Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Developing an Ontology on Information Governance Using Description Logic

Denis Moslavac

Course of Study:	Autonomous Systems
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dipl. Phys. Cataldo Mega
Commenced:	August 1, 2023
Completed:	February 1, 2024

Abstract

Currently, companies have high compliance requirements. Enterprises consult with Subject Matter Experts (SME) on regulatory compliance, for designing suitable compliance solutions, while simultaneously assessing for possible violations. While this is a crucial aspect of Information Governance (IG), this constitutes a complex and financially burdensome challenge, especially for startups or smaller companies. Furthermore, according to recent surveys, approximately 40% of companies are exposed to compliance risks due to insufficient implemented IG measures. To overcome this problem, this thesis introduces an Information Governance Ontology (IGONTO), which implements concepts and knowledge of the IG domain. IGONTO demonstrates how to capture IG knowledge in an ontology together with regulatory requirements, and possible solutions that satisfy regulatory compliance use cases. Moreover, individual architecture solutions are inferred based on regulatory requirements. IGONTO is developed as an aggregation of multiple ontologies, each specific to a sub-domain of the larger IG-domain. These sub-domains represent unique but relevant contexts, with concepts that establish connections between them. We designed IGONTO with a focus on the European General Data Protection Regulation (GDPR) and its regulatory requirements. Validation is performed through SHACL scripts implemented for each domain to ensure consistency. IGONTO's usefulness is verified by evaluating two use cases that show what companies of different sizes must implement to reach regulatory compliance. The evaluation demonstrates that IGSO can correctly answer SPARQL queries that help identify the necessary compliance measures as required by the 99 articles from GDPR.

Contents

List of Figures

List of Tables

Listings

Acronyms

ARM Association Rule Mining. 25

CCO Chief Compliance Officer. 51

CIGO Chief Information Governance Officer. 51

CSO Chief Security Officer. 51

DCAT Data Catalog Vocabulary. 23

DL Description Logic. 5, 11, 25, 28, 29, 33, 34, 45, 46, 66, 101, 113

DMC Document Management Component. 94

DPO Data Protection Officer. 51

DSL Domain Specific Language. 70

ECM Enterprise Content Management. 62, 92

EEA European Economic Area. 47, 82

EIM Enterprise Information Management. 15, 17, 117

ERM Electronic Records Management. 51

FOL First-Order Logic. 24, 25, 28

FR Functional Requirements. 91

GDPR General Data Protection Regulation. 3, 16, 27, 47, 57, 66

IG Information Governance. 3, 15, 17, 62, 82, 113, 117

IGONTO Information Governance Ontology. 3

ILP Inductive Logic Programming. 25

NFR Non-Functional Requirements. 91

OASIS Organization for the Advancement of Structured Information Standards. 70

OR Obligatory Requirements. 91

PII Personally Identifiable Information. 105, 116

RDBMS Relational Database Management System. 94

RESCAL Relational Single-Channel Analysis of Learning. 25

RIM Records and Information Management. 51, 78, 79, 89, 90, 91

RR Regulatory Requirements. 86

SHACL Shapes Constraint Language. 5, 35, 36, 37, 39, 42, 101

SME Subject Matter Experts. 3, 15

SWRL Semantic Web Rule Language. 6, 45, 47, 50, 56, 74, 75, 77, 79, 113, 115

TOSCA Topology and Orchestration Specification for Cloud Applications. 70

URI Uniform Resource Identifier. 41, 87, 101, 102

1 Introduction

Enterprise Information Management (EIM) is crucial to the success of modern enterprises and becomes complicated and challenging when required to satisfy the numerous compliance requirements. Regulatory requirements vary according to the jurisdictions in which the companies operate. Because many companies seek to expand to more regions, with additional requirements, overall compliance becomes even more complicated. Therefore, enterprises hire compliance SME's to address this challenge at higher costs.

SME's are crucial in bridging the knowledge gap between compliance needs and effective EIM, but the need to continuously adapt their IG strategy leads to high expenses. This financial burden poses a major challenge to startups and small- or medium-sized companies that must operate with a limited budget. Despite the financial challenges, both domain expertise and knowledge remain indispensable to ensure compliance and avoid non-conformance with laws and regulations.

This master's thesis aims to implement an ontology on IG. The term "ontology", of Greek origin, means "the study of being", which can be divided into *onto-* ("being" or "that which is") and *-logia* ("logical discourse"). While the origin of the term was already recorded by the Oxford English Dictionary in 1664 and has philosophical roots, its prominence in technical applications began with the development of the World Wide Web in the 1990s, which further accelerated interest in ontologies. In technical domains, ontologies are organized frameworks that define concepts and their relationships within a formal structure, enabling an unambiguous and systematic representation of complex and interrelated aspects of a specific domain.

This thesis suggests an ontology that encapsulates the knowledge of the information governance domain and implements a prototype to prove the approach. Given the complex nature of the IG domain, it was necessary to split the IG domain context into three sub-domains each with their respective ontology that are later merged into the one large ontology on IG. The three domain contexts include:

1. The Jurisdiction domain, which encompasses the regulations an enterprise must comply with.
2. The Enterprise domain, which includes various subdomains representing different aspects of an enterprise's structure
3. The Implementation domain, which, despite being a part of the enterprise, combines the knowledge of the first two domains, to determine the necessary actions and needs for compliance.

IGONTO aims at demonstrating that the knowledge of the IG domain can be effectively represented within an ontology and that companies can dynamically retrieve IG solution knowledge based on their profile or needs. Our investigation showed that regulations expose similar regulatory requirements, therefore we used the European General Data Protection Regulation (GDPR) to examine different representative use cases and extracted from those a set of regulatory requirements.

2 Context

This thesis is based on the previous contributions of Cataldo Mega in "An Ontology-based Knowledge Management Model on Information Governance", [IGONTO23]. Based on the situation illustrated in the introduction, Mega proposed an ontology-based framework (IGONTO) to simplify access to knowledge regarding the IG domain. His work encompasses a domain analysis, which forms the basis of implementing models describing several IG subdomains. These models are combined into one ontology and their concepts are mapped to certain services and components, which represent the solution to the knowledge retrieved from the ontology.

The domain analysis evaluates various important sources of the IG domain. These include the following:

- Associations, with detailed interpretations of regulatory mandates and requirements.
- ISO standards, that describe standardized best practices.
- Regulations, with descriptions of regulatory policies and rules within their jurisdiction.
- Enterprise Information Management (EIM) standards, which describe the architectural connections between governance and design concepts.
- Vendors, with their platform and service offerings that aggregate required functional components for implementing the enterprise-wide solution infrastructure.

The terms and concepts of each domain source are harmonized into one taxonomy. Several concept models were implemented based on this taxonomy to describe different subdomains within IG. The Organizational Model (ORG) focuses on the administrative view of organization units and their interaction with the organizational processes within the enterprise [IGONTO23, p. 174]. The concept of data and the description of Information and its value are illustrated in the Information Model (INF) [IGONTO23, p. 174], whereas its management and lifecycle are implemented in the Lifecycle Model (LCS) [IGONTO23, p. 175]. The System Model (SYS) entails core notions relevant to the design and architecture of solutions based on the requirements of an IG program [IGONTO23, p. 175]. The core functional components within IG are summarized in the Component Model (CPT) [IGONTO23, p. 175], and the Platform Model (PLT) [IGONTO23, p. 176] describes the execution environment and the services needed for the enterprise infrastructure [IGONTO23, p. 176]. The concepts of these models are implemented within the IGONTO ontology and its foundation.

Figure 2.1 describes the IGONTO framework in detail. IGONTO comprises five development stages of information governance: acquisition, pre-processing, processing, post-processing, storing, and archiving, using the enterprise repository. During the acquisition stage, vocabulary terms are gathered that best describe the IG domain context. The pre-processing stage determines key

concepts, defines policies and associated rules based on the information gathered in stage one, and classifies them into a taxonomy. Stage three processes the concepts found into an IG ontology (IGO), and creates class instances (individuals) based on the semantic schema into a knowledge graph (IGG). In the post-processing stage, blueprints and design solutions that satisfy individual use cases are retrieved, using SPARQL queries. Afterward, service templates are mapped to individual design concepts, thus providing all components required to complete an IG solution.

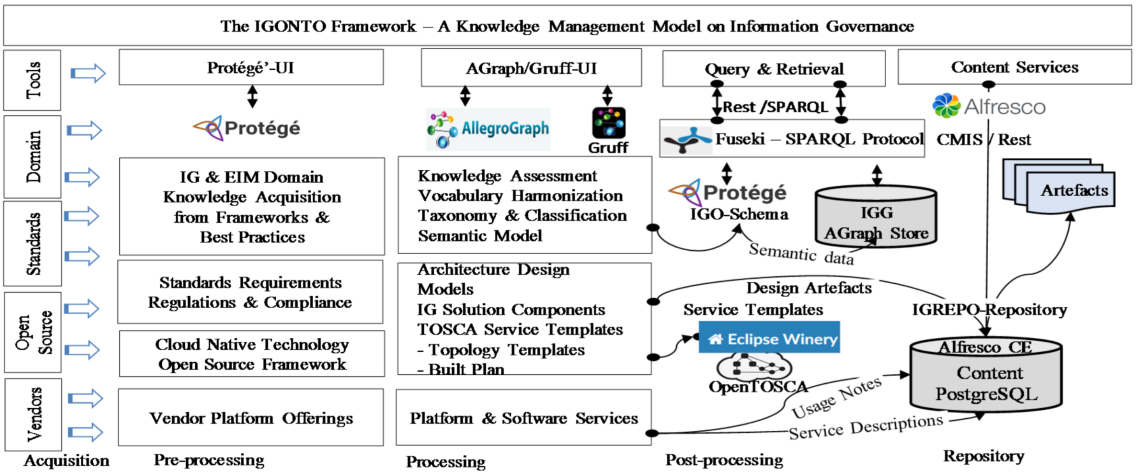


Figure 2.1: The IGONTO framework [IGONTO23].

This thesis focuses on the pre-processing and the processing stages as defined in [IGONTO23], in which the acquired knowledge is implemented into IGONTO, an ontology on information governance. The ontology development was done using the ontology editor, Protegé, with which we implemented the concepts of the pre-processing phase [Pro23]. To visualize query results, we utilized AllegroGraph and its interactive visualization tool, Gruff [23a]. The ontology implementation also includes validation logic, which was executed using Stardog [23d], an online database for ontologies.

3 Ontology Development - Prerequisites

This chapter provides the prerequisites of IGONTO and explains the implemented design decisions. Section 3.1 explains the general approaches. Section 3.2 analyzes reasoning parameters, which are important for inferring knowledge within our ontology, and the analysis of reasoning requirements in IGONTO. Based on this, the most suitable reasoning type for IGONTO is presented. Section 3.3 delves into the validation of the ontology, describing the structure and additional useful properties in IGONTO. Considering that multiple subdomains are implemented, Section 3.4 explains how these subdomains are brought together.

3.1 General Approach

This section describes three general approaches to the IGONTO ontology. Section 3.1.1 describes the ontology hierarchy used to build IGONTO. How object properties are implemented from the structural perspective is explained in Section 3.1.2, whereas the use of datatype properties is explained in Section 3.1.3.

3.1.1 Ontology Hierarchy

The IGONTO ontology consists of five top-level sub-ontologies. Each sub-ontology describes its subdomain within information governance. This allows the individual domains to be represented and implemented independently.

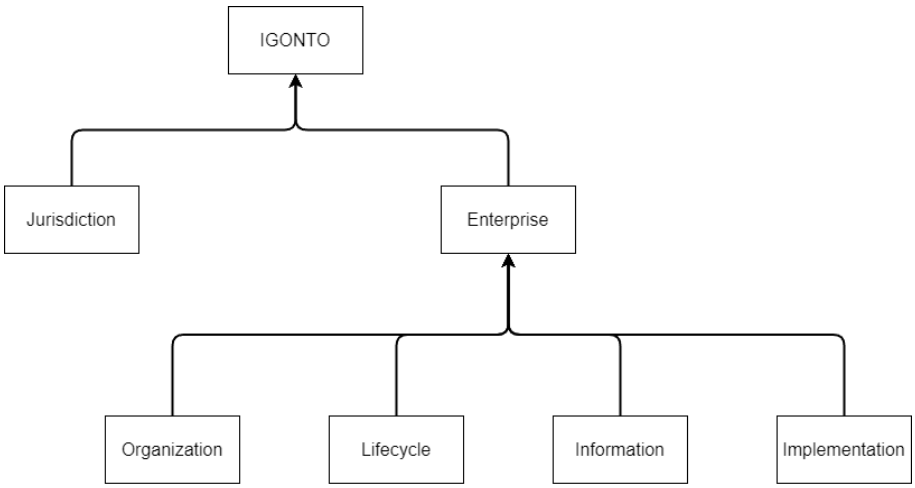


Figure 3.1: IGONTO topology.

Figure 3.1 shows the structure of the ontology. The IGONTO ontology is at the top level. This represents the final ontology consisting of the Jurisdiction ontology and the Enterprise ontology. The Jurisdiction is responsible for covering the legal aspect of information governance, whereas the Enterprise ontology implements the organization, structure, and relationships within an enterprise. The Enterprise ontology in turn consists of the Organization, Lifecycle, Information, and Implementation ontologies.
Merging Organization, Lifecycle, Information, and Implementation into the Enterprise ontology first, before directly merging it into IGONTO, is consistent with the concept of reuse. This intermediate step allows the Enterprise concept (or parts of it) to be potentially reused in a different context outside of IGONTO.
Each of these ontologies has its concepts, relations, attributes, and schema. By isolating individual sub-areas, they can be developed separately from other ontologies. This can have several advantages:

1. **Maintainability:** Each submodule can be developed, updated, and maintained independently of the others.
2. **Clarity:** The clear delimitation of individual subdomains increases the clarity of the entire system. This ensures proper understanding by users and developers.
3. **Reusability:** Small ontologies, like small software, are easy to reuse and embed in personal systems.
4. **Parallelism:** By isolating individual domains, it is easier to work in parallel on the overall system.

The division of the ontology also has some disadvantages:

1. **Consistency:** It makes maintaining a consistent ontology difficult. Changes within a sub-ontology can have an unwanted impact on the overall system. Therefore, it is important to develop a validation scheme for each ontology and the overall system to ensure quality and consistency. Therefore, the explanation of the framework used for validation is covered in Section 3.3, and its precise implementation regarding our ontology structure is described in Section 5.3.
2. **Integration:** Maintaining sub-ontologies requires consistent integration into the overall system. Similar to the problem of handling different branches in agile development (GitHub, etc.), it is not recommended to develop one ontology in isolation for a long time.

To mitigate such negative aspects, a process has been designed and outlined in 5.4.3. It can serve as a template for the further development of the prototype and as an initial reference point that can be modified if necessary. However, the key aspects presented there should be preserved.

3.1.2 Object Property Structure

Object properties are relationships between individuals linked to a domain and a range. The domain defines the set of subjects and the range defines the sub-set of objects that are connected. These relationships should be clear, but not too complicated. Object properties should automatically

provide the reader insight into the object that is connected to it through its object property characteristics. Similar to reading relationship names in class relationships of common UML class diagrams, insight into the meaning and context of the relationship should be automatically available, increasing readability and maintainability.

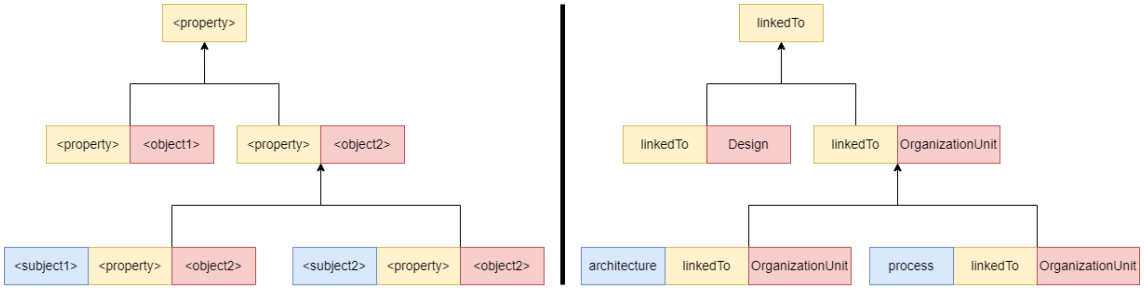


Figure 3.2: Object property structure.

In ontologies, object properties are implemented in a hierarchical structure. Figure 3.2 describes this concept outlining the implementation on the left side and is accompanied by an explicit example from IGONTO on the right side. Every object property has a base <*property*> that acts as the super property. In the given example, the super property is *linkedTo*. This super property is not used by any instance directly but acts as the top-level parent that aggregates all lower-level properties beneath it.
The next level and the first possible relationships are direct children of the super property, built from the base property (*linkedTo*) and the range of the class (<*object1*> / *Design*, *object2* / *OrganizationUnit*, ...), which should be connected by that object property. If the object class is only connected by exactly one subject class with the base property, there is no need to further detail the structure (see *linkedToDesign*). However, if multiple subject classes (<*subject1*> / *architecture*, *subject2* / *process*, ...) connect with the same base property to the same object class, additional relationships are required.
Without advanced differentiation, both classes would have the same base relationship, leading to possible inconsistencies or incorrect interpretations within the ontology. For instance, if *architecture* and *process* have no other unique and distinguishing characteristics and use the same relationship, the reasoner would interpret them as equivalent classes. On the one hand, this is semantically wrong because both classes do not describe the same concept. On the other hand, if the ontology explicitly rules that those classes must be disjointed, the reasoner would throw an error, indicating inconsistency.

Figure 3.3 visualizes the potential inconsistencies and presents the reasoner's explanation. Line 1 reveals that *Architecture* and *Process* are disjointed from each other. Lines 2 and 4 state that both classes are registered as the domain of the property, and line 3 reveals the connection of the process instance to the object. The reasoner interprets both instances as members of both classes based on these axioms, which contradicts the disjoint argument and leads to inconsistency. Removing the disjoint argument would resolve this error, but this would lead to the first problem: the reasoner equates both classes, which is consistent but semantically wrong.

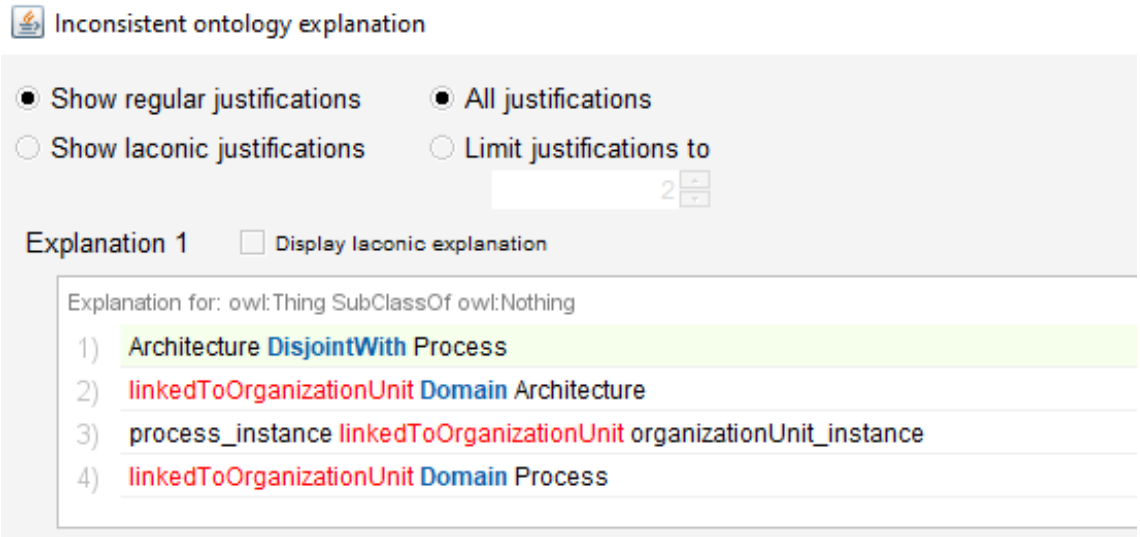


Figure 3.3: Example of reasoner inconsistency.

In addition to solving consistencies, this structure enhances readability and maintainability. Instances are always connected via the bottom level of the object property hierarchy. Therefore, in the example shown in 3.2, subject instances would be connected with *linkedToDesign, architectureLinkedToOrganizationUnit, and processLinkedToOrganizationUnit*. The object for every relationship becomes clear, namely *Design* and *OrganizationUnit*, which is obtained by the object property name itself. Without these suffixes, every instance would be connected via *linkedTo*, and the type of the connected object may be unknown. In the case of *linkedToOrganizationUnit*, the reader is also aware of the subject type, namely *architecture* and *process*. If desired, the subject could also be included in *linkedToDesign*, but to reduce unwanted overhead in reasoning and querying, the subject is only included if it is necessary to prevent inconsistency.
Likewise, if reasoning is enabled, the query can also be implemented only by using the super property *linkedTo*. Remembering the exact object property is unnecessary because all properties are subproperties of *linkedTo*, and the reasoner iterates through the whole property hierarchy. Asking what subjects are *linkedTo* what objects would lead to the same set of objects, like using the exact subproperties.

Organizing object properties in such a hierarchical way not only increases consistency but also the readability and maintainability of the ontology and queries. All instances are systematically connected, which enhances clarity and effective knowledge representation.

3.1.3 Datatype Property Structure

Datatype properties can be referred to by the more common name “attribute”. They describe classes and instances more explicitly and represent the most fine-grained description within an ontology. Data properties can be asserted in a domain and range similar to object properties. However, the difference arises from the fact that the range is not a set of classes but a set of datatypes. These datatypes range from simple ones, such as integers and strings, to more complex ones, such as

"datetime" and "hexbyte", as described by the XML schema [Wor12].
We imported datatype properties from the Data Catalog Vocabulary (DCAT) ontology [Wor21], which provides some data-related definitions. In addition, two data properties, *name* and *description*, were implemented on the top level of every sub-ontology, because these represent basic additional knowledge about classes and instances. *Name* can be considered a synonym for *rdfs:label*, which describes the id of a class or instance in human-readable terms. The *description* should detail the concepts in textual form.

3.2 Reasoning

Reasoning, in the context of ontology, describes the processes of inferring knowledge that is not directly stated within the ontology. Because reasoning plays a crucial role in ontologies, this chapter explores several reasoning methods and identifies the most suitable for IGONTO. Section 3.2.1 defines reasoning, the tasks it fulfills, and the parameters to consider from the perspective of a knowledge graph to select an appropriate reasoning method. Section 3.2.2 introduces several reasoning methods and their characteristics. Section 3.2.3 delves into the characterization of our IGONTO's knowledge graph. Finally, Section 3.2.4 deals with the proper selection of reasoning for our use case.

