KG-Prolog Mapper

Deliverable ID:	D4.2
Dissemination Level:	PU
Project Acronym:	AISA
Grant:	892618
Call:	H2020-SESAR-2019-2
Topic:	SESAR-ER4-01-2019
Consortium Coordinator:	FTTS
Edition date:	30 September 2021
Edition:	00.01.00
Template Edition:	02.00.02

Authoring & Approval

Authors of the document

Name/Beneficiary	Position/Title	Date
Bernd Neumayr/JKU	University Assistant (Senior PostDoc)	29 September 2021
Marlene Hartmann/JKU	Junior Researcher	24 September 2021

Reviewers internal to the project

Name/Beneficiary	Position/Title	Date
Tomislav Radišić/FTTS	Assistant Professor	27 September 2021
Michael Schrefl/JKU	Full Professor	27 September 2021
Ivana Hajdinjak/FTTS	Project Associate	27 September 2021

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

Name/Beneficiary	Position/Title	Date
Tomislav Radišić/FTTS	Project Coordinator	29 September 2021

Rejected By - Representatives of beneficiaries involved in the project

Name/Beneficiary	Position/Title	Date

Document History

Edition	Date	Status	Author	Justification
00.00.01	15/07/2021	First draft	B. Neumayr	New document
00.00.02	24/09/2021	First full version	B. Neumayr	Document complete
00.00.03	28/09/2021	Comments integrated	B. Neumayr	Comments from reviewers integrated
00.01.00	30/09/2021	First issue	B. Neumayr	First issue

Copyright Statement

AISA

AI SITUATIONAL AWARENESS FOUNDATION FOR ADVANCING AUTOMATION

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

Abstract

The AISA KG introduced with Deliverable D4.1 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 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*.

The design problem tackled by Task 4.2 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, so that Prolog programmers can easily develop Prolog programs, which read from and write to the KG. We investigate schema-oblivious and schema-aware KG-Prolog mapping. The schema-oblivious approach can be realized easily but is unwieldy for Prolog programmers when it comes to reading complex KG data. Schema-aware KG-Prolog mapping provides the contents of the KG in a form amenable to Prolog programmers according to the KG schema. We implement the schema-aware approach in three different variants and conduct preliminary performance studies for comparison. 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.

1 Table of Contents

List of Figures

Executive Summary

This document describes AISA Deliverable D4.2. The developed software prototypes are available open source as **GitHub** repository[1]. The deliverable consists of three realization variants of the schema-aware approach and the sample use and integration of the schema-oblivious and one variant of the schema-aware approach into the KG system introduced with Deliverable D4.1.

Intended Audience

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

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

1 Introduction

In Task 4.2 we have developed three variants of the RDFS/SHACL-to-Prolog mapper. This document describes these developments and the integration of the RDFS/SHACL-to-Prolog mapper into the proof-of-concept prototype KG system from Task 4.1.

1.1 Definitions

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

Knowledge Graph (KG). A knowledge graph is a persistent RDF dataset comprising data and metadata. An *RDF dataset* is a set of named RDF graphs each consisting of a set of RDF statements. An RDF statement is a triple of the form <subject, predicate, object>. An RDF dataset can also be seen as a set of <subject, predicate, object, graph> quadruples. In general, an RDF dataset might also contain one unnamed default RDF graph, but we do not make use of the default graph. The schema of the contents of the KG is specified using RDF Schema (RDFS)[2] and the Shapes Constraint Language (SHACL)[3].

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

1.2 Purpose of the document

The purpose of this document is to describe the work undertaken in Task 4.2 to develop building blocks for the implementation of a system that serves to assess the concept of AI situational awareness. The current version sets the way for the forthcoming developments of WP 4 as well as Task 5.1 in the AISA project. The KG system architecture proposed and the software described in this document may evolve together with evolving requirements in the remainder of WP4 and Task 5.1.

1.3 Structure and methodology

This document describes the software developed in Task 4.2 (Knowledge Graph to Prolog Mapper and its integration into the Knowledge Graph system). Chapter 2 gives a brief introduction to relevant

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

[3] https://www.w3.org/TR/shacl/

aspects of Prolog, describes the characteristics of schema-oblivious and schema-aware mapping approaches and provides a refined characterizaton of the design problem to be solved. Chapter 3 discusses the realization of the schema-aware approach and introduces three different realization variants. Chapter 4 describes the the conducted performance studies. Chapter 5 goes into detail about the handling of data types and missing values. Chapter 6 describes the integration of the KG-Prolog mapper in the Proof-of-Concept KG system introduced with Deliverable D4.1. Chapter 7 summarizes the results of Task 4.2.

Appendix A provides a glossary of used acronyms. Appendix B provides technical documentation about the Github repository and about conducting the performance studies.

1.4 Relations to other documents

The document is linked to project deliverables:

- AISA D2.1: Concept of Operations for AI Situational Awareness System
- AISA D4.1: Proof-of-concept KG System

2 KG Access from Prolog

In this chapter we introduce the problem of accessing the AISA KG from Prolog. We give a brief overview of relevant aspects of Prolog, discuss two generic approaches for accessing KGs from Prolog, the schema-oblivious and the schema-aware approach, and introduce our design goals and knowledge questions for the KG-Prolog mapper.

Let us, first, briefly revisit the proof-of-concept **KG system** introduced with Deliverable D4.1 with which the KG-Prolog mapper will be integrated The application-independent software components of the KG System are bundled in an instance of **Apache Jena Fuseki**[4] running as a separate **KG server** process. Application-specific engines are realized as **KG modules** implemented by subclassing Java classes provided with the Java library from D4.1 which builds on the Apache Jena Semantic Web Library[5] [3]. The functionality provided by these KG modules is invoked by a control component. Every invocation of a KG module produces a new named graph that is added to the KG. The control component together with the modules run as **KG manager** Java process. The KG modules and the control component communicate with the KG server via SPARQL Query[6], SPARQL Update[7], SPARQL Protocol[8], and the SPARQL Graph Store Protocol[9].

2.1 Prolog – A very brief introduction to relevant aspects

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

SWI-Prolog[10][1] 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 [2] as well as a SPARQL client library.

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.

[4] https://jena.apache.org/documentation/fuseki2/

[5] https://jena.apache.org/index.html

[6] https://www.w3.org/TR/sparql11-query/

[7] https://www.w3.org/TR/sparql11-update/

[8] https://www.w3.org/TR/sparql11-protocol/

[9] https://www.w3.org/TR/sparql11-http-rdf-update/

[10] https://www.swi-prolog.org/

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[11] 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 enables hybrid Prolog+Java applications to be designed and implemented so as to take best advantage of both language systems.

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.

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 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)** [2]of SWI-Prolog [1] 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 a HTTP SPARQL endpoint, as provided by the AISA KG server, from Prolog.

[11] https://jpl7.org/

2.2 Schema-oblivious approach

Following the schema-oblivious approach for accessing the KG from Prolog, every RDF quadruple is represented as a Prolog fact. Figure 1 shows the conceptual architecture of a lightweight implementation of the schema-oblivious 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.

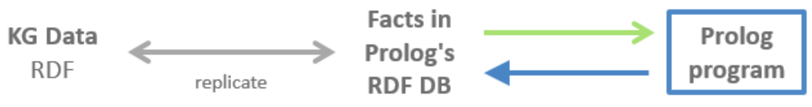
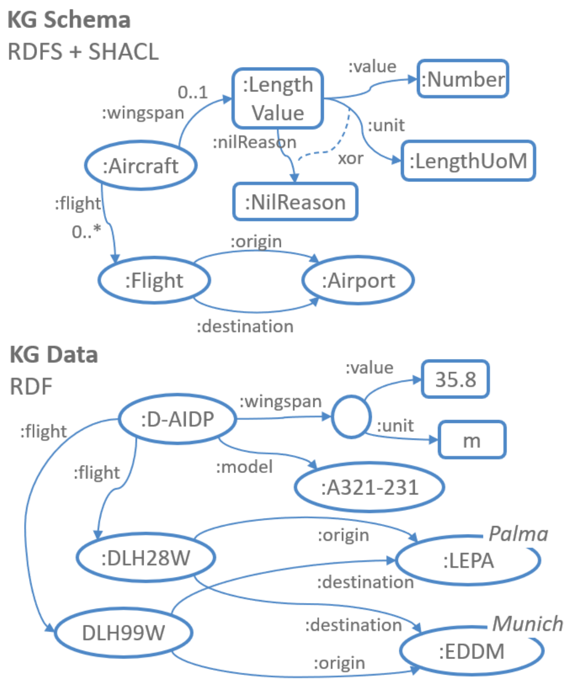


Figure 1 Conceptual architecture of the schema-oblivious approach

Figure 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.



