Investigating the use of Resource Description Framework to model and analyse a Knowledge Landscape

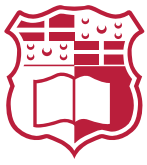
Andre' Vella

Supervisor: Prof. Mark Micallef

Co-Supervisor: Dr Chris Porter

May 2024

Submitted in partial fulfilment of the requirements for the degree of Bachelor of Science (Honours) Computing Science and Mathematics.



L-Università ta' Malta
Faculty of Information & Communication Technology

Abstract

The Knowledge Landscape (KL) of an organisation is defined as the synthesis of knowledge sources such as individuals, Knowledge Assets, and the relationships that may exist between them.

The Resource Description Framework (RDF) is a general framework for representing interconnected data on the web. This project focuses on using Resource Description Framework (RDF) to represent Knowledge Landscapes. As part of the investigation, an ontology-driven framework, RDF-KL is proposed to create an RDF-based knowledge graph that represents a Knowledge Landscape. The proposed ontology in the framework, called OntoKL, formalises Knowledge Landscape entities and their relationships, and is devised following an adaptation of the Ontology Development 101 methodology [1].

The aim of the knowledge graphs is to serve as a representational model that allows organisations to query their KL, thereby mitigating knowledge risks. The framework was implemented in Java using the ONT-API [2] and Apache Jena Framework [3], and the implementation was tested to ensure that it conforms to the framework's requirements. The utility of the framework for representing and analysing a KL is demonstrated with an example from a software engineering environment, where a Knowledge Landscape is converted into a knowledge graph. The representational and analytical capabilities of the graph are assessed by its ability to answer SPARQL queries translated from a set of competency questions derived from the literature review. This evaluation confirms that the knowledge graphs constructed using this framework, do answer the competency questions to some degree, and hence may indeed be effective for representing and analysing KLs. This work aims to encourage further exploration and development in the area of utilising RDF-knowledge graphs to represent and analyse KLs.

Acknowledgements

I would like to thank my supervisor, Prof. Mark Micallef, and my co-supervisor, Dr Chris Porter, for their guidance during my final year project. I would also like to express my gratitude to all the educators within the Faculties of ICT and Science, who, over the past five years, have equipped me with a valuable skill set. Additionally, I am grateful to my family and fiancée, Amber, for their unwavering support and patience throughout my studies.

Contents

List of Figures

List of Abbreviations

CID Converted Distance.

COD Converted Out Distance.

IRI Internationalized Resource Identifier.

KA Knowledge Asset.

KD Knowledge Distance.

KID Knowledge In Distance.

KL Knowledge Landscape.

KMR Knowledge Mobility Risk.

KOD Knowledge Out Distance.

KR Knowledge Risk.

KTR Knowledge Transfer Risk.

OWL W3C Web Ontology Language.

PMR Person Mobility Risk.

RDF Resource Description Framework.

RIC Relative In Centrality.

ROC Relative Out Centrality.

W3C World Wide Web Consortium.

1 Introduction

This chapter starts by introducing Knowledge Landscapes (KLs) and a set of tools for representing interconnected data. It then discusses constraints present in a current solution for representing a KL, showing how these tools can potentially address these issues. Finally, the chapter concludes by presenting the aims and objectives of this final year project and outlining the overall document structure.

1.1 Knowledge Landscape

For organisations carrying out knowledge-intensive tasks, the knowledge held by employees is an asset worth considering. For instance, Bjørnson and Dingsøyr [4] identify that in software organisations, where tasks are driven by knowledge, the main assets are not manufacturing plants, buildings, or machines, but rather the knowledge held by the employees. We refer to a member of an organisation as a Person, denoted by p, while knowledge considered important by an organisation is referred to as a Knowledge Asset (KA). A Knowledge Landscape (KL) of an organisation is defined as the synthesis of Persons, KAs, and the relationships that may exist between them. Furthermore, the landscape is a continuously changing environment due to factors such as persons joining and leaving the organisation, bringing and taking knowledge with them, and persons learning and transferring knowledge to others. A Knowledge Risk (KR) is any event that can increase the chance for an organisation to lose a valuable KA. The fewer KRs associated with an instance of a KL, the healthier the KL is said to be.

1.2 RDF, OWL, and Knowledge Graphs

The RDF-graph is one of the core data structures in the Resource Description Framework (RDF), which is a World Wide Web Consortium (W3C) standard for representing interconnected data on the Web [5]. The RDF-graph consists of RDF-triples, each a 3-tuple in the form of $\tau(s, p, o)$ with subject s, predicate p, and object o,. Each triple encodes a statement; for instance, $\tau(Andre, knowsAbout, graphs)$ encodes the statement $\underbrace{\text{Andre}}_{subject}$ $\underbrace{\text{knows about}}_{predicate}$ $\underbrace{\text{graphs}}_{object}$. The triples allow for the representation of information to be machine-readable but at the same time comprehensible to humans.

An ontology defines a set of representational primitives with which to model a domain of knowledge or discourse [6]. The representational primitives of an ontology

are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). W3C Web Ontology Language (OWL) [7], another W3C standard, explicitly represents ontologies that enable reasoning within a domain by using classes, attributes, and relationships that may be used to infer implicit facts from explicit ones.

Knowledge Graphs are defined as *graphs of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent relations between these entities* [8]. This definition is inclusive of RDF. Knowledge Graphs are recognised for modeling complex data effectively and have gained attention from academia and industry in recent years. To make Knowledge Graphs smarter and enable reasoning, we need to have precise definitions for terms. This can be achieved by integrating an ontology into a Knowledge Graph [5, 9, 10].

Figure 1.1 illustrates a general Knowledge Graph where black nodes and black, solid edges depict explicitly stated information, gray nodes and edges represents ontological information about the object properties *ofType* and *typeOf* (the former being the inverse of the latter), and the dashed edge indicates the inferred information obtained from both the explicit information and the ontology layer.

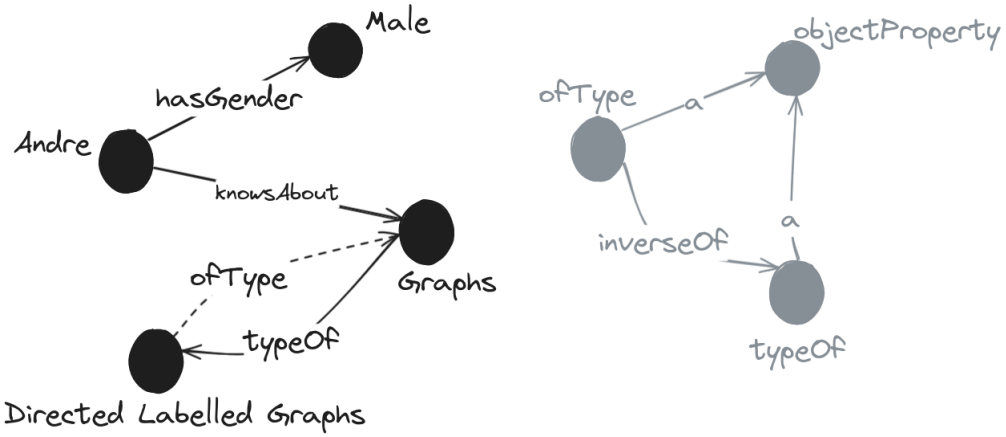
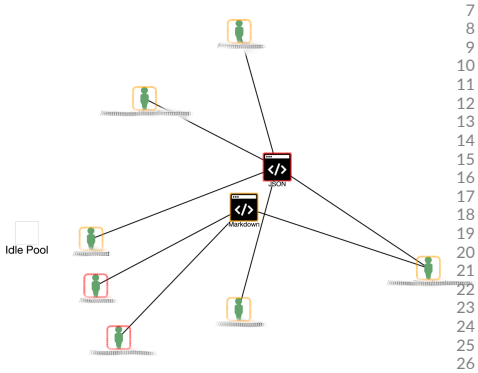


Figure 1.1 A visual representation of a smart Knowledge Graph with gray nodes and edges representing the semantic layer, black nodes and edges for explicit information, and a dashed edge for representing the inferred information.

1.3 Problem Definition and Motivation

Knowledge-intensive organisations needs to ensure that the knowledge they create is retained to maintain their competitive edge. For instance, organisations like software companies, facing high staff turnover, risk losing crucial knowledge when employees leave the organisation. To mitigate KRs, understanding and reasoning about KL is important.

Micallef and Porter developed a tool, “KnowledgeFabric” [11] to identify KRs. It analyses developers’ contributions to a source code repository, constructing a Knowledge Landscape as a simple graph (see Figure 1.2). Weighted edges between persons and knowledge assets signify their relation, with weights used to analyse the health of a given KL instance.



(a) Visual representation of KL

```
{
    "knowledgeMap": {
        "nodes": [
            {
                "id": "JSON",
                "name": "JSON",
                "type": 1
            },
            {
                "id": "johndoe@gmail.com",
                "name": "John Doe",
                "type": 1
            },
            ... (other nodes)
        ],
        "edges": [
            {
                "personId": "johndoe@gmail.com",
                "knowledgeAssetId": "JSON",
                "weight": 518,
                "calculatedWeight": 0
            },
            ... (other edges)
        ]
    }
}
```

(b) KL as a JSON object

Figure 1.2 KL in *KnowledgeFabric*

The resulting KL is considered limited in two aspects. Firstly, it only contains one type of relation (a weighted edge), and secondly, the graph represented as a JSON object lacks a semantic layer for defining entity types and properties.

1.4 Proposed Framework

To explore RDF for modeling Knowledge Landscapes (KLs), a framework for constructing RDF Knowledge Graphs with an underlying OWL ontology is proposed. This ontology formally defines Knowledge Landscape (KL) concepts and relationships, incorporating ideas from existing literature. By leveraging RDF’s representation capabilities and OWL’s formalisation, the framework enables the representation of diverse entity and facilitates KL reasoning. Representing a KL through an RDF Knowledge Graph with an embedded OWL ontology may offer deeper insights, aiding organisation in conceptualising their KL. Building such a model is challenging due to diverse relationships within the landscape, requiring a flexible representation to better understand KLs.

1.5 Aims and Objectives

This project aims to investigate the utilisation of RDF for representing Knowledge Landscapes (KLs) as Knowledge Graphs, allowing organisations to query their KL, and analyse various KRs that may be associated with it. For this aim, we set the following three objectives:

Objective 1 Propose a framework for constructing RDF Knowledge Graphs representing KLs, integrating an ontology layer to formally represent entities within the KL and their interconnections, drawing from existing literature.

Objective 2 Implement the framework to showcase its feasibility.

Objective 3 Develop a case scenario that effectively illustrates the application of the framework and demonstrates how the resulting graph can address a set of competency questions based on existing literature through the use of SPARQL queries

1.5.1 A note on scope

The scope of the framework is to investigate the use of RDF to represent and analyse a KL. A demonstration of how event listeners or software agents can utilise the framework to automatically construct a graph based on a set of events is beyond the scope of this project; however, an example of manual construction is included as part of the case study.

1.6 Document Structure

This document consists of five chapters. Chapter 1 introduces KLs, introduces tools related to RDF, and sets the project's aim and objectives. Chapter 2 provides additional background on RDF and OWL ontologies, and explores entities and relationships within a KL as part of the literature review. Chapter 3 discusses the explored framework and examines its application through a representation of a KL scenario in a Software Engineering organisation, built using a prototype implementation of the framework. Chapter 4 discusses the testing of the implementation to ensure compliance with the framework and evaluates its ability to represent and analyse a KL by addressing a set of competency questions, alongside discussing the results in the context of the modeled scenario. The final chapter concludes the project with a summary, critiques, and possible future works.

2 Background and Literature Review

This chapter provides a background on constructs that help clarify the tools and methods used in the project. This is followed by a literature review, introducing relevant theory concerning Knowledge Landscapes (KLs).

2.1 RDF Graphs

In Section 1.2, we defined an RDF-graph as a set of RDF-triples, $\tau(s, p, o)$, without specifying the components s, p, and o. The subject s must be an Internationalized Resource Identifier (IRI) [12] or a blank node, the predicate an IRI, and the object o can be an IRI, a blank node, or a literal. IRIs signify entities within the universe of discourse, termed resources in RDF ranging from physical objects to abstract concepts. Literals are tuples with a lexical form, a datatype IRI, and an optional language tag.

The datatype is characterised by a function known as the **lexical-to-value mapping**, which has domain referred to as the value space. This function associates a literal value with a lexical value. For example, consider the datatype $xsd : boolean$ [13] whose referent has lexical-to-value mapping given by $g = \{\text{“0”} \mapsto false, \text{“1”} \mapsto true, \text{“}false\text{”} \mapsto false, \text{“}true\text{”} \mapsto true, \}$, the literal $\langle xsd : boolean, \text{“0”} \rangle$ implies that $g(\text{“0”}) = false$, i.e., the lexical form “0” is a representative of the boolean value false. For the reader to understand this project, it is sufficient to know that graphs consist of triples relating entities to other entities or datatypes. However, for more in-depth definitions of RDF, one may refer to Appendix C.

2.1.1 SPAQRL

SPARQL [14] is a W3C recommendation that serves as a query language for RDF graphs. The results from these queries can themselves be RDF graphs or result sets. Result sets are presented in tabular form. A binding in a result set refers to a pair consisting of a variable and an RDF term, which could be an IRI, a Literal, or a Blank Node. Result sets are illustrated as tables, where variables are represented as column headers, and each row corresponds to a solution.

2.1.2 Tools

There are a number of tools for building, reading, and serialising RDF in various environments [3, 15–17]. For instance, Apache Jena [3] is a Java framework providing

an API to extract data from and write to RDF graphs. An RDF-graph built with Jena is referred to as a `Model`. Within the Jena framework, there is a tool called ARQ, which is a SPARQL-compliant engine used to run a query against an RDF graph represented by a Jena model.

2.2 OWL Ontology

The main component of an OWL ontology, is its set of axioms. Axioms are statements that define what is true in the domain [18]. OWL ontologies are primarily exchanged as RDF graphs [19]. In Appendix D, a detailed explanation of some fundamental axioms is provided, along with the respective RDF representations for some of them.

2.3 OWL Reasoners

An OWL reasoner is an inference machine that derives logical consequences from axioms in an OWL ontology. A recent survey [20] that analysed 73 OWL reasoners and 22 systems revealed that, majority of the reasoner are also able to check ontology consistency and querying whether the ontology contains some axiom. The survey noted positive aspects regarding reasoners such as scientific grounding, diversity in development environments, with Java being the most prominent, and active projects on GitHub encouraging contributions. However, challenges include limited documentation, many reasoners still in prototype phase, and poor maintenance where only 28 out of 73 are actively maintained.

2.3.1 Tools

Protege is a well-known ontology editor among ontologists and Semantic Web developers, developed by Stanford University[21]. It offers a user-friendly graphical interface for creating and managing OWL ontologies. Built upon OWL-API [22], a Java-based tool for ontology management, Protege not only facilitates parsing and manipulation of OWL ontologies but also supports reasoning capabilities. ONT-API[2], implements the interface of OWL-API, but takes a more RDF-centric approach by ensuring that all ontological changes, such as axiom additions or removals, are directly reflected in the RDF-graph itself rather than being serialised afterwards [19].

2.4 Ontology Development Methodology

There are different ontology development methodologies, with one of the most cited being the Ontology Development 101 by Noy and McGuinness, which has over 8000 citations on Google Scholar [1]. This methodology consists of seven steps, which are as follows.