3.2.1 Ontology Reasoning and Its Parameters

Reasoning forms a crucial aspect of ontologies, as it enables the extraction of meaningful information and the inference of implied knowledge based on the existing ontology. Ontological reasoning involves applying logical rules and algorithms to the ontology to derive new facts or validate existing ones [Tra08].
One of the main reasons for using reasoning in ontologies is to ensure consistency. Reasoning can detect and resolve ontological inconsistencies, thus ensuring that the knowledge representation is coherent and free of contradiction [KBM+11]. Additionally, reasoning validates correctness by checking for logical inconsistencies within an ontology [SBW19].
Moreover, reasoning enables ontology alignment and, therefore, facilitates interoperability and knowledge sharing by finding correspondences among different ontologies [SGSH16].
Ontologies can integrate knowledge from multiple sources by using reasoning algorithms, which align different ontologies based on their semantic and logical structure [QZC09].

Reasoning has numerous potential applications, and there exist multiple reasoning methods. However, method applicability can vary based on the context and ontology requirements. To determine the best method to use, one must first consider knowledge graph characteristics that influence the choice of the method. The following briefly explains the key aspects of the characteristics of a knowledge graph that are important in the choice of a suitable reasoning method.

1. One of the most critical parameters is the *size* of a knowledge graph, as it impacts the efficiency and scalability of reasoning. On the one hand, large knowledge graphs naturally require computationally intensive reasoning methods to handle the complexity and volume of data. On the other hand, smaller knowledge graphs are potentially easier to reason over using simpler reasoning methods [CJX20].
2. The *accuracy* of inferences is also crucial. This includes the required accuracy of making correct inferences and avoiding incorrect ones. Thus, the magnitude of the consequences in the case of wrong inferences has to be considered. Accuracy is also used in [TZW+22] to characterize reasoning methods.
3. Another aspect is the *frequency of data changes* in the knowledge graph and its structure. The more frequently and extensively the knowledge graph changes, the more dynamic the reasoning method should be. The need for dynamic reasoning methods to handle frequent data changes is highlighted by [CJX20], and the approach involves different techniques for reasoning over knowledge graphs.
4. Additionally, how *explicit and transparent the explanations* of inferences have to be is considered [TZW+22]. Are the resulting inferences enough, or are justifications and explanations needed for how these inferences were generated?
5. The last aspect builds *performance*. Naturally, reasoning methods should align with the performance requirements needed for the knowledge graph. A general decision is required regarding whether real-time results are needed for one's use case, or whether for instance, faster results are only for usability reasons and not because the data are required in real-time.

In conclusion, these parameters describe the most critical reasoning characterizations and should be considered when selecting the reasoning method. Some more specialized use cases may require more specialized parameters. For our case, this characterization is sufficient and it includes the most relevant ones.

3.2.2 Reasoning Methods

This section introduces the most well-known reasoning methods. We highlight the work of Tian et. al. [TZW+22], which offers a comprehensive overview of various approaches to reasoning. For each method explained, we also present the pros and cons to facilitate an evaluation and to determine which method is best suited for our knowledge graph implementation.

Reasoning Based on Logic Rules

Reasoning based on logic is further divided into two subcategories. The first subcategory is reasoning based on First-Order Logic (FOL). FOL provides a formal language for representing knowledge and performing logical inferences. Moreover, it serves as the basis for logical reasoning, enabling the derivation of new facts from existing ones [Fit90]. FOL reasoning is represented by rules, which are manually defined by experts. One example is the representation by Horn clauses, which resembles natural human language and increases the readability of the rules. A Horn clause is a disjunction of literals or constraints that imply certain knowledge when every constraint within

the clause is fulfilled. Its specific use for IGONTO is further explained in Section 4.6. Description Logic (DL) implements fragments of FOL to enhance usability and decidability at the expense of expressivity, as FOL is rather complex and lacks scalability. In this method, so-called TBoxes and ABoxes are defined. TBoxes are terminological axioms, which describe concepts and relations, and ABoxes are assertional sets, which consist of a variance of TBoxes and thus describe whole concepts. How DL differs from FOL is discussed in depth in Section 3.2.5.
Another logic-based reasoning method is reasoning based on statistics. The goal of this method is not to manually define rules but to automatically extract and apply the said rules with the help of machine learning approaches. Inductive Logic Programming (ILP) and Association Rule Mining (ARM) are distinct. While ILP aims to learn abstract rules from examples, ARM's goal is to detect and extract patterns in large amounts of data and identify association rules between different attributes. ILP achieves its goal by applying machine learning techniques and logic programming to automatically detect and model complex relationships in data. ARM, on the other hand, achieves this goal by using algorithms to find the association rules that describe how different attributes are connected. ILP dispenses manually defined rules and offers good reasoning via small-scale knowledge graphs. ARM offers high-confidence rules and is faster, so it can handle more complex and larger knowledge graphs.
Reasoning based on the graph structure refers to using the graph structure as a feature to draw conclusions. Specifically, the relationships between entities in the graph can be used to discover and infer new facts or relationships. A typical feature in a knowledge graph is the paths between entities, which play an important role in knowledge processing. This method is divided into global structured-based and local structured-based models. In the global structured-based model, the paths of the entire knowledge graph are used as a feature, which makes this method effective but also computationally expensive. On the other hand, the local structure-based model focuses on finer granularity by only examining the narrow and local graph structure of the relationship under investigation. Therefore, its computational costs are lower.

Knowledge Graph Reasoning Based on Representation Learning

The basic idea of representation learning is to convert complex data structures into vectors. There are several approaches, such as the tensor decomposition approach, distance model, and semantic matching model. The tensor decomposition approach uses the possibility of modeling interactions between entities and relations using a three-way tenor (entity-relation-entity). Relational Single-Channel Analysis of Learning (RESCAL) is an approach for tensor decomposition that reflects similarities within the graph structure by solving the simple tensor decomposition problem.
The distance model aims to model the distance between the embeddings of the subject and object entities in the knowledge graph. By minimizing the transformation error, this type of model learns low-dimensional embeddings of all the entities and relation types in the knowledge graph.
The semantic matching model uses a scoring function to determine similarity. With the scoring function, hidden semantics between entities and relations are matched and thus the validity of relation triples among these entities are measured. Because the parameters are numerous, this method suffers from a high complexity.

Knowledge Graph Reasoning Based on Neural Network

Knowledge graph reasoning based on neural networks reasons new entity-relationship representations by using neural networks in a knowledge graph. This approach allows complex and multi-hop relational reasoning, for which traditional logic-based methods are ineffective. Knowledge graph reasoning based on a neural network is effective for link prediction, entity classification, and question answering. However, as Nikolov and d'Aquin [Nd20] point out in their paper, the black-box nature of AI models leads to a lack of interpretability and does not explicitly explain the reasoning results.

3.2.3 IGONTO Characterization

This section discusses the characterization of the knowledge graph in terms of the parameters mentioned in Section 3.2.1 to choose the suitable reasoning method for our use case.

Parameter: Size

The first parameter pertains to the size of the knowledge graph. To classify our knowledge graph, what distinguishes a knowledge graph as small or large scale should be determined. However, defining such a threshold is challenging and cannot be categorically stated due to the contribution of several factors, including the number of classes, individuals, relations, and attributes, as well as the degree of interconnection among individual instances (i.e., the quantity of relationships and attributes each instance possesses). While it is feasible to query the precise metrics of our knowledge graph with the assistance of Protegé, as illustrated in Figure 3.4, no official reference metric designates a knowledge graph as "large" based on a specific number of instances.

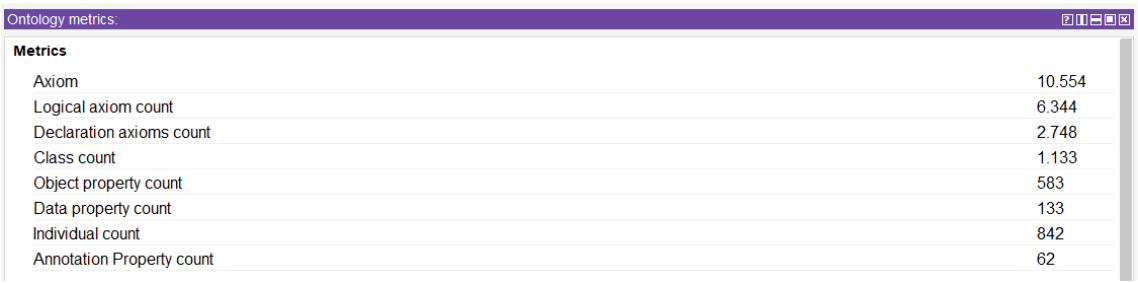


Figure 3.4: Knowledge graph metrics.

However, the storage space needed to save the ontology could be an approximate indication and metric of the graph's size. It is also feasible to transform a relational database into a knowledge graph [VMT13]. Moreover, the size of the knowledge graph can be compared with that of a relational database. Currently, the required storage space for our knowledge graph is approximately 19 MB. A database this size is considered rather small. Not many additional classes are going to be implemented, specifically not in a magnitude that alters the dimensions in which we currently reside. Nevertheless, IGONTO does not yet cover all domain knowledge and could increase in

size regarding its instances and relationships. In case these changes would alter the magnitude of IGONTO, a new analysis for reasoning requirements would be necessary. However, we analyze the current requirements for our prototype, and therefore, the knowledge graph is unequivocally small-scale.

Parameter: Accuracy

The inference accuracy is crucial to the use case. Every inference must be correct, and no inference should be excessive. The two case distinctions clarify the need for high accuracy:

1. If an inference is not derived when it should be, the consequences could be serious. For instance, a company is told that it does not require certain tools or processes because the knowledge is not inferred. This could violate the obligations of the GDPR and result in a large fine. For example, the hotel chain "Marriott Hotel" was sued and fined over 18.4 million euros because certain safety standards were not implemented [Hei21]. Therefore, if an inference related to this standard is missing, it would not be a satisfactory implementation.
2. In the other case, where an inference is derived when it is not necessary, the consequences may be an economic problem rather than a legal one. For instance, consider the inference state where a certain process or organization unit is needed even though it is not actually required. The company would have to provide resources to implement the unnecessary processes, which would cause economic damage.

In both cases, inaccurate inferences would be very harmful. We cannot afford "false positives" or "false negatives". Consequently, a high degree of accuracy in inferences is needed to guarantee sufficient GDPR compliance.

Parameter: Frequency of Data Changes

Here, how frequently new instances, relationships, or attributes are added and how often specific instances are altered in such a manner that a different model is formed must be clarified. A good indicator is the frequency of changes in the General Data Protection Regulation. The last update to the GDPR occurred on May 25, 2018. ISO standards, which are designed to provide years of stability and continuity, do not undergo frequent changes. Even if design or technology aspects are changed more frequently, the main complexity of IGONTO remains in inferring regulatory requirements for enterprises. Therefore, data within the mature IGONTO knowledge graph are unlikely to experience frequent alterations.

Parameter: Explicit Explanations of Inferences

In our use case, an explanation of inference is useful, if not necessary. For instance, if a company is told by querying the knowledge graph that it has to implement a certain tool or process, there is certainly interest in finding out why this is necessary in the first place. Each additional process costs money and resources, making it crucial to comprehend the reasons behind the inference. Accordingly, an explicit justification is necessary.

Parameter: Performance

Good performance is desirable. Here, however, a distinction must be made between whether real-time results are needed or whether it would be manageable to wait a few seconds. Our use case does not require results in real time. How often queries are executed and how often certain queries are repeated is not crucial but the quality and accuracy of the answers are.
In theory, a company may only need to execute each query once, as the final result remains unchanged. Repeated queries will yield the same processes and outcomes. In addition, ontologies can be materialized in advance. The process of materialization infers all reasoning knowledge and saves the inferred triples in a second, materialized version. Afterward, the queries are performed over the materialized version, whereas the original version remains available for further development processes. This way, no dynamic reasoning is required in query time, and therefore, reasoning performance does not play a crucial role. In this respect, performance in our knowledge graph is not a primary consideration and should not be the highest priority. Instead, fulfilling the other parameters would be much more important.

3.2.4 Reasoning Choice

Considering the properties of the knowledge graph, what is needed above all is a method for small-scale knowledge graphs with high accuracy. The rules and inferences must be comprehensible and explainable. Although a dynamic method would also be desirable, the dynamic methods presented here have poor accuracy, lack justification for the inferences, and are more suitable for large-scale knowledge graphs. Therefore, a reasoning method with logic-based reasoning is appropriate. Because OWL 2 uses $\mathcal{SROIQ}$ as the foundation of its description logic, Section 3.2.5 briefly describes the use of the description logic employed in our ontology.
Our approach uses reasoning with description logic for generating inferences, extended by SWRL rules. With the SWRL rules, complicated chained inferences can be implemented with high accuracy. Furthermore, every ontology has its own SHACL schema to guarantee consistency and validate our ontology.

3.2.5 Description Logic (DL)

This section describes the concept of description logic. The reason for its general use is explained in Section 3.2.5. Section 3.2.6 focuses on the basic DL and its theoretical concepts, whereas Section 3.2.7 explains the different DL features and what DL is used for in our ontology.

First-Order Logic (FOL) versus Description Logic (DL)

First-Order Logic (FOL) is a mathematical logic used to describe statements about objects in a specific domain. FOL is highly expressive but semi-decidable, which makes using it in computational contexts impractical [Pur06]. In addition, FOL is unintuitive for complex ontology domains because of its mathematical origin. Further, proving the consistency of FOL in ontologies is difficult due to scalability and complexity bottlenecks [SH20].
To find a compromise of expressiveness and scalability, certain fragments of FOL were simplified

with the help of constructors. The set of these constructors builds the DL and also its name. With these simpler constructors, more complex logical descriptions can be made. This explains why DL is not one "language" or one "logic" but a family of multiple DLs, involving a different set of constructors. The set of implemented constructors distinguishes the DLs from each other. Most DLs are decidable, easier to implement, and sufficiently expressive; therefore, they are often used in ontologies.

3.2.6 $\mathcal{ALC}$ - The Smallest Description Logic

In general, a DL consists of two or three architecture elements [DL96]:

1. **TBox:**
   The TBox includes terminological knowledge. Here, knowledge about the concepts of the domain is implemented, for instance, description logic about classes.
2. **ABox:**
   The ABox includes assertional knowledge. Here, knowledge about individuals is defined, for instance, by the definition of an individual type or the relationship to another individual.
3. **RBox:** The RBox contains role-centric knowledge. While TBoxes and ABoxes are included in every DL, RBoxes are optional. Basic DL, such as $\mathcal{ALC}$, does not include RBoxes. However, if DL needs to be more expressive, RBoxes are included to provide knowledge about roles, including relationships. For instance, a hierarchy of relationships can be implemented with RBoxes.

$\mathcal{ALC}$−Definition

To grasp the logic used in our ontology, we first define the building blocks of the base DL $\mathcal{ALC}$ which is short for *Attribute Language with Complement*. The following definitions of $\mathcal{ALC}$ are taken from [dFE05] and [HKS06] but changed in certain vocabulary to ensure they align with further definitions of the DL $\mathcal{SROIQ}$, as described in Section 3.2.7.
The basic building blocks of $\mathcal{ALC}$ are classes, roles/properties, and individuals. Even though roles are included, they are only used to connect concepts and individuals.

Definition 3.2.1 ($\mathcal{ALC}$ Atomic Types)
*Let $N_C = \{C, D, ...\}$ be a set of concept names interpreted as a subset from individuals $I = \{a, b, c...\}$ of a domain Δ. $N_R = \{R, S, ...\}$ is the set role interpreted as binary relations. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an **Interpretation**, where $\Delta^{\mathcal{I}}$ is the **domain** of the interpretation and $\cdot^{\mathcal{I}}$ the **interpretation function**, which maps the following:*

1. *Individual names $x \in I$ to domain elements $x^{\mathcal{I}} \in \Delta^{\mathcal{I}}$*
2. *Class names $C \in N_C$ to a set of domain elements $C^{\mathcal{I}} \in \Delta^{\mathcal{I}}$*
3. *Role names $R \in N_R$ to a set of domain element pairs $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} x \Delta^{\mathcal{I}}$*
4. *The top concept $\top$, where $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$*

5. *The bottom concept is* $\bot$*, where* $\bot^{\mathcal{I}} = \emptyset$

Definition 3.2.1 introduces the basic types of $\mathcal{ALC}$. The top concept $\top$ and the bottom concept $\bot$ are two special concepts, where every defined class is automatically a subclass of $\top$, while simultaneously $\bot$ is a subclass of every defined class.

Definition 3.2.2 ($\mathcal{ALC}$ Constructors)
Given the interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$*, concepts* C, D, E *and the role* R*.*
Let $P_{\mathcal{ALC}} = \{\top^{\mathcal{I}} | \bot^{\mathcal{I}} | (\neg C)^{\mathcal{I}} | (C \sqcap D)^{\mathcal{I}} | (C \sqcup D)^{\mathcal{I}} | (\exists R.C)^{\mathcal{I}} | (\forall R, C)^{\mathcal{I}}\}$ *be the set of production rules for creating* ***complex concepts****, where* $\top^{\mathcal{I}}$ *and* $\bot^{\mathcal{I}}$ *are interpreted as in Definition 3.2.1 and*

$$
\begin{aligned}
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\exists R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \exists y \in \Delta^{\mathcal{I}} ((x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}})\} \\
(\forall R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \forall y \in \Delta^{\mathcal{I}} ((x, y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}})\}
\end{aligned}
$$

Definition 3.2.3 ($\mathcal{ALC}$ TBox assertions)
Let $TBox_{\mathcal{ALC}} = \{C \sqsubseteq D, C \equiv D\}$ *be the set of concept assertions contained in a TBox. An interpretation* $\mathcal{I}$ *satisfies a TBox* $\mathcal{T}$ *(*$\mathcal{I} \models \mathcal{T}$*), where:*

$\mathcal{I} \models (C \sqsubseteq D)$ *holds, iff* $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ *for every interpretation* $\mathcal{I}$
$\mathcal{I} \models (C \equiv D)$ *holds, iff* $(C^{\mathcal{I}} \subseteq D^{\mathcal{I}}) \wedge (D^{\mathcal{I}} \subseteq C^{\mathcal{I}})$ *for every interpretation* $\mathcal{I}$

Definition 3.2.4 ($\mathcal{ALC}$ ABox assertions)
Let $ABox_{\mathcal{ALC}} = \{(a : C), (((a, b) : R)\}$ *be the set of concept assertions contained in an ABox. An interpretation* $\mathcal{I}$ *satisfies an ABox* $\mathcal{A}$ *(*$\mathcal{I} \models \mathcal{A}$*), where*

$\mathcal{I} \models (a : C)$ *holds, iff* $a^{\mathcal{I}} \in C^{\mathcal{I}}$
$\mathcal{I} \models (((a, b) : R)$ *holds, iff* $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

Considering these concepts, $\mathcal{ALC}$ is the smallest deductively complete DL. It forms the foundation for other DLs and only implements FOL fragments that are linked to TBoxes and ABoxes. Even though the concepts and constructors of $\mathcal{ALC}$ are already expressive, the lack of number restrictions and constructors regarding roles and properties is not sufficiently expressive for ontologies. Therefore, further DLs with better expressiveness were created, but every one of them includes $\mathcal{ALC}$ and its basic concepts.

3.2.7 $\mathcal{SROIQ}$ - Description Logic

What FOL fragments are implemented in a DL are represented by the name of the DL itself. Every letter represents an included fragment. Table 3.1 lists the existing set of different FOL fragments implemented in different DLs.

Symbol	Description
$\mathcal{ALC}$	Attribute Language with Complement
$\mathcal{S}$	ALC + Transitivity of Roles
$\mathcal{H}$	Role Hierarchies
$\mathcal{O}$	Nominals
$\mathcal{I}$	Inverse Roles
$\mathcal{N}$	Number Restrictions
$\mathcal{Q}$	Qualified Number Restrictions
$(\mathcal{D})$	Datatypes
$\mathcal{F}$	Functional Roles
$\mathcal{R}$	Role Constructors

Table 3.1: Description Logic features.

Because the description logic in OWL 2 is based on $\mathcal{SROIQ}(\mathcal{D})$ [W3C12], this description logic is detailed. The following definitions and explanations are taken from [HKS06] and aligned to fit in with the other definitions of $\mathcal{ALC}$, similar to Section 3.2.6
The implemented DL features are outlined in Table 3.1 and in the name of $\mathcal{SROIQ}(\mathcal{D})$ itself. $\mathcal{SROIQ}$ is first presented in detail, before including the Datatypes (D). The largest addition in $\mathcal{SROIQ}$, compared to $\mathcal{ALC}$, is the implementation of the RBox with its axioms and constructors. Definition 3.2.5 defines the inverse Roles as follows:

Definition 3.2.5 ($\mathcal{SHOIQ}$ - Inverse Roles)
Given the interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, the set of individuals $I = \{a, b, c...\}$ and the role $R \subseteq \Delta^{\mathcal{I}} x \Delta^{\mathcal{I}}$ from Definition 3.2.1, we define the universal role $U = \Delta^{\mathcal{I}} x \Delta^{\mathcal{I}}$. For each role $R \in \boldsymbol{R}$, we define the inverse role interpreted as:

$$(R^{-})^{I} = \{(y, x) | (x, y) \in R^{I}\}$$

Definition 3.2.6 is cited from [HKS06] and introduces the role hierarchy $\mathcal{R}_h$ and its regularity. Regularity prevents cyclic dependencies, which would lead to undecidability [HS05].

Definition 3.2.6 (((Regular) Role Inclusion Axioms) [HKS06])
*"Let $\prec$ be a regular order on roles. A **role inclusion axiom** (RIA for short) is an expression of the form $w \dot{\sqsubseteq} R$, where w is a finite string of roles not including the universal role U, and $R \neq U$ is a role name. **A role hierarchy** $\mathcal{R}_h$ is a finite set of RIAs. An interpretation I **satisfies** a role inclusion axiom $w \dot{\sqsubseteq} R$, written $I \models w \dot{\sqsubseteq} R$, if $w^{I} \subseteq R^{I}$. An interpretation is a **model** of a role hierarchy R_h if it satisfies all RIAs in R_h, written $I \models R_h$.*
*A RIA $w \dot{\sqsubseteq} R$ is $\prec$-**regular** if R is a role name, and*

1. $w = RR$,
2. $w = R^-$,
3. $w = S_1 \dots S_n$ *and* $S_i \prec R$, *for all* $1 \leq i \leq n$,
4. $w = RS_1 \dots S_n$ *and* $S_i \prec R$, *for all* $1 \leq i \leq n$,
5. $w = S_1 \dots S_n R$ *and* $S_i \prec R$, *for all* $1 \leq i \leq n$.

Finally, a role hierarchy R_h is regular if there exists a regular order $\prec$ such that each RIA in R_h is $\prec$-regular."

Similar to the TBox and ABox assertions, some assertions can be defined for roles. These include symmetry (*Sym(R)*), asymmetry (*Asy(R)*), transitivity (*Tra(R)*), reflexivity (*Ref(R)*), irreflexivity (*Irr(R)*), and disjointness (*Dis(D,S)*) of roles. These assertions, including the role hierarchy from Definition 3.2.6, build the role box defined in Definition 3.2.7.