Prolog Facts (in RDF DB)

```
% rdf( Subject, Predicate, Object, Graph )

rdf(:D-AIDP,rdf:type,:Aircraft,:g1).
rdf(:D-AIDP,:wingspan,_:b1,:g1).
rdf(_:b1,:value,'35.8',:g1).
rdf(_:b1,:unit,'m',:g1).
rdf(:D-AIDP,:model,:A321-231,:g1).
rdf(:D-AIDP,:flight,:DLH28W,:g1).
rdf(:D-AIDP,:flight,:DLH99W,:g1).

rdf(:DLH28W,rdf:type,:flight,:g1).
rdf(:DLH28W,:origin,:LEPA,:g1).
rdf(:DLH28W,:destination,:EDDM,:g1).

rdf(:DLH99W,rdf:type,:flight,:g1).
rdf(:DLH99W,:origin,:EDDM,:g1).
rdf(:DLH99W,:destination,:LEPA,:g1).
```

Figure 2 Schema-oblivious mapping example

2.3 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. 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.

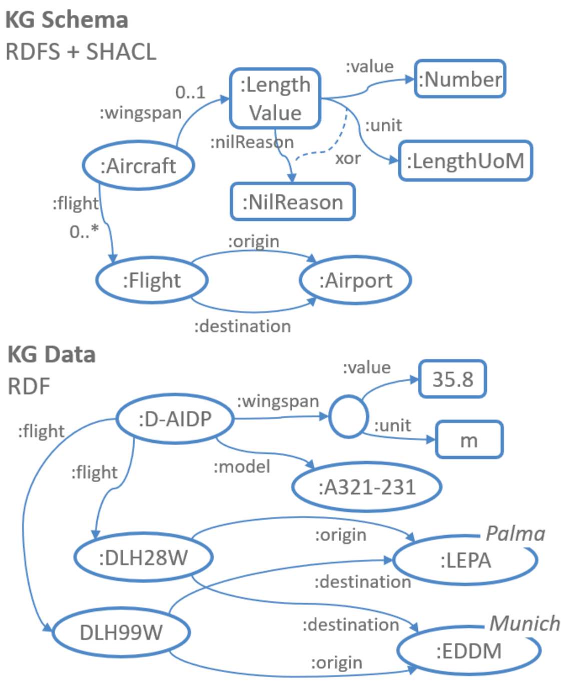
This KG schema directly represents the conceptual schema. The implementation of a schema-aware approach additionally has components for ma,pping generation at design/compile time and mapping execution. This approach is more convenient for Prolog programmers as knowledge about one object is already collected 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 in Prolog the KG schema. 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 programmes. The design goal of this approach is to improve the access to the AISA KG from Prolog.

Figure 3 shows an example for schema-aware mapping. The KG schema and the KG data is the same as in Figure 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 and facts can be interpreted the following way: `:Aircraft` from Graph `:g1` with the id `:D-AIDP` has an optional `:wingspan` val('35.8','m'), an optional `:model` `:A321-231` and a list of `:Flight [:DLH28W, :DLH99W]`. The `:Flight` from Graph `:g1` with the id `:D-AIDP` has 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. 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.



Prolog Schema (as comments)

```
% aircraft(graph,id,wingspan?,model?,flight*)
% flight(graph,id,origin,destination)
```

Prolog Facts

```
aircraft(:g1,:D-AIDP,val('35.8','m'),:A321-231,
         [:DLH28W, DLH99W]).
flight(:g1,:DLH28W,:LEPA,:EDDM).
flight(:g1,:DLH99W,:EDDM,:LEPA).
```

Figure 3 Schema-aware mapping example

2.4 Summary and Refinement of Design Problems

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.

In contrast to our initial plan in the project proposal, we now 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.

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 the nexth chapter. The access via SWI-Prolog's SPARQL client library sketched in the proposal is only one of several possibilities.

The overall design problem is to improve accessing a Knowledge Graph from Prolog by designing a KG-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 Knowledge Graph.

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.

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, so that advanced reasoning tasks in AISA that will be invoked recurrently can easily be realized as Prolog programs, which read from and write to the AISA Knowledge Graph.

3 Realization of schema-aware approach

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 three realization variants. Section 3.2 discusses the generation of 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 will be discussed in Chapter 4 and details about the mapping of data types and missing values common to all mapping variants will be discussed in Chapter 5.

3.1 Overview

Figure 4 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 below, the schema-specific mapping rules or queries will be executed, depending on the realization variant, 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 data against the KG schema using a SHACL validator (see Deliverable D4.1) in order to detect incorrect data prior to mapping. Depending on the realization variant, the input Prolog facts will be asserted in the Prolog database or made available only as virtual facts by Prolog rules. The **Prolog schema,** which is independent of the realization variant, comprises a description of the structure of mapped input facts as well as inheritance rules which represent subclass hierarchies from the KG schema. This schema will be inspected by the Prolog programmer when writing a **Prolog program** which accesses the input Prolog facts.

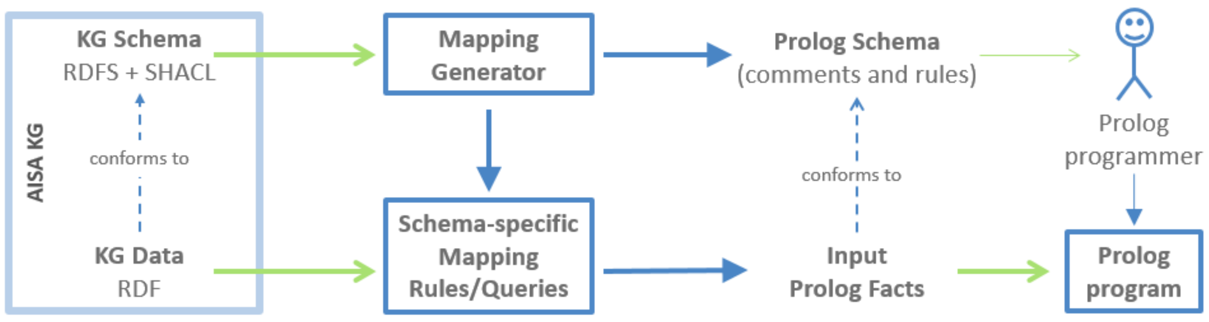


Figure 4 Conceptual architecture of the schema-aware approach

We have investigated and developed three different realization variants. Figure 5 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 in 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 B** 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.