Step 1 Determine the domain and scope of the ontology.

Step 2 Consider reusing existing ontologies.

Step 3 Enumerate important terms in the ontology.

Step 4 Define classes and class hierarchy.

Step 5 Define the properties of classes.

Step 6 Define the facets of the slots.

Step 7 Create instances.

This process is not necessarily a start to finish process, but rather an iterative one, where some steps may be repeated until the requirements are met.

2.5 SWRL

SWRL [23] is a rule language based on OWL, using RDF triples to express rules as implications with an antecedent (body) and a consequent (head). For example, we may have triples encoding the rule $hasParent(?x, ?y), hasBrother(?y, ?z) \rightarrow hasUncle(?x, ?z)$, indicating that if x has a parent y and y has a brother z, then z is x's uncle. The OWL reasoner HermiT, supports these rules when they are DL-safe to ensure decidability [24].

2.6 Knowledge Graphs

Hogan et al. offer an inclusive definition of a Knowledge Graph [8], describing a Knowledge Graph as any data graph structure intended to accumulate, expanding in size to convey knowledge about the real world. In this structure, nodes represent entities of interest, while edges represent various relations between them. This data graph adheres to a graph-based model, which may take the form of a directed graph with labelled edges, a property graph, among other possibilities. In this project, we represent knowledge graphs using RDF-graphs, which essentially function as directed graphs with labelled edges at their core.

2.6.1 What is the difference between an Ontology and a Knowledge Graph?

The term “knowledge graph” is often used interchangeably with “ontology”, though at times this usage is incorrect or misleading [25]. Knowledge graphs are typically described as very large ontologies. Additionally, some literature suggests that knowledge graphs extend ontologies by offering extra features, such as built-in reasoners for deriving new knowledge [25]. Due to these differences, Erhlinger and Wöb [25] define a knowledge graph as follows:

> *A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge.*

This definition is **deemed important** as it lays the foundation for the architecture of the proposed framework in this project (see Section 3.2).

2.7 Literature Review

In this section, we review literature related to properties of Knowledge Assets (KAs), properties of persons, representing Knowledge Landscapes (KLs), and methods for analysing Knowledge Risks (KRs). This literature is important for this project, as the overall aim of this project is to represent and analyse KLs using RDF.

2.7.1 Knowledge Assets

Knowledge Assets (KAs) represent knowledge that may or may not be known by persons in the organisation but has been deemed to be of value to it. No matter which definition of knowledge one adopts and which knowledge resource is considered, it is useful to appreciate various attributes of knowledge. An attribute is a dimension along which different instances of knowledge can vary. An attribute dimension may consist of a range of values or may be categorical. Knowledge attributes of interest form the axes of an attribute space [26]. Assessing various attributes of knowledge enhances one’s “understanding of knowledge resources and their processing” [26]. *In this project, we refer to such attributes as Knowledge Asset Features.* Several KA Features are identified by Micallef and Cachia [27] and are studied from a Software Engineering organisation perspective. The identified features are:

- **Category**: In a software engineering setting, there are three identified categories of knowledge: *computer science*, *business*, and *general*. Thus, in this setting, a KA can be assigned one of these three values for the Category feature, or none if the category is undefined.

- **Visibility**: A KA can be assigned to be Tacit or Explicit for the Visibility feature, or none if the visibility is undefined. Tacit knowledge refers to knowledge that resides purely within its *knower*, and explicit knowledge refers to a KA that has been explicitly articulated, codified, or otherwise communicated.
- **Social Classification**: A KA can be assigned as *individual* or *social* for the Social Classification feature, or none if the social classification is undefined. Individual knowledge refers to knowledge that is created and resides within an individual. On the other hand, social knowledge is created and inherent in the collective actions of a group, with no individual member possessing all the knowledge.
- **Operational Classification**: A KA can be assigned as *declarative (know-about)*, *procedural(know-how)*, *causal (know-why)*, *conditional (know-if)* and *relational (know-with)* for the Operational Classification feature, or none if the operational classification is undefined. Making this property visible may lead to a situation whereby management realise that there is a potentially harmful imbalance in the types of knowledge that its employees know (e.g. too much know-how as opposed to know-about).

In this project these four features KAs are referred to as base features and are to be considered when modelling KA in a landscape. However, since different organisations might have different requirements for categorising their KAs based on different needs, a representational model should allow for the extension of features.

Modelling how KAs can be related to other KAs allows organisations to reason about structural properties of their KL and also carry out more efficient knowledge transfer exercises, skills analysis, interview design, and so forth [27]. Micallef and Cachia [27], suggests three types of such relationships:

- **Related relationship**: This is a weak relationship since k_1 being related to k_2 means that k_1 is related in any way to k_2. Trivially, this relationship is symmetric.
- **Composition relationship**: A KA k_1 is said to be composed of another KA k_2 if k_2 represents a subset of the knowledge represented by k_1. This relationship is transitive since if we consider the relationship where k_2 is composed of k_3, then this means that k_3 represents a subset of the knowledge represented by k_2, and since k_2 represents a subset of the knowledge represented by k_1, it follows that k_3 represents a subset of the knowledge of k_1. Trivially, this relationship is asymmetric since a superset of the knowledge of k_1 cannot also be its subset.
- **Dependency relationship**: A KA k_1 is said to be dependent on k_2 if a person must learn/utilise k_2 first before using/learning k_1. For clear reasons, this property is

said to be asymmetric, and a composition relationship is said to be a sub-property of this relationship.

2.7.2 Persons and Knowledge

Persons are defined as members of an organisation. Teams are defined to be subsets of persons that share something common. Modelling teams allows for measure of knowledge with a human focus such as number of managers, number of directors, number of full time temporary employees, number of IT staff, etc [28]. Earl [29] proposes a taxonomy of strategies for knowledge management, one of which is called the Cartographic School. The main idea of this approach is to connect KA with persons in a "yellow pages" directory fashion, ensuring that knowledgeable people in the organisation are accessible to others for advice, consultation, or knowledge exchange . Mark and Cachia [30], elaborate further on this approach by modelling *how much* each person knows a particular KA, enabling them to detect knowledge mobility risks, Knowledge Transfer Risk (KTR)s, model staff turnover scenarios, and also analyse various structural characteristics of their organisational knowledge.

2.7.3 Analysing a Knowledge Landscape

Knowledge Risk (KR) refers to the event that can lead an organisation to loose its valuable KAs. In the following subsections, we explore two different types of risks and associated metrics for analysing them.

2.7.3.1 Detecting Knowledge Transfer Risk

Knowledge Transfer Risk (KTR) arises when transferring a KA between individuals becomes more difficult. Cummings and Teng found out that the more embedded the KA is, the harder it is to transfer [31]. Micallef and Cachia [27] define embeddedness as $embeddedness(k) = |dep(k)|$, where k is a KA and $dep(k)$ denotes the set of KAs, that k depends on. A higher embeddedness value indicates greater KTR. The authors also suggest a metric to measure the knowledge gap between two individuals, enhancing the likelihood of successful knowledge transfer. This metric is given by $distance(p_{src}, p_{rec}, k) = \sum_{k' \in dep(k)} dist_{abs}(p_{src}, p_{rec}, k')$, with p_{src} as the knowledge source, p_{rec} as the recipient of knowledge, and $dist_{abs}(p_{src}, p_{rec}, k)$ calculating the absolute difference in knowledge magnitudes. Negative values are adjusted to zero, since the primary focus is on situations where the source has more knowledge than the receiver.

2.7.3.2 Detecting Knowledge Mobility Risk

Knowledge Mobility Risk (KMR) describes the risk of a company losing valuable knowledge due to an employee's leaving the organisation. Micallef and Colombo [32] developed a method to categorise KMR as low, medium, or high, based on the Relative Out Centrality (ROC) Metric from Botafago et al. [33]. They also adapted the Relative Out Centrality Metrics [33], to assess Knowledge Assets from a person's perspective.

Definition 2.1 (Distance Matrix). *Let G be a directed graph with vertices $\{v_1, \ldots, v_n\}$. The distance matrix is said to be the $n \times n$ matrix C defined by:*

$$M_{ij} = \begin{cases} \infty & \text{no directed path between } v_i \text{ and } v_j \\ d(v_i, v_j) & \text{otherwise} \end{cases}$$

where $d(v_i, v_j)$ is the length if the shortest path between v_i and v_j.

Definition 2.2 (Converted Distance Matrix). *Let G be a directed graph with vertices $\{v_1, \ldots, v_n\}$. C is an $n \times n$ matrix defined by:*

$$C_{ij} = \begin{cases} M_{ij} & \text{if } M_{ij} \neq \infty \\ K & M_{ij} = \infty \end{cases}$$

where M is the distance matrix of G, K is a finite conversion constant, that is arbitrary but should be different from any finite entry in M. For example, we can set $K = |V(G)|$, ensuring that K is larger than every finite entry in M, i.e., since the length of a path is at most $|V(G)| - 1$.

Definition 2.3 (Centrality Metrics). *Let G be a directed graph with vertices $\{v_1, \ldots, v_n\}$, and C be the converted distance matrix for G. The Converted Out Distance (COD) for a node v_i is defined as $COD_{v_i} = \sum_{j=1}^{n} C_{ij}$. The converted distance Converted Distance (CID), is given by the sum of all entries in C, i.e. $CD = \sum_i \sum_j C_{ij}$. The ROC of v_i is given by $ROC_{v_i} = \frac{CD}{COD_{v_i}}$ and the Relative In Centrality (RIC) for a node v_i is given by $RIC_{v_i} = \frac{CD}{CID_{v_i}}$*

Definition 2.4 (Adapted Distance Matrix). *Let G be a directed graph, with $k_1, \ldots, k_n$ KA vertices and $p_1, \ldots, p_m$ person vertices. The distance matrix is said to be the $n \times m$ matrix C defined by:*

$$M_{ij} = \begin{cases} K & \text{if } knows(g, p_j, k_i) = 0 \\ \frac{maxKM}{knows(g,p_j,k_i)} & \text{otherwise} \end{cases}$$

where $maxKM$ is the highest knowledge magnitude assigned to knowledge relationships in the graph. K is a constant which represents infinity in the graph. K is taken to be the constant $maxKM + 1$.

Definition 2.5 (Adapted Centrality Metrics)**.** *Let G be a directed graph, with $k_1, \ldots, k_n$ KA vertices and $p_1, \ldots, p_m$ person vertices. The Knowledge Distance (KD), Knowledge In Distance (KID) for a KA vertex and the Knowledge Out Distance (KOD) for a person vertex are defined as follows:*

$$KD = \sum_{i=1}^{n} \sum_{j=1}^{m} M_{ij}$$

$$KID(k_i) = \sum_{j=1}^{m} M_{ij}$$

$$KOD(p_j) = \sum_{i=1}^{n} M_{ij}$$

The metric for KMR is defined as $KMR(k_i) = \frac{KID(k_i)}{KD}$. The metric for Person Mobility Risk (PMR) is defined as $PMR(p_j) = \frac{KOD(p_j)}{KD}$.

Method 1 (Classifying KMR)**.** *The following method is proposed to classify the KMR associated with a KA:*

1. *Let σ be the standard deviation of the KMR values for all KA vertices in the graph*
2. *Let μ be the average of the KMR values for all KA vertices in the graph*
3. *Colour all nodes with $KMR \leq \mu$ green (representing KAs with smallest KMR relative to the other KAs)*
4. *Colour all nodes with $\mu > KMR \leq (\mu + \sigma)$ orange (representing KAs with medium KMR relative to the other KAs) in this paper)*
5. *Colour all nodes with KMR > (μ + σ) red (representing KAs with highest KMR relative to the other KAs)*

2.8 Conclusion

This chapter provided a background on the tools that are relevant to this project, namely RDF-graphs, OWL ontologies which themselves can be expressed as RDF-graphs, and OWL reasoners. A review of the existing literature on various elements of a KL is provided, where the key components, which are the KAs and persons, were explored in more detail. Additionally, a list of metrics was provided to analyse various KRs as part of the review. In the next section, we introduce the framework that can be utilised to construct KLs in terms of RDF.

3 Proposed Framework: RDF-KL

In this chapter, we present a framework for constructing RDF knowledge graphs to represent Knowledge Landscapes (KLs), detailing its development and definition. The first section introduces competency questions formulated to guide and assess the framework's effectiveness in representing and analysing a KL. The second section outlines the framework's architecture. The third section covers the development and definition of the ontological layer. The fourth section describes the methods used to populate the ontology with data and how the resulting knowledge graph is generated. The fifth section provides some remarks. The sixth section introduces a prototype of the framework along with an explanation of how it can be used.

3.1 Defining the Competency Questions

Based on the literature review, a set of competency questions was derived to better understand what a representation of a Knowledge Landscape (KL) should be capable of representing and the type of analysis that can be conducted using it. The first five questions are user queries focused on retrieving data from the knowledge graph directly based on the input. These test the representational ability of KLs, including the successful representation of persons and teams (see Section 2.7.2), KAs (see Section 2.7.1), and the relationships between them. The remaining four questions test the analytical side, examining the structural properties such as composition and dependency relationships of KAs (see Section 2.7.1) and the processing of magnitude relationships (see Section 2.7.2) for deeper analysis and understanding. This includes measuring metrics for analysing a certain KR, such as using the Embeddedness metric to analyse KTR (see Section 2.7.3.1).

CQ1 Given a person and a KA, to what extent does the person know the KA?

CQ2 Who knows a given KA, and to what extent?

CQ3 What KAs are known by a person in a KL?

CQ4 What KAs have some specific attribute?

CQ5 What are the KAs known by a team and to what extent?

CQ6 What KAs are related to any way to a given KA?

CQ7 What are the prerequisite KAs that a person must know in order to learn/utilise a given KA?

CQ8 What KAs are considered to be subsets of knowledge encapsulated in a given KA?

CQ9 What are the KRs associated with a KL?

These questions are ordered in terms of complexity. The first five questions are considered to have low complexity as they are derived directly from the representation. Questions 6 to 8 are considered to have medium complexity since they require reasoning on the structural properties of Knowledge Assets (KAs). The final question, CQ9, is deemed to have the highest complexity, since it needs to utilise structural properties of the representation and use appropriate metrics used to analyse the associated KRs.

3.2 Architecture

The proposed framework utilises OWL to develop OntoKL, an ontology that semantically enriches terms and enables reasoning. This ontology is populated through specific methods and analysed using an OWL reasoner to infer new axioms. Additionally, an abstraction called an *updater* is explained, which allows organisations to enhance the resulting knowledge graph with custom rules specific to their needs. The populated ontology is represented as an RDF knowledge graph, which is then analysed through SPARQL queries to explore its representational features and the type of analysis that can be conducted.

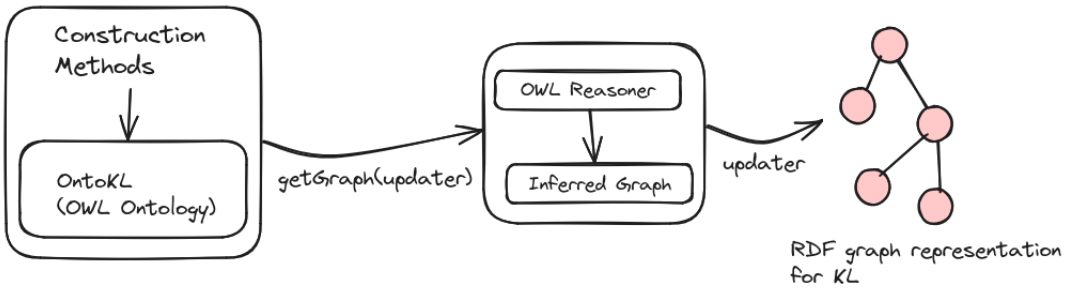


Figure 3.1 Visualising the architecture of RDF-KL

3.3 OntoKL

In this section, the process of setting up the requirements of OntoKL, the underlying ontology of RDF-KL is discussed, culminating in its definition.