Definition 3.2.7 (Role Box)
*Given the regular role hierarchy $\mathcal{R}_h$, the universal role U and individuals $x, y, z \in \Delta^{\mathcal{I}}$. Let $\mathcal{R}_a = \{Sym(R), Asy(R), Tra(R), Ref(R), Irr(R), Dis(R,S)\}$ be the set of **role assertions** for roles $R, S \neq U$. A **role box** is a set $\mathcal{R} = \mathcal{R}_h \cup \mathcal{R}_a$. An interpretation **models** $\mathcal{R}$ if $\mathcal{I} \models \mathcal{R}_h$ and $\mathcal{I} \models \phi$ for all role assertions $\phi \in \mathcal{R}_a$, where the elements of $\mathcal{R}_a$ are defined as*

$$
\begin{aligned}
Sym(R) &= \{(x,y) \in R^I \rightarrow (y,x) \in R^I\} \\
Asy(R) &= \{(x,y) \in R^I \rightarrow (y,x) \notin R^I\} \\
Tra(R) &= \{((x,y) \in R^I \wedge (y,z) \in R^I) \rightarrow (x,z) \in R^I\} \\
Ref(R) &= \{(x,x) | x \in \Delta^I\} \subseteq R^I \\
Irr(R) &= \{(x,x) | x \in \Delta^I\} \cap R^I = \emptyset \\
Dis(R,S) &= R^I \cap S^I = \emptyset
\end{aligned}
$$

In addition to the Role box and its assertions, some extensions regarding the constructors within $\mathcal{SROIQ}$ are implemented. Definition 3.2.8 presents the constructors in $\mathcal{SROIQ}$

Definition 3.2.8 ($\mathcal{SROIQ}$ - Production rules)
*Given the concept names, roles, and nominals interpreted as in Definition 3.2.1 and $P_{\mathcal{ALC}}$ as in Definition 3.2.3, let $P_{(\mathcal{SROIQ}/\mathcal{ALC})} = \{(\exists R.Self)^{\mathcal{I}}, (\geqslant nR.C)^{\mathcal{I}}, (\leqslant nR.C)^{\mathcal{I}}\}$ be the set of additional production rules, defined by the following equations, where **#M** denotes the **cardinality** of a set M:*

$$
\begin{aligned}
(\exists R.Self)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | (x,y) \in R^I\} \\
(\geqslant nR.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \#y.(x,y) \in R^I \rightarrow y \in C^{\mathcal{I}}) \geqslant n\} \\
(\leqslant nR.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \#y.(x,y) \in R^I \rightarrow y \in C^{\mathcal{I}}) \leqslant n\}
\end{aligned}
$$

$P_{\mathcal{SROIQ}} = P_{\mathcal{ALC}} \cup P_{(\mathcal{SROIQ}/\mathcal{ALC})}$ *completes the production rule set for complex concepts in* $\mathcal{SROIQ}$.

The additional constructors in Definition 3.2.8 enable the implementation of quantifiers within the DL. The assertion for the TBox in $\mathcal{SROIQ}$ is equivalent to the assertions presented in Definition 3.2.3. The assertions for the ABox are expanded by two, as defined in 3.2.9

Definition 3.2.9 ($\mathcal{SROIQ}$ - Production rules)
Given the concept names, roles and nominals interpreted as in Definition 3.2.1 and $ABox_{\mathcal{ALC}}$ *as in Definition 3.2.4, let* $ABox_{(\mathcal{SROIQ}/\mathcal{ALC})} = \{(a \doteq b), (a, b) : \neg R\}$ *be a set of additional assertions contained in an ABox. An interpretation* $\mathcal{I}$ *satisfies an ABox* $\mathcal{A}$ *(*$\mathcal{I} \models \mathcal{A}$*), where:*

$\mathcal{I} \models (a \doteq b)$ *holds, iff* $a^I \neq b^I$
$\mathcal{I} \models ((a, b) : \neg R)$ *holds, iff* $(a^I, b^I) \notin R^I$

$ABox_{\mathcal{SROIQ}} = ABox_{\mathcal{ALC}} \cup ABox_{(\mathcal{SROIQ}/\mathcal{ALC})}$ *completes the assertion set for the ABox in* $\mathcal{SROIQ}$.

(D) is also included as a DL fragment in OWL 2, allowing the implementation of datatypes. However, because datatypes do not play a crucial role in inferences in our ontology, we will not define them. The definitions for datatypes can be retrieved from [W3C12]. The constructors and assertions of $\mathcal{SROIQ}$ are defined, and Table 3.2 lists how these axioms and constructors are implemented in OWL 2.

Because these are only the main keywords, the use of these keywords and how they are implemented are outlined in example code 3.2.7. Here, the *Obligation* class from the Jurisdiction ontology has the existential constructor $\exists R.C$ implemented.
First, *rdf:type owl:Restriction* initiates the description logic and contains the constraints described by it. The term *restriction* in this context is also interesting but rational. Because ontologies act on the open world assumption, indirectly implemented knowledge can not be interpreted as negation. With DL, we further restrict the domain by defining the constraints the concepts have to comply with. Here, the constraint begins with the declaration of the object property that the rule is applied to (*owl:onProperty jurisdiction:programRequiredForRegulation*). Afterward, the keyword *owl:someValuesOn* is used on the class *jurisdiction:GDPR*. Therefore, this constraint restricts the instances from the class *Obligation* that need at least one relationship *obligationInvolvesArticle* connected with an instance from the class *GDPRArticles*. Other constructors are similarly implemented using their corresponding OWL keywords.

DL Syntax	OWL 2 Keyword
$\top$	owl:Thing
$\bot$	owl:Nothing
$C \sqcap D$	owl:intersectionOf
$C \sqcup D$	owl:unionOf
$\neg C$	owl:complementOf
$\exists R.C$	owl:someValuesFrom
$\forall R.C$	owl:allValuesFrom
$C \sqsubseteq D$	rdfs:subClassOf
$C \equiv D$	owl:equivalentClass
$a : C$	rdf:type
$(a, b) : R$	Object Property Assertion
(R^{-})	owl:inverseOf
$Sym(R)$	owl:SymmetricProperty
$Asy(R)$	owl:AsymmetricProperty
$Tra(R)$	owl:TransitiveProperty
$Ref(R)$	owl:ReflexiveProperty
$Irr(R)$	owl:IrreflexiveProperty
$Dis(R, S)$	owl:propertyDisjointWith
$(\exists R.Self)$	owl:hasSelf
$(\geqslant nR.C)$	owl:minCardinality
$(\leqslant nR.C)$	owl:maxCardinality
$(a \dot{\neq} b)$	owl:differentFrom
$((a, b) : \neg R)$	owl:NegativePropertyAssertion

Table 3.2: Mapping of DL Syntax to OWL 2 keywords.

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sh:    <http://www.w3.org/ns/shacl#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
@prefix jurisdiction:   <http://www.semanticweb.org/igonto/jurisdiction#> .

jurisdiction:Obligation rdf:type owl:Class ;
                        rdfs:subClassOf jurisdiction:IgontoTopJurisdictionDomain ,
                            [ rdf:type owl:Restriction ;
                              owl:onProperty jurisdiction:obligationInvolvesArticle ;
                              owl:someValuesFrom jurisdiction:GDPRArticles
                            ] .
```

Listing 3.1: Description Logic (DL) OWL implementation example.

3.3 Shapes Constraint Language (SHACL)

3.3.1 IGONTO Validation

Shapes Constraint Language (SHACL) is a language that focuses on validating RDF graphs. Typically, SHACL is implemented in a separate document, which is often referred to as a SHACL *schema*. The schema consists of *shapes* on which the RDF graph is validated. Each shape is a tuple ⟨s, t, d ⟩, where *s* describes the unique *id* of the shape, *t* the *target definition*, and *d* the *set of constraints* for the target. If the graph meets the constraints, it is considered *valid*. In the event that the RDF graph violates a constraint in the schema, the validation report throws an error with an explanation of which constraint failed and why. Because the schema is an RDF graph itself, it does not need additional infrastructure and can even be embedded in the ontology itself [PK21]. However, the schema is often separately stored in a file to ensure maintainability and usability.
Figure 3.5 shows a brief example of a SHACL shape in our jurisdiction ontology. Following the first triple, which is the definition of the shape including its name, the target of the shape is defined. In our example, the target is a whole class, and therefore all instances from this class (here: *EU*) are considered a *focus node*. Four types of possible target definitions exist.

1. *sh:targetNode*: Specifies an individual instance as the target node of this shape and its constraints.
2. *sh:targetClass*: Specifies a class, and therefore all of its instances as the target node of this shape and its constraints.
3. *sh:targetSubjectsOf*: Specifies a property, and therefore all constraints of the shape apply to all the subjects of that property.
4. *sh:targetObjectsOf*: Specifies a property, and therefore all constraints of the shape apply to for all the objects of that property.

After specifying the instances where this shape is responsible, the set of constraints is then presented. Recursively, these constraints can be implemented as their own property shapes with their own constraints, either by defining a new shape (*hasRegulationShape*) or by implementing these constraints directly. Although constraints can be implemented directly, defining new shapes is recommended for traceability reasons in the validation report if invalid constraints occur. The targets of these constraints are called the “target nodes”. Here, the target nodes are all instances from *jurisdiction:Regulation* and the attribute *rdfs:label*. The reference to a property in the original knowledge graph is defined in *sh:path*. The following lines represent the constraints of the said property. For example, the constraints for the property *rdfs:label* are that the datatype of the label should be a string, and the focus node must have a minimum of one label.

Figure 3.6 shows an example of a violation report. Here, both constraints failed. The report describes which constraint component failed (*sh:sourceConstraintComponent*), the instance (*sh:focusNode*), and the property (*sh:resultPath*) that violated the constraint, along with a message explaining why it failed. This message can also be redefined for each shape and purpose. It also illustrates why it is useful to define new property shapes for each property. In the first violation, the *sh:sourceShape* indicates in which constraint and shape the error occurred. On the second violation, a generic identifier for that property shape is generated because, in our schema, the property shape is anonymous and undefined.

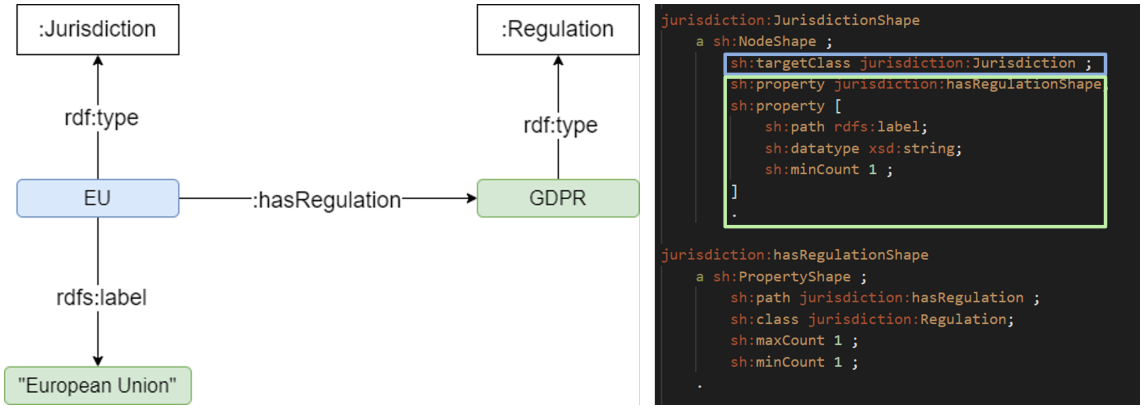


Figure 3.5: SHACL example.

```
_:bnode_2442e70f_afc7_4fa9_bcac_75f646016df8_682 a sh:ValidationReport ;
   sh:conforms false .
_:bnode_2442e70f_afc7_4fa9_bcac_75f646016df8_683 a sh:ValidationResult ;
   sh:resultSeverity sh:Violation ;
   sh:sourceShape jurisdiction:hasRegulationShape ;
   sh:sourceConstraintComponent sh:MinCountConstraintComponent ;
   sh:focusNode jurisdiction:EU ;
   sh:resultPath jurisdiction:hasRegulation ;
   sh:resultMessage "There must be >= 1 values" .
_:bnode_2442e70f_afc7_4fa9_bcac_75f646016df8_682 sh:result _:bnode_2442e70f_afc7_4fa9_bcac_75f646016df8_683 .
_:bnode_2442e70f_afc7_4fa9_bcac_75f646016df8_684 a sh:ValidationResult ;
   sh:resultSeverity sh:Violation ;
   sh:sourceShape _:anon-genid-416d7c76244f4204b8fd6481a1619411-0 ;
   sh:sourceConstraintComponent sh:MinCountConstraintComponent ;
   sh:focusNode jurisdiction:EU ;
   sh:resultPath rdfs:label ;
   sh:resultMessage "There must be >= 1 values" .
_:bnode_2442e70f_afc7_4fa9_bcac_75f646016df8_682 sh:result _:bnode_2442e70f_afc7_4fa9_bcac_75f646016df8_684 .
```

Figure 3.6: SHACL validation report example.

3.3.2 PropertyGroups

As mentioned in Section 3.3.1, the main purpose of SHACL is to validate RDF graphs. In addition to validation, SHACL can also be implemented directly into the ontology.
Analyzing how SHACL is implemented reveals that SHACL and its concepts are also RDF graphs. Listing 3.3.2 presents a part of the SHACL ontology. Lines 2, 9, and 16 show that a NodeShape and a PropertyShape are also classes. When defining a concrete nodeShape or a propertyShape as a *sh:NodeShape / sh:PropertyShape* in a schema (see 3.5 JurisdictionShape / hasRegulationShape), those shapes are in reality instances of the class sh:NodeShape / sh:PropertyShape. This structure allows object properties to be treated as instances and, therefore, subjects by creating the corresponding PropertyShape.

With these PropertyShapes and the ability to treat relationships as subjects, more information and knowledge can be created and involved in our ontology. One of the additional abilities is *PropertyGroups*.
A PropertyGroup is a concept of the SHACL ontology, where PropertyShapes can be put into one

thematic concept, allowing us to classify object properties, hence improving the ontology structure and knowledge representation. This new type of knowledge can also be used in SPARQL queries.

```
@prefix sh:    <http://www.w3.org/ns/shacl#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

sh:Shape
  a rdfs:Class ;
  rdfs:label "Shape"@en ;
  rdfs:comment "A shape is a collection of constraints that may be targeted for
        certain nodes."@en ;
  rdfs:subClassOf rdfs:Resource ;
  rdfs:isDefinedBy sh: .

sh:NodeShape
  a rdfs:Class ;
  rdfs:label "Node shape"@en ;
  rdfs:comment "A node shape is a shape that specifies constraint that need to be
        met with respect to focus nodes."@en ;
  rdfs:subClassOf sh:Shape ;
  rdfs:isDefinedBy sh: .

sh:PropertyShape
  a rdfs:Class ;
  rdfs:label "Property shape"@en ;
  rdfs:comment "A property shape is a shape that specifies constraints on the
        values of a focus node for a given property or path."@en ;
  rdfs:subClassOf sh:Shape ;
  rdfs:isDefinedBy sh: .
```

Listing 3.2: Definition of SHACL [23e].

Every (sub-)ontology has its own PropertyGroups implemented in its SHACL schema to ensure consistency and a closed implementation environment. Figure 3.7 shows how each PropertyGroup that is implemented. The first prefix of each PropertyGroup begins with the name of the ontology. Afterward, the name of the PropertyGroup follows. If an ontology is a sub-ontology of another, the corresponding PropertyGroup of that ontology will also be grouped into the super-ontology PropertyGroup. For instance, the PropertyGroup *OrganizationInversePathPropertyGroup* will be part of the *EnterpriseInversePathPropertyGroup* which will be part of the overall *InversePathPropertyGroup* that is implemented on the top-level IGONTO ontology. Listing 3.3.2 shows how a PropertyGroup is implemented in the SHACL ontology. Like the PropertyShape and the NodeShape, a PropertyGroup is also a *rdfs:Class*.
Therefore, the implementation of the *InversePathPropertyGroup* example is implemented as shown in Listing 3.3.2 at the top-level IGONTO ontology. Thus, in the enterprise ontology, the declaration of the *igonto:InversePathPropertyGroup* subclass relation is missing, ensuring the concept of semantical self-containment mentioned before. Recursively, the same applies to the organization ontology with the *enterprise:EnterpriseInversePathPropertyGroup*.

```
@prefix sh:    <http://www.w3.org/ns/shacl#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

sh:PropertyGroup
  a rdfs:Class ;
  rdfs:label "Property group"@en ;
  rdfs:comment "Instances of this class represent groups of  property shapes that belong
    together."@en ;
  rdfs:subClassOf rdfs:Resource ;
  rdfs:isDefinedBy sh: .
```

Listing 3.3: SHACL PropertyGroup definition [23e].

```
@prefix sh:    <http://www.w3.org/ns/shacl#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix enterprise:   <http://www.semanticweb.org/igonto/enterprise#> .
@prefix igonto:   <http://www.semanticweb.org/igonto/igonto#> .
@prefix organization:   <http://www.semanticweb.org/igonto/organization#> .

igonto:InversePathPropertyGroup
        a sh:PropertyGroup .

enterprise:EnterpriseInversePathPropertyGroup
        a sh:PropertyGroup ;
        rdfs:subClassOf igonto:InversePathPropertyGroup .