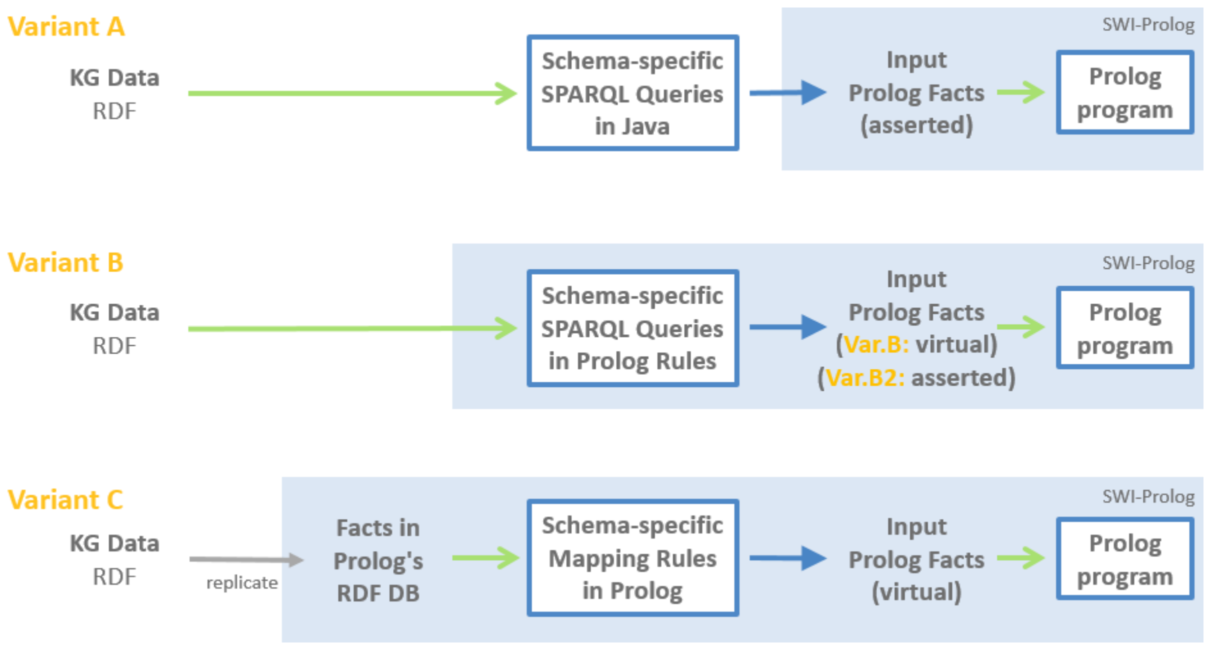

Variant A
KG Data
RDF
Schema-specific SPARQL Queries in Java
SWI-Prolog
Input Prolog Facts (asserted)
Prolog program
Variant B
KG Data
RDF
Schema-specific SPARQL Queries in Prolog Rules
SWI-Prolog
Input Prolog Facts (Var.B: virtual) (Var.B2: asserted)
Prolog program
Variant C
KG Data
RDF
replicate
Facts in Prolog's RDF DB
Schema-specific Mapping Rules in Prolog
SWI-Prolog
Input Prolog Facts (virtual)
Prolog program

Figure 5 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 realize the subclass hierarchies from the KG schema. The Prolog schema is independent of the the three realization variants.

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. According to the Jena API, which is used for this project, a target shape is a SHACL shape, which is defined as an `rdfs:Class`. For each target shape, a schema comment is generated:

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

The given example is the predicate schema for `aixm:OrganisationAuthorityAssociation` with the properties `aixm:type`, `aixm:annotation` and `aixm:theOrganisationAuthority`. The question mark next to Type means that the value is optional, which is defined in the SHACL schema by a missing `sh:minCount (or sh:minCount = 0)` and a `sh:maxCount` = 1. The wildcard next to Annotation marks that this property is a list of zero or multiple concatenated values, which is defined in the SHACL schema by a missing `sh:minCount (or sh:minCount = 0)` and a missing `sh:maxCount` (or `sh:maxCount` > 1). Regarding TheOrganisationAuthority, there is no question mark or wildcard after the argument name in the predicate schema. This means that there is only one value and this value is mandatory.

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

For each inheritance defined in the SHACL schema, an inheritance rule will be printed to the fact file. An inheritance rule consists of the super class and the respective subclass:

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

In this example, the inheritance rule aixm_AirportHeliportResponsibilityOrganisation_Combined is defined. This inheritance rule consists of aixm_PropertiesWithSchedule and its sub class aixm_AirportHeliportResponsibilityOrganisation.

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 gives 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 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 is available[12] open source 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, the first thing that happens is a connection establishment to the Jena Fuseki server and the upload of the input files. After the upload of 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 over and over again. 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 proper data types for 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 and the use of modules in Prolog are also printed to the facts file. Furthermore, inheritance rules for `sh:subClassOf` relations between NodeShapes, 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 run. 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 4.2) are saved to `/output/performance_results.csv`.

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.

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

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 B[13] and its subvariant B2[14] are available open source for experimentation. Before execution of the mapping, Jena Fuseki 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, the first thing that happens is a connection establishment to the Jena Fuseki server and the upload of the input files. After the upload of 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 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. After the creation of the facts file, a short prolog program `/output/program.pl` is consulted and run. 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 4.3) are saved to `/output/performance_results.csv`.

3.5 Mapping Variant C – Mapping Rules in Prolog

Mapping variant C also consists of a Java program. The relevant part of the Knowledge Graphs 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.

Mapping variant C[15] is available open source for experimentation. Before execution of the mapping, Jena Fuseki server should be started. The starting point and main method of the mapping can be found

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

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

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

in the `Shacl2PrologLauncher.java` file. When executing the Shacl2PrologLauncher, the first thing that happens is a connection establishment to the Jena Fuseki server and the upload of the input files. After the upload of the input files, the schema and the data are fetched from Jena Fuseki 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 `sh:subClassOf` relations are printed to the facts file. Next to the static context, inheritance rules and the facts rules are generated and printed. After the creation of the facts file, a short prolog program `/output/program.pl` is consulted and run. 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 4.4) are saved to `/output/performance_results.csv.`

4 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 times include the loading of schema and data into the KG, the mapping generation from the schema, the execution of the generated mappings 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 fragement. This corresponds to the amount of *new* data we expect the AISA KG system has to deal with per 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 4.1 provides details on the setup of the performance studies. Sections 4.2, 4.3, and 4.4 discuss the measurements obtained for each of the realization variants described in the previos chapter. Section 4.5 summarizes the preliminary performance studies by comparing the total execution times of the different variants.

4.1 Setup of Performance Studies

The following performance measurements were generated by running start_performance_test.bat 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. In order to get proper results, Jena Fuseki server needs to be restarted before every execution run of a mapping variant.

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. As mentioned above, 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 48500 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 section 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 steps “Fetching shacl schema” and “Creating KnowledgeGraphClasses and KnowledgeGraphProperties” are mapping-specific, but independent from the number of data copies.

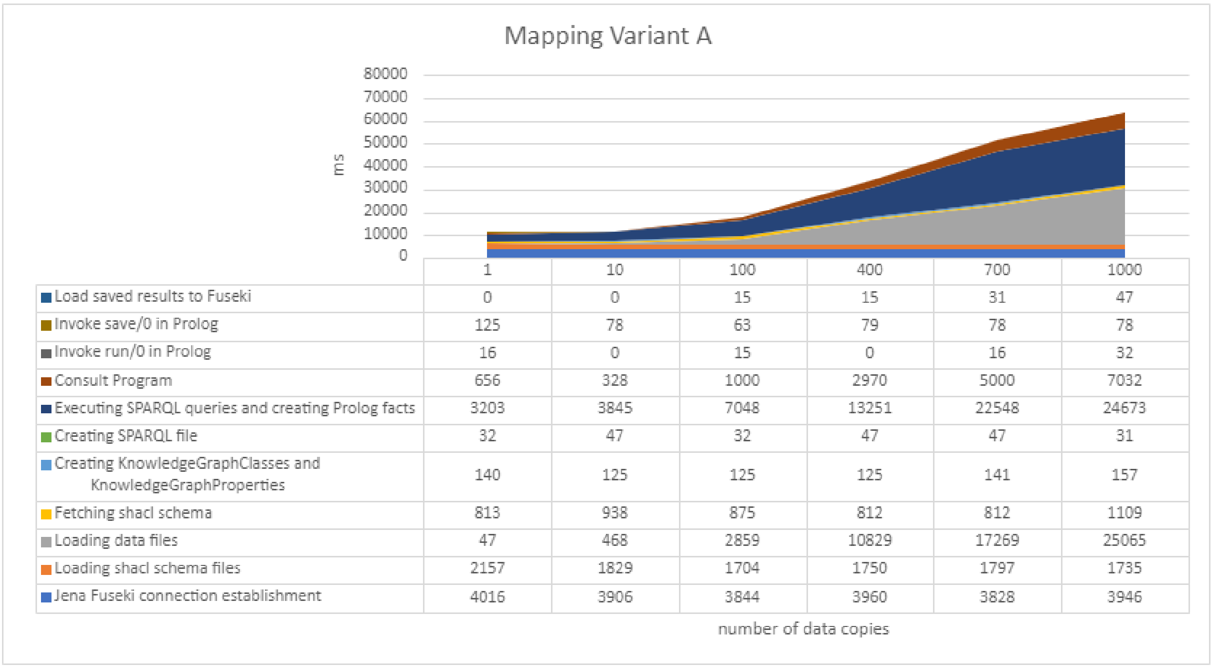
The performance of each variant dependent part will be described more in detail in the following subsections.