3.3.1 Requirements

The requirements that must be met by the ontology are defined to ensure clarity of what the ontology is meant to achieve. This section is based on the Ontology Development 101 methodology [1]. The ontology development begins from scratch, i.e., other ontologies are not utilised to build OntoKL. This should not be an issue since OntoKL is designed to work within this framework's context, so no controlled grammar is required. However, if OntoKL was embedded in other environments, other considerations might be required.

3.3.1.1 Determining the domain and scope of the ontology

The domain and scope of the ontology are initially defined by addressing four basic questions outlined by Noy and McGuinness [1].

Question 1 What is the domain that the ontology will cover?
The ontology will cover the domain of KLs. It needs to model persons, KAs, and the various interactions that may exist between them. We want the ontology to allow for the creation of a knowledge graph representation of a KL. Since we aim to achieve a realistic KL representation, the ontology should include related entities and properties from existing literature.

Question 2 For what we are going to use the ontology?
The ontology is used as the semantic layer for the RDF knowledge graph representation of KL, constructed using a number of proposed methods, which not only serves as a base for a reasoner to infer information from explicit axioms and entities but also gives meaning to what entities are, thus facilitating querying.

Question 3 For what types of questions the information in the ontology should provide answers?
The ontology is designed to define entities and axioms that enable organisations to reason about their landscape effectively. To assess whether the ontology achieves this objective, we demonstrate its capability to answer the competency questions outlined in Section 3.1.

Question 4 Who will use and maintain the ontology?
The ontology is designed for users implementing the framework to build a graph representation of their KL. Users can be actual persons or event listener programs that update the graph in response to events. As the framework is in its initial iteration, researchers may explore and address its limitations, refining the ontology over time. Additionally, organisations may maintain and expand the ontology to suit their specific requirements.

3.3.1.2 Important terms in the ontology

From the literature review, a list of terms was obtained that relate to representing a Knowledge Landscapes (KLs), without concern for the overlap between the concepts they represent, the relations among the terms, or any properties that the concepts may have.

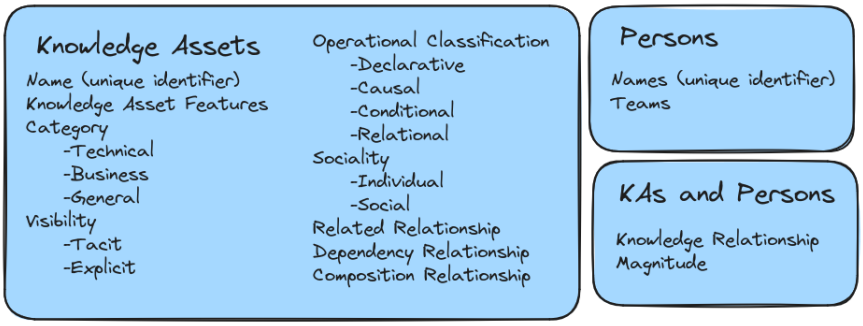


Figure 3.2 List of relevant terms

3.3.1.3 Classes, class hierarchy and properties

The main classes in the ontology are taken to be the classes representing the set of persons and KAs, explicitly defined as disjoint to ensure no individual can be simultaneously a person and a KA.

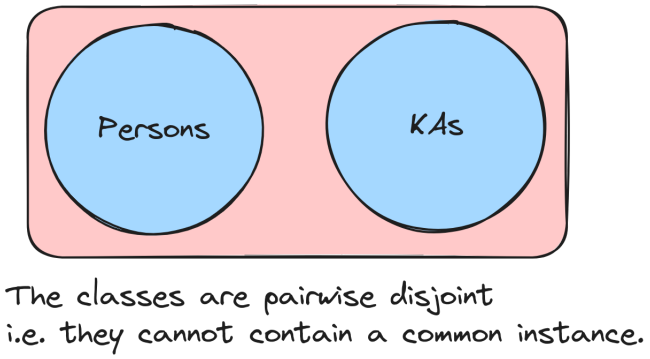


Figure 3.3 Classes as sets

Teams are considered to be any named subclass of the class of persons. To represent KA features and their values, the design pattern “Values as subclasses partitioning a feature” [34, Pattern 2] is used. This structure allows for the scalability of the ontology by enabling the addition of new features and values. A class representing all KA features is defined, where each subclass of this class represents a KA feature. Each subclass forms a disjoint union of value sets for possible values a feature may have. KAs are linked to these values through functional object properties, assuming for

the sake of simplicity that a KA has only one value per feature. To accommodate multiple values for a single feature, a hybrid value can be created. The initial set of KA features includes Category, Visibility, Social Classification, and Operational Classification with value sets $\{Technical, Business\}$, $\{Tacit, Explicit\}$, $\{Individual, Social\}$, and $\{Declarative, Procedural, Causal, Conditional, Relational\}$ respectively . This framework facilitates the extension of these base features and their value sets within the ontology (see Section 3.3.3). These four base features, are to be shipped with a default OntoKL ontology. Moreover, the design pattern with these base features can be visualised in terms of a Venn Diagram as shown in Figure 3.4.

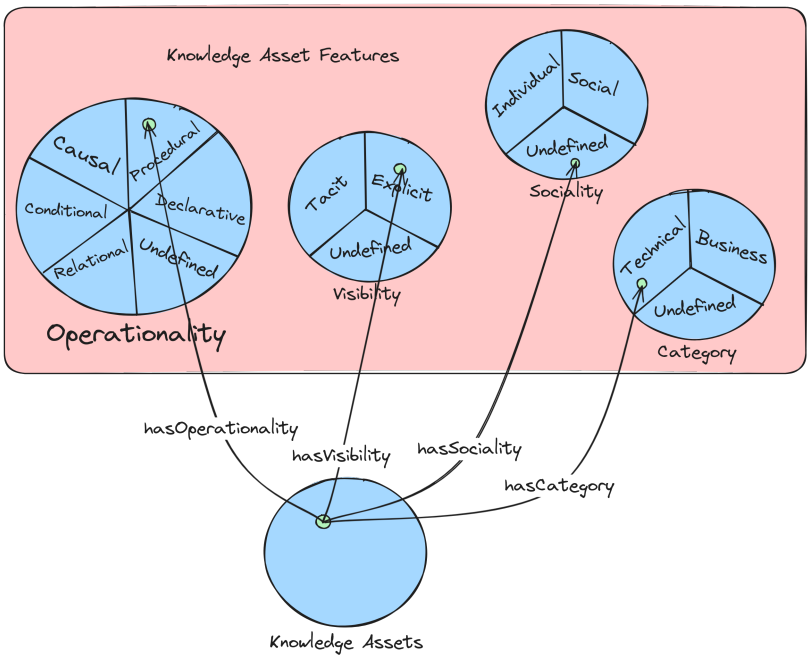


Figure 3.4 Feature values as value sets

A 3-ary representation [35] is used to model what KA is known by a person and to what extent, as this relationship cannot be straightforwardly represented by a triple. Non-negative integers represent magnitudes assigned to represent extent of knowledge [32].

A knowledge observation class is defined, with instances related to KAs, persons, and non-negative integer literals via object properties hasKnowledgeAsset and hasPerson, and the data property hasMagnitude. The domain of hasKnowledgeAsset, hasPerson, and hasMagnitude is the knowledge observation class, and their ranges are the class of KAs, the class of persons, and values of type *xsd:nonNegative*.

Each instance in the knowledge observation class must be uniquely identified by the person and KA associated with it. This ensures each person is connected to a particular KA with only one magnitude, preventing ambiguity, as a person cannot know

a KA to two different extents. Each knowledge observation instance must be correlated with one KA, one person, and a non-negative value, defined using cardinality restriction axioms.

Figure 3.5 illustrates the chosen modelling, with an example of a 3-ary relationship relating an instance of a knowledge observation that related to a person John, to a KA Java, and a magnitude 3.

The properties of "related to", "composed of", and "depends on" are modelled as object properties, with their domains and ranges specified to be the class of KAs. Their respective structural properties are established through the asymmetric, symmetric, transitive, and sub-object property axioms.

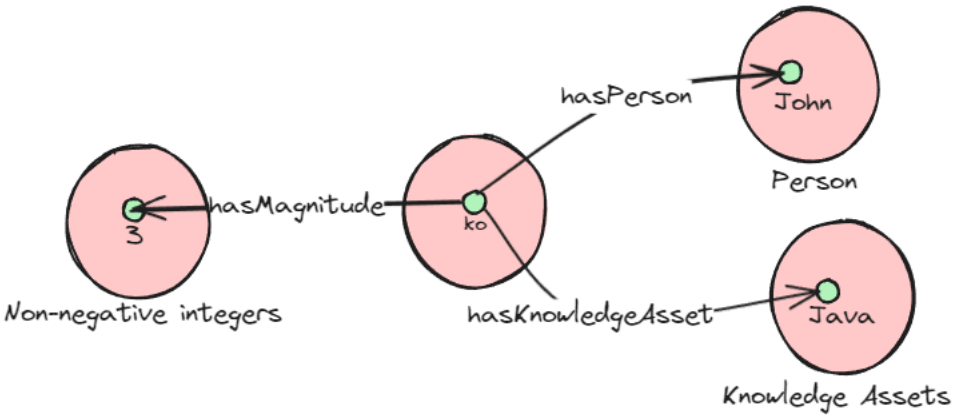


Figure 3.5 Knowledge Observation: John knows Java with magnitude 3

3.3.2 Definition

This subsection outlines the criteria for an OWL ontology to qualify as a valid OntoKL ontology. All entities within the ontology share a common namespace, which forms the base IRI. The namespace selection is implementation-dependent; for example, the prototype (Section 3.6) uses the namespace, linked to the framework's GitHub Page source code. For clarity, the prefix *kl* is typically used to represent the namespace, though alternatives or no prefix might be used in different implementations. Entities are denoted as *kl:localname*, where *kl:* expands to the base IRI. Subsequent sections detail the classes, properties, and attributes essential for an ontology to meet the OntoKL requirements.

3.3.2.1 Classes, properties, and characteristics

The ontology needs to declare the following entities by using a declaration axiom.

- The ontology must declare the following classes: *kl:Person*, *kl:KnowledgeAsset*, *kl:KnowledgeObservation*, *kl:KnowledgeAssetFeature*, *kl:Category*,

kl:TechnicalCategoryValue,*kl:BusinessCategoryValue*, *kl:GeneralCategoryValue*, *kl:Visibility*, *kl:TacitVisibilityValue*, *kl:ExplicitVisibilityValue*, *kl:Sociality*,*kl:IndividualSocialityValue*,*kl:SocialSocialityValue*, *kl:Operationality*, *DeclarativeOperationalityValue*, *kl:ProceduralOperationalityValue*, *kl:CausalOperationalityValue*, *kl:ConditionalOperationalityValue*, and *kl:RelationalOperationalityValue*.

- The ontology must declare the following object properties: *kl:relatedTo*, *kl:dependsOn*, *kl:composedOf*, *kl:hasCategory*, *kl:hasVisibility*, *kl:hasSociality*,*kl:hasOperationality*, *kl:hasPerson*, *kl:hasKnowledgeAsset*.
- The ontology must declare the following data property: *kl:hasMagnitude*.

After specifying the declaration axioms, axioms need to be specified to define the following characteristics:

- Utilising the object property domain axiom, the domain of *kl:relatedTo*, *kl:dependsOn*, *kl:hasCategory*, *kl:hasVisiblity*,*kl:hasSociality*,*kl:hasOperationality*, and *kl:composedOf* is specified as *kl:KnowledgeAsset*. The domain of *kl:hasPerson*, *kl:hasKnowledgeAsset* is set to be *kl:KnowledgeObservation*.
- Utilising the object property range axiom, the range of *kl:relatedTo*, *kl:dependsOn*, and *kl:composedOf* is set to be *kl:KnowledgeAsset*. The ranges of *kl:hasCategory*, *kl:hasVisiblity*,*kl:hasSociality*,*kl:hasOperationality* are set to be *kl:Category*,*kl:Visibility*,*kl:Sociality*, and *kl:Operationality* respectively. The ranges of *kl:hasPerson*, *kl:hasKnowledgeAsset* are set to be *kl:Person* and *kl:KnowledgeAsset* respectively.
- Utilising the data property domain axiom, the domain of *kl:hasMagnitude* is set to be *kl:KnowledgeObservation*.
- Utilising the data property range axiom, the range of *kl:hasMagnitude* is set to be the datatype *xsd:nonNegative*.
- Utilising the symmetric object property axiom, *kl:relatedTo* is defined to be symmetric.
- Utilising the transitive object property axiom or SWRL rule $r(a,b), r(b,c) \rightarrow r(a,c)$, *kl:relatedTo*, and *kl:composedOf* are defined to be transitive.
- Utilising the asymmetric object property axiom, *kl:dependsOn* and *kl:composedOf* are defined to be asymmetric.
- Utilising the sub object property axiom, *kl:composedOf* is defined to be the sub property of *kl:dependsOn*.

- Utilising the functional object property axiom, *kl:hasCategory*,*kl:hasVisibility*,*hasSociality*,*hasOperationality* are defined to be functional.
- Utilising the disjoint classes axiom, it is defined that the classes *kl:Person* and *kl:KnowledgeAsset* are mutually exclusive.
- Utilising the subclass axiom, it is defined that *kl:Visibility*, *kl:Category*, *kl:Sociality*, and *kl:Operationality* are subclasses of *kl:KnowledgeAssetFeature*.
- Utilising the disjoint union axiom for classes, it is defined that *kl:Visibility* is the disjoint union of *kl:TacitVisibilityValue* and *kl:ExplicitVisibilityValue*. It is defined that *kl:Category* is the disjoint union of *kl:GeneralCategoryValue*, *kl:BusinessCategoryValue*, and *kl:TechnicalCategoryValue*. It is defined that *kl:Sociality* is the disjoint union of *kl:SocialSocialityValue* and *kl:IndividualSocialityValue*. It is defined that *kl:Operationality* is the disjoint union of *kl:CausalOperationalityValue*, *kl:ProceduralOperationalityValue*, *kl:RelationalOperationalityValue*, *kl:ConditionalOperationalityValue*, and *kl:DeclarativeOperationalityValue*.
- Utilising the has key axiom for classes, it is defined that *kl:KnowledgeObservation* has keys *kl:hasPerson* and *kl:hasKnowledgeAsset*.
- Utilising the subclass and cardinality restriction axioms: every individual of kl:KnowledgeObservation is related to exactly one magnitude via *kl:hasMagnitude*. Every individual of kl:KnowledgeObservation is related to exactly one KA via *kl:hasKnowledgeAsset*. Every individual of kl:KnowledgeObservation is related to exactly one person via *kl:hasPerson*.