organization:OrganizationInversePathPropertyGroup
        a sh:PropertyGroup ;
        rdfs:subClassOf enterprise:EnterpriseInversePathPropertyGroup .
```

Listing 3.4: SHACL PropertyGroup example.

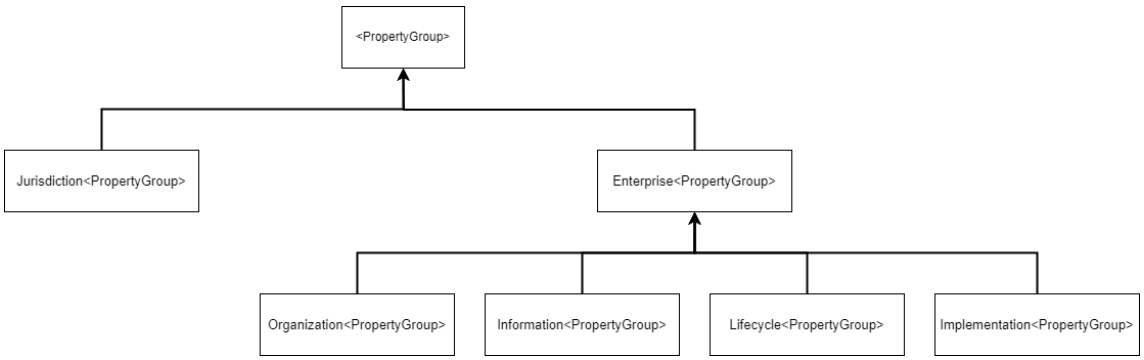


Figure 3.7: PropertyGroup hierarchy implementation.

PropertyGroups have several positive impacts on the ontology, including:

1. **Maintainability:**
   Grouping helps the developer to combine constraints on several object properties simultaneously. Each object property within a group inherits the constraints defined in that group, making it easier to maintain these constraints and reducing the overall lines of code.
2. **Clarity:**
   Grouping can be particularly helpful when, for instance, it is not clear whether the connection of an ObjectProperty is generated via inference or statically implemented and always present. PropertyGroups can inform the developer or user about the purpose of the ObjectProperty, leading to a clearer understanding of the overall system.
3. **Better expression:**
   PropertyGroups can offer very useful knowledge by expressing the meaning of an object property, for instance, when the object property explicitly indicates that there should be no connection between two instances. Because ontologies operate under the open-world assumption, a nonexistent connection cannot be interpreted as an explicitly unwanted connection. By grouping object properties within a *NegativeObjectPropertyGroup*, it becomes possible to treat the object property as a relationship that explicitly denotes the absence of a link between those two instances.
4. **Useability:**
   Implementing PropertyGroups can enhance usability in query writing. For example, when a query should only include inferences or negative object properties from an instance, having groups makes filtering easier. Without them, every unwanted relationship must first be explicitly known by its name and meaning. With PropertyGroups, the query can be easily implemented by filtering out the members of a group that represents that specific semantic meaning.

We implemented three PropertyGroups to improve the structure of the ontology. Each one serves a different purpose and is used differently.

1. **InversePath- & PathPropertyGroup:**
   Every relationship has an inverse one that oppositely connects subjects and objects. These two relationships are normal object properties with an additional axiom, *owl:inverseOf*, forming the bi-directional link between both of them. While inverse properties enhance an ontology's structure, the hierarchy can become obscured without further modifications. In our case, the preferred direction of thought begins with the Jurisdiction and the Enterprise ontology, extending down to the Implementation ontology. This is due to the fact that the implementation of the enterprise architecture depends on the regulation, what compliance requirements are to be met, and the organization units involved.
   Although the OWL ontology already allows the definition of properties as *owl:InverseFunctionalProperty*, there are limitations regarding this property, as the object of an *owl:InverseFunctionalProperty* uniquely determines the subject. In other words, the subject of this *InverseFunctionalProperty* can only have a relationship to exactly one object [23b]. This would not be suitable for many classes in our ontology.
   To maintain the hierarchy and structure while implementing inverse properties, relationships

are grouped into either *PathPropertyGroup*, indicating the *top-down* direction in the hierarchy, or the *InversePathPropertyGroups* for the opposite direction.

2. **InferenceObject- & NormalObjectPropertyGroup:**
   The *InferenceObjectPropertyGroup* only contains object properties that are derived from SWRL rules. Since inferences are also defined as normal object properties in ontologies, a separation without explicitly looking at the rules is not possible. To enhance clarity, relationships explicitly defined in the ontology development process and not inferred are put into the *NormalObjectPropertyGroup*, whereas inferences are put into the *InferenceObject-PropertyGroup*.
3. **NegativeObject- & PositiveObjectPropertyGroup:**
   The same clarity problem applies to object properties, which explicitly denote the negative relationship to objects. Without grouping these relationships, the problem can appear later on to misinterpret the result. To filter these properties out of the results or explicitly contain them in queries without naming each one of them, object properties that are meant negatively are divided from normal positive object properties.

Object properties can also be members of multiple groups. For instance, a SWRL-inferred relationship can be meant as a negative one (for example "*enterpriseDoesNotNeed*"). A practical use besides the additional structuring for these PropertyGroups and the advantages of these structures is shown in Section 5.4.

3.4 Ontology Fusion

Although splitting the ontology has some advantages, these ontologies must be combined in such a way that they work without errors in the overall system. The merging of ontologies is shown schematically in Figure 3.8. It describes two different ontologies, Ontology 1 and Ontology 2, with their concepts, schemas, and instances. The merging is handled differently.

3.4.1 Concept Fusion

In the present case, we always have at least one class as an intersection between two ontologies. This is implemented as a concept in both ontologies. These two classes describe the same concept, but either one of them is meant to be an abstract class of the other or both to describe the same thing but from a different point of view.

Figure 3.10 relates to the fusion of two classes where one of them is implemented as abstract, based on the example of the class *Capability*. Here, the original *Capability* class is implemented in the Organization ontology. This class is significant because it bridges the Jurisdiction with the Organization ontology. In both cases, description logic has to be implemented from the Jurisdiction and the Organization's point of view. Therefore, the implementation of an abstract class in Jurisdiction that describes the same concept with the same meaning is needed. The

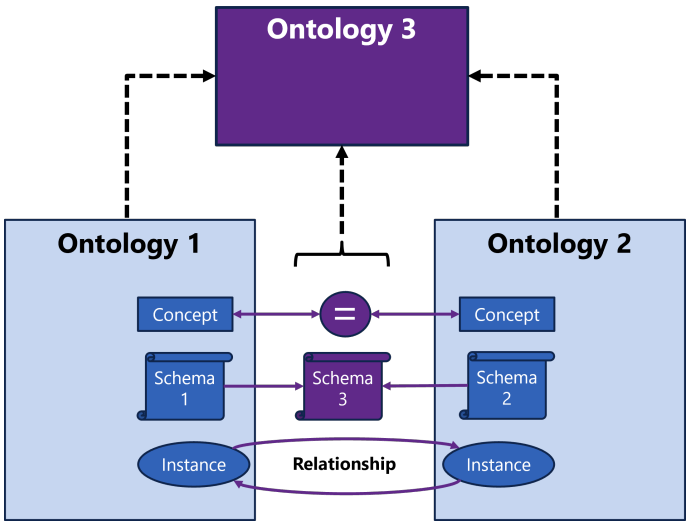


Figure 3.8: Ontology fusion.

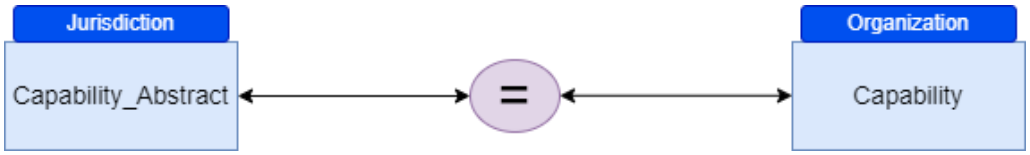


Figure 3.9: Example fusion of abstract classes.

CapabilityAbstract class enables the implementation of description logic from the Jurisdiction side, whereas the description logic from the organizational side is implemented on *Capability*. With this approach, both ontologies are inherently consistent with the ability to implement their description logic in the connecting class. When setting both classes equal on the top ontology, both classes inherit the description logic from each other. Without this approach, the description logic would have to be implemented manually on the top level, thus decreasing maintainability and making both classes dependent on the top-level description logic implementation. If a class is meant abstractly, it will have the suffix *Abstract* in its Uniform Resource Identifier (URI).

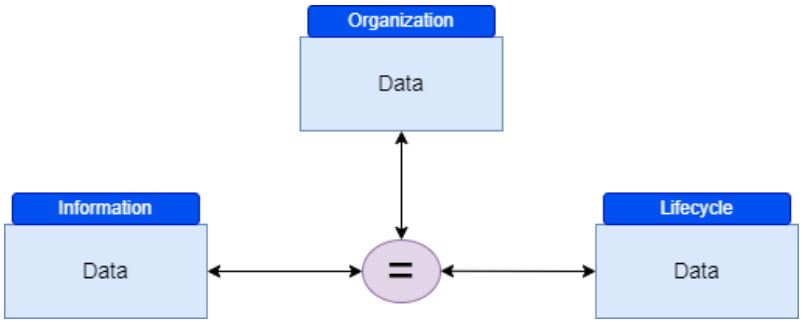


Figure 3.10: Example fusion of classes with different a point of view.

An example of the fusion of classes that share the same concept but from a different perspective is shown in Figure 3.10. While the name of the class *Data* is the same, the perspective from which this class is seen differs in every ontology. In the Information ontology, data are described in terms of their concrete values, the organizational aspect details which departments are responsible for maintaining data, and the Lifecycle ontology places it within the context of a Record lifecycle. These classes will also be set equal at the top level.

3.4.2 Schema Fusion

The schema fusion describes how the SHACL validation schema from two different ontologies are combined. As already mentioned in Section 3.3, every ontology has its own SHACL validation schema implemented, including its own PropertyGroups. These validation schemas can also be used on the top-level ontology. Therefore, combining two or more schemas is limited to these implementations:

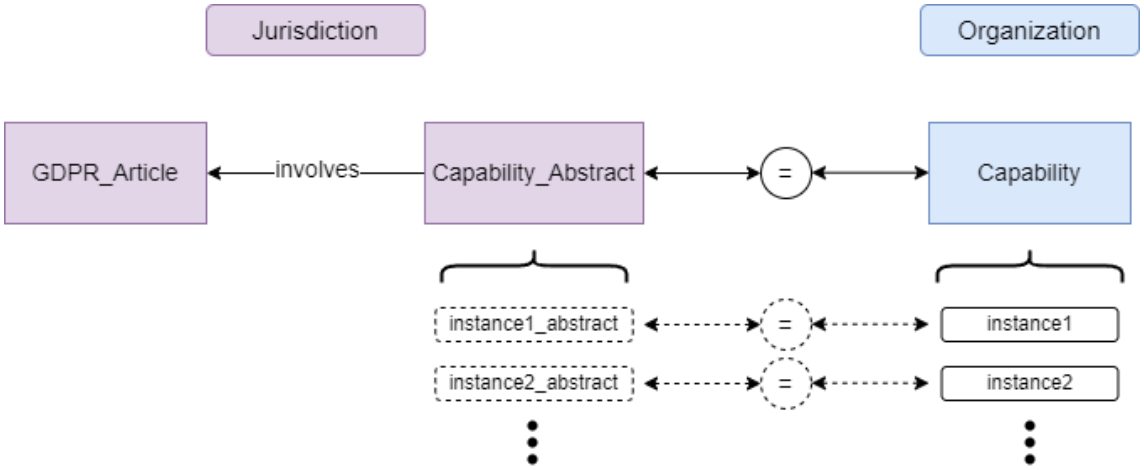


Figure 3.11: Example of a schema fusion from abstract classes.

- **Abstract classes with their non-abstract equivalents:**
  As discussed in Section 3.4.1, implementing abstract classes within the ontology is one possibility for combining ontologies. These abstract classes also implement the required DL. Figure 3.11 illustrates the necessary implementation needs in case those instances were also realized within abstract classes. As previously mentioned, *CapabilityAbstract* resides in the ontology Jurisdiction. In theory, implementing abstract instances is possible and allows validation of the involved relationship between *CapabilityAbstract* and *GDPRArticle*. However, the establishment of semantic equivalence between every abstract instance and its non-abstract variant would be necessary. This would result in even more implementation needs than simply validating the involved relationship at the top-level schema. In addition, each instance would have an unnecessary abstract duplicate, leading to overhead regarding storage and query performance. Because abstract classes are only implemented to later connect both ontologies and are not important for the Jurisdiction sub-ontology, validation at the bottom level is not required. Thus, validation still takes place at the top level to validate the relationships involved in the overall system.

- **PropertyGroups:**
  As mentioned in Section 3.3.2, every ontology has its own PropertyGroups. To utilize PropertyGroups from all ontologies simultaneously, these PropertyGroups become members of another PropertyGroup at the top-level schema, as shown in code snippet 3.3.2.
- **SWRL rules:**
  SWRL rules are implemented exclusively at the top level, as they integrate multiple classes and relationships across different ontologies. Thus, these rules and their corresponding inferences are validated in the top-level schema.

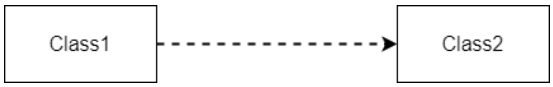
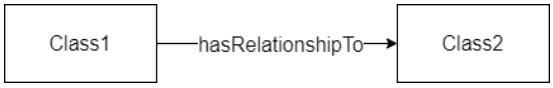
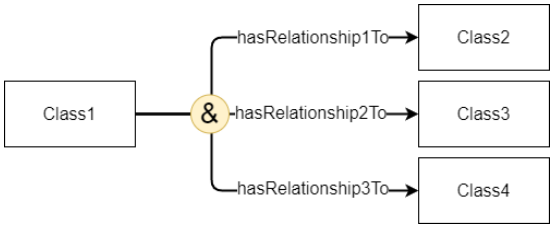
Apart from these implementations, the rest of the validation schema from every sub-ontology can be used exactly as they are to validate their part.

3.4.3 Instance Connection

As already mentioned, the implementation of instances for abstract classes leads to unwanted and unnecessary overhead. Returning to the example in Figure 3.11, the description logic for *Capability* is automatically carried over, including the *involves* relationship. However, explicit connections between instances are not automatically established. Therefore, in this case, the connection between the instances from *Capability* and *GDPRArticles* has to be connected via the *involves* relationship to conform to the description logic rules. The same principle applies to any connection between a regular class and its abstract counterpart, in which the non-abstract variant is implemented in another ontology along with its instances.

4 IGONTO Development

After illustrating various general approaches and design decisions associated with the development of IGONTO, this chapter describes the concrete implementation. Every sub-ontology is presented, along with the fusion of the whole system, including the development of validation, description logic, and SWRL rules. The concepts, general classes, and their relationship are based on previous work from [IGONTO23]. This chapter focuses on the specific implementation of these concepts, combined with the general approaches mentioned in Chapter 3.
Each sub-ontology described will include a visual representation. Every relationship has an inverse relationship, though to avoid overloading the visual representation, only the top-down direction will be visualized. Although some ontologies have SWRL rules that apply only for connections within their domain, the discussion of all SWRL inferences will be included in a separate section at the end of this chapter. This structure was chosen because the validation and implementation of all SWRL inferences occur at the top level, which enhances the overall structure and organization. Some classes are highlighted in green to denote that they contain subclasses, but are excluded from the overall visualization to enhance clarity. These subclasses do not have relationships other than *isSubClassOf* the corresponding superclass. Therefore, they are not critical for the overall view and are explained in more detail in the textual description. Table 4.1 explains how relationships and the description logic between classes are implemented in the following visualizations.

Visualization	Explanation
	The dashed arrow symbolizes a subclass relation between two or more classes. The class from which the arrow originates is the subclass. In this case, *Class1* is a *subclass* of *Class2*.
	If the arrow is not dashed, it indicates a relationship from one class to another. The relationship name is written above the arrow. In this case, *Class1* has a *relationship* to *Class2*.
	If an "&" stands before multiple arrows, all descendant relationships belong to one DL. Here, the DL would be implemented in Protegé as follows: *Class1 and* *(hasRelationShip1To Class2) and* *(hasRelationShip2To) Class3) and* *(hasRelationShip3To Class4)*

Continued on next page

Table 4.1 continued from previous page

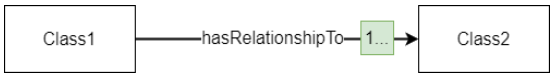
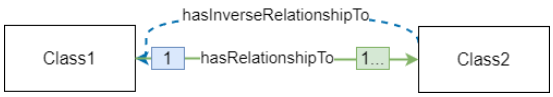
Visualization	Explanation
	Classes can have different colors. If the class is blue, it is an abstract class. If the class is green, it has further subclasses that are not visualized for clarity but are mentioned in a table within the section. If the class is orange, it is already visualized at another location within the graph but duplicated to make relationships shorter.
	On every relationship arrow, certain entities are noted. These entities explain the description logic behind the relationship: • If the entity has a 1 with dots, the description logic keyword “some” is used, indicating that the subject of this relationship is connected to at least one instance from the object class. • If the entity has a 1 without dots, the description logic keyword “exactly” is used, indicating that the subject of this relationship is connected to exactly one instance from the object class. • If the entity has a 0 with dots the description logic keyword “min 0” is used, indicating that the subject of this relationship may be connected to at least one instance from the object class.
	Every relationship has a corresponding inverse relationship that connects the subject and the object in the opposite direction. For clarity, inverse relationships are not shown in the graph, but they are practically always present. Sometimes the description logic differs in the opposite direction. To still show the inverse DL, the entity number of the inverse relationship (here, the blue 1) is included.

Table 4.1: Graph legend.

4.1 Jurisdiction Ontology

The jurisdiction ontology is at the top level, encompasses the jurisdiction domain of a government and its regulations, and bridges the path between jurisdiction and enterprise. Figure 4.1 visualizes the concept level. As it connects the jurisdiction and the enterprise domain, several abstract classes are implemented (see Section 3.4.1). Each abstract class is colored blue, and its origins lie in the organization ontology.

The *Enterprise* represents a business or organization that *operates in* one or more *Countries*. Every *Country belongs to* a *Jurisdiction*, which is the legal entity of a region and has the authority to govern and legislate within its boundaries. A *Jurisdiction has* a *Regulation* with which *Countries* are *regulated*, it establishes and enforces rights and obligations for enterprises, business partners and customers. Every *Regulation requires* an enterprise-wide governance *Strategy* to achieve regulatory compliance and a governance *Program* with specific initiatives and projects that implement the required measures to satisfy compliance needs. *Country* has the sub-classes *US* and *EEA* including its subclass *EU*, to connect countries with the right jurisdiction. Countries in the EU must comply with GDPR, whereas states in the US need to comply with various federal and state laws, each serving a different purpose. The European Economic Area (EEA) includes every country that needs to comply with GDPR. This hierarchy is important because not every country in the EEA is also in the EU, such as Norway. In the IGONTO prototype, we developed the concept of information governance based regulatory compliance using GDPR to represent regulatory requirements in general. Therefore, the concept of *European Union* was implemented as a subclass of *Regulation*, with further sub-classes implementing the *GDPR* structure. *GDPR* is the european *Regulation* and *includes GDPR Articles*. The GDPR has 11 *Chapters*, each *including* a varying number of *Articles*. A total of 99 articles form the foundation of GDPR. Every *Article includes Keywords*, to quickly summarize the intention behind each article.

Associations such as ARMA or CGOC, which are based on GDPR, have defined *Capabilities* that describe how to achieve compliance at different levels. Whereas ARMA concentrates on providing implementation guidelines based on the GDPR Principles, CGOC defines capabilities in functional processes. Because they describe GDPR, every *Capability involves* some *GDPR requirements as specified in their respective Articles*. Since these associations describe compliance from different points of view, identifying commonalities between them is challenging. Therefore, we connect these capabilities with more generally defined *Rights* and *Obligations* using SWRL inferences to form a unified set of *Requirements*. This is realized because each of these classes *involves GDPR Articles*. The exact implementation is explained in the separate Section 4.6, which deals with all SWRL inferences in this ontology. Originally, these capabilities were implemented in the organization ontology, but also as an abstract class here because they are based on GDPR and its articles.

Obligations and *Rights* include sub-classes, each describing a specific obligation/right. Because Obligations and Rights are fundamental for the connection between the (non)-functional *Requirements* and the *Capabilities*, each connection to the *GDPR Articles* involves an annotation property with a citation or explanation, as well as why the article is involved in this obligation or right

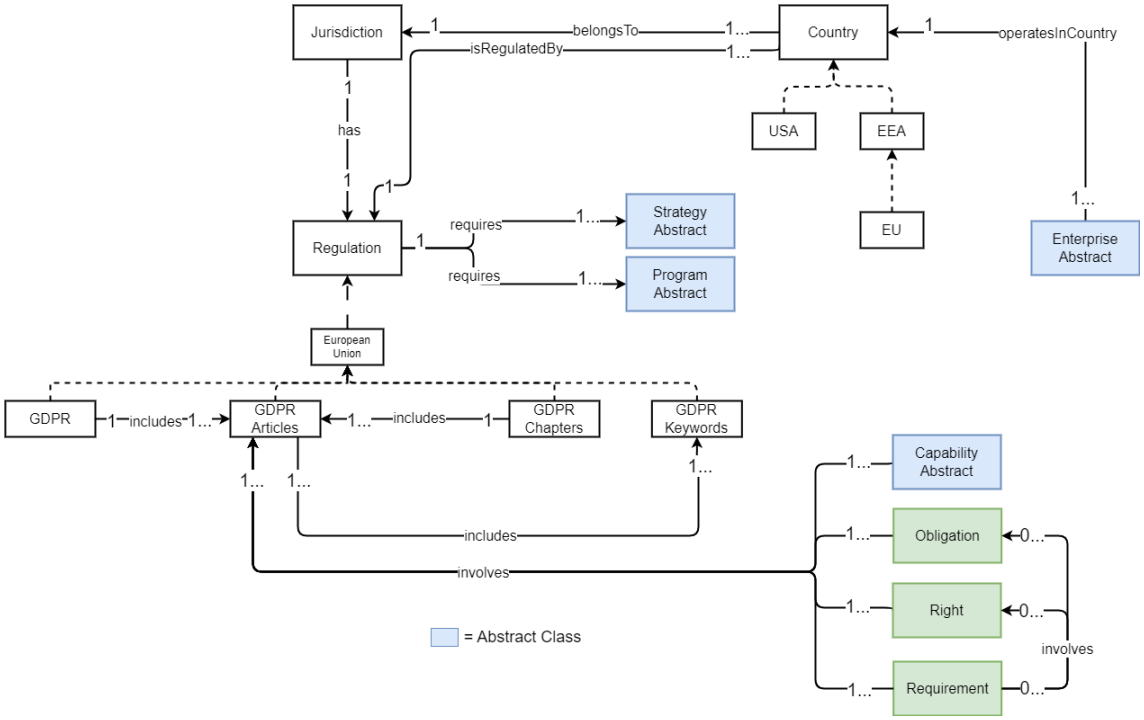


Figure 4.1: Visualization of the Jurisdiction ontology.

instance. Figure 4.2 illustrates this approach based on the *Obligation To Ensure Integrity Of Data Instance*. *Article 5* is involved in this obligation, and the explanation is added as a comment to the object property as a citation from *Article 5*.

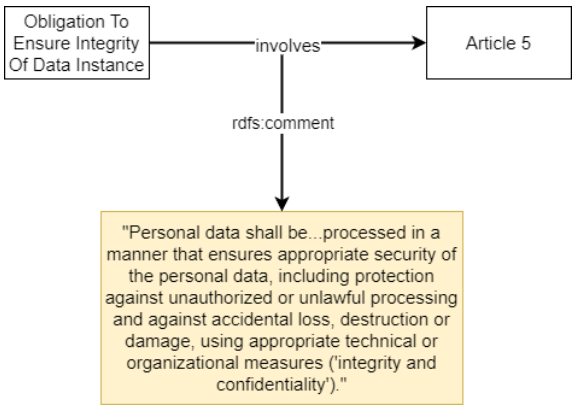


Figure 4.2: Visualization of annotation property.

Table 4.2 lists the further taxonomy of *Obligations*, *Rights*, and *Requirements*. The intention of each *Obligation*, *Right* and *Requirement* can be derived from the class name itself.
Additionally, obligations, rights, and requirements are semantically grouped whenever possible. These include the following:

- Obligations for the Data Controller

- Obligations for the Data Processor
- Data Subject rights
- Functional Requirements
- Non-Functional Requirements

Superclass	Subclass
Obligation	**DataControllerObligation**, **DataProcessorObligation**, ObligationToAppointDataProtectionOfficer, ObligationTo NotifyOfPersonalDataBreach, ObligationToPerformDataProtectionImpactAssessment
DataControllerObligation	ObligationToCooperateWithSupervisoryAuthority, ObligationOfCross-borderDataTransfer, ObligationToEnsureAccountabilityOfDataController, ObligationToEnsureAccountabilityOfProcessor, ObligationToEnsureAccuracyOfData, ObligationToEnsureConfidentialityOfData, ObligationToEnsureIntegrityOfData, ObligationToFulfillConsentRequirement, ObligationToLimitPurposeDataProcessing, ObligationToLimitStoragePeriod, ObligationToMinimizeData
DataProcessorObligation	ObligationToCaptureRecordsOfDataProcessingActivity, ObligationToFairnessDataProcessing, ObligationToLawfulnessDataProcessing, ObligationToTransparencyOfDataProcessing
Right	**DataSubjectRight**
DataSubjectRight	RightRelatedToAutomatedDecision-making, RightRelatedToAutomatedProfiling, RightToAccess, RightToBeForgotten, RightToBeInformed, RightToComplain, RightToDataPortability, RightToErasure, RightToObject, RightToRectification, RightToRestrictProcessing
Requirement	**FunctionalRequirement**, **Non-FunctionalRequirement**
FunctionalRequirement	FR.Access, FR.SearchFR.Audit, FR.Dispose, FR.Store, FR.Secure, FR.Transfer, FR.Classify, FR.Privacy, FR.eDiscover, FR.Record, FR.Archive, FR.Replicate, FR.Manage, FR.Protect, FR.Administration, FR.Load, FR.Query, FR.TransactionSupport, FR.ReferentialIntegrity, FR.Collect, FR.Hold, FR.Retain, FR.Control
Non-FunctionalRequirement	**NFR.Availability**, **NFR.ServiceQuality**, NFR.Backup, NFR.Resiliency, NFR.Recovery, NFR.Restore, NFR.LoadBalancing, NFR.Elasticity, NFR.DisasterRecovery, **NFR.TransactionSupport**, **NFR.Scale**
NFR.Availability	NFR.HighAvailability
NFR.Scale	NFR.DynamicScale

Continued on next page

Table 4.2 continued from previous page

Superclass	Subclass
NFR.ServiceQuality	NFR.ServiceLevelAgreement
NFR.TransactionSupport	NFR.TwoPhaseCommit, NFR.ReferentialIntegrity

Table 4.2: Hierarchy of Obligations, Rights and Requirements.

4.2 Organization Ontology

The principle underlying this sub-ontology is presenting information governance from an organizational point of view. The main part consists of a description of the *Enterprise*, its *Organizational Units*, and how they interact with and produce *Data*.
The *Enterprise* superclass has three subclasses: *SmallEnterprise*, *MediumEnterprise*, and *LargeEnterprise*. These subclasses represent enterprises with a certain number of employees. *Small Enterprises* have fewer than 250 employees, *MediumEnterprises* have between 250 and 5000 and *LargeEnterprises* represent enterprises with an employee number larger than 5000. These subclasses also represent the use case covered in this prototype to show the functionality of IGONTO. By GDPR mandate compliance obligations depend on the size of an enterprise. These different obligations and implementation requirements are dynamically created through SWRL rules and multiple sub-ontologies. The exact rules and how they are implemented are discussed in Section 4.6.

The *Governance* class represents the governance side and how the *Enterprise* interacts with it at a top level. *Governance* has three subclasses: *Governance Criterion*, *Governance Body*, and *Governance Strategy*. Every *Enterprise needs* a *GovernanceStrategy* to manage its resources effectively. Calahan et. al. [CBK04] emphasized that IT governance and therefore the needed strategies are crucial for organizations to achieve their business objectives and goals. The *GovernanceStrategy* defines these *Objectives* and *Goals*. *Objectives* set by the strategy ensure adherence to governance criteria, such as *Quality*, *Compliance*, and *Transparency*. These criteria, which need to be fulfilled, ensure the alignment of governance strategy within the organization's legal, ethical, and operational standards. This alignment of strategy in (IT)-Governance with the *Enterprise Risk Management* positively impacts an organization's overall performance [Sir21].

Every *Enterprise needs* a *Steering Committee*, a group of high-level stakeholders, who are essential in directing project sponsors and involving managers and supervisors in the project [SL14]. The *Steering Committee defines* a *Program* and a set of *Strategies*. The *Program* represents a set of activities with the intention to implement parts of the *Strategy*. Every *Program specifies Principles*, which are fundamental guidelines for steering the program. A *Strategy* is a plan to *realize* long-term *Objectives* and to *define Policies*, that *define Rules* to govern all actions within an enterprise.

Every *Rule executes* or governs the creation and management of *Records*. A *Record* is documented information of activities or transactions. [HH13] highlighted the importance of enforcing business rules within the data lifecycle so that they ensure data quality and consistency. A *Process* is a series

of actions that *creates Records*, which document the decisions taken and the outcome of the *Process*. *Records govern* management and use of data, which include information about creation, processing, storage, retrieval, protection, disposition, and other needs [Ben10]. These steps are summarized in *Record Lifecycle*, which are *used on Records*.

Every *Enterprise has* multiple *Organization Units*, which refer to different segments such as *Departments* or *Business Unit* that fulfill different functions within the enterprise. While *Departments* focus more on specific functions, *Business Units* operate more independently and focus on specific market segments or product lines. The specific functions of the Departments are divided into *Change Management*, *IT*, *Legal*, *Privacy and Security*, *Records Management*, and *Risk and Compliance*. Every *Department has Employees*, who work in specific *Departments*. Because departments are functional organization units, each *Department creates Data*.
