog engine and AISA KG system for the schema-oblivious approach together with one variant of the schema-aware approach. The integrated system realizes the schema-oblivious approach and variant C of the schema-aware approach. We opted for variant C, because the Prolog programmer has the full KG also available in the form of RDF quadruples and can flexibly choose between schema-aware and schema-oblivious approach. With variant C it is possible to cope with very

frequent additions of new data to the KG and to incrementally make the new data to Prolog available.

8.2 Future Work

With this thesis we improved accessing the KG by creating different variants of a schema-aware RDF-Prolog mapper. However, the use of dedicated model transformation languages was only discussed in theory. Especially, the comparison between the complexity and size of model transformations written in ATL and Java, investigated by Höppner et al. [HKT22], were discussed in Chapter 7. The mappers are implemented using plain Java and the technologies mentioned in Chapter 2.3. The used source metaschema is adapted to handle datatypes and constraints, which are in the input files we got from project partners of AISA and the target metaschema is adapted to the expected results for the project.

Possible extensions to this work include implementing the schema-aware mapper using various model transformation tools and benchmarking it against our solution.

Finally, the metaschemas may be expanded to handle data types and constraints, which are not covered by our implementation yet.

List of Figures

List of Listings

Glossary

1NF First Normal Form. 74

AI Artificial Intelligence. 1, 2

AISA Artificial Intelligence Situational Awareness. xi, xiii, xv, 1, 2, 4, 6, 16, 18, 46, 52, 54, 71, 79

AIXM Aeronautical Information Exchange Model. 57, 58

DBMS Database Management System. 72

FIXM Flight Information Exchange Model. 57, 58

JDBC Java Database Connectivity. 74

KG Knowledge Graph. xi, xiii, xv, 1–7, 9, 13, 15–18, 21–23, 26, 46–48, 52, 57, 58, 65–70, 76, 79, 80

MDE Model Driven Engineering. 75

OWL Ontology Web Language. 75

RDF Resource Description Framework. xi, xiii, 1–4, 6, 9–13, 15–17, 21–23, 26, 45, 57–60, 65–69, 71–75, 79, 80

RDFS Resource Description Framework Schema. 1–3, 6, 7, 9, 10, 13, 16, 18, 21, 23, 25, 26, 45, 46, 48, 57, 66, 70–76

SHACL Shapes Constraint Language. 1–3, 6, 7, 9, 10, 16, 18, 21, 23–27, 35–37, 39–42, 45–49, 52–54, 57, 58, 62, 63, 66, 70, 76, 83, 84

SPARQL SPARQL Protocol and RDF Query Langauge. 1, 2, 6, 11, 15, 16, 21, 22, 25, 26, 29, 30, 41, 48, 49, 53, 60, 62, 70

UML Unified Modeling Language. 2

URI Uniform Resource Identifier. 14, 53, 71

XML Extensible Markup Language. 68

Bibliography

[ADF04] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the XML-to-relational mapping problem. In Alberto H. F. Laender, Dongwon Lee, and Marc Ronthaler, editors, *Sixth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2004), Washington, DC, USA, November 12-13, 2004*, pages 31–38. ACM, 2004.

[AILS16] Jesús Manuel Almendros-Jiménez, Luis Iribarne, Jesús J. López-Fernández, and Ángel Mora Segura. PTL: A model transformation language based on logic programming. *J. Log. Algebraic Methods Program.*, 85(2):332–366, 2016.

[ais] AI Situational Awareness Foundation for Advancing Automation. `https://aisa-project.eu/`. Accessed: 2023.24.02.

[aix] Aeronautical Information Exchange Model. `https://www.aixm.aero/`. Accessed: 2023.25.02.

[AJI11] Jesús Almendros-Jiménez and Luis Iribarne. ODM-based UML Model Transformations using Prolog. In *International Workshop on Model-Driven Engineering, Logic and Optimization: friends or foes*, pages 382–394, 01 2011.

[Apo74] Tom M Apostol. *Mathematical analysis; 2nd ed.* Addison-Wesley series in mathematics. Addison-Wesley, Reading, MA, 1974.

[atl] Atlas Transformation Language. `https://www.eclipse.org/atl/`. Accessed: 2022.24.06.

[BBG+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006.

[BCG19] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. The Future of Model Transformation Languages: An Open Community Discussion. *J. Object Technol.*, 18(3):7:1–11, 2019.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Ian Horrocks and James A. Hendler, editors, *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2002.