3.3.3 Note on feature extension

This section discusses how an OntoKL ontology is extended to facilitate the addition of new KA features and their values. Given a newly identified KA feature named $\langle featureName \rangle$ with values named $\langle v_1 \rangle$, ..., $\langle v_n \rangle$, the ontology can be extended by declaring a class *kl:*$\langle featureName \rangle$. This class is then defined as a subclass of *kl:KnowledgeAssetFeature*. Classes *kl:*$\langle v_1 \rangle\langle$*featureName*$\rangle$*Value*, ..., *kl:*$\langle v_n \rangle\langle$*featureName*$\rangle$*Value* are declared, and the class kl:$\langle featureName \rangle$ is defined to be the disjoint union of these classes. An object property *kl:has*$\langle$*featureName*$\rangle$ is declared to handle the relationship between a KA and a newly defined feature. To add a new value named $\langle v \rangle$ to an existing feature represented by the class *kl:*$\langle featureName \rangle$, a class *kl:*$\langle$v$\rangle$ is created to represent the value. The subsets of the feature class representing the value sets of the feature are obtained, and then the feature class is redefined to be the disjoint union of the existing features and the newly identified feature. The removal of features and

their values is not discussed, given that this is the first iteration of the framework and the removal of a particular feature or value does not have a consequence on the structure of other existing features and values.

3.4 Knowledge Graph Construction

In this section, methods to populate the OntoKL, allowing for the construction of the knowledge graph, are outlined.

3.4.1 Note on unique naming assumption

OWL does not assume unique names for individuals, meaning two different IRIs can refer to the same individual. To eliminate ambiguity that could result in incorrect reasoning (such as inferring that two individuals are the same, even when they are not), any individual created from the construction method is specified to be distinct from every existing individual using the "different from" axiom for individuals.

3.4.2 Methods

The first step in transitioning from an ontology to a knowledge graph is populating the ontology with data. An RDF-KL implementation should include methods acting as an interface for populating the ontology and constructing the knowledge graph. These methods include asserting new persons and Knowledge Assets if not already declared, removing existing Knowledge Assets and persons, establishing and deleting relationships between persons and Knowledge Assets, and managing teams. For a more exhaustive explanation of the methods and how they are to be employed, one may refer to Appendix A, Section A.1. After executing each method, **the ontology is checked for consistency using a reasoner**. If any inconsistencies are found, such as an individual incorrectly classified as both *kl:KnowledgeAsset* and *kl:Person*, a person incorrectly related to a KA of differing magnitudes, or a KA having multiple values for a single feature, the recent changes are reverted to **maintain the ontology's consistency**.

3.4.3 Generating the RDF Knowledge Graph

After creating a populated ontology, the framework should generate a graph by applying a reasoner to infer new properties from the structural characteristics of object properties linking two KAs. Then, the resulting graph is augmented using an *updater* (see Section 3.4.3.1). Note that an ontology can be converted to an RDF graph if specified using structural specification [18], with the necessary RDF mapping [19]. If

specified using RDF syntax, direct mapping into an RDF graph is not required. Although the former method is straightforward, as it simplifies specifying ontological properties, it may become more expensive to map it into an RDF graph as the ontology size increases.

3.4.3.1 Updater

An organisation should be allowed to make sense of graph data. An abstract method `update(graph)` should be provided, allowing the organisation to take the generated graph and update it with information based on custom set specifications set by the organisation. A use case would be to model transitivity through knowledge dependency [27]. A graph that is updated is referred to as an augmented graph and should be free of any inconsistencies.

3.5 Remarks

In this section, a framework, RDF-KL, has been defined, illustrating how an RDF graph can be constructed. This framework introduces construction methods that serve as an interface for populating the graph with data. The integration of the ontology layer enables structured and semantically enriched data representation within the RDF graph. Moving forward, the feasibility of the RDF-KL framework is validated through a prototype implementation. The subsequent section presents a simple prototype of the framework.

3.6 Prototype Implementation

A prototype of the framework is provided to demonstrate its feasibility. The tools used to implement the framework were ONT-API [2], HermiT [24], and Jena [3]. ONT-API was chosen because it manages ontologies as a Jena model, aligning with the thesis's objective of exploring RDF's role in knowledge graph representation. HermiT was selected for reasoning tasks due to its high success rate in consistency validation [36] and compatibility with ONT-API.

ONT-API was used to implement OntoKL and its construction methods. The resulting knowledge graph was generated by applying the populated ontology to the HermiT OWL reasoner and an *updater*. This results in a Jena model that represents the RDF graph of a Knowledge Landscape (KL).

The implementation is deployed as a Maven package, utilising ONT-API and Jena libraries (see Appendix A.2 for usage details). For specific explanations regarding a

particular package, class, or method, refer to the JavaDoc at `https://andimon.github.io/rdf-kl/`.

3.6.1 Updater Implementations

This subsection focuses on the implementation of the *updater* abstraction, as described in Section 3.4.3.1. The NullUpdater and BaseUpdater classes implement the updater interface, representing the abstraction. The NullUpdater interacts with the knowledge graph but does not extract any information, whereas the BaseUpdater models the transitivity of knowledge via composition [27] and dependency of KAs.

Given two KAs and a person p, in transitivity through composition, if k_1 is composed of k_2 and p knows k_2, then p also knows k_1 to a lesser extent. The exact magnitude of knowledge is randomly assigned between 0 and n, where n is the degree to which p knows k_1. In transitivity through knowledge dependency, if k_1 depends on k_2 and p knows k_1, then p knows k_2 to a generally higher degree. Here, the magnitude is randomly assigned between n and $maxMag$, with n indicating how well p knows k_1 and $maxMag$ being the maximum possible magnitude. These random costs are considered for preliminary analysis since the calculation of magnitudes is deemed out of scope for this project and further research is needed.

The NullUpdater serves as a no-op, allowing the graph to be generated without adding specific information. In contrast, the BaseUpdater exemplifies how additional information can be deduced by simulating transitivity through KA relationships.

3.6.2 Non-simple properties

During implementation testing, HermiT encountered an exception due to object properties being defined as transitive and either asymmetric or irreflexive. To resolve this, a workaround involved defining a property characteristic excluding transitivity with axioms and adding a SWRL Rule for transitivity. This approach prevented exceptions. The issue stems from HermiT's compatibility with a subset of OWL 2 called OWL 2 DL [24], which prohibits certain property characteristics for decidability. HermiT ensures decidability by restricting SWRL rules to individuals named in the ABox axioms [24]. Consequently, the framework was updated to allow transitive properties to be defined using SWRL rules (refer to Section 3.3.2).

3.6.3 Using the prototype

This section explains how the prototype can be used to construct a KL. A scenario in a simulated Software Engineering environment illustrates how to use the framework to

construct a graph.

3.6.4 Construction of an RDF Knowledge Landscape

3.6.4.1 Example Scenario

We consider a scenario adapted from [27], which depicts a knowledge landscape. This scenario is extended to include additional features outlined in the framework, such as assigning values to Knowledge Asset features like visibility, representing teams as subsets of individuals, and adding new Knowledge Asset (KA) features. An organisation might measure knowledge age as a dimension ranging from new to established to old. As an instance of knowledge ages, its applicability, accessibility, utility, and validity can fluctuate [37]. We consider this KA feature in the extended scenario.

The organisation employs a team consisting of four people: Jane, Chris, John and Peter. The organisation identified ten valuable Knowledge Assets (KAs): object-oriented programming, Java, SQL, web development, Smalltalk, JDBC, Servlets, Haskell, Elixir and Functional Programming. Java is set to be dependent on object-oriented programming, Smalltalk is said to be dependent on object-oriented programming, JDBC and Servlets are said to be part of Java, and Servlets are said to be related to web development. Elixir and Haskell are said to be dependent on functional programming. Jane knows Smalltalk with magnitude 30, and Chris knows Java with magnitude 23. John knows Elixir and Haskell with magnitudes 26 and 23 respectively, and Peter knows Elixir and Haskell with magnitudes 21 and 17 respectively. The KAs have the following attributes: Java is said to have a feature category with a technical value, and a feature sociality with a social value; OOP is said to have a feature category with a technical value; SQL has a sociality feature with an individual value, an operationality feature with a procedural value, and a category feature with a business value; SmallTalk has a category feature with a technical value. Elixir and Haskell are said to have a feature category with a technical value, OOP, Java, Web Development, JDBC, Servlets, Haskell have a feature Age with an established value, Smalltalk have a feature Age with Old value, and Elixir have a feature Age with a new value. John and Peter are placed in a team called functional programmers, while Jane and Chris are placed in a team called object-oriented programmers. Moreover, John and Jane are in a team called Managers.

3.6.4.2 Setup

To utilise the ontology, an OntoKL instance is created. The construction methods involve creating a KnowledgeLandscapeConstructor instance and passing OntoKL into it. If no ontology is passed, a default OntoKL instance without extensions is used.

```
1 OntoKL ontoKL = new OntoKL();
```

```
KnowledgeLandscapeConstructor knowledgeLandscapeConstructor = new
    KnowledgeLandscapeConstructor(ontoKL);
```

Listing 3.1 Setting up the ontology and construction methods

3.6.4.3 Creating KAs

The construction method `knowledgeAssetIdentification` can be used to create Knowledge Assets (KAs). For example, Listing 3.2 shows how the prototype is used to create the Knowledge Asset (KA) Java with the features given in scenario (see Section 3.6.4.1).

```
knowledgeLandscapeConstructor.knowledgeAssetIdentification("Java", Set.of(Visibility.EXPLICIT,
    Category.TECHNICAL, Age.ESTABLISHED));
```

Listing 3.2 Adding a new KA

3.6.4.4 Creating Persons and Teams

The construction methods `personIdentification`, `createTeam`, `addPersonToTeam` can be used to create persons and teams. For example, Listing 3.3 shows how the prototype is used to create a person, a team, and adding the person to the team.

```
knowledgeLandscapeConstructor.personIdentification("Jane");
knowledgeLandscapeConstructor.createTeam("ObjectOrientedProgrammers");
knowledgeLandscapeConstructor.addPersonToTeam("ObjectOrientedProgrammers", "Jane");
```

Listing 3.3 Creation of person and team

3.6.4.5 Creating the structural properties of KAs

The construction methods `dependsOn`, `composedOf`, and `relatedTo` can be used to specify structural properties of Knowledge Assets (KAs) and their respective values. For example, Listing 3.4 shows how the prototype is used to create a dependency relationship between Java and OOP.

```
knowledgeLandscapeConstructor.dependentOn("Java", "OOP")
```

Listing 3.4 Structural property example

3.6.4.6 Creating the Knowledge Observations

The construction method `knowledgeObservation` can be used to reify a ternary relationship where an observation is connected to a KA, a person, and a non-negative number representing the extent to which the KA is known. For example, Listing 3.5 shows how the prototype is used create a knowledge relationship between Jane and Smalltalk with a magnitude of 16.

```
knowledgeLandscapeConstructor.knowledgeObservation("Jane", "SmallTalk", 16)
```

Listing 3.5 Knowledge Observation example

3.6.4.7 Extending the Knowledge Asset feature set

The new KA feature, age, can be created using the RDF-KL implementation by extending OntoKL, as shown in Listing 3.6. The `addFeature` method creates the feature class *kl:Age*, the value classes *kl:NewAgeValue*, *kl:EstablishedAgeValue*, *kl:OldAgeValue*, makes the feature class equivalent to the disjoint union of the value classes, and establishes the relation *kl:hasAge* with domain *kl:KnowledgeAsset* and range *kl:Age*. To use the age feature in the construction methods, an ENUM implementing the Feature interface must be defined, as shown in Listing A.2 in the User Guide.

```
ontoKL.addFeature("Age", Set.of("New", "Established", "Old"));
```

Listing 3.6 Adding a new KA feature to the Ontology

3.6.5 Generating the RDF graph

The RDF graph utilising HermiT to generate the graph with inferences based on the structural properties of KLs and utilising BaseUpdater to simulate transitivity through dependency and composition (see Section 3.4.3.1), can be obtained using the prototype as given in Listing 3.7. Where the `Model` class is the representation of an RDF graph within the Jena framework.

```
BaseUpdater baseUpdater = new BaseUpdater();
Model knowledgeLandscapeGraphModel = knowledgeLandscapeConstructor.getGraph(baseUpdater);
```

Listing 3.7 Getting the RDF graph with inferences

3.7 Conclusion

In this chapter, we presented a framework, RDF-KL, for representing a KL, where we explored its architecture, its ontological layer (OntoKL), the methods used to populate the ontology and generate the RDF graph, along with a prototype implementation of this framework. Furthermore, we presented a scenario depicting a KL within a simulated Software Engineering organisation (Section 3.6.4.1), illustrating how the prototype can be applied to construct such a scenario. In the next chapter, we evaluate this framework in terms of this scenario.

4 Evaluation and Testing

4.1 Methodology

The framework's viability was tested by confirming that the prototype conforms to it. The model's effectiveness in representing and analysing a Knowledge Landscape (KL) was evaluated by translating the competency questions from Section 3.1 into SPARQL queries and executing them against the scenario in Section 3.6.4.1. Each query was written as part of a standalone test.

4.2 Testing the implementation

Unit tests[1] were carried out to ensure the correctness of the implementation by verifying that it conforms to the RDF-KL framework (see Chapter 3) through a series of JUnit tests. The tests include those that test for happy execution paths and unhappy execution paths.

For the OntoKL ontology (see Section 3.3), unit tests were written to ensure conformity to the OntoKL definition and assess extensibility. The "Happy Paths" tests scenarios like verifying that the ontology contains the expected axioms, and asserting that the expected axioms are added to the ontology when adding features and values and , while the "Unhappy Paths" tests checked error handling, such as attempting to add existing KA features or values.

Similarly, construction method tests evaluated both expected and unexpected behavior. "Happy Paths" tests confirmed the expected triples are added after calling cetain methods such as person and KA identification, while "Unhappy Paths" tests checked error conditions, like attempting contradictory actions or violating ontology constraints, such as having a person related to a KA with two different magnitudes which contradicts the hasKey axiom (see Section 3.3).

For the Updater component, testing focused on ensuring that new KA relationships are created through transitivity, specifically via dependency and composition of KAs (see Section 3.4.3.1).

In conclusion, these tests aim to ensure that the implementation is faithful to the requirements of the framework. In the next section we evaluate the ability of the RDF-KL framework's to represent and analyse a KL using a set of of competency questions defined in Section 3.1

[1] `https://github.com/andimon/rdf-kl/tree/dev/rdf_knowledge_landscape/src/test/java/com/andimon/rdfknowledgelandscape`

4.3 Evaluating the competency of the framework

This section uses the implementation to model the extended scenario described in Section 3.6.4.1. Additionally, we use the `BaseUpdater` to enhance the knowledge relationships within a KL, as detailed in Section 3.6.1. To ensure reproducibility, a fixed seed value is set for the magnitudes assigned by `BaseUpdater`. The framework's competency is evaluated through a series of competency questions, which are translated into SPARQL queries. We discuss the results obtained from executing these queries against the RDF-graph constructed using the implementation of the framework for the scenario in Section 3.6.4.1.