The *Change Management* implements strategies and adopts approaches to transform organizational goals, processes, and technologies. The key objectives include not only finding and preparing for changes to improve performance but also minimizing resistance while adopting these changes. *IT* departments have a wide range of responsibilities that are crucial for the enterprise, such as administration, accounting, and data integration. The *Legal* department oversees the regulatory part by managing legal risks, thus ensuring compliance. *Privacy and Security* summarizes the departments responsible for privacy and security within the enterprise, such as *Access Control*, *Audit*, and *Monitoring*. *Records Management* is responsible for all functions and processes in connection with records. *Records and Information Management (RIM)* and *Electronic Records Management (ERM)* are two specifications. While the RIM contains the management of all records, the ERM concentrates on digital records such as emails, and digital documents. *RiskAndCompliance* also includes *EnterpriseRiskManagement* and *Information Risk Management* as specifications. Both have the goal of identifying, managing, and reducing risks. While *EnterpriseRiskManagement* concentrates on identifying risks regarding the overall enterprise, the scope of *InformationRiskManagement* covers risks regarding the management and security of information.

Every *Organization Unit has* a specific *Role* they fulfill. Every specific role is grouped into one of the following concepts: *Governance Role, Data Protection Role*, or *Business Role*. *Governance Roles* focuses on monitoring and managing the overall enterprise governance strategy and compliance. *Governance Roles* include the following:

- *Chief Compliance Officer (CCO)*, who is responsible to ensure lawfull and regulatory compliance.
- *Chief Security Officer (CSO)*, who is responsible for physical and digital security.
- *Chief Information Governance Officer (CIGO)*, who is responsible for leading the strategy rearding information governance
- *Data Protection Officer (DPO)*,which involves advising and supervising the enterprise regarding data compliance.

Data Protection Roles concentrate on data protection and data security. The *Data Controller* determines data processing methods while ensuring data protection principles. The *DataProcessor* acts on behalf of the *DataController* and must ensure confidentiality and security while processing data. The *Data Subject* is the owner of the processed data and has several rights that need to be

ensured throughout the whole process.
Business Roles include positions that are linked to the management, planning, and implementation of projects. *Executive Sponsors* often lead positions that bridge the gap between upper management and project implementation, which the *Project Manager* is responsible for. *Stakeholders* represent internal or external parties with a certain interest in the project and have a direct impact on the project's success.

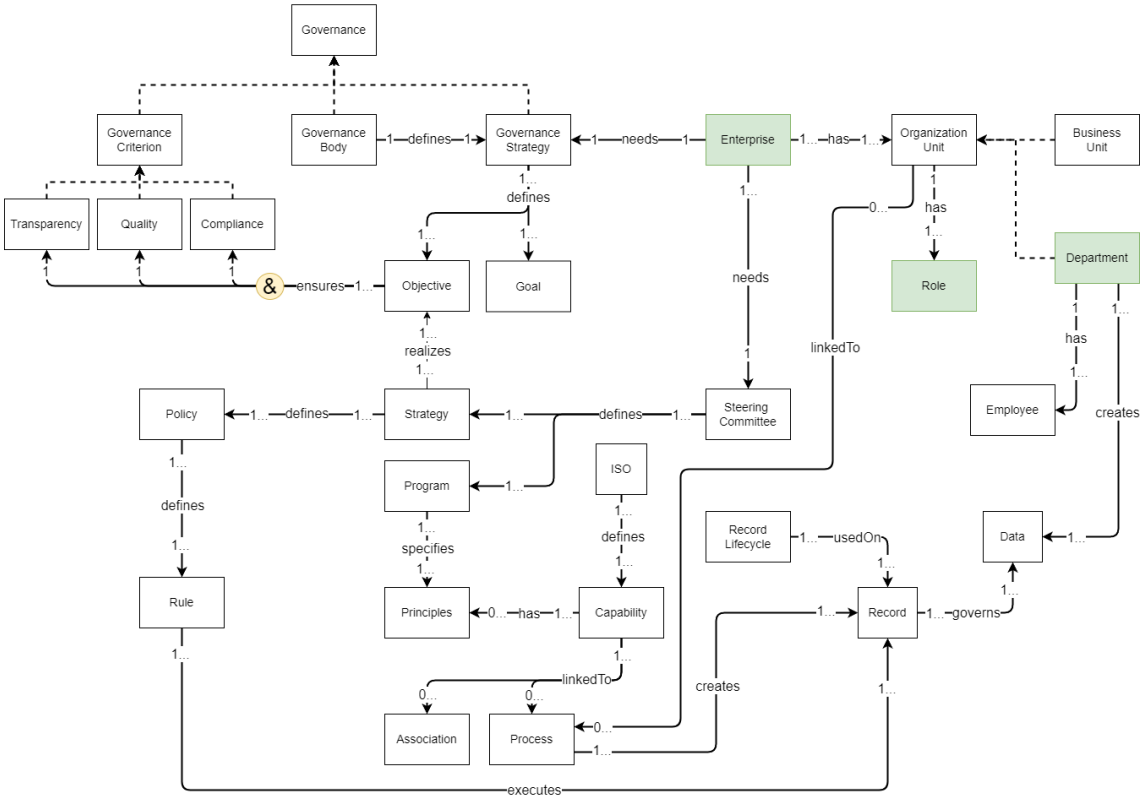


Figure 4.3: Visualization of the Organization ontology.

4.3 Lifecycle Ontology

The *Lifecycle* is relatively small and focuses on the lifecycle of data, extending the organization ontology in the Record Lifecycle. *Data drives* the *Record Lifecycle*, which handles the creation, management, and retention of records. A well-defined information lifecycle and its governance are crucial for the reduction of data growth, risk, and cost to optimize information value [Lam14].
A *Record Lifecycle* is divided into *Service* and *Process*. *Service* represents a set of activities to manage, access or utilize data/records (e.g. information). This includes the following:

- *Access*, representing methods to access information by an authorized user.

- *Find*, representing methods to locate specific information.
- *Rendering*, representing methods to convert information into usable formats.
- *Retrieve*, representing methods to retrieve specific information from a database.
- *Service Search*, representing structural methods to search and retrieve information based on specific queries or criteria.
- *Session*, representing user interaction periods with the system to access and manipulate data.
- *Submission*, representing methods to upload data into the system for later processing, management, or storage.

A *Process* is a sequence of steps to achieve a specific record or data management goal. These goals include processes for record creation up to their disposal and archiving:

- *Archive*, representing processes to transfer records into long-term storage for preservation.
- *Classify*, representing categorization processes of information, based on predefined metrics.
- *Index*, representing indexing methods to enhance the searchability of information.
- *Measure*, representing the application of predefined metrics on information to enhance efficiency and compliance.
- *Metric*, representing standardized units for measurement to evaluate effectivity and efficiency.
- *Monitor*, representing processes to oversee data management to ensure functionality.
- *Retain*, representing processes for keeping information as long as required in a certain repository or process, to meet policy rules or other requirements.
- *Search*, representing processes to locate information.
- *Secure*, representing processes regarding security measurements to protect information from unauthorized access.

Processes create Statistics, providing various information regarding record management. *Reportings* are *produced* by these *Statistics*, to assist upper management in decision-making and compliance.

4.4 Information Ontology

The information ontology describes the concept of data and its information value. *Data* represents raw facts used and collected by organizations and can be interpreted in several ways. *Data* is *equivalent to* an item, which represents a concrete individual unit of information. Every *Item* has a *Version* to represent its state over time. Because data can have multiple structures, *Document, Asset, Content*, and *Metadata* are defined as subclasses of *Data*. *Documents* are structured units of recorded information. A Document *has* a *Rendition*, which represents the document in a different format for a different purpose (different file format, different language, etc.), and an *Edition*, which documents updates and changes. *Assets* describe strategic, financial, or operational information valuable to the enterprise. *Content* focuses on the information conveyed by the data itself. *Metadata* provide *Information* about data, which is a subclass of *Metadata*. This *Information* can include

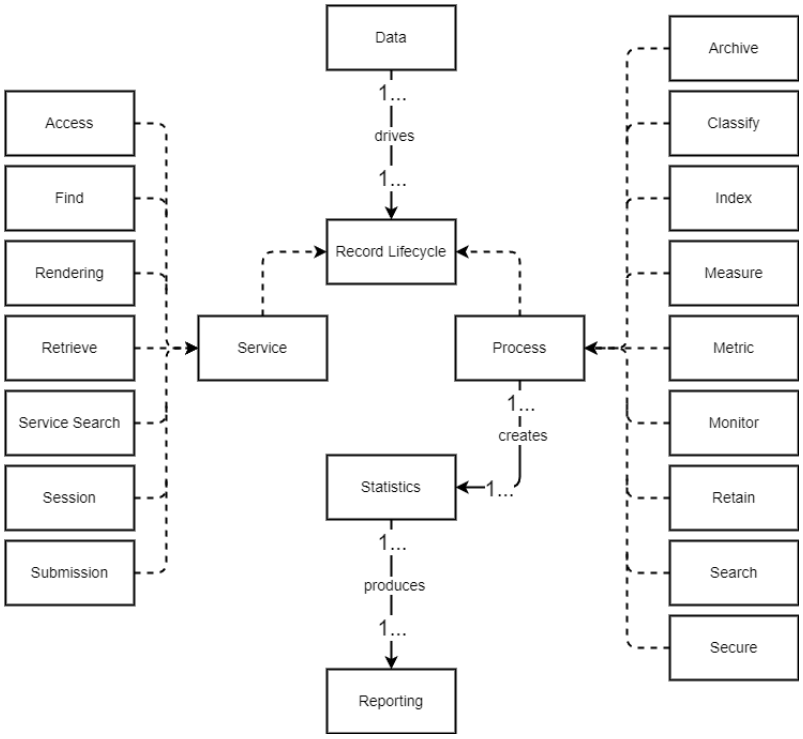


Figure 4.4: Visualization of the Lifecycle ontology.

Archival Information, Content Information, or a *Description*. These pieces of *Information* are *contained in Files*, which in turn are *contained in Container* - a broader term used for organizing and storing records.

Container organize files and include their information. *Data* can also be represented as a vector, comprising *containing Value*, *exposing Risk*, and *producing Cost*. Calculating these components aids in process decisions regarding data management. For instance, if the data value is too low and the costs of keeping it in a certain repository are too high, the data may be archived for long-term storage.
Records classify and therefore categorize and organize *Data*, thus represents a crucial component in effective data management [Fra13]. *Records* have a *RecordControlStructure*, which is used to manage and control records within organizations. These *RecordControlStructures* include the following:

- *Control Data*, representing records of sensitive data, such as *PersonnelFiles, PII-Data, and AuditData.*
- *Report*, representing records of data that provide analytical information for decision-making.
- *Hold*, representing an organizational directory to retain records, often for audits or legal matters.
- *Retention*, representing policies to manage the duration of record keeping in certain repositories.

- *Audit Trail*, representing recordings of changes or actions performed on records to ensure security, integrity, and compliance [Bjo75].
- *Category*: A classification of records.

Every *Record uses* a *DataDictionary*. The *DataDictionary* forms the *vocabularyFor* the *Taxonomy*, which provides uniform terms and definitions for categorization and classification, and therefore, allows the effective organization of information [Mil07]. The Data Dictionary ensures consistency, and clarity and is crucial for effective record management [BP08].

A *Policy* is a set of guidelines to govern proceedings and activities within an enterprise. *Policies grant Priviliges* to *Roles*, defining what actions a role can perform. *Policies monitor metrics* to evaluate their effectiveness and to enable necessary changes. A *Policy has* a *Goal*, representing an objective, a desired outcome, or a principle such as accountability that the policy aims to achieve. These policies and their goals need to be implemented. This is done by first *applying Rules*, which are specific regulations that must be followed. These *Rules* build the foundation and *define* a *Schedule*, which is *used in Retention* to ensure that records are kept for an appropriate duration.

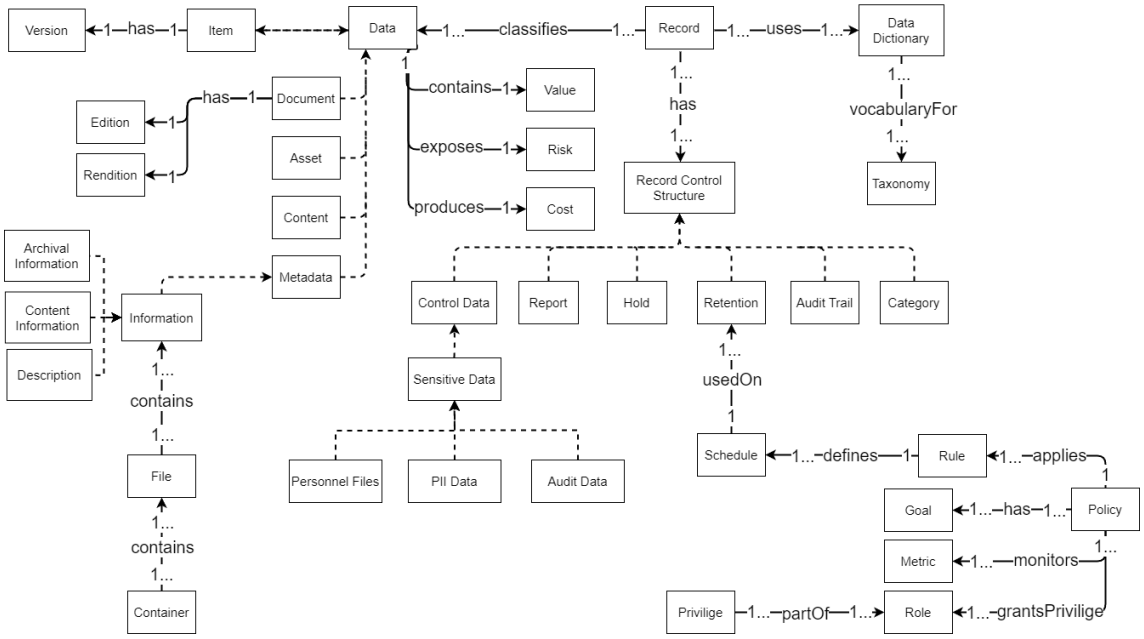


Figure 4.5: Visualization of the Information ontology.

4.5 Implementation Ontology

The *Implementation* ontology provides a structured approach for creating unique blueprints for architecture solutions and their design components based on the individual company's needs. Knowledge inferred from the other ontologies aids in inferring solutions in this domain. Although the *Implementation* ontology is a direct subdomain to the *Enterprise* such as *Organization*, *Information*, and *Lifecycle*, its granularity is higher. Therefore *Implementation* contains multiple subdomains

to grasp all necessary concepts in a structured manner. Some *Implementation* sub-ontologies semantically represent the same domain as those already implemented but from a different point of view. Thus, even though these domains are similar, some changes are implemented to represent the domain in the implementation context.

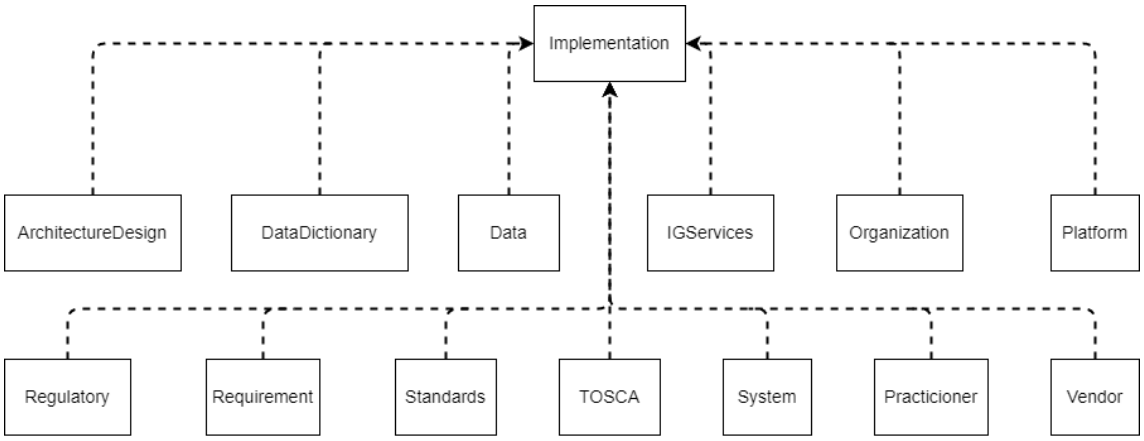


Figure 4.6: Visualization of the Implementation ontology hierarchy.

Figure 4.6 demonstrates the hierarchy of the *Implementation* ontology. Even though *Implementation* has various subdomains, some of them are rather simple but still kept in their individual ontologies to provide a clearer structure.

4.5.1 ArchitectureDesign Ontology

The *ArchitectureDesign* bridges the knowledge of other ontologies with the implementation domain and forms the framework of the individual enterprise solutions. Enterprises from the top level are connected with SWRL rules to concepts of this domain. These concepts represent the different architecture solutions, where these solutions are in turn implemented with the other domains within the *Implementation* ontology.

Figure 4.7 shows the ArchitectureDesignDomain. The subclasses of the green-highlighted classes are presented in Table 4.3, to simplify the visualization. The first subclass level contains concepts about *Function, DataQuality, DataArchitectureManagement, DesignComponent, Method, Lifecycle, SoftwareDesignPattern* and *Solution*.
DataArchitectureManagement is crucial to organizing data assets to ensure certain *DataQuality*. Both aspects are involved in the architecture solution, as different implementation solutions require different data quality and management efforts. *DesignComponents* describe the building blocks of the system architecture. The combination of all components defines the structure and behavior of the system. *Methods* refer to the necessary procedures within the architectural solution. *Function* is a synonym for *Methods* used in other domains. Therefore, we equate both classes. *Lifecycle* describes the different stages of an architectural entity, as distinct stages may require alternate functional solutions. *SoftwareDesignPatterns* are proven efficient, standardized solutions that provide scalable design strategies for the architectural domain.

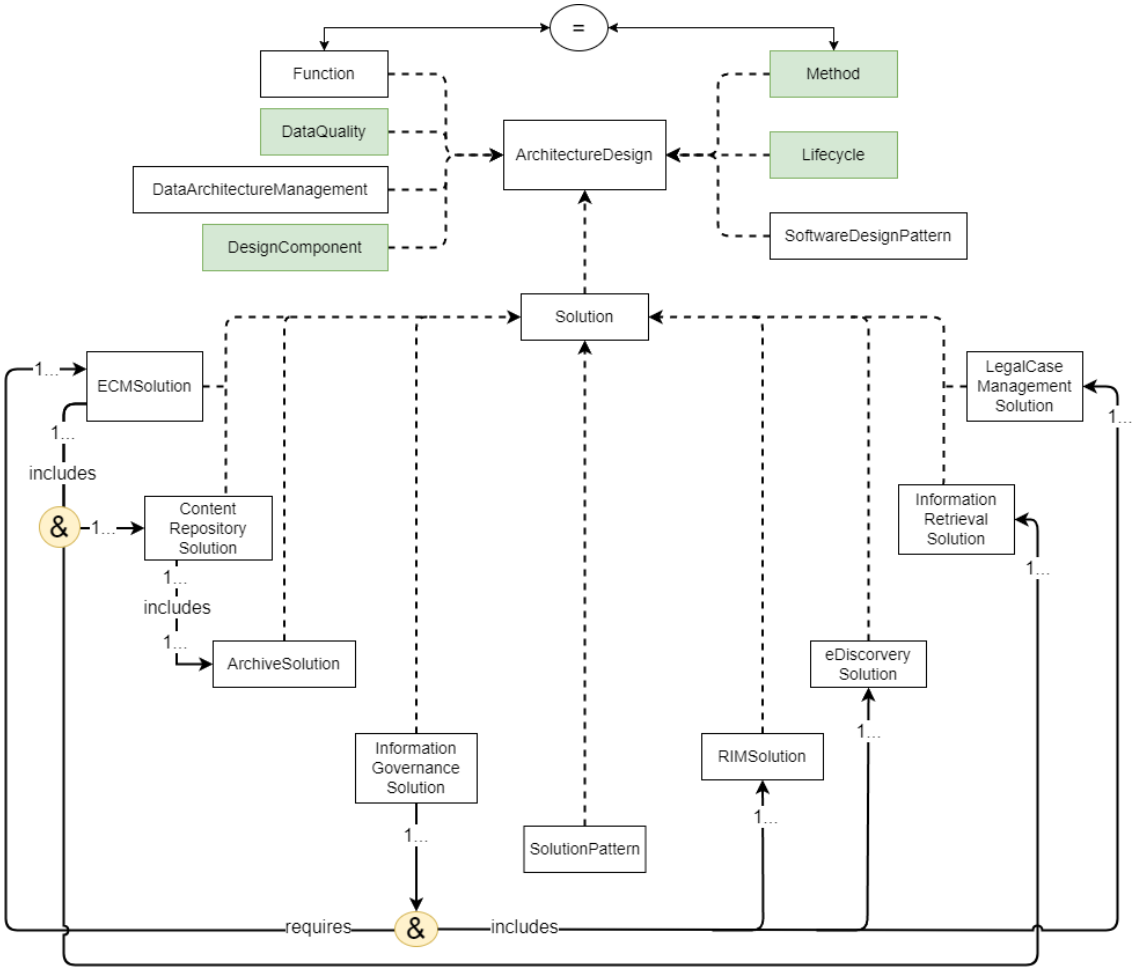


Figure 4.7: Visualization of the ArchitectureDesign ontology.

In *Solution*, diverse examples of solutions are implemented, to which the enterprises at the top level of the ontology are connected. *ArchiveSolutions* focuses on long-term data retention and access. *ECMSolutions* represent architecture implementations needed to organize information across the enterprise, such as documents, and assets. *ContentRepositorySolutions* are more specialized management architectures that focus on organizing, storing, and retrieving digital content. Therefore, *ECMSolutions* often *include ContentRepositorySolutions* and *InformationRetrievalSolutions*. Because the digital content also has to be archived, *ContentRepositorySolutions* includes *ArchiveSolutions*. The *InformationGovernanceSolution* is a more extensive solution architecture and represents the combination of solutions if all GDPR obligations have to be implemented. Therefore, *InformationGovernanceSolution* is a complex class, which contains solution architectures focused on retrieving information (*InformationRetrievalSolution*), managing records *RIMSolution*, discovery processes (*eDiscoverySolution*) and *requires* an *ECMSolution*. Like the ECMSolution, it includes some *ContentRepositorySolutions*. That these four solutions are all together part of the *InformationGovernanceSolution*, is visually illustrated by the "&" in the graph. The *LegalCaseManagementSolution* focuses on tracking case progress and the management of

further legal matters, such as billing and evidence. Each solution is solved by a template of various *SolutionPatterns*. These include best practices for solving common problems.

Superclass	Subclasses
DataQuality	DataQualityAwerness, DataQualityBusinessRule, DataQualityDefect, DataQualityManagement, DataQualityMetric, DataQualityPerformance, DataQualityServiceLevel, OperationalDataQualityProcedure
DesignComponent	**CatalogComponent**, ClassificationComponent, ContentManagementComponent, GovernancePortal, GovernanceComponent, TransactionManagementComponent, ControlComponent, IMSComponent, **FulltextServer**, ContentManagementPortal, StorageManagementComponent, RIM, RelationalDatabase, Repository, SoftwareComponentPattern, LoadBalancing, TransformationComponent, EndUserPortal, IndexingComponent, **InformationManagement**
CatalogComponent	ContentCatalog, ContentObjectCatalog, RecordsCatalog
Content Management Component	MetadataManagement, RepositoryManagement, SearchAndRetrievalManagement, TransactionManagement
FulltextServer	FulltextIndexServer, FulltextSearchServer
Information Management	**AnalyticsImp**, BusinessIntelligence, BusinessIntelligenceManagement, InformationRetrieval
Lifecycle	InformationLifecycle, LifecycleStage
Method	AccessMethod, TransferMethod, SecureMethod, StoreMethod, ManageMethod, DisposeMethod, SearchMethod, RecordMethod, TwoPhaseCommitFunctionMethod, eDiscoverMethod, AuditMethod, AdministerMethod, ArchiveMethod, PrivacyMethod, ControlMethod, QueryMethod, HoldMethod, LoadMethod, ClassifyMethod, CollectMethod, ProtectMethod, RetainMethod
AnalyticsImp	AI, MachineLearning

Table 4.3: Remaining hierarchy of the ArchitectureDesign ontology.

4.5.2 DataDomain Ontology

The *DataDomain* Ontology is similar to the *Information* Ontology but focuses on data management in the context of architecture solutions. Figure 4.8 shows the visualization of the domain.
The direct subclasses of the domain are *DataType, DataRisk, DataCost, DataValue, SpecialCategory, Policy, Schedule, Rule* and *dcat:Catalog*. The biggest subclass is *DataType*, which describes various forms of data. Every *DataType has DataCost, DataRisk* and *DataValue*, which represents financial aspects of data, potential threats associated with data, and the importance and relevance of data, respectively. All three concepts are merged, and thus each data point is represented by a vector containing risk, cost, and value. Every piece of data *consistsOf* a *Policy* which

recursively *consistsOf* the combination of a *Schedule* and its *Rules*. The class *dcat:Catalog* involves organization and indexing of data, which is again divided into *ObjectCatalog, CatalogComponent*, and *RecordsCatalog*. The *SpecialCategory* refers to data that requires special management, such as sensible data.
The subclasses of *DataType* describe different data variations, which include *AssetType, CategoryType, ObjectType, TransferType, DispositionType, LifecycleType, StorageType, MetadataType*, and a TriggerType. *AssetType, CategoryType*, and *ObjectType* classify and categorize data, which is crucial for effective data management. *TransferType, DispositionType* and *LifecycleType* refer to different stages of data management that are important for maintaining data integrity. More detailed concepts are implemented in the *DataDomain* as examples. For instance, *ContentObjects*, such as multimedia files and *Documents* are subclasses of *BusinessData*, which encompasses data related to business processes. Here, a *Document* is detailed and *consistsOf DocumentMetadata*, which is a subclass of *MetadataType*, and *ContentObjects*. From these *ContentObjects*, more specific types exist, such as *Rendition*. Additionally, an example of using connections for *Databases* is implemented. Here, *RelationalSchema*, *RelationalDatabase* and a *IndexingScheme* are part of a *Database*. The *RelationalDatabase* is a type of *Database* that stores data in a structured format and is *composedOf* a *RelationalSchema*, which defines its structure. *ContentCatalog* and *RecordsCatalog* are further examples that illustrate the relationship with the *Database* concept. Here, *ContentCatalog requires* a *RelationalSchema* and a *RelationalDatabase* to function effectively. The *RecordsCatalog isSearchable* within a *RelationalSchema* and *isComposedOf* of a *RecordsCategory*, which is a subclass of *CategoryType* and implies that the *RecordsCatalog* is a collection of records. These concepts now have to be extended to all *DataTypes*, *Catalogs*, and *Databases*, as listed in Table 4.4, in the next implementation iteration of this prototype.

Superclass	**Subclasses**
AssetType	BusinessAsset, ContentAsset, DataAsset, DesignAsset, **HRAsset**, IPAsset, ITAsset
HRAsset	HRData
BusinessData	BusinessAsset, ContentAsset, DataAsset, DesignAsset, **HRAsset**, IPAsset, ITAsset
ContentObject	MultimediaObject, OfficeDocument, **Rendition**
Rendition	HTMLDocumentRendition, PDFDocumentRendition
Document	Book, LegalDocument, Notice, ReferenceDocument, Report
RecordsCategory	AccessManagement.RC, EvaluationAndDisposal.RC, GovernmentRelation.RC, InformationManagement.RC
Database	FulltextIndex, RelationalSchema, IndexingScheme, RelationalDatabase, NewSQLDatabase, SQLDatabase, NoSQLDatabase
DispositionType	DispositionRecord, DispositionSchedule, DispositionTrigger
LifecycleType	ContentLifecycle, InformationLifecycle, StorageLifecycle, DataLifecycle
MetadataType	AccessControlList, Information, DocumentMetadata, Record
ObjectType	BusinessObject, **ContentObject**
Policy	DispositionPolicy, StoragePolicy, RetentionPolicy, TransferPolicy

Continued on next page

Table 4.4 Continued from previous page

Superclass	Subclasses
RetentionType	RetentionRecord, RetentionPeriod, RetentionSchedule
Rule	DispositionRule, RetentionRule, StorageRule, TransferRule
Schedule	RetentionSchedule
StorageType	BlockStorage, StorageInformation, KeyValueStorage, HierarchicalStorage, TieredStorage, ObjectStore, StorageLocation
ClassificationScheme	DocumentModel, RecordsPlan
TransferType	TransferRecord, TransactionRecord
TriggerType	DispositionTrigger, RetentionTrigger, StorageMigrationTrigger, TransferTrigger

Table 4.4: Remaining hierarchy of the DataDomain ontology.

4.5.3 IGServicesDomain Ontology

The *IGServiesDomain* includes a list of *Services* and represents their hierarchy. A *Service* is provisioned by a *ServiceProvider* and has to expose certain *Capabilities* such to satisfy the *ServiceConsumer* needs. Every service has *ServiceAgreements* to outline conditions between the *ServieProvider* and the client*ServiceConsumer*. The *ServiceLevelAgreement* defines specific service quality levels that are expected from the service provider and includes metrics to measure *ServiceQuality*. A *ServicePattern* represents re-useable templates of different services organized in different categories. *IGServiesDomain* is a hierarchical representation of the domain, and its concepts are presented in Table 4.5. All services that are subclasses of the *IGServiceDomain* in the first order represent services with different objectives, which again can contain multiple sub-services where the individual focus can be obtained from its name. The top services include the following:

- *RecordsManagementService*, which focuses on managing and maintaining records.
- *StorageService*, which focuses on storing data to ensure GDPR principles such as availability and integrity.
- *LegalCaseManagementService*, which focuses on managing data for legal cases such as evidence or legal documents.
- *RegulatoryService*, which focuses on services that guide information management through regulatory compliance.
- *InformationRetrievalService*, which focuses on services to effectively retrieve information.
- *ContentManagementService*, which focuses on managing digital content such as multimedia files.
- *RenderingService*, which focuses on converting information into different formats.
- *eDiscoveryService*, which focuses on the discovery of digital content for legal matters.
- *UtilityService*, which focuses on basic utility services such as networking.

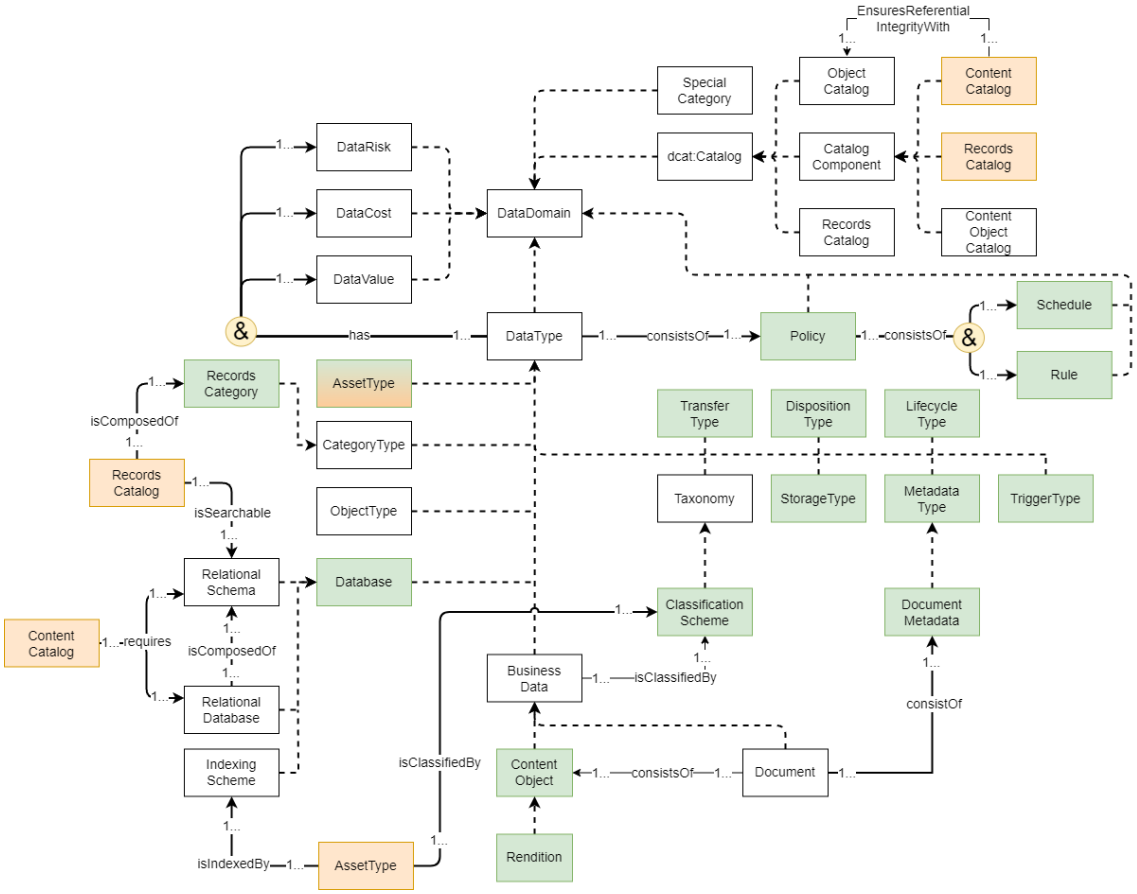


Figure 4.8: Visualization of the DataDomain ontology.

- *DocumentManagementService*, which focuses on document management within their different lifecycle stages.
- *DataService*, which focuses on services directly related to data, such as analysis.

Some services include multiple sub-services, where each focus can be read from its name.

Superclass	Subclass
IGServiceDomain	Capability, RecordsManagementService, ServicePattern, StorageService, LegalCaseManagementService, ServiceAgreement, dctype:Service, ServiceProvider, RegulatoryService, InformationRetrievalService, **ContentManagementService**, InformationManagementService, ServiceLevelAgreement, RenderingService, Service, eDiscoveryService, UtilityService, DocumentManagementService, ServiceQuality, DataService
ContentManagementService	AccessService, TransferService, LoadService, ArchiveService, ManageService, QueryService, AdministerService, SecureService, CollectService, AuditService, StoreService, ControlService, SearchService
DataService	DatabaseFunction, DatabaseOperation, DataManagementSupervision
eDiscoveryService	**ServerDiscoveryService**
ServerDiscoveryService	**DataDiscoveryService**
DataDiscoveryService	DataDiscoveryRequest, DataDiscoveryResult
RecordsManagementService	ClassifyService, HoldService, RetainService, ManagedRecordService, PrivacyService, DisposeService, eDiscoverService, ProtectService
ServiceLevelAgreement	Archive, Gold, Bronze, Silver, Platinum
ServiceProvider	CloudServiceProvider
ServiceQuality	arc:Availability, arc:Scale, arc:Elasticity, arc:Backup, arc:Restore, arc:DynamicScale, arc:DisasterRecovery, arc:LoadBalancing, arc:Recovery, arc:Resiliency, arc:HighAvailability
UtilityService	ControlType, DataType

Table 4.5: Hierarchy of the IGServiceDomain ontology.

4.5.4 ImplementationDomain Ontology

The *ImplementationDomain* contains a hierarchical description of several implementation components for IG, as presented in Table 4.6. *EnterpriseContentManagement* refers to components for managing and implementing the organizational aspects of Enterprise Content Management (ECM), such as *Archive, Case, Content, Document, eDiscovery* and *Records*. *InformationManagement* focuses on components for implementing information-related processes, such as *Analysis*, *Business-Intelligence*, and *InformationRetrieval*. Similar to other domains, the *InformationPattern* describes

standardized methods within its domain. Because IG does not only consist of processes but also the collaboration of all resources, *InteractiveResource* represents external tools for interactive engagement with data resources, employees, and the system.

Superclass	Subclass
ImplementationDomain	**DataManagement**, **WorkflowManagement**, **InformationManagement**, **EnterpriseContentManagement**, ImplementationPattern, EnterpriseInformationManagementPortal, ITGovernance
DataManagement	DataAssessment, DataClassification, DataCollection
ImplementationDomain	EnterpriseContentManagement, InformationManagement, ImplementationPattern, InteractiveResource
EnterpriseContentManagement	ArchiveManagement, RecordsManagement, eDiscoveryManagement, DocumentManagement, CaseManagement, **ContentManagementComponent**
ContentManagementComponent	MetadataManagement, SearchAndRetrievalManagement, TransactionManagement, RepositoryManagement
DocumentManagement	DocumentRelatedBusinessCases
InformationManagement	**AnalyticsImp**, InformationRetrieval, BusinessIntelligenceManagement, BusinessIntelligence, OperationalSupportSystems
AnalyticsImp	AI, MachineLearning, KnowledgeBase
WorkflowManagement	ContentCentricWorkflow, WorkflowProcess

Table 4.6: Hierarchy of the ImplementationDomain ontology.

4.5.5 OrganizationDomain Ontology