[BM07] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1–12. ACM, 2007.

[Bra13] Max Bramer. *Logic Programming with Prolog*. Springer, 2013.

[CH03] K. Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*, oct 2003.

[CHWL07a] Richard Cyganiak, David Hyland-Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. *W3C Proposed Recommendation*, 01 2007.

[CHWL07b] Richard Cyganiak, David Hyland-Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. *W3C Proposed Recommendation*, 01 2007.

[DBC14] Eric Prud'hommeaux David Beckett, Tim Berners-Lee and Gavin Carothers. RDF 1.1 Turtle. *W3C Proposed Recommendation*, 02 2014.

[Fag06] Ronald Fagin. Inverting schema mappings. In Stijn Vansummeren, editor, *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, pages 50–59. ACM, 2006.

[FBJV05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Applying Generic Model Management to Data Mapping. In Véronique Benzaken, editor, *21èmes Journées Bases de Données Avancées, BDA 2005, Saint Malo, France, 17-20 octobre 2005, Actes (Informal Proceedings)*, 2005.

[fix] Flight Information Exchange Model. https://fixm.aero/. Accessed: 2023.25.02.

[FKPT05] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.

[git] KG-Prolog Mapper. https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper/. Accessed: 2022.02.02.

[GTG21] Stefan Götz, Matthias Tichy, and Raffaela Groner. Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review. *Softw. Syst. Model.*, 20(2):469–503, 2021.

[HKT22] Stefan Höppner, Timo Kehrer, and Matthias Tichy. Contrasting dedicated model transformation languages versus general purpose languages: a historical perspective on ATL versus Java based on complexity and size. *Software and Systems Modeling*, 21:1–33, 04 2022.

[HN23] Marlene Hartmann and Bernd Neumayr. jku-win-dke/aisa-kg-prolog-mapper: v1.0.0. Apr 2023.

[Jam92] P. James. *Knowledge graphs*, pages 97–117. Elsevier, Ireland, 1992. 1991 Workshop on Linguistic Instruments in Knowledge Engineering ; Conference date: 17-01-1991 Through 18-01-1991.

[jena] Apache Jena. https://jena.apache.org/. Accessed: 2022.29.04.

[jenb] Apache Jena Fuseki. https://jena.apache.org/documentation/fuseki2/. Accessed: 2022.29.04.

[jku] Johannes Kepler University Linz. https://www.jku.at/. Accessed: 2023.25.02.

[jpl] JPL. https://jpl7.org/. Accessed: 2022.02.02.

[KBC+19] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Dániel Varró. Survey and classification of model transformation tools. *Softw. Syst. Model.*, 18(4):2361–2397, 2019.

[KK17] Holger Knublauch and Dimitris Kontokostas. Shapes constraint language (SHACL). *W3C Proposed Recommendation*, 07 2017.

[KN09] Saurabh Kejriwal and N. S. Narayanaswamy. TRANS: Schema-Aware Mapping of OWL Ontologies into Relational Databases. In Sanjay Chawla, Kamalakar Karlapalem, and Vikram Pudi, editors, *Proceedings of the 15th International Conference on Management of Data, December 9-12, 2009, International School of Information Management, Mysore, India*. Computer Society of India, 2009.

[KR06] Tim Kraska and Uwe Röhm. Genea: Schema-Aware Mapping of Ontologies into Relational Databases. In Laks V. S. Lakshmanan, Prasan Roy, and Anthony K. H. Tung, editors, *Proceedings of the 13th International Conference on Management of Data, December 14-16, 2006, Delhi, India*, pages 92–103. Tata McGraw-Hill Publishing Company Limited, 2006.

[MCY16] Zongmin Ma, Miriam A. M. Capretz, and Li Yan. Storing massive resource description framework (RDF) data: a survey. *Knowl. Eng. Rev.*, 31(4):391–413, 2016.

[mde] MDETOOLS. `http://www.mdetools.com/`. Accessed: 2022.19.03.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema Mapping as Query Discovery. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 77–88. Morgan Kaufmann, 2000.

[Neu21] Bernd Neumayr. Proof-of-concept KG system. `https://aisa-project.eu/downloads/AISA_4.1.pdf`, 2021.

[NH21] Bernd Neumayr and Marlene Hartmann. KG-Prolog Mapper. `https://aisa-project.eu/downloads/AISA%20D4.2.pdf`, 2021.