4.2 Performance Results of Mapping Variant A

In order to conduct preliminary performance studies, we ran mapping variant A with 1, 10, 100, 400, 700, 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 shows the results of running the mapper with different number 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

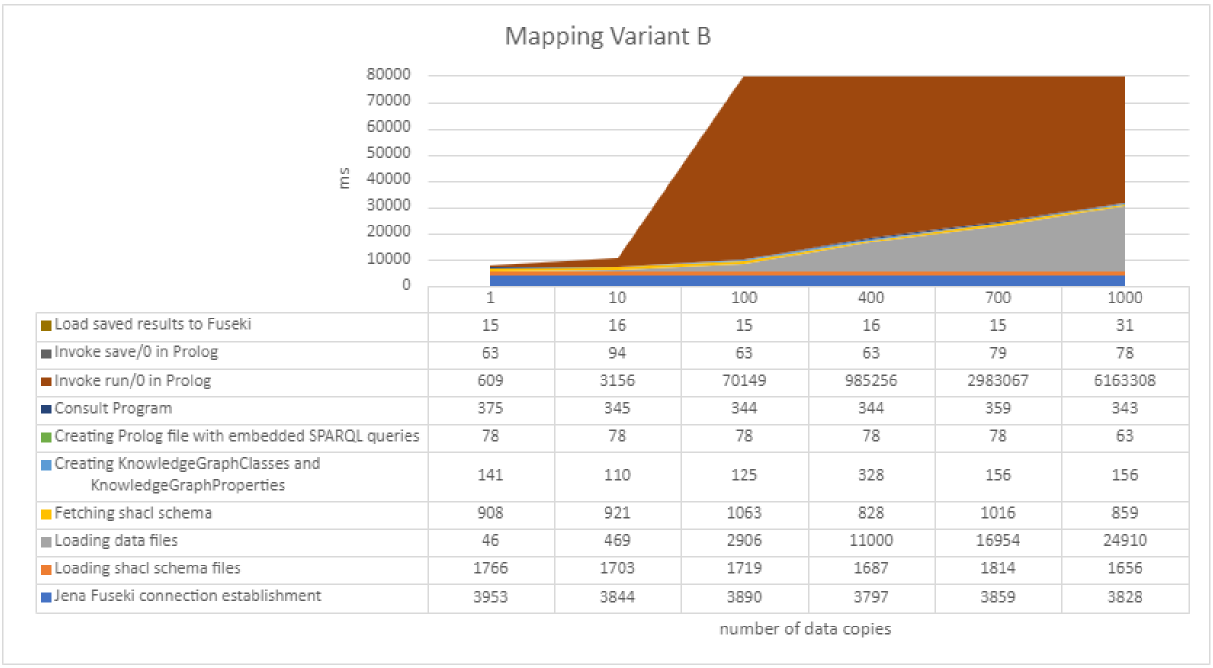
Figure 6 Performance results of mapping variant A. KG data size scaled from 1 data copy (485 RDF quadruples) to 1000 data copies (485000 RDF quadruples).

4.3 Performance Results of Mapping Variant B

In order to conduct preliminary performance studies, we ran mapping variant B with 1, 10, 100, 400, 700, 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 7 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).

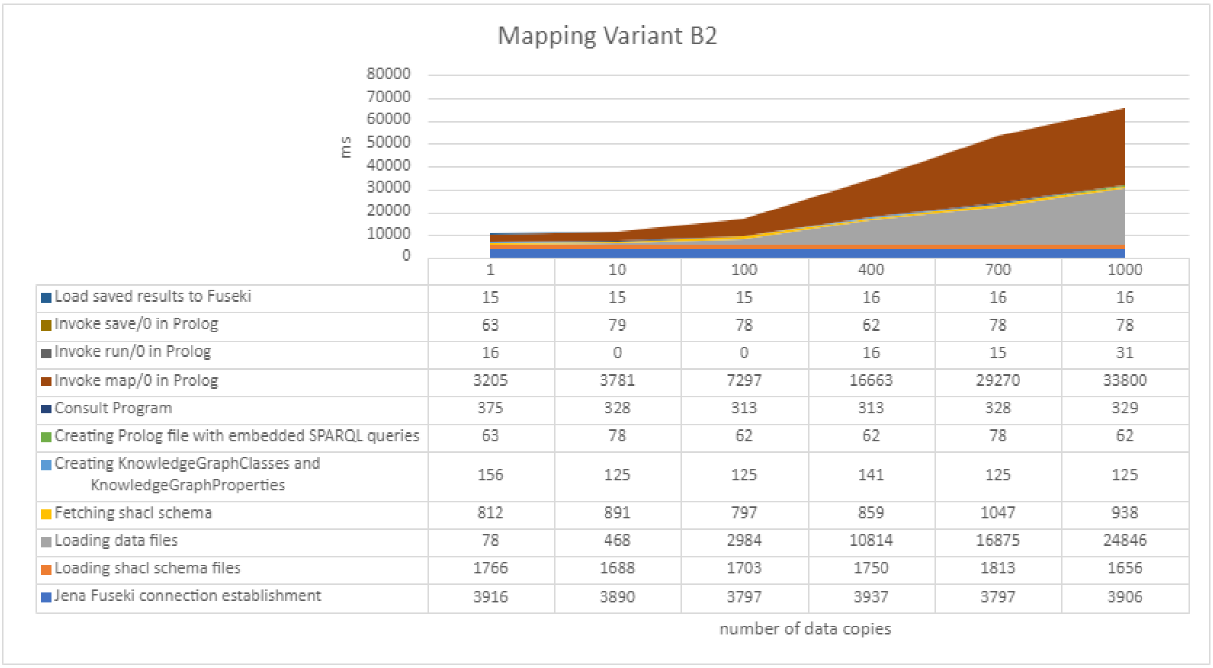
- Creating KnowledgeGraphclasses and KnowledgeGraphProperties: This step is mapping-specific, but independent from the number of data copies.
- Creating Prolog file with embedded SPARQL queries: This step is mapping-specific for variant B, but does not take much time in total, because information, which 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 execution 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 7 Performance results of mapping variant B

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 8 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 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

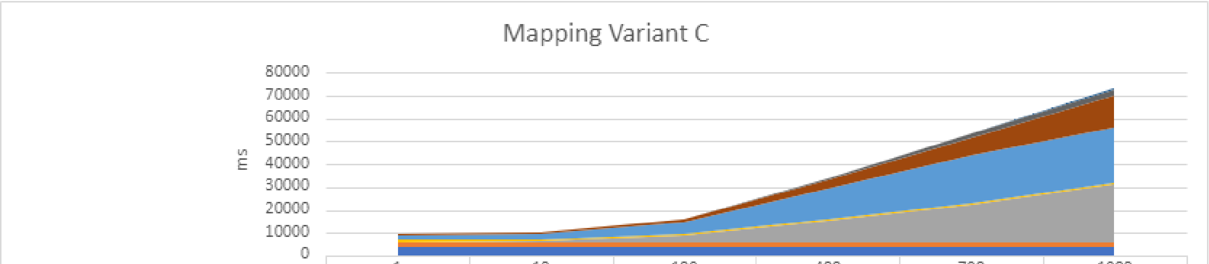
Figure 8 Performance results of mapping variant B2

4.4 Performance Results of Mapping Variant C

In order to conduct preliminary performance studies, we ran mapping variant C with 1, 10, 100, 400, 700, 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 9 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.
- 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.
- Consult Program: This step is dependent on the mapping variant. Consulting /output/program.pl reads program.pl as Prolog resource and loads the data set file into Prolog by calling facts.pl. Therefore, this step scales with the number of data copies.
- Invoke run/0 in Prolog: This step is dependent on the mapping variant.
- Invoke save/0 in Prolog: (same as in varaint 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