The OrganizationDomain Ontology is similar to the Organization ontology described in Section 4.3. Even though these ontologies are similar, the implementation domain needs the addition of the classes around *Occurrence*, as illustrated in Table 4.7. Occurrence refers to certain actions, events, or triggers that bridge the concepts of organization and implementation. *Organizations* may perform some *Actions* to achieve their objectives or initiate certain processes. *AssessmentEvents* represent evaluation initiatives such as audits or reviews. Certain events can also be bound to their specific *LifecycleOccurence*. Events that may involve more critical lifecycle stages could involve more critical processes. Organizations also can *Request* certain information. Finally, the *TriggeringEvent* implies events that initiate or trigger certain processes. For instance, a request could trigger processes to retrieve certain records from a database.

Superclass	Subclass
OrganizationDomain	Occurrence
Occurrence	Action, AssessmentEvent, LifecycleOccurrence, Request, TriggeringEvent
Action	CorporateAction, RegulatoryAction

Table 4.7: Hierarchy of Occurrence in the OrganizationDomain ontology.

4.5.6 PlatformDomain Ontology

The *PlatformDomain* Ontology describes various aspects related to platform services and their architecture. The hierarchy is described in Table 4.8. These patterns are derived from Fehling et. al. [FLR+14]. We will not discuss every pattern in detail, as only the hierarchy and conceptualization of these patterns are important. To further explain each cloud computing aspect, please refer to their contribution.
The first distinction is made between the concepts of *CloudComputingPattern*, representing standardized design matters in cloud computing, and *CloudPlatform*, representing some existent cloud platforms such as *AmazonCloud, GoogleCloud, IBMCloud, MicrosoftCLoud*, and *RadhatCloud*. A complete *CloudComputingPattern* includes the following:

- *CloudApplicationArchitecturePattern*, which describes the architecture of cloud applications.
- *CloudApplicationManagementPattern*, which describes methods for cloud computing management.
- *CloudComputingFundamental*, which describes fundamental characterization of the cloud.
- *CloudOfferingPattern*, which describes a set of technical functionalities a cloud can provide.
- *CompositeCloudApplicationPattern*, which describes more complex cloud application characteristics.

In this use case, a *KubernetesPattern* is additionally implemented, as the IGONTO framework uses one. *CloudApplicationArchitecturePattern* is further divided into four subclasses. The *CloudApplicationComponent* contains individual building blocks needed to implement a cloud application. These building blocks need to be integrated with each other and the system and are represented in the *CloudIntegration*. Each cloud application has one or more core concepts, represented in *FundamentalCloudArchitecture*. Because cloud applications serve multiple customers (tenants) at once, the needed components are described in *Multi-Tenancy*.
A *CloudApplicationManagementPattern* can be divided into a *ManagementComponent*, including functionalities and tools for efficient cloud management, and *ManagementProcess*, involving the required processes for effective cloud management.
The *CloudComputingFundamental* characterizes the type of workload that is performed on cloud applications, represented in *ApplicationWorkload*. A cloud application can be deployed in different ways. The different models are conceptualized in *CloudDeploymentModel*. The implemented service offerings of a cloud application are implemented in the *CloudServiceModel*.
CloudEnvironment, CommunicationOffering, ProcessingOffering and *StorageOffering* describe different aspects within the *CloudOfferingPattern*. The *CloudEnvironment* specifies the cloud

infrastructure, and each variable is associated with availability, scalability, and flexibility. The communication used within a cloud application is detailed in *CommunicationOffering*. Since cloud application demands can vary, different *ProcessingOfferings* are used to enhance processing performance. Since different demands also affect the way we store data, aspects of the data storage are summarized in *StorageOffering*.
These patterns can also be combined to define new patterns. Fundamental compositions are described in *NativeCloudApplication*, whereas *HybridCloudApplication* details solutions in the case of the distribution of application components.

Superclass	**Subclass**
PlatformDomain	**CloudComputingPattern**, **CloudPlatform**
CloudComputingDomain	**CloudApplicationArchitecturePattern**, **CloudOfferingPattern**, KubernetesPattern, **CloudApplicationManagementPattern**, **CompositeCloudApplicationPattern**, **CloudComputingFundamental**
CloudPlatform	AmazonCloud, IBMCloud, GoogleCloud, RadhatCloud, MicrosoftCloud
CloudApplication ArchitecturePattern	**CloudApplicationComponent**, **FundamentalCloudArchitecture**, **CloudIntegration**, **Multi-Tenancy**
CloudApplication ManagementPattern	**ManagementComponent**, **ManagementProcess**
CloudComputing Fundamental	**ApplicationWorkload**, **CloudDeploymentModel**, **CloudServiceModel**
CloudOfferingPattern	**CloudEnvironment**, **CommunicationOffering**, **StorageOffering**, **CloudPlatformService**, **ProcessingOffering**
CompositeCloud ApplicationPattern	**HybridCloudApplication**, **NativeCloudApplication**
CloudApplication Component	BatchProcessingComponent, DataAbstractor, Timeout-BasedMessageProcessor, Multi-ComponentImage, Transaction-BasedProcessor, StatefulComponent, ProcessingComponent, UserInterfaceComponent, DataAccessComponent, IdempotentProcessor
CloudIntegration	ApplicationComponentProxy, IntegrationProvider, CompliantDataReplication, RestrictedDataAccessComponent, MessageMover
FundamentalCloud Architecture	DistributedApplication, LooseCoupling
Multi-Tenancy	LooseCoupling, SharedComponent, Tenant-IsolatedComponent
ManagementComponent	ElasticityManager, ElasticLoadBalancer, ManagedConfiguration, ProviderAdapter, ElasticQueue, Watchdog

Continued on next page

Table 4.8 continued from previous page

Superclass	Subclass
ManagementProcess	ElasticityManagementProcess, ResiliencyManagementProcess, StandbyPoolingProcess, UpdateTransitionProcess, FeatureFlagManagementProcess
ApplicationWorkload	ContinuouslyChangingWorkload, Once-in-a-LifetimeWorkload, UnpredictableWorkload, StaticWorkload, PeriodicWorkload
CloudDeploymentModel	CommunityCloud, HybridCloud, PublicCloud, PrivateCloud
CloudServiceModel	Infrastructure-as-a-Service, Software-as-a-Service, Platform-as-a-Service
CloudEnvironment	ElasticInfrastructure, EnvironmentBasedAvailability, ElasticPlatform, Node-BasedAvailability
CommunicationOffering	VirtualNetworking
ProcessingOffering	ExecutionEnvironment, MapReduce, Hypervisor
StorageOffering	BlobStorage, BlockStorage, EventualConsistency, StorageSystem, KeyValueStore, RelationalDatabase, StrictConsistency
HybridCloudApplication	HybridApplicationFunction, HybridBackend, HybridBackup, HybridUserInterface, HybridProcessing, HybridData
NativeCloudApplication	ContentDistributionNetwork, Three-TierCloudApplication, Two-TierCloudApplication

Table 4.8: Hierarchy of the PlatformDomain ontology.

4.5.7 RegulatoryDomain Ontology

The *RegulatoryDomain* ontology presented in Figure 4.9 is similar to the *Jurisdiction*, described in Section 4.1. Therefore, only the description of new classes is going to be considered. The implementation domain requires a more detailed conceptualization of the regulatory domain. Everything *isRelatedTo* a *Regulation*. Because this relationship is implemented at the top level, its subclasses inherit this DL. An *Activity* is a specific operation, such as the *DataOperation*, which *isDefinedBy* the *GDPR*, *EconomicActivity* and *LegalActivity*. Each *Activity isRegulatedBy* a *Regulation*. The formal representation of regulations is realized by the *Law*. These regulations describe *Duties*, which is a more general concept that includes certain types of obligations. The representation that organizations or persons are subject to these regulations is represented by *LegalEntity*. *Penalty, Privacy-firendlySetting, SecurityMeasure, Violation* and *VitalInterest* are additional keywords describing regulatory aspects, which may be useful for certain solutions. The *Obligations* and *Rights* in the *Jurisdiction* ontology 4.1 are further specified in *RegulatoryRequirementType, Obligation*, and *Right*, whereas the *RegulatoryRequirementType* is equivalent to the *RegulatoryRequirement* class in the RequirementDOmain ontology. Additionally, each process can be described regarding its purpose and characteristics of the regulatory aspects with *DataProcessingType*, where *PersonalDataPRocessing* is a specification for personal data. Each process *requiresConsent* from the *DataSubject* and *requiresComplianceWith* the *GDPR*. The GDPR *definesPrinciplesFor* the *DataSubjectRight*, *DesirableBehavior* and *DataControllerObligation*,

while being a subclass of *DataProcessorObligation* in combination with the *DataControllerObligation*, as the data controller determines how the data processor should process personal data. The complete hierarchy of all green-highlighted classes is listed in Table 4.9.

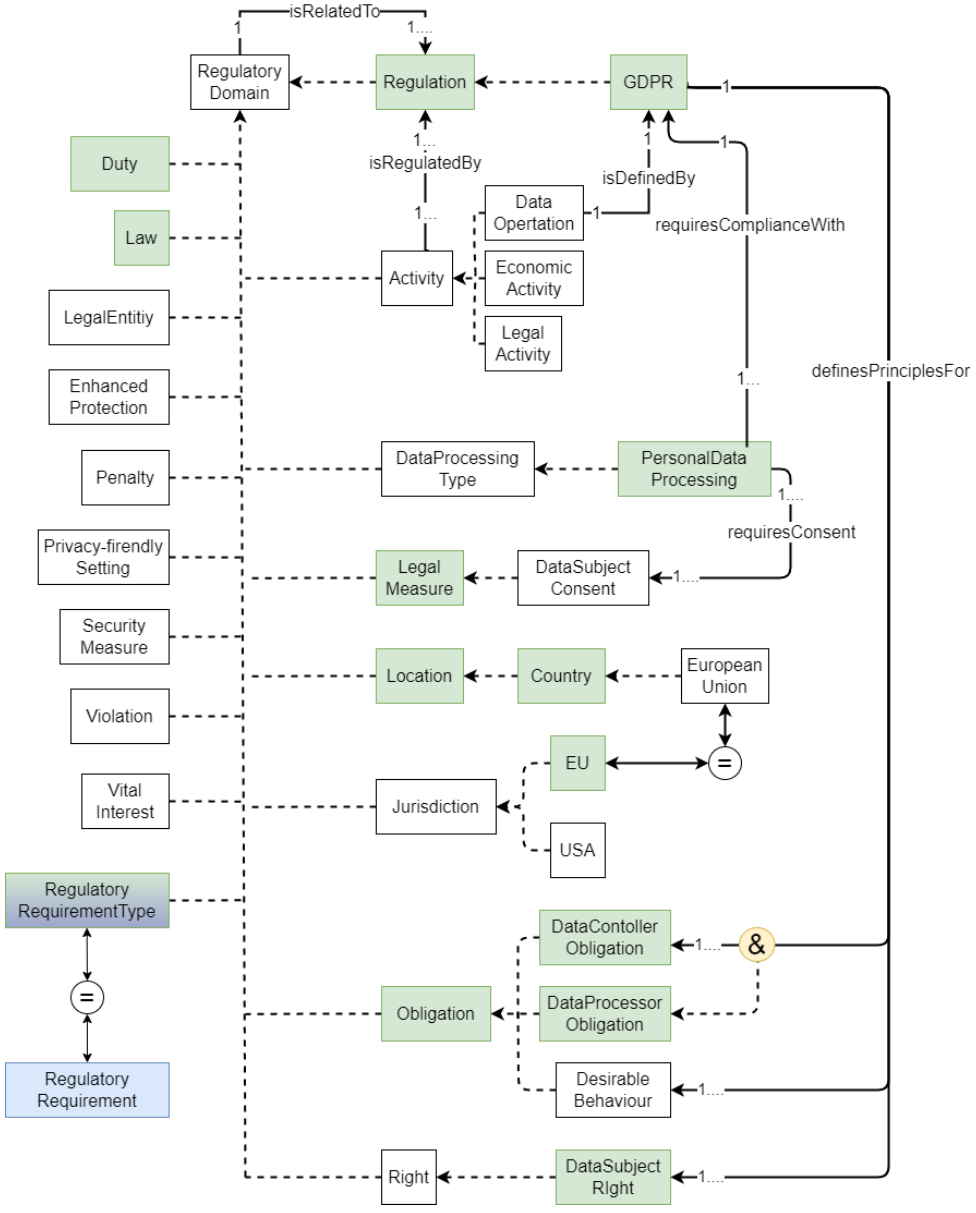


Figure 4.9: Visualization of the RegulatoryDomain ontology hierarchy.

Superclass	**Subclass**
Duty	Compensation, Consistency, LegalObligation, **Violation**, Responsibility, VitalInterest
Violation	PrivacyBreaches

Continued on next page

Table 4.9 continued from previous page

Superclass	Subclass
PersonalDataProcessing	PersonalDataTransfer, ProcessingWithFairness, ProcessingPurpose, ProcessingWithLawfulness, PurposeOfProcessing, ProcessingWithTransparency, ProcessingWithConfidentiality, ProcessingWithDataMinimization, ProcessingWithAccuracy, ProcessingWithStorageLimitation, ProcessingWithIntegrity
EU	Germany, Italy
Law	LawfulBasis, StatuteLaw
LegalEntity	**Association**, **DataProcessor**, **DataController**, DataOwner, NaturalPerson, LegalPerson, DataSubject
DataProcessor	SubProcessor
LegalMeasure	**Association**, **DataProcessor**, **DataController**, DataOwner, NaturalPerson, LegalPerson, DataSubject, SubProcessor, Claim, SecurityMeasure, RestrictionMeasure, Cooperation, Penalty, Privacy-friendlySetting, LegalRecourse, Liability, DataSubjectConsent
Location	BackupLocation, org:GeographicRegion, RemoteLocation, StorageLocation, **Country**
Country	**EuropeanUnion**, USA
Obligation	**DataControllerObligation**, ObligationToAppointDataProtectionOfficer, ObligationToNotifyOfPersonalDataBreach, **DataProcessorObligation**, DesirableBehavior, ObligationToPerformDataProtectionImpactAssessment
DataProcessorObligation	ObligationToCaptureRecordsOfDataProcessingActivity, ObligationToFairnessDataProcessing, ObligationToLawfulnessDataProcessing, ObligationToTransparencyOfDataProcessing
DataControllerObligation	ObligationOfCross-borderDataTransfer, ObligationToCooperateWithSupervisoryAuthority, ObligationToEnsureConfidentialityOfData, ObligationToEnsureAccuracyOfData, ObligationToEnsureAccountabilityOfProcessor, ObligationToLimitPurposeDataProcessing, ObligationToFulfillConsentRequirement, ObligationToLimitStoragePeriod, ObligationToMinimizeData, ObligationToEnsureAccountabilityOfDataController, ObligationToEnsureIntegrityOfData
Regulation	DSGVO, GDPR, USPrivacyAct

Continued on next page

Table 4.9 continued from previous page

Superclass	Subclass
RegulatoryRequirementType	RR.AppointDataProtectionOfficer, RR.SupportEnterpriseWithRemoteLocation, RR.RightToErasure, RR.LimitPurposeDataProcessing, RR.ImplementManageDataSubjectConsent, RR.EnsureConfidentialityOfData, RR.NotifyOfPersonalDataBreach, RR.EnsureLawfulnessDataProcessing, RR.SupportEnterpriseWithBackupLocation, RR.MinimizeDataSubjectData, RR.EnsureAccuracyOfData, RR.SupportLegitimateInterest, RR.RightToBeForgotten, RR.RightToObject, RR.RightToAccess, RR.EnsureFairnessDataProcessing, RR.EnsureAccountabilityOfDataProcessor, RR.CaptureRecordsOfDataProcessingActivity, RR.RightToDataPortability, RR.FulfillConsentRequirement, RR.RightToRectification, RR.SupportEnterpriseWithMainEstablishment, RR.ImplementDataProcessingRestrictionMeasure, RR.ImplementLegalRecourse, RR.EnsureTransparencyOfDataProcessing, RR.LimitStoragePeriod, RR.PerformDataProtectionImpactAssessment, RR.RightRelatedToAutomatedProfiling, RR.CooperateWithSupervisoryAuthority, RR.RightRelatedToAutomatedDecision-making, RR.EnsureAccountabilityOfDataController, RR.EnforceDesirableBehavior, RR.Cross-borderDataTransfer, RR.EnsureIntegrityOfData, RR.RightToBeInformed, RR.ImplementSecurityMeasure, RR.RightToRestrictProcessing, RR.SupportEnterpriseWithStorageLocation
DataSubjectRight	LegitimateInterest, RightToRestrictProcessing, RightToBeForgotten, RightToObject, RightToBeInformed, RightRelatedToAutomatedProfiling, RightRelatedToAutomatedDecision-making, RightToErasure, RightToAccess, RightToRectification, RightToDataPortability

Table 4.9: Hierarchy of the remaining RegulatoryDomain ontology.

4.5.8 RequirementDomain Ontology

The *RequirementDomain* Ontology extends the functional and non-functional requirements from the *Jurisdiction* ontology explained in Section 4.1. In addition, the equated *RegulatoryRequirementTypes* from the *RegulatoryDomain* are part of it. These requirements for the implementation solution are more detailed, and their hierarchy is listed in Table 4.10. These requirements line up with certain processing and service names that have already been discussed in this thesis and represent the need for their implementation.

Superclass	**Subclass**
RequirementDomain	**UseCaseScenario**, RegulatoryRequirement, Non-FunctionalRequirement, FunctionalRequirement
FunctionalRequirement	FR.Access, FR.Record, FR.eDiscover, FR.ReferentialIntegrity, FR.Store, FR.Manage, FR.Query, FR.TransactionSupport, FR.Protect, FR.Privacy, FR.Audit, FR.Retain, FR.Classify, FR.Transfer, FR.Collect, FR.Control, FR.Replicate, FR.Administration, FR.Dispose, FR.Archive, FR.Secure, FR.Hold, FR.Search, FR.Load
Non-FunctionalRequirement	**NFR.Availability**, NFR.Backup, NFR.DisasterRecovery, NFR.Resiliency, NFR.Recovery, NFR.Elasticity, **NFR.Scale**, NFR.Restore, NFR.LoadBalancing, **NFR.ServiceQuality**, **NFR.TransactionSupport**
NFR.Availability	NFR.HighAvailability
NFR.Scale	NFR.DynamicScale
NFR.ServiceQuality	NFR.ServiceLevelAgreement
NFR.TransactionSupport	NFR.ReferentialIntegrity, NFR.TwoPhaseCommit
UseCaseScenario	UseCase, UserStory

Table 4.10: Hierarchy of the RequirementDomain ontology.

4.5.9 TOSCA Ontology

Topology and Orchestration Specification for Cloud Applications (TOSCA) is an open cloud standard for defining the topology and orchestration of cloud applications. TOSCA was developed by the Organization for the Advancement of Structured Information Standards (OASIS) and initially published in 2013. TOSCA uses Domain Specific Language (DSL) to define interoperable descriptions of applications, thereby enabling portability and automated cloud management, expanding customer choice, improving reliability, and reducing cost. The initial publication was implemented in XML, and 2017, it was enhanced with a YAML syntax to enhance readability, accessibility, and conciseness. Every piece of information regarding TOSCA is retrieved from its official YAML documentation [OAS19]. Because TOSCA is a wide domain, only superficial aspects are going to be explained to understand the implemented hierarchy. To further explain each TOSCA aspect, the reader is encouraged to refer to the documentation. The TOSCA ontology is required in

the post-processing of IGONTO (see Figure 2.1) and therefore its vocabulary and hierarchy are implemented in this ontology.
TOSCA's main parts are topology and orchestration. The topology describes the application structure, whereas the orchestration illustrates the deployment and management of the application. Figure 4.10 describes the hierarchy of the TOSCA ontology.

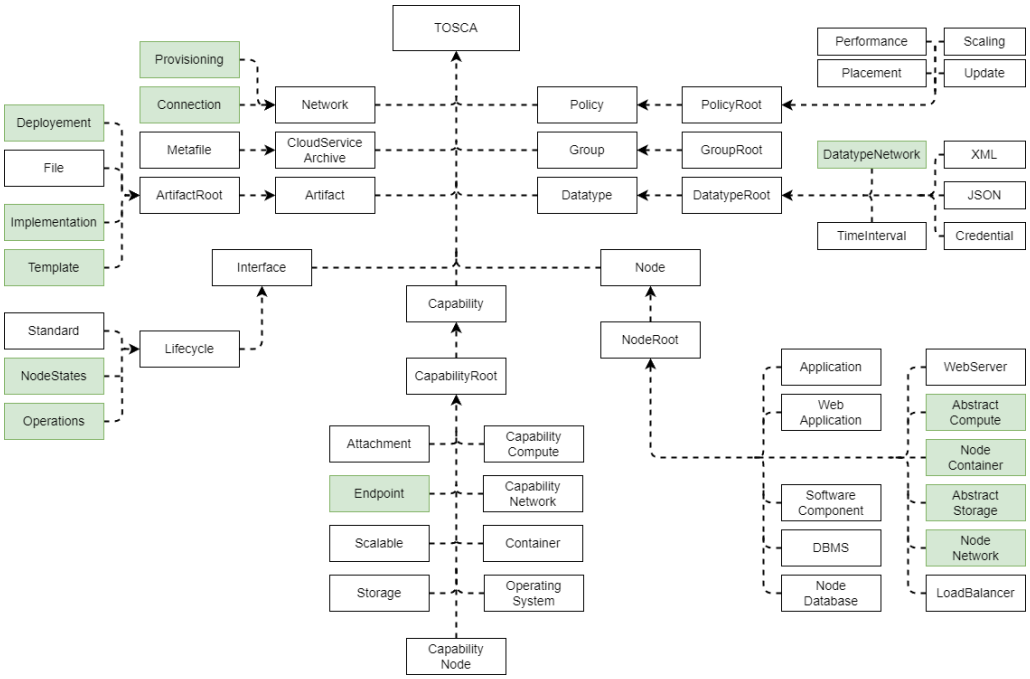


Figure 4.10: Visualization of the TOSCA ontology.

TOSCA defines Node templates and Relationship templates, which are the most important building blocks of the topology. Because the relationship templates are implemented to connect different nodes within TOSCA, they are implemented as object properties in this ontology. TOSCA defines *Root* types for multiple concepts for portability, and therefore they are also implemented as classes within the ontology. *Nodes* are the primary elements in the TOSCA topology and describe individual components in the topology structure. These components are semantically further divided into *Application, WebApplication, SoftwareComponent, DBMS, NodeDatabase, WebServer, AbstractCompute, NodeContainer, AbstractStorage, NodeNetwork,* and *LoadBalancer*. Each one of them describes their respective component and can be *Grouped*. *DataTypes* describe complex datatypes with the node, such as *XML, JSON, Credential, TimeInterval,* and *DatatypeNetwork*. Because some vocabulary, such as "DatatypeNetwork" are used multiple times in TOSCA, a prefix has to be added to ensure unique identifiers. In addition, some nodes in TOSCA can be implemented as abstracts and referenced in other templates. Nodes can be further described by adding *Capabilities* to them. These include the following:

- *CapabilityNode*, which describes the basic capabilities of the node.
- *CapabilityNetwork*, which describes that the node can provide addressability.

- *CapabilityCompute*, which describes that the node can provision hosting on a compute resource.
- *Storage*, which describes that the node can provide a storage location.
- *Endpoint*, which extends the network capability and includes information about the endpoint.
- *Attachment*, which defines the attachment capability of a node. *OperatingSystem*, which describes capabilities regarding the used operating system within the node. *Scalable*, which describes the scalability capability of the node. *Container*, which describes that the node can contain or host other nodes.

Artifacts specify packages and files used in *Nodes*. These include representative artifacts regarding *Deployement, Files, Implementation*, and *Templates*. The *Network* class models the network connectivity semantics within TOSCA. *Connection* networks represent semantics that need to be implemented to use the target services, and *Provisioning* describes the logical model for managing network components and services. These elements constitute the TOSCA topology template.
Multiple topology templates can be summarized into a service template, which serves as a reusable container and provides the orchestration with the needed information. *Policies* are used within the service templates to guide node management regarding *Performance, Scaling, Placement,* and *Update*. The *Interface* defines the node *Lifecycle* and possible *Relationships*. *Lifecycle* includes possible node *Operations* that, if executed, lead to different *NodeStates*. In *Standard*, predefined and standardized concepts, operations, and practices are implemented. Table 4.11 illustrates the remaining classes within the TOSCA ontology. Based on the application requirements defined in the service template and the infrastructure capabilities of the service provider, the orchestration decides what cloud environment is optimal for the TOSCA application.

Superclass	**Subclasses**
Provisioning	ControllingNetworkFulfillment, DeclarativeNetworkProvisioning, ImplicitNetworkProvisioning
Connection	PeerToPeer, SourceToTarget, TargetToSource
Deployment	**Image**
Image	VM
Implementation	Bash, Python
Template	Ansible, Helm, DockerCompose, Kubernetes
Endpoint	Admin, Public, Database
DatatypeNetwork	NetworkInfo, PortInfo, PortDef, PortSpec
NodeStates	Error, Creating, Deleting, Starting, Stopping, Initial, Configuring, Created, Configured, Started
Operations	Delete, Stop, Start, **Configure**, Create
Configure	RemoveTarget, RemoveSource, PostConfigSource, AddTarget, AddSource, PreConfigTarget, PostConfigTarget, PreConfigSource
NodeContainer	NodeContainerApplication, NodeContainerRuntime
NodeNetwork	**NodeNetworkTypes**

Continued on next page

Table 4.11 continued from previous page

Superclass	Subclasses
NodeNetworkTypes	BindsTo, Linkable, Port, LinksTo
AbstractCompute	NodeCompute
AbstractStorage	Blockstorage, ObjectStorage

Table 4.11: Hierarchy of the remaining TOSCA ontology.

4.5.10 PractitionerDomain, StandardsDomain, SystemsDomain and VendorDomain Ontology

Because the last domains are rather small and only describe the taxonomy of their domain, all three are included in Table 4.12. The practitioner domain lists various agencies, associations, and the concept of maturity level. The standards domain represents a list of several ISO standards. These standards provide standardized best practices and are important for providing crucial knowledge for implementing certain processes. The SystemsDomain includes several concepts of IT infrastructure, each summarizing specific processes and functionalities. Finally, the VendorDomain represents vendors, where companies provide certain products and services that can be integrated to implement certain solutions.

Domain	Subclasses
PracticionerDomain	**Agency**, **Association**, **MaturityIndex**
Agency	CRL, NARA, PROV
Association	**ARMA**, CGOC, DAM, DAMA, EDRM, ISACA
ARMA	Accountability, Transparency, Disposition, Availability, Retention, Compliance, Protection, Integrity
MaturityIndex	MI-Level-1, MI-Level-2, MI-Level-3, MI-Level-4, MI-Level-5
StandardsDomain	ISOStandards
ISOStandards	I**SO14000**, ISO17000, **ISO15000**, **ISO27000**, ISO23081, ISO20000
ISO14000	ISO14271
ISO15000	ISO15489, ISO15939
ISO27000	ISO27001:2019, ISO27701:2022, ISO27013:2021
SystemsDomain	**System**

Continued on next page

Table 4.12 continued from previous page

Superclass	Subclasses
System	ArchiveSystem, CaseManagementSystem, InformationRetrievalSystem, NetworkSystem, eDiscoveryManagementSystem, RecordsManagementSystem, DocumentManagementSystem, StorageSystem, EnterpriseContentManagementSystem, RDBMSystem, SystemPattern, CommunicationSystem, TransferSystem, RenderingSystem
VendorDomain	Alfresco, IBM, **Microsoft**, **Google**, Vendor, **Redhat**, Opentext, **Amazon**, **CloudNativeFoundationComputingFoundation**, Oracle
Amazon	AWS
CloudNativeFoundationComputing Foundation	CNCF, Kubernetes
Google	GCP
Microsoft	Azure
Redhat	Openshift

Table 4.12: Hierarchy of the remaining Domains ontology.

4.6 Semantic Web Rule Language (SWRL) Rules

This section describes all implemented SWRL rules. There are two different levels of how these rules are implemented. Every rule is first implemented as a Horn clause, which is then translated into semantic and syntactically coherent OWL syntax based on the OWL RDF/XML exchange syntax. Thus, the implementation of SWRL rules is made easier with Horn clauses while simultaneously providing expressive power [HPBT05]. The first rule will include a more detailed description, including the OWL translated code to give an impression of the appearance of the OWL code, whereas the others are only explained with the Horn clauses, as the translated code is often large and more difficult to read.

4.6.1 Jurisdiction Rules

The jurisdiction rules primarily deal with inferences regarding *Capabilities, Obligations, Rights* and *Requirements*. The first rule connects the capabilities with the obligations. As mentioned in Section 4.1, every article relates to obligation and capabilities. The first SWRL rule, 4.6.3, connects capabilities with obligations that involve the same articles.

Rule: S1

capabilityInvolvesArticle(?cap, ?article) ∧
obligationInvolvesArticle(?obligation, ?article)

→ **capabilityInvolvesObligation**(?cap, ?obligation)

The first two clauses take the relationships *capabilityInvolvesArticle* and *obligationInvolvesArticle* as a requirement for the rule. Variables are denoted with a "?" at the beginning of the Horn clauses. Thus, *?cap* represents the set of instances that are the subjects of the relationship, and *?article* represents the set of objects to which the subjects are connected. Since every relationship in our ontologies has only one domain and one range, as described in Section 3.1.2, we do not need further specifications. If this were not the case, we would need an additional *Capability(?cap)* horn clause to state that the subjects should only be instances of the class *Capability*. The same would go for every other variable. This keeps the length of each Horn clause to a minimum and, also, the translated OWL code. The same principle applies to the *obligationInvolvesArticle* relationship, where *?obligation* is the set of obligation instances and *?article* is the set of articles to which each obligation is connected.
The requirements of the rule and the resulting inferences are divided by an →. In this inference, we state that the set of capabilities and obligations that are connected to the same articles should have the relationship *capabilityInvolvesObligation*, where the capabilities are the subject and the obligations are the object.
Figure 4.6.1 illustrates the translated OWL code in Turtle format. The first lines of the implementation include:

Line 1-4: All namespace prefixes to reduce the rule length.

Line 6: Information about whether the SWRL rule is enabled or not and, therefore, whether it is active or not.

Line 7: Optional explanation of this rule in the form of a comment.

Line 8: The label or identifier.

Line 9: The declaration of the rule type.

The actual implementation of this rule begins at line 10. The *swrl#body* section from lines 10-16 describes the antecedent (if part) of the rule. It consists of an *AtomList*, a list of conditions for the rule to apply in the first place. Lines 11-16 check, whether a *capabilityInvolvesArticle* relationship exists between *cap* and *article*. Lines 17-23 check the second condition (*obligationInvolvesArticle*). The code *rdf:rest rdf:nil* at line 24. describes a special rdf resource and represents an empty list, which is used to indicate the end of the conditions.