[ope] Opensky Network. `https://opensky-network.org/`. Accessed: 2021.17.03.

[PH03] Zhengxiang Pan and Jeff Heflin. DLDB: Extending Relational Databases to Support Semantic Web Queries. In Raphael Volz, Stefan Decker, and Isabel F. Cruz, editors, *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[PHS13] Eric Prud'hommeaux, Steve Harris, and Andy Seaborne. SPARQL 1.1 Query Language. `http://www.w3.org/TR/sparql11-query`, 2013.

[PS07] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. `https://www.w3.org/TR/rdf-sparql-query/`, 01 2007. Accessed: 2022.29.04.

[RGK+09a] Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Steven Seida, and Bhavani Thuraisingham. R2D: A Bridge between the Semantic Web and Relational Visualization Tools. In *Proceedings of the 3rd IEEE International Conference on Semantic Computing (ICSC 2009), 14-16 September 2009, Berkeley, CA, USA*, pages 303–311. IEEE Computer Society, 2009.

[RGK+09b] Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Steven Seida, and Bhavani Thuraisingham. R2D: Extracting Relational Structure from RDF Stores. In *2009 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2009, Milan, Italy, 15-18 September 2009, Main Conference Proceedings*, pages 361–366. IEEE Computer Society, 2009.

[RGK+09c] Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Steven Seida, and Bhavani Thuraisingham. Relationalizing RDF stores for tools reusability. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 1059–1060. ACM, 2009.

[RO08] Ruth Raventós and Antoni Olivé. An object-oriented operation-based approach to translation between MOF metaschemas. *Data Knowl. Eng.*, 67(3):444–462, 2008.

[ses] SESAR Joint Undertaking. `https://www.sesarju.eu/`. Accessed: 2023.25.02.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Softw.*, 20(5):42–45, 2003.

[sky] Skyguide Swiss Air Navigation Services Ltd. `https://www.skyguide.ch/`. Accessed: 2023.25.02.

[slo] Slot Consulting Ltd. `https://www.slotconsulting.hu/`. Accessed: 2023.25.02.

[SS86] Hans-Jörg Schek and Marc H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, 1986.

[Stö07] Harald Störrle. A PROLOG-based Approach to Representing and Querying Software Engineering Models. In Philip T. Cox, Andrew Fish, and John Howse, editors, *Proceedings of the VLL 2007 workshop on Visual Languages and Logic in Coeur d'Aléne, Idaho, USA, 23rd September 2007 as part of the 2007 IEEE Symposium on Visual Languages and Human Centric Computing VL/HCC 07*, volume 274 of *CEUR Workshop Proceedings*, pages 71–83. CEUR-WS.org, 2007.

[swi] SWI-Prolog. `https://www.swi-prolog.org/`. Accessed: 2020.26.04.

[TC07] Wajee Teswanich and S. Chittayasothorn. A Transformation from RDF Documents and Schemas to Relational Databases. In *2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 38 – 41, 09 2007.

[TCK05] Yannis Theoharis, Vassilis Christophides, and Gregory Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*, volume 3729 of *Lecture Notes in Computer Science*, pages 685–701. Springer, 2005.

[TRT21] Mia Bazina Tomislav Radišic, Dorea Antolovic and Ivan Tukaric. Facts, Rules and Queries Capturing En-Route ATC Operations. `https://aisa-project.eu/downloads/AISA%20D4.4.pdf`, 2021.

[tub] Technische Universität Braunschweig, Institute of Flight Guidance. `https://www.tu-braunschweig.de/en/iff`. Accessed: 2023.25.02.

[unia] Faculty of Transport and Traffic Sciences at University of Zagreb. `https://www.fpz.unizg.hr/oms/?lang=en`. Accessed: 2023.25.02.

[unib] Universidad Politécnica de Madrid. `https://www.upm.es/`. Accessed: 2023.25.02.

[unic] Zurich University of Applied Sciences (ZHAW), School of Engineering. `https://www.zhaw.ch/en/engineering/`. Accessed: 2023.25.02.

[Wie14] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

[WSTL10] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *CoRR*, abs/1011.5332, 2010.

[WSW03] Jan Wielemaker, Guus Schreiber, and Bob J. Wielinga. Prolog-Based Infrastructure for RDF: Scalability and Performance. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003, Proceedings*, volume 2870 of *Lecture Notes in Computer Science*, pages 644–658. Springer, 2003.