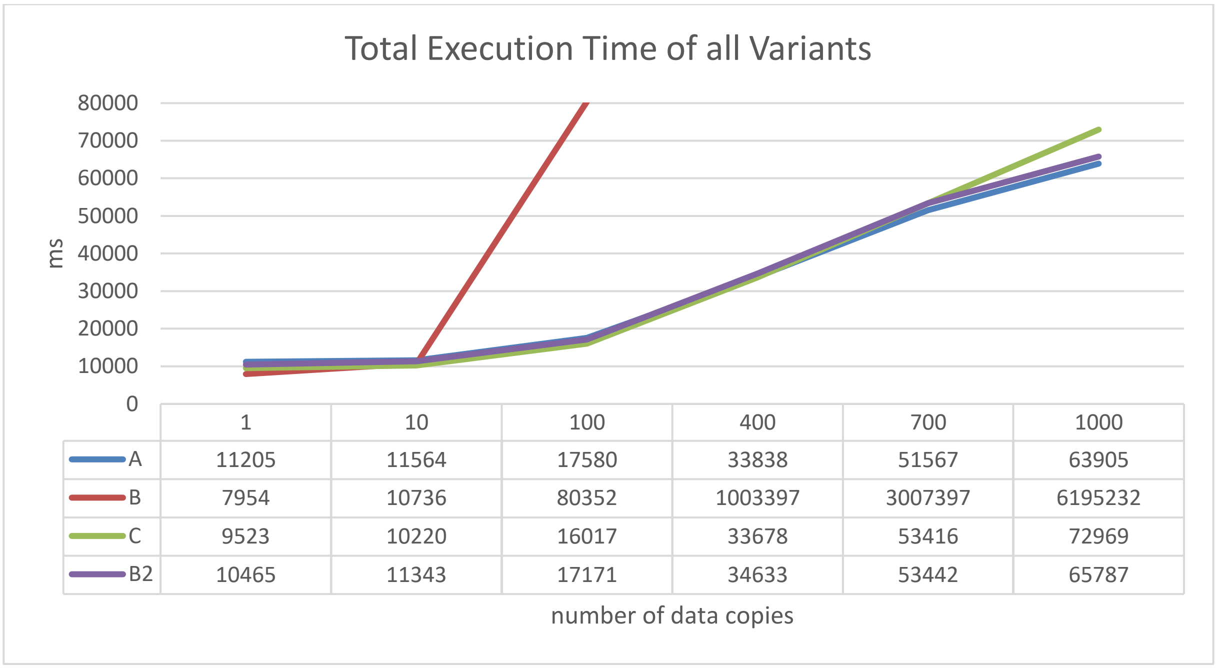
number of data copies

Figure 9 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.

4.5 Summary

We conducted preliminary performance studies regarding three variants and one subvariant of the schema-aware KG-Prolog mapper. Figure 10 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.



	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 10 Total execution time of different mapping variants

5 Handling of Data Types and Missing Values

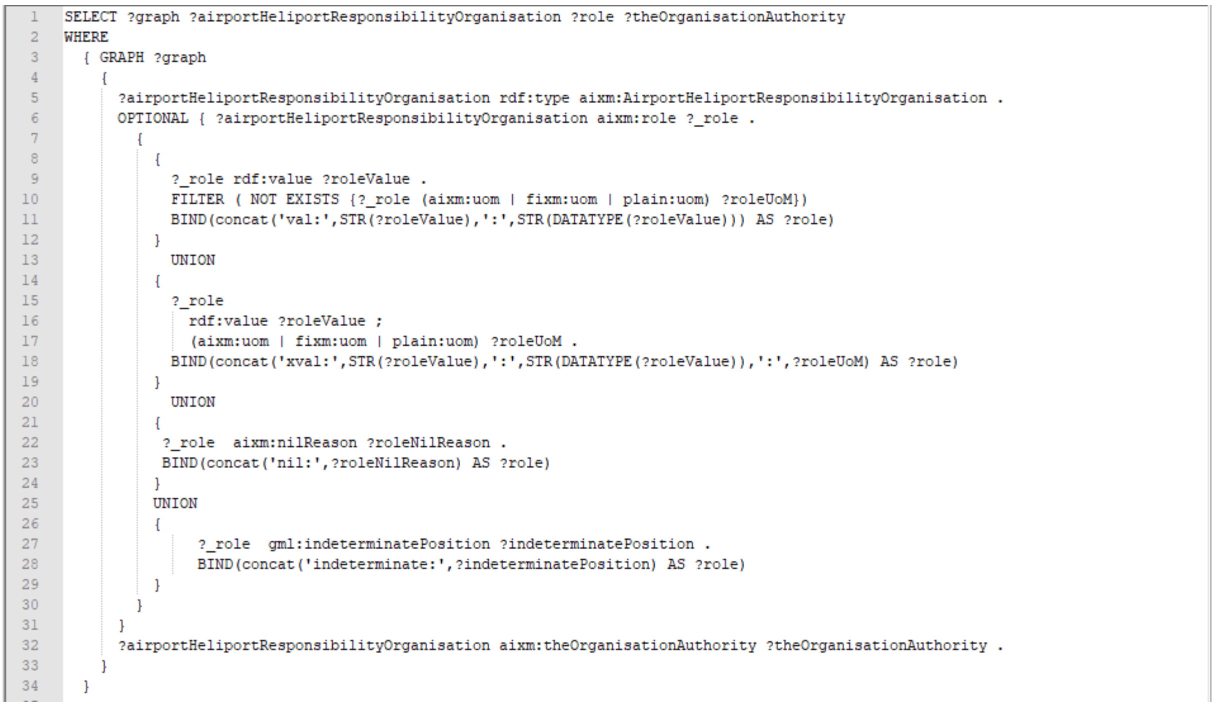
One important part of the implementation is the correct handling of data types, lists and missing values. Our goal was to process the data types into a Prolog readable format and to get exactly the same results from each mapping variant. Generally, The mappers only consider a limited number of data types relevant for the project: `xsd:string`, `xsd:integer`, `xsd:decimal`, `xsd:unsignedInt` and `xsd:dateTime`.

In this chapter we first discuss, in Section 5.1, how reading and processing values and data types is solved differently for each realization. The remaining sections will show examples of mapping nilReason (Section 5.2), values without unit of measurement (Section 5.3), values with unit of measurement (Section 5.4), indeterminate positions and dateTimes (Section 5.5), missing values (Section 5.6), and, finally, the mapping of multi-valued properties to Prolog lists (Section 5.7). 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.

5.1 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 result set will be processed in the right format for Prolog in the next step in Java.

Figure 11 shows an example SPARQL query for `aixm:OrganizationAuthorityAssociation`.



```
SELECT ?graph ?airportHeliportResponsibilityOrganisation ?role ?theOrganisationAuthority
WHERE
  { GRAPH ?graph
    {
      ?airportHeliportResponsibilityOrganisation rdf:type aixm:AirportHeliportResponsibilityOrganisation .
      OPTIONAL { ?airportHeliportResponsibilityOrganisation aixm:role ?_role .
        {
          {
            ?_role rdf:value ?roleValue .
            FILTER ( NOT EXISTS {?_role (aixm:uom | fixm:uom | plain:uom) ?roleUoM})
            BIND(concat('val:',STR(?roleValue),':',STR(DATATYPE(?roleValue))) AS ?role)
          }
            UNION
          {
            ?_role
              rdf:value ?roleValue ;
              (aixm:uom | fixm:uom | plain:uom) ?roleUoM .
            BIND(concat('xval:',STR(?roleValue),':',STR(DATATYPE(?roleValue)),':',?roleUoM) AS ?role)
          }
            UNION
          {
           ?_role  aixm:nilReason ?roleNilReason .
           BIND(concat('nil:',?roleNilReason) AS ?role)
          }
          UNION
          {
               ?_role  gml:indeterminatePosition ?indeterminatePosition .
               BIND(concat('indeterminate:',?indeterminatePosition) AS ?role)
          }
        }
      }
      ?airportHeliportResponsibilityOrganisation aixm:theOrganisationAuthority ?theOrganisationAuthority .
    }
  }
```