```
@prefix swrla.owl:   <http://swrl.stanford.edu/ontologies/3.3/swrla.owl#> .
@prefix swrl:   <http://www.w3.org/2003/11/swrl#> .
@prefix igonto:   <http://www.semanticweb.org/igonto/igonto#> .
@prefix jurisdiction:   <http://www.semanticweb.org/igonto/jurisdiction#> .

[  swrla.owl:isRuleEnabled "true"^^xsd:boolean ;
   rdfs:comment "" ;
   rdfs:label "S1" ;
   rdf:type swrl:Imp ;
   swrl:body [ rdf:type swrl:AtomList ;
                           rdf:first [ rdf:type swrl:IndividualPropertyAtom ;
                                       swrl:propertyPredicate
                                       jurisdiction:capabilityInvolvesArticle ;
                                       swrl:argument1 igonto:cap ;
                                       swrl:argument2i igonto:article
                                     ] ;
                           rdf:rest [ rdf:type swrl:AtomList ;
                                      rdf:first [ rdf:type swrl:IndividualPropertyAtom ;
                                                  swrl:propertyPredicate
                                                  jurisdiction:obligationInvolvesArticle ;
                                                  swrl:argument1 igonto:obligation ;
                                                  swrl:argument2 igonto:article
                                                ] ;
                                      rdf:rest rdf:nil
                                    ]
                         ] ;
   swrl:head [ rdf:type swrl:AtomList ;
                           rdf:first [ rdf:type swrl:IndividualPropertyAtom ;
                                       swrl:propertyPredicate
                                       jurisdiction:capabilityInvolvesObligation ;
                                       swrl:argument1 igonto:cap ;
                                       swrl:argument2 igonto:obligation
                                     ] ;
                           rdf:rest rdf:nil
                         ]
 ] .
```

Listing 4.1: SWRL Rule S1 OWL implementation.

If the conditions are met, *swrl#head* from lines 27-33 describes the consequence and the inferred rule that needs to be applied. Line 34 ends the head, similar to line 24.
Although the rule is concise, the implementation is extensive. Larger rules will consequently result in even larger OWL translations. Therefore, the following rules in this chapter are explained only through Horn clauses.

Rule: S2

capabilityInvolvesArticle(?cap, ?article) ∧
rightInvolvesArticle(?right, ?article)

→ **capabilityInvolvesRight**(?right, ?obligation)

Rule: S3

capabilityInvolvesObligation(?cap, ?obligation) ∧
requirementInvolvesObligation(?req, ?obligation)

→ **capabilityInvolvesRequirement**(?cap, ?req)

The same principle from *Rule S1* is applied to *Rule S2*. The difference is that S2 is about the connection to *Rights* rather than *Obligations*. It is also possible to include the outcomes of some rules as input in other rules. For instance, *Rule S3* connects *Capabilities* with *Requirements*, based on the result from Rule *S1*. In theory, these rules can be included in only one large SWRL rule. However, just as large methods in common software decrease software quality, large SWRL rules decrease understandability and manageability in ontologies and, therefore, decrease their quality [ROM11]. Related work [HOD10] also implemented visualizations for SWRL rules to improve SWRL tools, thus revealing the downside of large rule sets. Therefore, if possible, SWRL rules should be divided to keep them short.

4.6.2 Organization Rules

Organization rules focus on connecting enterprises with capabilities. Enterprises that are considered "large", such as those with more than 250 employees, have to comply with all GDPR requirements. Therefore, *Rule S4* is a simple rule where every capability gets connected to a large enterprise. Thereafter, further queries can be executed to determine what obligations, rights, and requirements each capability includes based on *Rules S1-S3*.

Rule: S4

IGCapability(?capability) ∧ **LargeEnterprise**(?large)

→ **enterpriseNeedsToFullfillCapability**(?large, ?capability)

If an enterprise has fewer than 250 employees, more specialized rules are needed, which are represented in *Rules S5-S7*. According to GDPR Article 30 (5), enterprises with fewer than 250 employees are exempt from the obligation to maintain records of processing activities and, therefore, do not need a Records Information Management system. Therefore, small enterprises do not need to possess the capabilities that are linked to the RIM and only make inferences about capabilities that are linked to Legal (*Rule S5*), IT (*Rule S6*) or Privacy and Security *Rule S7*.

Rule: S5

processLinkedToOrganizationUnit(?process, ?legal) ∧
capabilityLinkedToProcess(?capability, ?process) ∧ **Legal**(?legal) ∧
Small_Enterprise(?enterprise)

→ **enterpriseNeedsToFullfillCapability**(?enterprise, ?capability)

Rule: S6

processLinkedToOrganizationUnit(?process, ?it) ∧
capabilityLinkedToProcess(?capability, ?process) ∧ **IT**(?it) ∧
SmallEnterprise(?enterprise)

→ **enterpriseNeedsToFullfillCapability**(?enterprise, ?capability)

Rule: S7

processLinkedToOrganizationUnit(?process, ?pas) ∧
capabilityLinkedToProcess(?capability, ?process) ∧
PrivacyAndSecurity(?pas) ∧ **SmallEnterprise**(?enterprise)

→ **enterpriseNeedsToFulfillCapability**(?enterprise, ?capability)

To include the RIM as knowledge within the small enterprise, *Rule S8* was implemented as a negative object property to explicitly express that the small enterprise does not need a RIM, and therefore, the associated capabilities are also not needed. Further, a justification relationship is utilized to directly state which article is responsible for this negative inference.

Rule: S8

RecordsAndInformationManagement(?rim) ∧
capabilityLinkedToProcess(?capability, ?process) ∧
processLinkedToOrganizationUnit(?process, ?rim) ∧
justifiedByArticle(?rim, Article30)(?enterprise) ∧
IGCapability(?capability) ∧ **Process**(?process)

→ **justifiedByArticle**(?capability, Article30) ∧
enterpriseDoesNotNeedOrganizationUnit(?enterprise, ?rim) ∧
enterpriseDoesNotNeedCapability(?enterprise, ?capability) ∧
justifiedByArticle(?rim, Article30)

4.6.3 Solution Rules

The solution rules connect the enterprise use cases with the correct solution implementation. In our use case, other solutions are inferred depending on whether the enterprise needs an RIM. As illustrated in *Rule S9*, a large enterprise is connected to the information governance solution, which includes every possible solution, including the RIM, eDiscovery, and Content Management solutions.
Rule S10 describes the solution connection for small enterprises. Because they do not need a RIM, small enterprises only need to implement a basic ECM and archive solution that relates to an ECM reference model, that represents an architecture solution pattern. Similar to *Rule 8*, a negative object property is implemented to state that small enterprises do not need a RIM solution.

Rule: S9

LargeEnterprise(?enterprise) ∧ **InformationGovernanceSolution**(?infog)
∧ **SolutionPattern**(?pattern)

→ **needsSolution**(enterprise, ?infog) ∧ **needsSolution**(?enterprise, ?pattern)

Rule: S10

SmallEnterprise(?enterprise) ∧
RecordsAndInformationManagement(?rim) ∧
enterpriseDoesNotNeedOrganizationUnit(?enterprise, ?rim) ∧ **ECMSolution**(?ecm) ∧
ReferenceModel(?pattern) ∧ **ArchiveSolution**(?archive) ∧ **RIMSolution**(?rimSol)

→ **needsSolution**(?enterprise, ?ecm) ∧
needsSolution(?enterprise, ?pattern) ∧
needsSolution(enterprise, ?archive) ∧
enterpriseDoesNotNeedSolution(?enterprise, ?rimSol)

5 Evaluation

The evaluation covers several aspects of the IGONTO prototype. Section 5.1 presents the results of several SPARQL queries and demonstrates the functional correctness of the prototype in relation to the IGONTO concept. The performance of these queries is discussed in Section 5.2. Moreover, Section 5.3 explains the technique used for validating IGONTO, thus ensuring its consistency. The potential for further development of IGONTO, along with its design decisions, is demonstrated in Section 5.4.

5.1 Use-Case Scenario

To demonstrate and prove how enterprises of varying sizes can achieve GDPR compliance, we developed a typical use case scenario and a set of competency questions to test our IGONTO prototype. In this section, we will iterate through various competency questions formulated through SPARQL queries and will include visual representations of the query results in Gruff. The actors for this use case consist of the following concepts:

1. **Small Enterprise:** Represent the set of companies with fewer than 250 employees.
2. **Large Enterprise:** Represent the set of companies with more than 250 employees.

Besides the employee sizes, both sets of enterprises belong to the same jurisdiction and operate in the same country. To shorten the query length, certain PREFIXES are included in every query, which abbreviates the namespaces of every ontology. Without these prefixes, the entire namespace would have to be included for every class, relationship, and attribute, which increases the query readability.
Listing 5.1 illustrates the prefixes which are included in every query.

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sh:    <http://www.w3.org/ns/shacl#>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:   <http://www.w3.org/2002/07/owl#>
PREFIX igonto:   <http://www.semanticweb.org/igonto/igonto#>
PREFIX organization:   <http://www.semanticweb.org/igonto/organization#>
PREFIX jurisdiction:   <http://www.semanticweb.org/igonto/jurisdiction#>
PREFIX lifecycle:   <http://www.semanticweb.org/igonto/lifecycle#>
PREFIX information:   <http://www.semanticweb.org/igonto/information#>
PREFIX enterprise:   <http://www.semanticweb.org/igonto/enterprise#>
PREFIX igso-arc:   <http://www.example.org/cm/igso/2023/igso-arc#>
PREFIX igso-svc:   <http://www.example.org/cm/igso/2023/igso-svc#>
PREFIX igso-org:   <http://www.example.org/cm/igso/2023/igso-org#>
```

```
PREFIX igso-ddy:   <http://www.example.org/cm/igso/2023/igso-ddy#>
PREFIX igso-data:   <http://www.example.org/cm/igso/2023/igso-data#>
PREFIX igso-req:   <http://www.example.org/cm/igso/2023/igso-req#>
PREFIX igso-impl:   <http://www.example.org/cm/igso/2023/igso-impl#>
PREFIX igso-plt:   <http://www.example.org/cm/igso/2023/igso-plt#>
PREFIX igso-ass:   <http://www.example.org/cm/igso/2023/igso-ass#>
PREFIX igso-stds:   <http://www.example.org/cm/igso/2023/igso-stds#>
PREFIX igso-sys:   <http://www.example.org/cm/igso/2023/igso-sys#>
PREFIX igso-vend:   <http://www.example.org/cm/igso/2023/igso-vend#>
```