4.3.1 CQ1: Given a person and a KA, to what extent does the person know the KA?

To evaluate this question, we considered the example of determining the level of Chris's familiarity with Java. This involved identifying the individual in the knowledge observations class linked to Chris and Java respectively. This graph pattern was queried as illustrated in Figure 4.1a. The formatted result set of the query, presented in Figure 4.1b, indicated that Chris knows Java with a magnitude of 23, which is consistent with the scenario modelled.

```
1 PREFIX kl:<namespace>
2 SELECT ?extent
3 WHERE {
4     ?ko a kl:KnowledgeObservation .
5     ?ko kl:hasKnowledgeAsset kl:Java .
6     ?ko kl:hasPerson kl:Chris .
7     ?ko kl:hasMagnitude ?extent
8 }
9
```

(a) SPARQL Query

```
----------
| extent |
==========
| 23     |
----------
```

(b) Result set

Figure 4.1 Addressing CQ1

4.3.2 CQ2: Who knows a given KA, and to what extent?

To evaluate this question, we considered the example of determining which individuals were knowledgeable about Elixir. This involved identifying all individuals in the knowledge observation class that were linked to Elixir through the object property hasKnowledgeAsset. For these knowledge observations, we then obtained the associated individuals and their extents. This graph pattern was queried as illustrated in Figure 4.2a. The formatted result set of the query, presented in Figure 4.2b, showed that Peter and John knew Elixir to extents of 21 and 26, respectively, which is consistent with the scenario modelled.

```
PREFIX kl:<namespace>
SELECT ?person ?extents
WHERE {
    ?ko a kl:KnowledgeObservation .
    ?ko kl:hasKnowledgeAsset kl:Elixir .
    ?ko kl:hasPerson ?person .
    ?ko kl:hasMagnitude ?extent
}
```

(a) SPARQL Query

person	extent
kl:Peter	21
kl:John	26

(b) Result set

Figure 4.2 Addressing CQ2

4.3.3 CQ3: What KAs are known by a person?

To evaluate this question, we considered the example of determining which Knowledge Assets are known by Chris. This involved identifying all individuals in the knowledge observation class that are related to Chris via object property hasPerson. For all these individuals we obtained the KAs and magnitudes that are related to via properties hasKnowledgeAsset and hasMagnitude respectively. This graph pattern was queried as illustrated in Figure 4.3a. The formatted result set of the query, present in Figure 4.3b, indicated that Chris knows Java with an extent of 23 which is explicitly stated in the scenario being considered. However since we utilised `BaseUpdater` to model knowledge relationships through transitivity of dependency and composition of KAs, the result set also is included implicit information that Chris knows Servlets and JDBC with a higher extent than Java since we are considering that JDBC and Servlets are part of the Java framework, and knows OOP with a higher extent than Java since Java depends on OOP.

```
PREFIX kl:<namespace>
SELECT ?ka ?extents
WHERE {
    ?ko a kl:KnowledgeObservation .
    ?ko kl:hasKnowledgeAsset ?ka .
    ?ko kl:hasPerson kl:Chris .
    ?ko kl:hasMagnitude ?extent
}
```

(a) SPARQL Query

ka	extent
kl:Servlets	28
kl:JDBC	26
kl:Java	23
kl:OOP	27

(b) Result set

Figure 4.3 Addressing CQ3

4.3.4 CQ4: What KAs have some specific attribute?

This competency question is broken down into different queries. In the first query illustrated in Figure 4.4a), we examined the features that a KA can be related to, by asking what the subclasses of the class of knowledge features are. The result set for this query, indicated in Figure 4.4b, shows that the knowledge graph contains the four base features and the newly established Age feature, which is as expected in the

context of the scenario modelled. The values of each feature form its disjoint union. To find out the values defined for a feature, we asked what the subclasses of a particular feature are. For example, in the query shown in Figure 4.5a, we obtained the values that a KA can be associated with for the newly established feature Age. Now that we have established what the KLs features and their values are, we can find out for example which KAs have an age set to old and a category set to technical, by getting the individuals that a KA is related to via hasCategory and hasAge, where these values are in the Technical and Old value sets respectively. This example is translated in query shown in Figure 4.6a. The result set for this query, is shown in Figure 4.6b, shows that, only SmallTalk, is assigned with an assigned old age and a technical category. This is as expected in the context of the scenario modelled. We conclude this subsection with a query, given in Listing 4.1, which provides a comprehensive list of attributes for all Knowledge Assets, using the OPTIONAL clause to still return a result when a feature value is not found, when a feature value is not bound we use COALESCE to return "Undefined" when a KA is not related to any particular feature. Due to length limitations, the result set for this query is given in Appendix B, Figure B.1.

```
1 PREFIX kl:<namespace>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT ?feature
4 WHERE {
5     ?feature rdfs:subClassOf kl:KnowledgeAssetFeature .
6 }
7
```

(a) SPARQL Query

feature
kl:Category
kl:Sociality
kl:Age
kl:Visibility
kl:Operationality

(b) Result set

Figure 4.4 Getting Knowledge Asset Features

```
1 PREFIX kl:<namespace>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT ?value
4 WHERE {
5     ?value rdfs:subClassOf kl:Age .
6 }
7
```

(a) SPARQL Query

value
kl:EstablishedAgeValue
kl:OldAgeValue
kl:NewAgeValue

(b) Result set

Figure 4.5 Getting values for a Knowledge Asset Feature

```
PREFIX kl:<namespace>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?ka
WHERE {
    ?ka a kl:KnowledgeAsset .
    ?ka kl:hasCategory ?technicalValue .
    ?ka kl:hasAge ?ageValue .
    ?technicalValue a kl:TechnicalCategoryValue .
    ?technicalValue a kl:OldAgeValue .
}
```

(a) SPARQL Query

```
----------------
| ka           |
================
| kl:SmallTalk |
----------------
```

(b) Result set

Figure 4.6 Getting a Knowledge Assets with particular features

```
PREFIX kl:<namespace>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?ka ?feature ?value
WHERE {
    ?ka a kl:KnowledgeAsset .
    ?feature rdfs:subClassOf kl:KnowledgeAssetFeature .
    OPTIONAL {
        ?featureProperty rdfs:domain kl:KnowledgeAsset .
        ?featureProperty rdfs:range ?feature .
        ?ka ?featureProperty ?featureValAssignment .
        ?featureVal rdfs:subClassOf ?feature .
        ?featureValAssignment a ?featureVal .
    }
    BIND(COALESCE(?featureVal,"Undefined") AS ?value)
}
```

Listing 4.1 Exhaustive SPARQL Query for getting KAs with attributes

4.3.5 CQ5: What are the KAs known by a team and to what extent?

In this subsection, we considered two types of queries used to answer this question. In the first query, given in Figure 4.7a, we retrieved all the KAs known by a specific team, such as the team of managers. The result set of this query is given in Figure 4.7b and shows the knowledge relationships John and Jane are related to. It is to be noted that in these relationships, we have two implicit knowledge relationships between John and Functional Programming and Jane and OOP, which have been inferred by the BaseUpdater since SmallTalk is dependent on OOP, and Elixir and Haskell are dependent on the Functional Programming Paradigm.

In the second query given in Listing 4.2, showcase how one may associate a magnitude with a team, where the arithmetic average of the magnitudes of persons within a team is considered. The aim of this query is to showcase how one may query a graph for a team's extent using a metric. The investigation of different metrics for associating a magnitude to a team may be explored in further studies, as the metrics for magnitudes go beyond the scope of this FYP.

```
PREFIX kl:<namespace>
PREFIX rdfs:
      <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?team ?member ?ka ?extent
WHERE {
    ?team rdfs:subClassOf kl:Person .
    ?member a ?team .
    ?ko kl:hasPerson ?member .
    ?ko kl:hasKnowledgeAsset ?ka .
    ?ko kl:hasMagnitude ?extent .
    FILTER(?team=kl:Managers)
}
```

(a) SPARQL Query

team	member	ka	extent
kl:Managers	kl:Jane	kl:OOP	30
kl:Managers	kl:Jane	kl:SmallTalk	30
kl:Managers	kl:John	kl:FunctionalProgramming	28
kl:Managers	kl:John	kl:Elixir	26
kl:Managers	kl:John	kl:Haskell	23

(b) Result set

Figure 4.7 Getting all Knowledge Assets known by a team's members

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?ka (AVG(COALESCE(?memberExtent,0)) AS ?extent)
WHERE {
    ?team rdfs:subClassOf kl:Person .
    ?ka a kl:KnowledgeAsset .
    ?member a kl:Person .
    ?member a ?team .
    OPTIONAL {
        ?ko kl:hasPerson ?member .
        ?ko kl:hasKnowledgeAsset ?ka .
        ?ko kl:hasMagnitude ?memberExtent .
    }
    FILTER(?team=kl:Managers)
}
```

Listing 4.2 Associating a magnitude to a team

4.3.6 CQ7 What are the prerequisite KAs that a person must know in order to learn/utilise a given KA?

This question can be answered by considering the dependency relationships. Since a k_1 is dependent on k_2, it means that a person must learn/utilise k_2 first in order to learn/utilise k_1. For example, the query for finding which KL "Java" depends on is given in Figure 4.8a. The results are expected since JDBC and Servlets are defined to be parts of the Java framework; hence, the reasoner is able to infer the dependency relationship since the composed-of relationship is a subset of the dependent-on relationship. On the other hand, Java is explicitly stated to be dependent on OOP in the scenario modelled. Answering CQ6 and CQ8 is analogous to this question and are answered by considering the "related to" and "composed of" relationships, respectively. Their discussion is omitted from this chapter; however it should be noted that the respective results were as expected with respect to the scenario modelled.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?ka
WHERE {
    ?ka a kl:KnowledgeAsset .
    ?ka1 kl:dependsOn ?ka
    FILTER (?ka1 = kl:Java)
}
```

(a) SPARQL Query

ka
kl:JDBC
kl:Servlets
kl:OOP

(b) Result set

Figure 4.8 Addressing CQ7

4.3.7 CQ9 What are the KRs associated with a KL

To answer this question, we consider some metrics available from the literature to investigate two types of risk: KTR (see Section 2.7.3.1) and KMR (see Section 2.7.3.2).

4.3.7.1 Investigating KTR

The more KAs a given KA depends on, the more embedded it is said to be. The higher the embeddedness (see Section 2.7.3.1) of a KA, the greater the risk it poses to knowledge transfer activities. We incorporated the embeddedness metric in the query given in Figure 4.9a, which involves counting the number of assets that a KA depends on. To facilitate reading, the results have been ordered in descending order to indicate which Knowledge Assets pose a higher risk of unsuccessful transfer. In the context of our scenario, Java poses the higher risk since it has a higher number of dependents, which are SmallTalk and JDBC, since they are defined to be part of the Java framework and OOP, as explitly stated in the scenario.

```
SELECT ?ka (COUNT(?ka2) AS ?embeddedness)
WHERE {
    ?ka a kl:KnowledgeAsset .
    OPTIONAL { ?ka kl:dependsOn ?ka2 . }
}
GROUP BY ?ka
ORDER BY DESC(?embeddedness)
```

(a) SPARQL Query

ka	embeddedness
kl:Java	3
kl:Elixir	1
kl:Haskell	1
kl:SmallTalk	1
kl:FunctionalProgramming	0
kl:JDBC	0
kl:OOP	0
kl:SQL	0
kl:Servlets	0
kl:WebDevelopment	0

(b) Result set

Figure 4.9 Measuring Embeddedness

4.3.7.2 Investigating KMR

A LOW, MEDIUM, or HIGH KMR value is obtained for each KA by incorporating Method 1 into a query. The query is given in Appendix B, Listing B.2, due to space

limitations. Subqueries [38] are used to compute various metrics such as entries of the distance matrix, KD, KID, and KMR (see Section 2.7.3.2). Investigating the results (see Figure B.2), the only two KAs with a high KMR were SQL and Web Development. These two KAs were not associated with any person in the scenario, hence the high risk. Java got a medium KMR, and Haskell got a low KMR. This may be due to the fact that in the current scenario, only one person knows Java and two persons know Elixir. Furthermore, the magnitudes in which Java and Elixir are known are similar. Thus, we may consider the number of persons who know a particular KA as the factor in Java posing a higher KMR than Elixir.

4.3.8 Conclusion

In this chapter, we presented how testing was undertaken to ensure that the prototype is faithful to the framework's requirements. Once we had established this faithfulness, the framework's ability to represent and analyse a Knowledge Landscape was tested against a set of competency questions derived from the literature. The framework was able to answer queries regarding the representation aspect of a KL, such as what KAs are associated with a person and to what extent. We also tested the analytical side of a KL, for example by querying what KAs are part of or depend on another knowledge asset through the structural properties of KAs. Although CQ9 (What KRs are associated with a KL?) was answered by means of analysing KMR and KTR, there might exist other KRs that require further information to be deduced.

Additionally, it should be noted that pure SPARQL does not support certain functions like the square root, limiting the inclusion of certain metrics in graph queries. However to our advantage, many query engines, such as ARQ, offer custom functions [39], including the square root, which is essential for calculating standard deviation. In our case, this is needed to calculate the standard deviation of the KMRs.

In the next chapter, we conclude this project by summarising the work done, discussing threats to the validity of the approach taken, and outlining possible future work that could further build on this project.

5 Conclusion

5.1 Summary of work

This project outlined a framework for creating an RDF knowledge graph based on existing literature (see Section 3), achieving the first objective set in Section 1.5. A prototype implementation of the framework was presented and tested to ensure it met the framework's requirements, thus meeting the second objective. The competency analysis (see Section 4.3) fulfilled the third objective by developing a case scenario that illustrated the framework's application and its ability to address the competency questions, verifying the overall aim of investigating the viability of RDF for representing and analysing a KL.

5.2 Threats to Validity

Due to the subjective nature of knowledge management, additional relationships and entities might be considered or may be suitable within the representation landscape. Although the framework can identify KRs in a KL to some degree, by analysing KTR and KMR, other risks may only be identifiable if the framework models additional characteristics. The evaluation may have been more comprehensive with a case study involving an organisation, especially for assessing KRs. However, the evaluation still demonstrated the usefulness of RDF in representing and analysing KLs through the ability to answer the competency questions.

5.3 Future Work

The use of RDF was investigated to represent and analyse a KL. Exploring various metrics for measuring knowledge magnitude relationships and embedding them in the framework could provide more realistic findings. Augmenting the framework by exploring new areas of knowledge management and reusing relevant publicly available ontologies to enhance the semantic layer can improve its robustness and applicability. Furthermore, exploring the use of software agents can be explored to construct a KL using this framework based on a set of events in order to construct a more dynamic landscape, thereby enabling the possibility to integrate the framework into an existing organisational environment. In this context, one may consider building on the work done in two previous final-year projects by Piscopo [40] and Ruggier [41], who constructed knowledge maps by analysing Git statistics.

References

[1] N. Noy and D. Mcguinness, "Ontology Development 101: A Guide to Creating Your First Ontology," *Knowledge Systems Laboratory*, vol. 32, Jan. 2001.

[2] OntAPI, *Ont-api*, version 3.0.5, 2019. [Online]. Available: https://github.com/owlcs/ont-api, [Accessed April 23, 2024].

[3] Apache Software Foundation, *A free and open source Java framework for building Semantic Web and Linked Data applications*. [Online]. Available: https://jena.apache.org/, [Accessed April 23, 2024].

[4] F. O. Bjørnson and T. Dingsøyr, "Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used," *Information and Software Technology*, vol. 50, no. 11, pp. 1055–1068, Oct. 2008, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2008.03.006. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.03.006.

[5] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, "OWL 2 Web Ontology Language Primer (Second Edition)," *W3C recommendation*, vol. 27, no. 1, p. 123, 2009. [Online]. Available: https://www.w3.org/TR/owl2-primer/, [Accessed April 23, 2024].