Figure 11 SPARQL query of mapping variant A

Given the data:

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

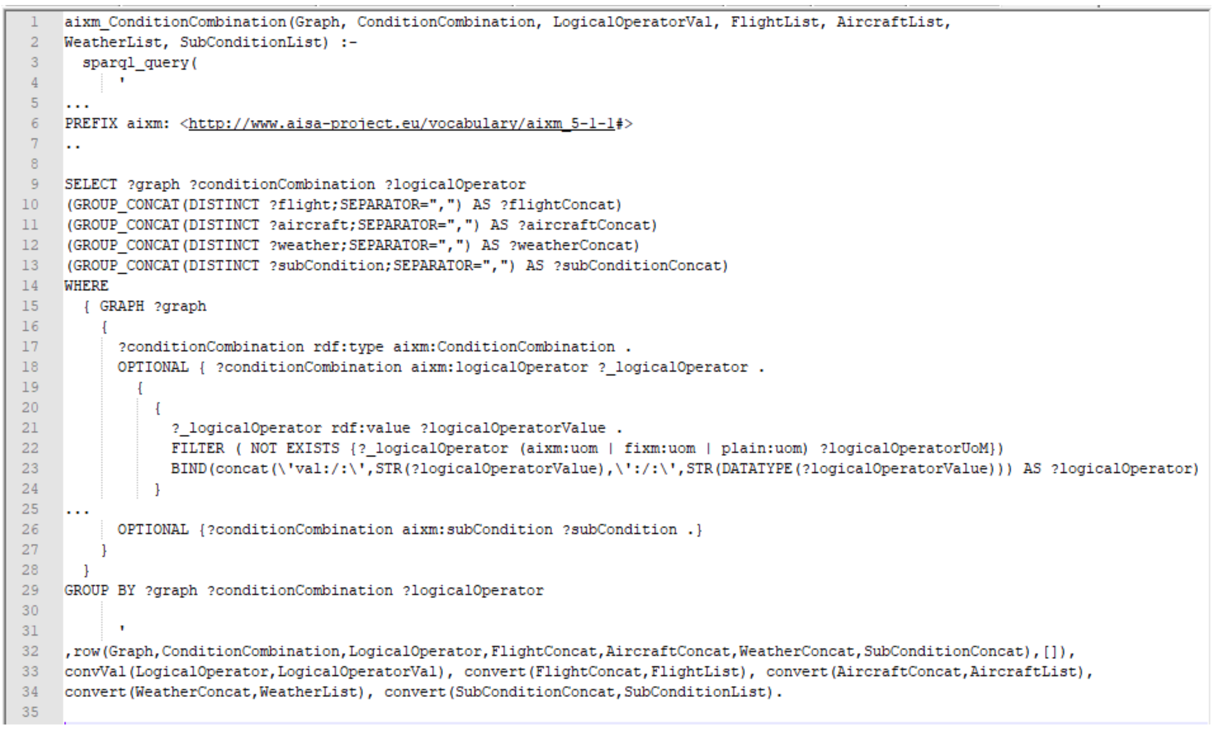
the binding for variable `?role` in the result set of the SPARQL query will be:

```
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')
```

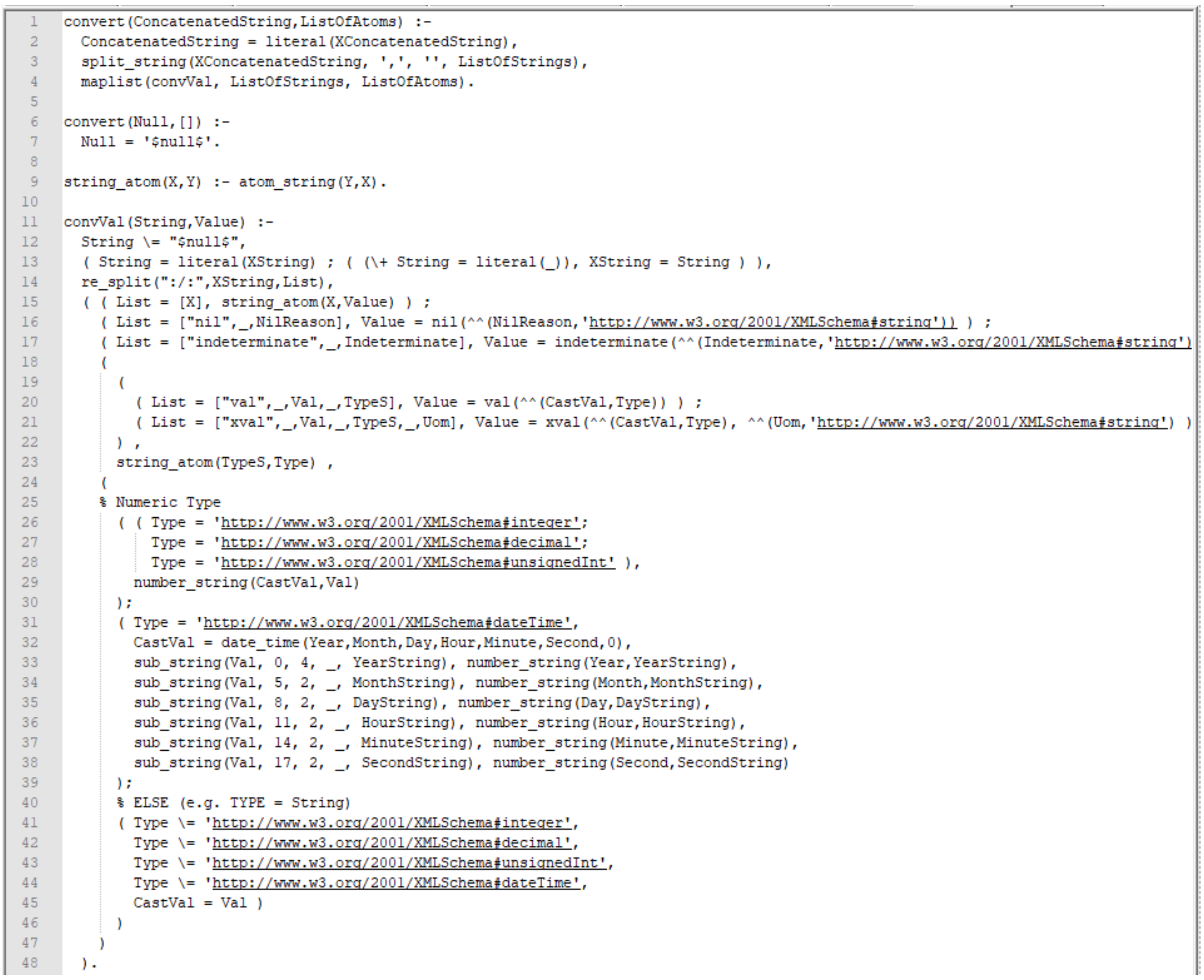
Mapping variant B also puts the value and the data type together within the SPARQL query, which can be seen in Figure 12, just like in variant A. However, the processing of the result set happens in Prolog and is called within the Prolog module:



```
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
```