[6] T. Gruber, "Ontology," in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 1963–1965, ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1318. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_1318.

[7] D. L. McGuinness and F. V. Harmelen, "OWL Web Ontology Language Overview," *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[8] A. Hogan *et al.*, "Introduction," in *Knowledge Graphs*. Cham: Springer International Publishing, 2022, pp. 1–4, ISBN: 978-3-031-01918-0. DOI: 10.1007/978-3-031-01918-0_1. [Online]. Available: https://doi.org/10.1007/978-3-031-01918-0_1.

[9] A. Hogan *et al.*, "Knowledge graphs," *ACM Computing Surveys*, vol. 54, no. 4, pp. 1–37, Jul. 2021, ISSN: 1557-7341. DOI: 10.1145/3447772. [Online]. Available: http://dx.doi.org/10.1145/3447772.

[10] C. Peng, F. Xia, M. Naseriparsa, and F. Osborne, *Knowledge graphs: Opportunities and challenges*, 2023. arXiv: 2303.13948 [cs.AI].

[11] M. Micallef, "Visualising and maintaining a healthy knowledge fabric," Presented at the EuroSTAR Conference, Jun. 2023. [Online]. Available: https://conference.eurostarsoftwaretesting.com/event/2023/visualising-and-maintaining-a-healthy-knowledge-fabric/.

[12] M. J. Dürst and M. Suignard, *Internationalized Resource Identifiers (IRIs)*, RFC 3987, Jan. 2005. DOI: 10.17487/RFC3987. [Online]. Available: https://www.rfc-editor.org/info/rfc3987.

[13] P. V. Biron, K. Permanente, and A. Malhotra. "W3c xml schema definition language (xsd) 1.1 part 2: Datatypes," W3C. (2012), [Online]. Available: https://www.w3.org/TR/xmlschema11-2/.

[14] S. Harris and A. Seaborne, "SPARQL 1.1 query language," W3C, W3C Recommendation, Mar. 2013. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[15] N. Humfrey, *EasyRDF - a PHP library designed to make it easy to consume and produce RDF*, https://www.easyrdf.org/, [Accessed April 23, 2023].

[16] D. Krech *et al.*, *RDFLib*, version 7.0.0, Aug. 2023. DOI: 10.5281/zenodo.6845245. [Online]. Available: https://github.com/RDFLib/rdflib.

[17] A. Bendiken, B. Lavender, and G. Kellogg, *Rdf.rb is a pure-ruby library for working with resource description framework (rdf) data.* https://rubygems.org/gems/rdf, [Accessed April 23, 2024].

[18] B. Motik, P. F. Patel-Schneider, and B. Parsia, "OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax," W3C, W3C Recommendation, Dec. 2012. [Online]. Available: https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/, [Accessed: April 23, 2024].

[19] B. C. Grau, I. Horrocks, B. Parsia, A. Ruttenberg, and M. Schneider, *OWL 2 Web Ontology Language Mapping to RDF Graphs (Second Edition)*, 2012. [Online]. Available: https://www.w3.org/TR/owl2-mapping-to-rdf/, [Accessed April 23, 2024].

[20] K. Abicht, *OWL Reasoners still useable in 2023*, 2023. arXiv: 2309.06888 [cs.AI].

[21] M. A. Musen, "The protégé project: A look back and a look forward," *AI Matters*, vol. 1, no. 4, pp. 4–12, 2015. DOI: 10.1145/2757001.2757003. [Online]. Available: https://doi.org/10.1145/2757001.2757003.

[22] M. Horridge and S. Bechhofer, “The owl api: A java api for working with owl 2 ontologies,” in *Proceedings of the 6th International Conference on OWL: Experiences and Directions - Volume 529*, ser. OWLED’09, Chantilly, VA: CEUR-WS.org, 2009, pp. 49–58.

[23] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosofand, and M. Dean. “SWRL: A semantic web rule language combining OWL and RuleML,” W3C. (May 2004), [Online]. Available: http://www.w3.org/Submission/SWRL/.

[24] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, “Hermit: An owl 2 reasoner,” *Journal of Automated Reasoning*, vol. 53, Oct. 2014. DOI: 10.1007/s10817-014-9305-1.

[25] L. Ehrlinger and W. Wöß, “Towards a definition of knowledge graphs,” in *International Conference on Semantic Systems*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:8536105.

[26] C. W. Holsapple, “Knowledge and its attributes,” in *Handbook on Knowledge Management 1: Knowledge Matters*, C. W. Holsapple, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 165–188, ISBN: 978-3-540-24746-3. DOI: 10.1007/978-3-540-24746-3_9. [Online]. Available: https://doi.org/10.1007/978-3-540-24746-3_9.

[27] E. Cachia and M. Micallef, “An event-driven cartographic approach to modelling software engineering knowledge,” in *Proceedings of the International Conference on Knowledge Management and Information Sharing (IC3K 2011) - KMIS*, INSTICC, SciTePress, 2011, pp. 18–27, ISBN: 978-989-8425-81-2. DOI: 10.5220/0003630400180027.

[28] R. Bose, “Knowledge management metrics,” eng, *Industrial management + data systems*, vol. 104, no. 6, pp. 457–468, 2004, ISSN: 0263-5577.

[29] M. Earl, “Knowledge management strategies: Toward a taxonomy,” *Journal of Management Information Systems*, vol. 18, no. 1, pp. 215–233, 2001, ISSN: 07421222. [Online]. Available: http://www.jstor.org/stable/40398522.

[30] E. Cachia and M. Micallef, “On knowledge management in software development life cycles,” in *Workshop in ICT (WICT)*, University of Malta. Faculty of Information and Communication Technology, Msida, 2010, pp. 1–5.

[31] J. Cummings and B.-S. Teng, “Transferring r&d knowledge: The key factors affecting knowledge transfer success,” *Journal of Engineering and Technology Management*, vol. 20, pp. 39–68, Jun. 2003. DOI: 10.1016/S0923-4748(03)00004-3.

[32] M. Micallef and C. Colombo, “An event-driven language for cartographic modelling of knowledge in software development organisations,” 2011. [Online]. Available: https://www.um.edu.mt/library/oar/handle/123456789/30375.

[33] R. Botafogo, E. Rivlin, and B. Shneiderman, “Structural analysis of hypertexts: Identifying hierarchies and useful metrics,” *ACM Trans. Inf. Syst.*, vol. 10, pp. 142–, Apr. 1992. DOI: 10.1145/146802.146826.

[34] A. Rector, “Representing specified values in OWL: “value partitions” and “value sets”,” W3C, W3C Note, May 2005, https://www.w3.org/TR/2005/NOTE-swbp-specified-values-20050517/.

[35] A. Rector and N. Noy, “Defining n-ary relations on the semantic web,” W3C, W3C Note, Apr. 2006, https://www.w3.org/TR/2006/NOTE-swbp-n-aryRelations-20060412/.

[36] A. N. Lam, B. Elvesæter, and F. Martin-Recuerda, “A performance evaluation of owl 2 dl reasoners using ore 2015 and very large bio ontologies,” *Proceedings of the DMKG*, 2023.

[37] C. W. Holsapple, “Knowledge and its attributes,” in *Handbook on Knowledge Management 1: Knowledge Matters*, C. W. Holsapple, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 165–188, ISBN: 978-3-540-24746-3. DOI: 10.1007/978-3-540-24746-3_9. [Online]. Available: https://doi.org/10.1007/978-3-540-24746-3_9.

[38] R. Angles and C. Gutiérrez, “Subqueries in sparql,” in *Alberto Mendelzon Workshop on Foundations of Data Management*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:15040641.

[39] J. Documentation, *Functions in ARQ*, Available: https://jena.apache.org/documentation/query/library-function.html, [Accessed April 23, 2024].

[40] J. Piscopo, “Constructing and analysing knowledge maps via source code repository analysis,” Bachelor’s dissertation, University of Malta, 2022. [Online]. Available: https://www.um.edu.mt/library/oar/handle/123456789/107033.

[41] D. Ruggier, “Investigating the impact of pull requests on knowledge map construction,” Bachelor’s dissertation, University of Malta, 2023. [Online]. Available: https://www.um.edu.mt/library/oar/handle/123456789/113315.

[42] S. Mikhailov, M. Petrov, and B. Lantow, “Ontology visualization: A systematic literature analysis,” in *BIR Workshops*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:6353605.

[43] S. Lohmann, V. Link, E. Marbach, and S. Negru, “WebVOWL: Web-based Visualization of Ontologies,” in *Knowledge Engineering and Knowledge Management*, P. Lambrix *et al.*, Eds., Cham: Springer International Publishing, 2015, pp. 154–158, ISBN: 978-3-319-17966-7.

[44] S. Lohmann, S. Negru, F. Haag, and T. Ertl, “Visualizing ontologies with VOWL,” *Semantic Web*, vol. 7, no. 4, pp. 399–419, 2016.

Appendix A RDF-KL Framework

A.1 Construction Methods

The following lists the construction methods designed to populate the ontology with data. Note that "fresh" denotes that a created instance is assigned a new IRI (name).

- **personIdentification(personName)**: First, it verifies that *kl:personName* is not already declared. Then, it declares a fresh individual named *kl:personName*. The individual is being set to be an instance of *kl:Person* using the class assertion axiom.
- **removePerson(personName)**: First, it verifies that *kl:personName* is an instance of *kl:Person*. If so, the all axioms associated with *kl:personName* are removed, as well as all axioms related to instances of *kl:KnowledgeObservation* that are linked to *kl:personName* via the property *kl:hasPerson*.
- **knowledgeAssetIdentification(knowledgeAssetName,featureSet)**: $featureSet$ is a set whose elements are pairs of the type form ($\langle$featureName$\rangle$, $\langle$valueName$\rangle$). First, it verifies that *kl:knowledgeAssetName* is not already declared as an individual. Furthermore the method, verifies each feature and value is defined in OntoKL, that is, for each element in the feature set, *kl:*$\langle$*featureName*$\rangle$ is a subset of *kl:*$\langle$*KnowledgeAssetFeature*$\rangle$, and *kl:*$\langle$*valueName*$\rangle\langle$*featureName*$\rangle$*Value* is one of the sets forming the disjoint union of *kl:*$\langle$*featureName*$\rangle$. If both verifications hold, an individual named *kl:knowledgeAssetName* is declared. The newly created individual is set to be an instance of *kl:KnowledgeAsset* using the class assertion axiom. Then, a fresh individual is declared and set to be an instance of class *kl:*$\langle$*valueName*$\rangle\langle$*featureName*$\rangle$*Value*. Utilising an object property assertion axiom, kl:knowledgeAssetName is related to this instance using the property *kl:has*$\langle$*featureName*$\rangle$.
- **removeKnowledgeAsset(knowledgeAssetName)**: First, it verifies if *kl:knowledgeAssetName* is an instance of *kl:KnowledgeAsset*. If so, the method removes all axioms associated with *kl:knowledgeAssetName*, as well as all axioms related to instances of *kl:KnowledgeObservation* that are linked to *kl:knowledgeAssetName* via the property *kl:hasKnowledgeAsset*. Furthermore, all the associated feature values are removed, that is the individuals that are instances of value classes of KAs features that are related to *kl:hasKnowledgeAssetName* via an object property *kl:has*$\langle featureName\rangle$ are removed.

- **knowledgeObservation(knowledgeAssetName, personName, n)**: First, verifies that *kl:knowledgeAsset* and *kl:personName* are instances of classes *kl:KnowledgeAsset* and *kl:Person* respectively. If both verifications hold, then the method reifies a ternary relationship by declaring a fresh individual. Subsequently, this individual is instantiated as an instance of *kl:KnowledgeObservation* by applying the class assertion axiom. Using the object property assertion axiom, the individual is correlated with *kl:personName* and *kl:knowledgeAssetName* using the object properties *kl:hasPerson* and *kl:hasKnowledgeAsset*, respectively. Lastly, using the data property assertion axiom, the individual is linked to the value *n* via the property *kl:hasMagnitude*.

- **relatedTo(knowledgeAsset1Name,knowledgeAsset2Name)**: Establishes a generic relation between *kl:knowledgeAsset1Name* and *kl:knowledgeAsset2Name*, provided that both entities are instances of *kl:KnowledgeAsset*. It utilises the object property assertion axiom to state that *kl:knowledgeAsset1Name* is related to *kl:knowledgeAsset2Name* through the property *kl:relatedTo*.

- **dependentOn(knowledgeAsset1,knowledgeAsset2)**: Establishes a dependence relationship between *kl:knowledgeAsset1* and *kl:knowledgeAsset2*, provided that both entities are instances of *kl:KnowledgeAsset*. It utlises the the object property assertion axiom to state that *kl:knowledgeAsset1* is dependent on *kl:knowledgeAsset2* through the property *kl:dependsOn*.

- **composedOf(knowledgeAsset1,knowledgeAsset2)**: Establishes a composition relationship between *kl:knowledgeAsset1* and *kl:knowledgeAsset2*, provided that both entities are instances of *kl:KnowledgeAsset*. It utlises the the object property assertion axiom to state that *kl:knowledgeAsset1* is composed of *kl:knowledgeAsset2* through the property *kl:composedOf*.

- **deleteRelatedTo(knowledgeAsset1,knowledgeAsset2), deleteDependentOn(knowledgeAsset1,knowledgeAsset2), deleteComposedOf(knowledgeAsset1,knowledgeAsset2)**: Methods that allow the removal of an object property assertion of one of the three aforementioned properties between two KAs.

- **createTeam(teamName):** verifies whether *kl:teamName* is already declared. If not, it proceeds to declare a new class *kl:teamName*. Subsequently, utilising the subclass axiom, this class is specified to be a subclass of *kl:Person*.

- **addPersonToTeam(teamName, personName):** Confirms that *kl:personName* is an instance of *kl:Person*, then verifies that *kl:teamName* is a subclass of *kl:Person*, and checks if the person individual is not already an instance of the team class. If all

three conditions are met, use the class assertion axiom *kl:personName* is set to be an instance of *kl:teamName*.

- **deleteTeam(teamName), removePersonFromTeam(teamName,personName)**: Methods that allow the deletion of a team class involve deleting all of the axioms directly related to it, such as declarations and class assertions, and the removal of a person, *kl:personName*, from a team, *kl:teamName*.

A.2 Implementation

The framework has been implemented and deployed as a Maven package. The repository containing all the source code is located at `https://github.com/andimon/rdf-kl`. The project has been packaged as a Maven package, available at `https://github.com/andimon/rdf-knowledge-landscape/packages/2098096` on GitHub. A guide for using the Maven package, hosted on GitHub, as a dependency for a project can be found on GitHub.[1]

A.2.1 User Guide

This section gives a series of examples of how a knowledge landscape representation can be constructed.

```
ontoKL.addFeature("Age", Set.of("New", "Established", "Old"));
```

Listing A.1 Adding a new KA feature to the Ontology

```
package com.andimon.rdfknowledgelandscape.testscenario;

import com.andimon.rdfknowledgelandscape.features.Feature;

import static com.andimon.rdfknowledgelandscape.factories.KnowledgeLandscapeProperties.*;

public enum Age implements Feature {
    OLD(DEFAULT_NAMESPACE.getValue(String.class)+"OldAgeValue"),
    ESTABLISHED(DEFAULT_NAMESPACE.getValue(String.class)+"EstablishedAgeValue"),
    NEW(DEFAULT_NAMESPACE.getValue(String.class)+"NewAgeValue");

    private final String valueIRI;

    Age(String value) {
        this.valueIRI = value;
    }

    @Override
```

[1] https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-apache-maven-registry

```
    public String getFeatureIRI() {
        return DEFAULT_NAMESPACE.getValue(String.class)+"Age";
    }

    @Override
    public String getValueIRI() {
        return valueIRI;
    }
}
```

Listing A.2 Enum for Age KA feature

```
ontoKL.addFeature("Age", Set.of("New", "Established", "Old"));
KnowledgeLandscapeConstructor knowledgeLandscapeConstructor = new
    KnowledgeLandscapeConstructor(ontoKL);
knowledgeLandscapeConstructor.knowledgeAssetIdentification("Java", Set.of(Visibility.EXPLICIT,
     Category.TECHNICAL, Age.ESTABLISHED));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("OOP", Set.of(Category.TECHNICAL,
    Age.ESTABLISHED));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("SQL", Set.of(Operationality.
    PROCEDURAL, Category.BUSINESS, Sociality.INDIVIDUAL));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("SmallTalk", Set.of(Category.
    TECHNICAL,Age.OLD));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("Servlets", Set.of(Age.ESTABLISHED)
    );
knowledgeLandscapeConstructor.knowledgeAssetIdentification("WebDevelopment", Set.of(Age.
    ESTABLISHED));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("JDBC", Set.of(Age.ESTABLISHED));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("Haskell", Set.of(Category.
    TECHNICAL,Age.ESTABLISHED));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("Elixir", Set.of(Category.TECHNICAL
    ,Age.NEW));
knowledgeLandscapeConstructor.knowledgeAssetIdentification("FunctionalProgramming", Set.of(
    Category.TECHNICAL));
```

Listing A.3 Adding a new KAs

```
knowledgeLandscapeConstructor.personIdentification("Jane");
knowledgeLandscapeConstructor.personIdentification("Chris");
knowledgeLandscapeConstructor.personIdentification("John");
knowledgeLandscapeConstructor.personIdentification("Peter");
```

Listing A.4 Adding new persons

```

knowledgeLandscapeConstructor.createTeam("FunctionalProgrammers");
knowledgeLandscapeConstructor.createTeam("ObjectOrientedProgrammers");
knowledgeLandscapeConstructor.createTeam("Managers");
//Adding Persons to Teams
knowledgeLandscapeConstructor.addPersonToTeam("ObjectOrientedProgrammers", "Jane");
knowledgeLandscapeConstructor.addPersonToTeam("ObjectOrientedProgrammers", "Chris");
knowledgeLandscapeConstructor.addPersonToTeam("FunctionalProgrammers", "John");
knowledgeLandscapeConstructor.addPersonToTeam("FunctionalProgrammers", "Peter");
knowledgeLandscapeConstructor.addPersonToTeam("Managers", "John");
knowledgeLandscapeConstructor.addPersonToTeam("Managers", "Jane");
```

Listing A.5 Adding new teams

```
knowledgeLandscapeConstructor.dependentOn("Java", "OOP");
knowledgeLandscapeConstructor.dependentOn("SmallTalk", "OOP");
knowledgeLandscapeConstructor.dependentOn("Haskell", "FunctionalProgramming");
knowledgeLandscapeConstructor.dependentOn("Elixir", "FunctionalProgramming");
// If k1 is a part of k2 then k2 is composed using k1
knowledgeLandscapeConstructor.composedOf("Java", "JDBC");
knowledgeLandscapeConstructor.composedOf("Java", "Servlets");
knowledgeLandscapeConstructor.relatedTo("Servlets", "WebDevelopment");
```

Listing A.6 Specifying structural properties of KAs

```
knowledgeLandscapeConstructor.knowledgeObservation("Jane", "SmallTalk", 16);
knowledgeLandscapeConstructor.knowledgeObservation("Chris", "Java", 32);
knowledgeLandscapeConstructor.knowledgeObservation("John", "Elixir", 17);
knowledgeLandscapeConstructor.knowledgeObservation("John", "Haskell", 20);
knowledgeLandscapeConstructor.knowledgeObservation("Peter", "Elixir", 16);
knowledgeLandscapeConstructor.knowledgeObservation("Peter", "Haskell", 30);
```

Listing A.7 Creating Knowledge Observations

A.2.2 Running tests

IntelliJ IDEA is used to run the tests. A successful test run contains 63 tests, all passing successfully.

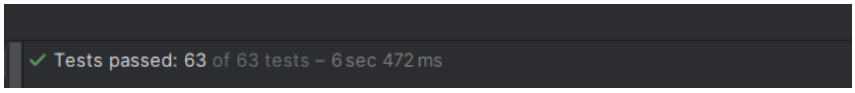


Figure A.1 Number of tests

A.3 Visualising the ontology

Visualisation is also an important task related to ontologies. It is mainly based on a mapping from information to a graphical representation in order to facilitate data interpretation [42]. The OntoKL ontology has been encoded in Turtle syntax [2]. The resulting Turtle document is uploaded to the WebVOWL visualisation service for visualising ontologies [43]. For detailed information on the visual notation used by the service, one may refer to the detailed information in [44].

[2] https://github.com/andimon/rdf-kl/blob/dev/rdf_knowledge_landscape/src/test/java/com/andimon/rdfknowledgelandscape/ontology/SerialisationTest.java

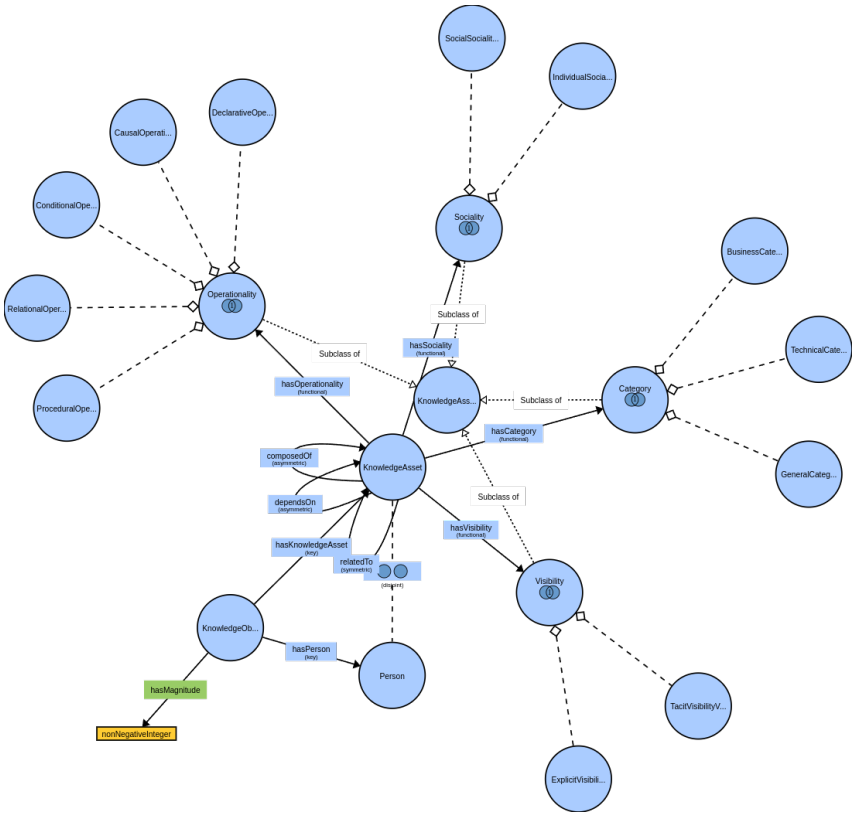


Figure A.2 OntoKL visualised using WebVowl

Appendix B Further SPARQL queries and results

B.1 Queries

```
SELECT ?perSrc ?perRec ?ka (SUM(IF(COALESCE(?perSrcMag, 0) - COALESCE(?perRecMag, 0) < 0, 0,
    COALESCE(?perSrcMag, 0) - COALESCE(?perRecMag, 0))) AS ?knowledgeDistance)
WHERE {
    ?perSrc a kl:Person .
    ?perRec a kl:Person .
    ?ka a kl:KnowledgeAsset .
    OPTIONAL {
        ?ko kl:hasKnowledgeAsset ?ka .
        ?ko kl:hasPerson ?perSrc .
        ?ko kl:hasMagnitude ?perSrcMag .
    }
    OPTIONAL {
        ?ko kl:hasKnowledgeAsset ?ka .
        ?ko kl:hasPerson ?perRec .
        ?ko kl:hasMagnitude ?perRecMag .
    }
    FILTER (?perSrc != ?perRec)
}
GROUP BY ?perSrc ?perRec ?ka
ORDER BY DESC(?knowledgeDistance)
```

Listing B.1 SPARQL query for analysing the distance between two persons

```
PREFIX kl: <namespace>
PREFIX afn: <http://jena.apache.org/ARQ/function#>
SELECT ?ka ?kmrVal WHERE {
    {
        SELECT ?ka ?kmr
        WHERE {
            {
                SELECT ?ka (SUM(?knowledgeMatrix) AS ?knowledgeInDistance)
                WHERE {
                    # Calculate Knowledge In Distance Here
                    SELECT ?person ?ka
    (IF(COALESCE(?mag,0)=0,MAX(?genericMag)+1,MAX(?genericMag)/?mag) AS ?knowledgeMatrix)
                    WHERE {
                        ?person a kl:Person .
                        ?ka a kl:KnowledgeAsset .
                        OPTIONAL {
                            ?ko kl:hasKnowledgeAsset ?ka .
                            ?ko kl:hasPerson ?person .
                            ?ko kl:hasMagnitude ?mag .
                        }
                        OPTIONAL {
                            ?genericKo kl:hasKnowledgeAsset ?genericKA .
                            ?genericKo kl:hasPerson ?genericPerson .
                            ?genericKo kl:hasMagnitude ?genericMag .
                        }
                    }
```

```
                    GROUP BY ?person ?ka ?mag
                }
                GROUP BY ?ka
            }
            {
                # Calculate Knowledge Distance Here
                SELECT (SUM(?knowledgeMatrix) AS ?knowledgeDistance)
                WHERE {
                    {
                        SELECT ?person ?ka (IF(COALESCE(?mag, 0) = 0, MAX(?genericMag) + 1,
    MAX(?genericMag) / ?mag) AS ?knowledgeMatrix)
                        WHERE {
                            ?person a kl:Person .
                            ?ka a kl:KnowledgeAsset .
                             OPTIONAL {
                                ?ko kl:hasKnowledgeAsset ?ka .
                                ?ko kl:hasPerson ?person .
                                ?ko kl:hasMagnitude ?mag .
                            }
                            OPTIONAL {
                                ?genericKo kl:hasKnowledgeAsset ?genericKA .
                                ?genericKo kl:hasPerson ?genericPerson .
                                ?genericKo kl:hasMagnitude ?genericMag .
                            }
                        }
                        GROUP BY ?person ?ka ?mag
                    }
                }
            }
            BIND( ?knowledgeDistance/?knowledgeInDistance AS ?kmr)
        }
        GROUP BY ?ka ?kmr
    }
    {
        SELECT ((SUM((?kmr - ?mean)*(?kmr - ?mean)))/(COUNT(?kmr)) AS ?variance) ?mean WHERE {
            {
                SELECT ?ka ?kmr
                WHERE {
                    {
                        SELECT ?ka (SUM(?knowledgeMatrix) AS ?knowledgeInDistance)
                        WHERE {
                            # Calculate Knowledge In Distance Here
                            SELECT ?person ?ka
    (IF(COALESCE(?mag,0)=0,MAX(?genericMag)+1,MAX(?genericMag)/?mag) AS ?knowledgeMatrix)
                            WHERE {
                                ?person a kl:Person .
                                ?ka a kl:KnowledgeAsset .
                                OPTIONAL {
                                    ?ko kl:hasKnowledgeAsset ?ka .
                                    ?ko kl:hasPerson ?person .
                                    ?ko kl:hasMagnitude ?mag .
                                }
                                OPTIONAL {
                                    ?genericKo kl:hasKnowledgeAsset ?genericKA .
                                    ?genericKo kl:hasPerson ?genericPerson .
                                    ?genericKo kl:hasMagnitude ?genericMag .
```

```
                                }
                            }
                            GROUP BY ?person ?ka ?mag
                        }
                        GROUP BY ?ka
                    }
                    {
                        # Calculate Knowledge Distance Here
                        SELECT (SUM(?knowledgeMatrix) AS ?knowledgeDistance)
                        WHERE {
                            {
                                SELECT ?person ?ka (IF(COALESCE(?mag, 0) = 0,
MAX(?genericMag) + 1, MAX(?genericMag) / ?mag) AS ?knowledgeMatrix)
                                WHERE {
                                    ?person a kl:Person .
                                    ?ka a kl:KnowledgeAsset .
                                     OPTIONAL {
                                        ?ko kl:hasKnowledgeAsset ?ka .
                                        ?ko kl:hasPerson ?person .
                                        ?ko kl:hasMagnitude ?mag .
                                    }
                                    OPTIONAL {
                                        ?genericKo kl:hasKnowledgeAsset ?genericKA .
                                        ?genericKo kl:hasPerson ?genericPerson .
                                        ?genericKo kl:hasMagnitude ?genericMag .
                                    }
                                }
                                GROUP BY ?person ?ka ?mag
                            }
                        }
                    }
                    BIND( ?knowledgeDistance/?knowledgeInDistance AS ?kmr)
                }
                GROUP BY ?ka ?kmr
            }
            {
                SELECT (AVG(?kmr) AS ?mean) WHERE {
                    {
                        SELECT ?ka ?kmr
                        WHERE {
                            {
                                SELECT ?ka (SUM(?knowledgeMatrix) AS ?knowledgeInDistance)
                                WHERE {
                                    # Calculate Knowledge In Distance Here
                                    SELECT ?person ?ka
(IF(COALESCE(?mag,0)=0,MAX(?genericMag)+1,MAX(?genericMag)/?mag) AS ?knowledgeMatrix)
                                    WHERE {
                                        ?person a kl:Person .
                                        ?ka a kl:KnowledgeAsset .
                                        OPTIONAL {
                                            ?ko kl:hasKnowledgeAsset ?ka .
                                            ?ko kl:hasPerson ?person .
                                            ?ko kl:hasMagnitude ?mag .
                                        }
                                        OPTIONAL {
                                            ?genericKo kl:hasKnowledgeAsset ?genericKA .
```

```
                                        ?genericKo kl:hasPerson ?genericPerson .
                                        ?genericKo kl:hasMagnitude ?genericMag .
                                    }
                                }
                                GROUP BY ?person ?ka ?mag
                            }
                            GROUP BY ?ka
                        }
                        {
                            # Calculate Knowledge Distance Here
                            SELECT (SUM(?knowledgeMatrix) AS ?knowledgeDistance)
                            WHERE {
                                {
                                    SELECT ?person ?ka (IF(COALESCE(?mag, 0) = 0,
    MAX(?genericMag) + 1, MAX(?genericMag) / ?mag) AS ?knowledgeMatrix)
                                    WHERE {
                                        ?person a kl:Person .
                                        ?ka a kl:KnowledgeAsset .
                                         OPTIONAL {
                                            ?ko kl:hasKnowledgeAsset ?ka .
                                            ?ko kl:hasPerson ?person .
                                            ?ko kl:hasMagnitude ?mag .
                                        }
                                        OPTIONAL {
                                            ?genericKo kl:hasKnowledgeAsset ?genericKA .
                                            ?genericKo kl:hasPerson ?genericPerson .
                                            ?genericKo kl:hasMagnitude ?genericMag .
                                        }
                                    }
                                    GROUP BY ?person ?ka ?mag
                                }
                            }
                        }
                        BIND( ?knowledgeDistance/?knowledgeInDistance AS ?kmr)
                    }
                    GROUP BY ?ka ?kmr
                }
            }
        }
        GROUP BY ?mean
    }
    BIND(afn:sqrt(?variance) AS ?sdev)
    BIND (
      COALESCE(
        IF(?kmr <= ?mean, "LOW", 1/0),
        IF( ?mean < ?kmr  && ?kmr <= (?mean+?sdev), "MEDIUM", 1/0),
        IF(?kmr > (?mean+?sdev), "HIGH", 1/0),
        "Undefined"
      ) AS ?kmrVal
    )
}
```