Figure 12 Prolog module with embedded SPARQL query of mapping variant B

With convVal(Value,ValueVal) and convert(Value,ValueList) in line 33 and 34 the Prolog data type mapping methods (see Figure 13) are called, which process the SPARQL query results in the right format:



```
 1  convert(ConcatenatedString,ListOfAtoms) :-
 2    ConcatenatedString = literal(XConcatenatedString),
 3    split_string(XConcatenatedString, ',', '', ListOfStrings),
 4    maplist(convVal, ListOfStrings, ListOfAtoms).
 5
 6  convert(Null,[]) :-
 7    Null = '$null$'.
 8
 9  string_atom(X,Y) :- atom_string(Y,X).
10
11  convVal(String,Value) :-
12    String \= "$null$",
13    ( String = literal(XString) ; ( (\+ String = literal(_)), XString = String ) ),
14    re_split(":/:",XString,List),
15    ( ( List = [X], string_atom(X,Value) ) ;
16      ( List = ["nil",_,NilReason], Value = nil(^^(NilReason,'http://www.w3.org/2001/XMLSchema#string')) ) ;
17      ( List = ["indeterminate",_,Indeterminate], Value = indeterminate(^^(Indeterminate,'http://www.w3.org/2001/XMLSchema#string')
18      (
19        (
20          ( List = ["val",_,Val,_,TypeS], Value = val(^^(CastVal,Type)) ) ;
21          ( List = ["xval",_,Val,_,TypeS,_,Uom], Value = xval(^^(CastVal,Type), ^^(Uom,'http://www.w3.org/2001/XMLSchema#string') )
22        ) ,
23        string_atom(TypeS,Type) ,
24      (
25      % Numeric Type
26        ( ( Type = 'http://www.w3.org/2001/XMLSchema#integer';
27            Type = 'http://www.w3.org/2001/XMLSchema#decimal';
28            Type = 'http://www.w3.org/2001/XMLSchema#unsignedInt' ),
29          number_string(CastVal,Val)
30        );
31        ( Type = 'http://www.w3.org/2001/XMLSchema#dateTime',
32          CastVal = date_time(Year,Month,Day,Hour,Minute,Second,0),
33          sub_string(Val, 0, 4, _, YearString), number_string(Year,YearString),
34          sub_string(Val, 5, 2, _, MonthString), number_string(Month,MonthString),
35          sub_string(Val, 8, 2, _, DayString), number_string(Day,DayString),
36          sub_string(Val, 11, 2, _, HourString), number_string(Hour,HourString),
37          sub_string(Val, 14, 2, _, MinuteString), number_string(Minute,MinuteString),
38          sub_string(Val, 17, 2, _, SecondString), number_string(Second,SecondString)
39        );
40        % ELSE (e.g. TYPE = String)
41        ( Type \= 'http://www.w3.org/2001/XMLSchema#integer',
42          Type \= 'http://www.w3.org/2001/XMLSchema#decimal',
43          Type \= 'http://www.w3.org/2001/XMLSchema#unsignedInt',
44          Type \= 'http://www.w3.org/2001/XMLSchema#dateTime',
45          CastVal = Val )
46        )
47      )
48    ).
```

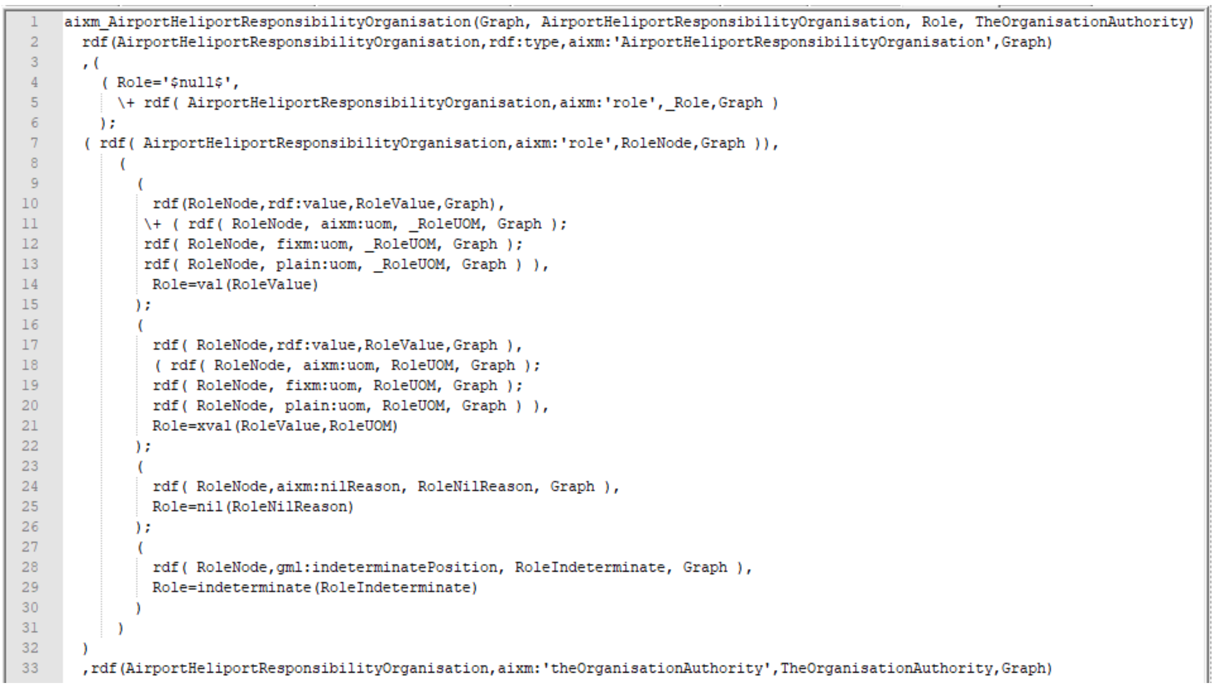
Figure 13 Prolog methods for data type handling of mapping variant B

`convert(ConcatenatedString,ListOfAtoms)`, see Line 1 in Figure 13, 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 Figure 13, is called to handle empty lists.

`convVal(String,Value)`, see Line 11 in Figure 13, 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. Figure 14 shows an example for such a mapping rule.



```
aixm_AirportHeliportResponsibilityOrganisation(Graph, AirportHeliportResponsibilityOrganisation, Role, TheOrganisationAuthority)
  rdf(AirportHeliportResponsibilityOrganisation,rdf:type,aixm:'AirportHeliportResponsibilityOrganisation',Graph)
  ,(
    ( Role='$null$',
      \+ rdf( AirportHeliportResponsibilityOrganisation,aixm:'role',_Role,Graph )
    );
  ( rdf( AirportHeliportResponsibilityOrganisation,aixm:'role',RoleNode,Graph )),
      (
        (
          rdf(RoleNode,rdf:value,RoleValue,Graph),
         \+ ( rdf( RoleNode, aixm:uom, _RoleUOM, Graph );
         rdf( RoleNode, fixm:uom, _RoleUOM, Graph );
         rdf( RoleNode, plain:uom, _RoleUOM, Graph ) ),
          Role=val(RoleValue)
        );
        (
          rdf( RoleNode,rdf:value,RoleValue,Graph ),
          ( rdf( RoleNode, aixm:uom, RoleUOM, Graph );
          rdf( RoleNode, fixm:uom, RoleUOM, Graph );
          rdf( RoleNode, plain:uom, RoleUOM, Graph ) ),
          Role=xval(RoleValue,RoleUOM)
        );
        (
          rdf( RoleNode,aixm:nilReason, RoleNilReason, Graph ),
          Role=nil(RoleNilReason)
        );
        (
          rdf( RoleNode,gml:indeterminatePosition, RoleIndeterminate, Graph ),
          Role=indeterminate(RoleIndeterminate)
        )
      )
  )
  ,rdf(AirportHeliportResponsibilityOrganisation,aixm:'theOrganisationAuthority',TheOrganisationAuthority,Graph)
```

Figure 14 Mapping rule of mapping variant C

5.2 NilReasons

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

Example: `aixm:AirportHeliportTimeSlice` has a property `aixm:designatorIATA`, which is defined as a DataTypeNodeShype `aixm:CodeIATAType`.

```
aixm:AirportHeliportTimeSlice
        a             sh:NodeShape , aixm:FeatureNodeShape , rdfs:Class ;
        sh:and        ( aixm:AIXMTimeSlice ) ;
…
        sh:property  [ sh:maxCount  1 ;
                       sh:node      aixm:CodeIATAType ;
                       sh:order     4 ;
                       sh:path      aixm:designatorIATA
                     ] ;
… .
aixm:CodeIATAType  a  aixm:DataTypeNodeShape , sh:NodeShape ;
        sh:and       ( aixm:CodeIATABaseType ) ;
        sh:property  [ sh:maxCount  1 ;
                       sh:node      aixm:NilReasonEnumeration ;
                       sh:order     1 ;
                       sh:path      aixm:nilReason
                     ] ;
        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:minCount  1 ;
                                       sh:order     1 ;
                                       sh:path      aixm:nilReason
                                     ]
                      ]
                    ) .
```

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 following data:

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

The mapping result will look the following way:

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

5.3 Values without Unit of Measurement

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

Example: `aixm:AirportHeliportTimeSlice` has a property `aixm:designator`, which is defined as a DataTypeNodeShape `aixm:CodeAirportHeliportDesignatorType`.

```
aixm:AirportHeliportTimeSlice
      a             sh:NodeShape , aixm:FeatureNodeShape , rdfs:Class ;
      sh:and        ( aixm:AIXMTimeSlice ) ;
…
      sh:property   [ sh:maxCount  1 ;
                      sh:node      aixm:CodeAirportHeliportDesignatorType ;
                      sh:order     1 ;
                      sh:path      aixm:designator
                    ] ;
….
aixm:CodeAirportHeliportDesignatorType
      a             aixm:DataTypeNodeShape , sh:NodeShape ;
```

```
        sh:and        ( aixm:CodeAirportHeliportDesignatorBaseType ) ;
        sh:property   [ sh:maxCount  1 ;
                        sh:node      aixm:NilReasonEnumeration ;
                        sh:order     1 ;
                        sh:path      aixm:nilReason
                      ] ;
        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:minCount  1 ;
                                         sh:order     1 ;
                                         sh:path      aixm:nilReason
                                       ]
                        ]
                      ) .
```

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 following data for `aixm:AirportHeliportTimeSlice`:

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

The mapping result is:

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

5.4 Values with Unit of Measurement

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

Example: `aixm_AirportHeliportTimeSlice` has a property `aixm:fieldElevation`, which is defined as a DataTypeNodeShape `aixm:ValDistanceVerticalType`.

```
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
                                      ]
                       ]
                     ) .
```

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

Given the following data for `aixm:AirportHeliportTimeSlice`:

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

The mapping result is:

```
% aixm_AirportHeliportTimeSlice(Graph, AirportHeliportTimeSlice, …
FieldElevation?, …
```

```
aixm_AirportHeliportTimeSlice(graph:'149_donlon-data.ttl', s1:'AHP_EADH', …
xval("18"^^xsd:'string',"M"^^xsd:'string'), …).
```

5.5 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: `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
                     ] .
```

Given the following data:

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

The resulting fact and mapping of the `xsd:dateTime` and `gml:indeterminatePosition` will look the following:

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

5.6 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 fall back to the atom **'$null$'**, which is also used for variables that are unbound in SPARQL, when using sparql_query.

Example: `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
                     ] ;
…
```

In donlon-data.ttl, this property is missing:

```
  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.
```

The missing value is represented by '$null$' and the final fact after mapping looks the following:

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

5.7 Lists

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

Example: `aixm_AirportHeliportTimeSlice`, which is defined in donlon-shacl.ttl 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
                     ] ;
…
```

In donlon-data.ttl, an `aixm:AirportHeliportTimeSlice` with 2 cities is defined:

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

The final fact after mapping looks the following:

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

If no data for a list is available, then an empty list is represented by []:

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

Other previously mentioned data type mapping rules are also considered within lists; e.g.:
`[val("D1"^^xsd:'string'), val("L"^^xsd:'string'), val("B1"^^xsd:'string')]`

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

With Deliverable D4.1 (Section 3) we introduced the architecture for the AISA KG system, a compact Java library supporting this architecture, and a small proof-of-concept KG system exemplifying the architecture as well as the usage of the Java library.

In this section we describe the integration of Prolog engine and KG-Prolog mapping in the proof-of-concept KG system. 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.

From 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 available the full KG also in the form of RDF quadruples and can flexibly choose between schema-aware and schema-oblivious approach.
- With variant C it is straightforward to cope with very frequent additions of new data to the KG and to incrementally make available the new data to Prolog.

Figure 15 shows the overall approach. The mapping generator gets as input the KG schema and produces as output the schema of Prolog predicates (documenting the schema of input prolog facts for the Prolog programmer which will inspect this docementation 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 to the KG directly via Prolog’s RDF DB.

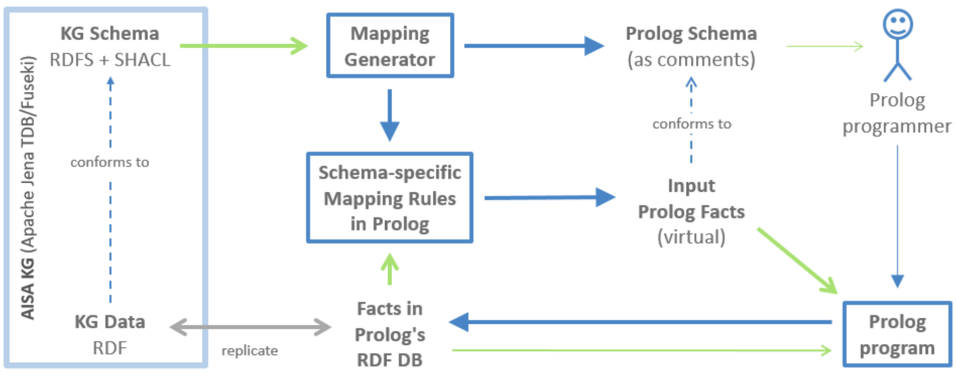


Figure 15 Conceptual architecture of the integrated KG-Prolog mapper

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 Figure 16 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/0 method which is invoked each time the Prolog-based KG module runs.

The generic Prolog code (see Figure 16) 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 in Figure 16) is called from Java (Line 36 in Figure 17) prior to calling the run/0 method (Line 37 in Figure 17).

```
 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 ).
```

Figure 16 Prolog program global.pl

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

Figure 17 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*. Figure 18 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' ) ) .
```

Figure 18 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. 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 get 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 whch 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 gets replicated to the KG.

7 Results

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 sketched in the proposal. 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) execute 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.

References

[1] Wielemaker, Jan, et al. "Swi-prolog." *Theory and Practice of Logic Programming* 12.1-2 (2012): 67-96.

[2] Wielemaker, Jan, Guus Schreiber, and Bob Wielinga. "Prolog-based infrastructure for RDF: Scalability and performance." *International Semantic Web Conference*. Springer, Berlin, Heidelberg, 2003.

[3] Carroll, Jeremy J., et al. "Jena: implementing the semantic web recommendations." *Proceedings of the 13th international World Wide Web conference* (Alternate Track Papers & Posters). 2004.

Appendix A Glossary

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

Table 1 Table of acronyms

Appendix B Technical Documentation

B.1 Overview of GitHub repository

Java code, Prolog programs and example data and schema (RDF, RDF Schema and SHACL) are available in GitHub repository https://github.com/jku-win-dke/AISA-KG-Prolog-Mapper/ .

The repository consists of the following projects and dependencies:

- at.jku.dke.aisa.kg.sample.adsb
    - depends on: at.jku.dke.aisa.kg
- at.jku.dke.aisa.kg.sample.prolog
    - depends on: at.jku.dke.aisa.kg
- at.jku.dke.aisa.kg
    - depends on: at.jku.dke.aisa.mapperC
- at.jku.dke.aisa.mapperA
- at.jku.dke.aisa.mapperB
- at.jku.dke.aisa.mapperC

Additionally, the installation of swipl 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").

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.

B.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: Jena Fuseki 3.17 does not work, rather use the given version 3.16 or try with the latest. Note: In case you want to restart the mapping, make sure to close the Jena Fuseki server first if started manually.

Input files

There are 3 input files which are used for mapping: the SHACL schema, the data and the prefixes.

KG-Schema can be defined in multiple RDFS/SHACL files which will be unionend 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 sever at runtime. The shacl schema is used to create a shacl graph and parse shacl shapes. The graph and the shapes are needed for creation of the SPARQL queries and the Prolog facts.

All shacl schema files which are in the /input/schema/ folder will be 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 will be later retrieved by the SPARQL queries and the results processed to Prolog facts.

All data files which are in the /input/data/ folder will be 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`

Output Files

Depending on which variant is executed, the mapper outputs either 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`

Performance result

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`

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|MappperC.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.

B.3 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 whether the different mapping variants produce the same facts. The test script test.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.pl files are loaded. Then Prolog methods are called, which will show up differences between the results of the files.

Example:

```
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).
```

If the test succeeds the output looks as follows:

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