Listing B.2 SPARQL query for analysing KMR for each Knowledge Asset

B.2 Results

ka	feature	value
kl:Java	kl:Category	kl:TechnicalCategoryValue
kl:Haskell	kl:Category	kl:TechnicalCategoryValue
kl:JDBC	kl:Category	"Undefined"
kl:WebDevelopment	kl:Category	"Undefined"
kl:SQL	kl:Category	kl:BusinessCategoryValue
kl:FunctionalProgramming	kl:Category	kl:TechnicalCategoryValue
kl:OOP	kl:Category	kl:TechnicalCategoryValue
kl:Elixir	kl:Category	kl:TechnicalCategoryValue
kl:SmallTalk	kl:Category	kl:TechnicalCategoryValue
kl:Servlets	kl:Category	"Undefined"
kl:Java	kl:Sociality	"Undefined"
kl:Haskell	kl:Sociality	"Undefined"
kl:JDBC	kl:Sociality	"Undefined"
kl:WebDevelopment	kl:Sociality	"Undefined"
kl:SQL	kl:Sociality	kl:IndividualSocialityValue
kl:FunctionalProgramming	kl:Sociality	"Undefined"
kl:OOP	kl:Sociality	"Undefined"
kl:Elixir	kl:Sociality	"Undefined"
kl:SmallTalk	kl:Sociality	"Undefined"
kl:Servlets	kl:Sociality	"Undefined"
kl:Java	kl:Age	kl:EstablishedAgeValue
kl:Haskell	kl:Age	kl:EstablishedAgeValue

Figure B.1 Result for query shown in Listing 4.1 (redacted for length)

ka	kmrVal
kl:SmallTalk	"MEDIUM"
kl:Servlets	"MEDIUM"
kl:FunctionalProgramming	"LOW"
kl:Haskell	"LOW"
kl:Java	"MEDIUM"
kl:Elixir	"LOW"
kl:SQL	"HIGH"
kl:OOP	"LOW"
kl:JDBC	"MEDIUM"
kl:WebDevelopment	"HIGH"

Figure B.2 Results for query shown in Listing B.2

Appendix C RDF

Definition C.1 (Literal). *Let $X = LEX \times (\mathcal{D} \setminus \{langStringIRI\})$ and $Y = LEX \times \{langStringIRI\} \times LANGTAGS$. The set of all literals is represented by set of tuples $\mathcal{L} = X \cup Y$.*

Blank nodes refer to items that are disjoint from IRIs and literals, used in statements to express the existence of a resource with a given relationship, without explicitly naming it. The internal structure of a blank node is arbitrary, depending solely on an implementation of the RDF-graph.

Definition C.2 (Blank node). *Let $\mathcal{U}$ be the universal set. A blank node is an arbitrary element in the set $\mathcal{U} \setminus (\mathcal{I} \cup \mathcal{L})$. We denote blank nodes by b_i with $i \in \mathbb{N}$ and the set of all blank nodes by $\mathcal{B}$.*

With the definitions of IRIs, literals, and blank nodes established, we can conclude this section with a complete definition of an RDF graph and a definition of collections.

Definition C.3 (RDF graph revised). *An RDF graph is a set $\{\tau(s, p, o) \ : \ \langle s, p, o\rangle \in S\}$, where $S \in \mathcal{P}((\mathcal{B} \cup \mathcal{I}) \times \mathcal{I} \times (\mathcal{B} \cup \mathcal{I} \cup \mathcal{L}))$, i.e., S is any subset of $(\mathcal{B} \cup \mathcal{I}) \times \mathcal{I} \times (\mathcal{B} \cup \mathcal{I} \cup \mathcal{L})$. In this project, unless specified otherwise, the term "graph" is assumed to refer to an RDF graph.*

RDF also provides vocabulary to denote lists using a head-tail notation. For example, if we want to represent a list $[:Andre, :Mark, :Chris]$, we include the following triples in the graph: $\tau(b_0, rdf{:}first, :Andre)$; $\tau(b_0, rdf{:}rest, b_1)$; $\tau(b_1, rdf{:}first, :Mark)$; $\tau(b_1, rdf{:}rest, b_2)$; $\tau(b_2, rdf{:}first, :Chris)$ and $\tau(b_2, rdf{:}rest, rdf{:}nil)$.

Definition C.4 (Collection). *A graph G is said to have a collection $C = [X_0, X_1, \ldots X_n]$ with elements in $\mathcal{B} \cup \mathcal{I}$ iff*
$\tau(b_0, rdf{:}first, X_0), \tau(b_0, rdf{:}rest, b_1), \tau(b_1, rdf{:}first, X_1), \tau(b_1, rdf{:}rest, b_2),$
$\ldots, \tau(b_n, rdf{:}first, X_n), \tau(b_n, rdf{:}rest, rdf{:}nil) \in G$, *where b_i with $i \in \{1, \ldots, n\}$ are arbitrary but distinct blank nodes, $rdf{:}first$, $rdf{:}rest$ are used to obtain head and tail of lists respectively and $rdf{:}nil$ denotes an* empty list, *which is used as an end of list indicator. We denote the set containing the aforementioned triples as, $\tau(X, R, (\ X_0\ X_1 \ldots\ X_n\ ))$, so therefore $\tau(X, R, b_0)$, signifies that some entity X is associated with C under predicate $R \in \mathcal{I}$.*

Appendix D OWL

In this appendix, an explanation of some fundamental axioms is provided, along with their respective RDF representations.

D.1 Declaration

To help with managing an ontology, OWL includes a declaration axiom where each class, property, or individual should be declared in an ontology before it can be used within that ontology and any ontologies that import it. For example, the declaration of an entity, CN, as a class is represented by the triple $\tau(CN, rdf:type, owl{:}Class)$.

D.2 Classes

Classes are sets generally used to model collections of individuals that have something in common. For instance, if we define a class of dogs denoted by the IRI $:Dogs$, we can assign an individual, such as $:Ruby$, to this class, signifying that $:Ruby$ is a dog. This can be stated by a class assertion axiom represented by the triple $\tau(:Ruby, rdf{:}type, :Dog)$. A class I_1 is a subclass of I_2 if all individuals in I_1 are also in I_2. To allow the reasoner to draw conclusions, we specify this relation via a subclass axiom, represented by the triple $\tau(I_1, rdfs{:}subclassOf, I_2)$. Generally, a subclass axiom, is valid if the statement "every I_1 is an I_2" holds true. For instance, if we have classes for dogs and animals, it is correct to state that $\tau(:dogs, rdfs{:}subclassOf, :animals)$ since every dog is indeed an animal.

D.2.1 Equivalent Classes

Two classes I_1 and I_2 are said to be equivalent if they contain the exact same set of individuals. This relationship is represented by the triple, $\tau(I_1, owl{:}equivalentClass, I_2)$. Explicitly stating when classes are known to be equivalent is important for enabling the reasoner to draw accurate conclusions. Stating that a graph contains the axiom $\tau(I_1, owl{:}equivalentClass, I_2)$ is equivalent to stating that the ontology graph contains the axioms $\tau(I_1, rdfs{:}subclassOf, I_2)$ and $\tau(I_2, rdfs{:}subclassOf, I_1)$.

D.2.2 Class Disjointness

There may exist classes I_1 and I_2 that have no common elements. When we know that two classes I_1 and I_2 are disjoint, it's important to specify this as an axiom to enable a

reasoner to make the desired inferences. The axiom is represented by the triple $\tau(I_1, owl{:}disjointWith, I_2)$.

D.2.3 Enumeration of Individuals

A class I_1 can be stated to represent a finite set of individuals $I_2, I_3, \ldots, I_{n-1}, I_n$. This relation can be specified by inclusion of triples, $\tau(I_1, owl{:}equivalentClass, b_0)$, $\tau(b_0, rdf{:}type, owl{:}Class)$, $\tau(b_0, owl{:}oneOf, (I_2, \ldots I_n))$.

D.3 Complex Classes

Classes can be constructed using axioms such as intersection, union, and complement. Let us consider declared classes I_1, I_2, and I_3. We can say that I_1 is equivalent to the intersection of classes I_2 and I_3 if and only if all instances common to I_2 and I_3 are in I_1. Similarly, we can say that I_1 is equivalent to the union of classes I_2 and I_3 if and only if an individual is in I_2 or I_3, then the same individual will be in I_1. Furthermore, we can say that I_1 is equivalent to the complement of I_2 if and only if all individuals that are in I_1 are not in I_2, and all individuals that are not in I_1 are in I_2. We can nest operators to build more complex classes, and these operators can be used in subclass relations. For example, we can state that I_1 is a subclass of the intersection of I_2 and I_3.

D.4 Individuals

D.4.1 Equality and inequality individuals

OWL does not make a unique name assumption, meaning the same individual can be referred to by two different IRIs. For two individuals I_1 and I_2, we can explicitly state whether I_1 and I_2 refer to the same individual or different individuals. For example, the triple $\tau(I_1, owl{:}differentFrom, I_2)$ states that individuals I_1 and I_2 are different.

D.5 Object Properties

We have described individuals, their membership in classes, and how classes can be related based on their instances. Often, we need to specify how individuals can be related to other individuals. For this, we can use object properties, and the object property assertion axiom. For example, the object assertion axiom relating individuals I_1 and I_2 by an object property I_3 is represented by a triple $\tau(I_1, I_3, I_2)$. Note that generally, the order in which two individuals are related by an object property matters.

For example, if I_1 is related to I_2 under the property $fatherOf$, then I_2 should not be related to I_1 under the same property. This is a common source of error in modelling, and "of" or "has" constructions in object properties (e.g., $fatherOf$ or $hasFather$) can help clear any confusion [5], by allowing for a direct and unique reading.

D.5.1 Negative Object Property

We can also state that two individuals I_1 and I_2 are not related by some object property I_3. This axiom can be stated by the following triples $\tau(b_0, rdf{:}type, owl{:}NegativePropertyAssertion)$, $\tau(b_0, owl{:}sourceIndividual, I_1)$, $\tau(b_0, owl{:}assertionProperty, I_3)$, and, $\tau(b_0, owl{:}targetIndividual, I_2)$.

D.5.2 Subproperties

For two declared object properties I_3 and I_4, we say that I_3 is a subproperty of I_4 if for all individuals I_1 and I_2 such that I_1 is related to I_2 under I_3, then I_1 is related to I_2 under I_4. The subclass axiom is represented by the following triple $\tau(I_3, rdfs{:}subPropertyOf, I_4)$.

D.5.3 Domain and Range Restrictions

For a given object property I_3, we can declare its domain and range as some classes I_4 and I_5 respectively. This allows a reasoner to infer class membership. So let's say that two individuals I_1 and I_2 are related by I_3. Then, with the established domain and range of I_3, a reasoner is able to infer that I_1 is an instance of class I_4 and I_2 is an instance of class I_5. For example, the domain axiom is stated by the triple $\tau(I_3, rdfs{:}domain, I_4)$.

D.6 Datatype Properties

In the previous subsection, we explored how we can relate individuals. In many cases, individuals need to be described using data values. For each assigned value, we can make use of the XML Schema Datatypes [13].

D.6.1 Negative Datatype Property

This subsection is analogous to subsection D.5.1. For example, we may state that under datatype property I_3, I_1 is not associated with the some value.

D.6.2 Domain and range restrictions

This subsection is analogous to subsection D.5.3. In this case, for the assigned range, it is a datatype instead of a class. For example, a domain and range assertion of a datatype property, let's say I_3, with class I_1 and datatype $xsd{:}string$, respectively.

D.7 Property Restrictions

D.7.1 Existential Quantification

One restriction for a property is called existential quantification, that defines a class I_1 as the set of all individuals that are connected to some individual in class I_2 under some object property I_3.

D.7.2 Universal Quantification

There exists another restriction called universal quantification, which defines a class I_1 as the set of all individuals that, under some particular relation I_3, all the individuals related to instances in I_1 under I_3 must be instances of I_4. Note that individuals that do not have any relation with another individual under I_3 are in I_1 and are in I_4 since the statement "if such individual is related to instances in I_1 under I_3, then the instances it is related to must be an instance of I_4" is vacuously true.

D.7.3 Cardinality restrictions

OWL offer constucts to specify the maximum, minimum, or exact number of individuals an individual of a class can be related to.

D.7.4 Properties Characteristics

We can specify the following characteristics for object properties: Given declared object properties I_1, I_2, I_3, we can state the following characteristics about them:

- **inverse:** If I_1 is stated to be the inverse of I_2, then if individual I_4 is related to individual I_5 by I_1, then individual I_5 is related to individual I_4 by I_2.
- **symmetry**: If I_1 is stated to be symmetric, then if I_1 relates I_4 with I_5, then I_1 also relates I_5 with I_4.
- **asymmetry**: If I_1 is stated to be asymmetric, then if I_1 relates I_4 with I_5, then I_1 cannot relate I_5 with I_4.

- **disjointness**: If I_1 and I_2 are stated to be disjoint, then no two distinct individuals can be interlinked by both I_1 and I_2.
- **reflexive**: If I_1 is stated to be reflexive, then it relates individuals to themselves.
- **irreflexive**: If I_1 is stated to be irreflexive, then it does not relate individuals to themselves.
- **functional**: If I_1 is stated to be functional, then every individual can be linked by the I_1 property to at most one other individual.
- **inverse functional**: If I_1 is stated to be inverse functional, then the inverse of I_1 is functional.
- **transitive**: If I_1 is stated to be transitive, then I_1 interlinks two individuals I_4 and I_6 whenever it interlinks individual I_4 with individual I_5 and individual I_5 with individual I_6 for some individual I_5.

For example, the triple $\tau(I_1, owl{:}inverseOf, I_2)$ is used to state that I_1 is the inverse of I_2, or $\tau(I_1, rdf{:}type, owl{:}SymmetricProperty)$ is used to state that I_1 is a symmetric property. Given a datatype property, we can state that it is functional.

D.7.5 Keys

In OWL, a set of data or object properties, $I_1, \ldots, I_n$ can be designated as a key for a class I_0. This signifies that every instance of the class is uniquely identified by the combination of values that these properties hold in relation to I_0. In RDF, this is specified by triples $\tau(I_0, owl{:}hasKey, (I_1 \ldots I_n))$.

D.8 Others

This appendix provided an outline of OWL. For a comprehensive view of axioms, i.e., statements that are made in an ontology, how OWL manages ontologies, and various sub languages of OWL, one is recommended to follow the “OWL 2 Web Ontology Language Primer” [5].