Listing 5.1: All PREFIXES included in every SPARQL query.

5.1.1 General Queries

This section lists the relevant competency questions that an enterprise of a given size would ask, along with seeking qualified answers around more general questions regarding IG.

1. Which countries are subject to GDPR?
2. What are the rights of the data subject?
3. What are the obligations of the data controller and data processor?
4. What data subject rights imply which functional- and non-functional requirements?
5. What data controller obligations imply which functional- and non-functional requirement?

Query 1 retrieves all countries that are subject to GDPR, whereas Figure 5.1 presents the result. Here, all countries are visualized, including countries outside the EU and inside the European Economic Area (EEA).
Query 2 lists all data subject rights and its results are visualized in Figure 5.2.
Query 3 focuses on the resulting GDPR obligations. These obligations are further specialized in *DataControllerObligation* and *DataProcessorObligation* and visualized in Figure 5.3 and Figure 5.4.
Query 4 lists all rights and associated requirements. These requirements are either *Functional-Rquirements* or *NonFuncitonalRequirements*. To enhance visualization, the graph is divided into two sub-graphs. Figure 5.5 shows the *NonFuncitonalRequirements* graph, and Figure 5.6 contains the graph regarding *FunctionalRequirements* .
Query 5 is similar to Query 4 but focuses on the data controller obligations, including their involved requirements. Because the results are too numerous, a visualization is not included.

```
SELECT ?country WHERE {
?country jurisdiction:isRegulatedBy jurisdiction:GDPR.
}ORDER BY ?country
```

Listing 5.2: Implementation of SPARQL Query 1.

```
SELECT ?dataSubjectRight WHERE {
?dataSubjectRight rdf:type jurisdiction:DataSubjectRight.
}ORDER BY ?dataSubjectRight
```

Listing 5.3: Implementation of SPARQL Query 2.

```
SELECT ?dataControllerObligation ?dataProcessorObligation
WHERE {
    {
        ?dataControllerObligation rdf:type jurisdiction:DataControllerObligation.
    }
    UNION
    {
        ?dataProcessorObligation rdf:type jurisdiction:DataProcessorObligation.
    }
}
ORDER BY ?dataControllerObligation ?dataProcessorObligation
```

Listing 5.4: Implementation of SPARQL Query 3.

```
SELECT ?dataSubjectRight ?funcRequirement ?nonFuncRequirement WHERE {
?dataSubjectRight rdf:type jurisdiction:DataSubjectRight.
{
?funcRequirement rdf:type jurisdiction:FunctionalRequirement.
?dataSubjectRight jurisdiction:rightInvolvedInRequirement ?funcRequirement.
}
UNION
{
?nonFuncRequirement rdf:type jurisdiction:NonFunctionalRequirement.
?dataSubjectRight jurisdiction:rightInvolvedInRequirement ?nonFuncRequirement.
}
}ORDER BY ?dataSubjectRight ?funcRequirement ?nonFuncRequirement
```

Listing 5.5: Implementation of SPARQL Query 4.

```
SELECT ?dcObligation ?dcRequirement
WHERE {
        ?dcObligation rdf:type jurisdiction:DataControllerObligation.
        ?dcObligation jurisdiction:obligationInvolvedInRequirement ?dcRequirement.
}
ORDER BY ?dcObligation ?dcRequirement
```

Listing 5.6: Implementation of SPARQL Query 5.

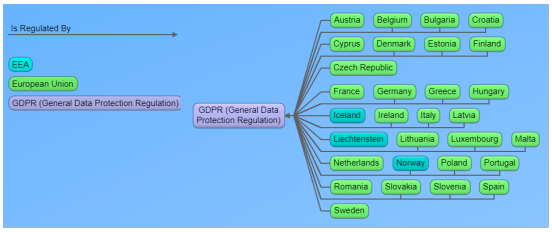


Figure 5.1: Gruff result visualization of Query 1.

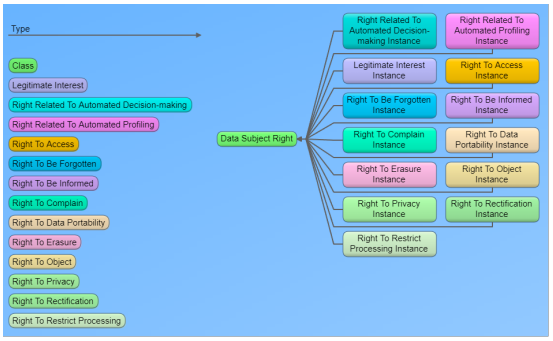


Figure 5.2: Gruff result visualization of Query 2.

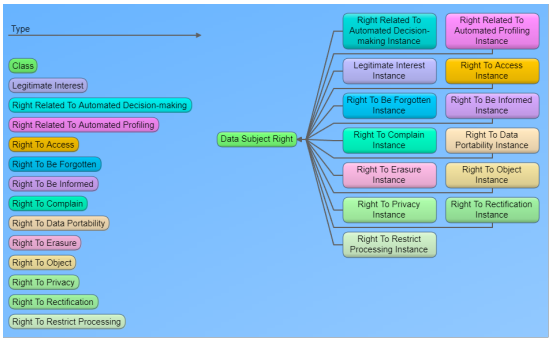


Figure 5.3: Gruff result visualization of Query 3 (1).

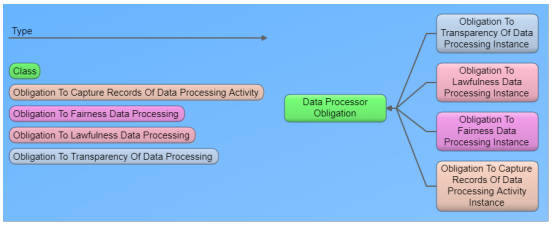


Figure 5.4: Gruff result visualization of Query 3 (2).

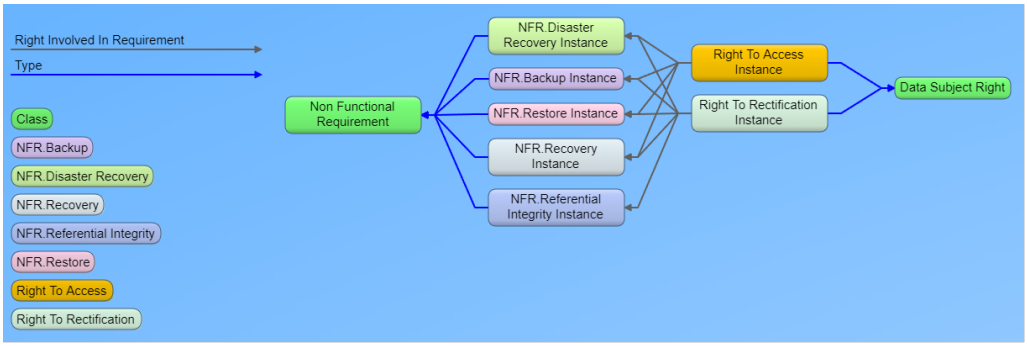


Figure 5.5: Gruff result visualization of Query 4 (1).

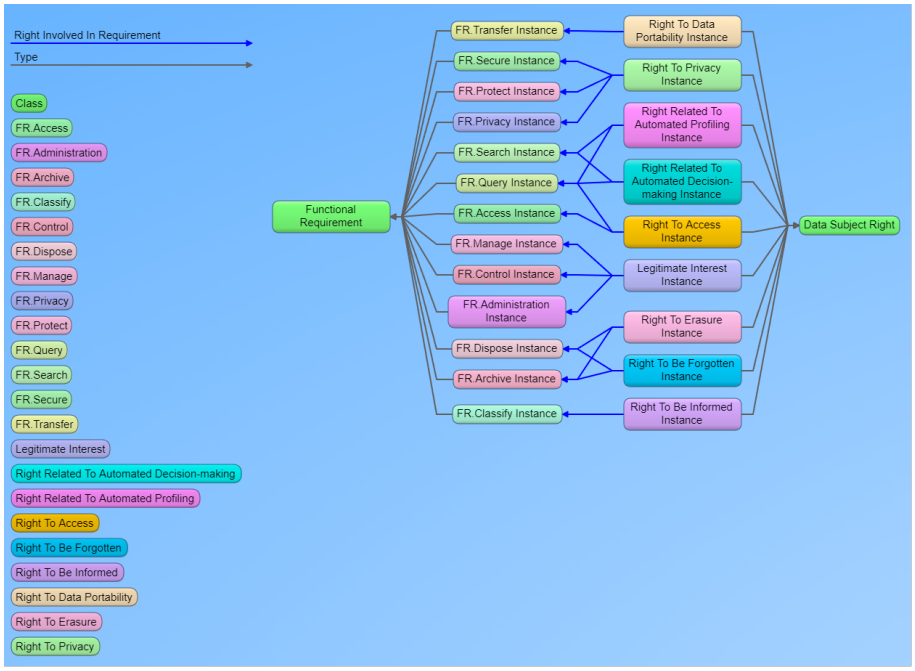


Figure 5.6: Gruff result visualization of Query 4 (2).

5.1.2 Use-case Queries

The queries start at the top level of IGONTO and move down to the implementation level. Query 6 provides answers to the competency question of which regulations a company must comply with, in a particular country.
The variable *?smallEnterprise* represents the concept of small enterprises. Because only one instance from the class *SmallEnterprise* was implemented, which represents the concept of small enterprises at the instance layer, we can easily retrieve that specific instance by asking for all instances that are members of the class *SmallEnterprise*. Afterward, we query the *operatesIn* connection to retrieve the instance of the country in which the small enterprise operates. The country is represented in the variable *?country*. If the enterprise operated in multiple countries, *?country* would involve multiple instances of the class *Country*. In case the query should contain a specific country, the variable *?country* must be replaced with the desired instance (e.g. *jurisdiction:Germany*). After linking to the country instance, we recursively ask for the *?regulation* that the country *is regulated by*. Similarly, we query the connection *belongsToJurisdiciton*, to retrieve the *?jurisdiction*.
Figure 5.7 shows the results of Query 6 in Gruff, where the instances are displayed on the right side and the legend on the left side.

```
SELECT ?smallEnterprise ?country ?regulation ?jurisdiction WHERE{
  ?smallEnterprise a organization:SmallEnterprise.
  ?smallEnterprise jurisdiction:operatesIn ?country.
  ?country jurisdiction:isRegulatedBy ?regulation.
  ?country jurisdiction:belongsToJurisdiction ?jurisdiction.
}ORDER BY ?smallEnterprise
```

Listing 5.7: Implementation of SPARQL Query 6.

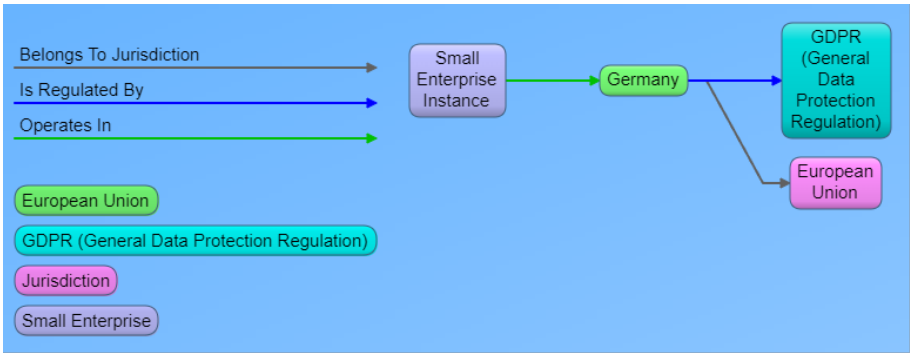


Figure 5.7: Gruff result visualization of Query 6.

When asking what the Regulatory Requirements (RR) we need to comply with, the capabilities required to fulfill the RR must be identified. Associations like ARMA and CGOC interpreted the GDPR and defined a large number of capabilities, classifying them into different maturity levels that range from level 1 (not existent) to level 5 (overachieved). To be GDPR-compliant, capabilities from level 3 need to be fulfilled. Query 7 deals with the following questions:

- What minimum *Capabilities* do I need to fulfill to be GDPR compliant, and what do they entail?
  To retrieve the capabilities required for minimum GDPR compliance, we queried all capabilities represented by the variable *?capability* that are *partOfMaturityLevel organization:3Essential*, which represents maturity level 3 of 5.
- What *Requirements, Obligations*, and *Rights* are satisfied by these capabilities?

  We then retrieve the set of requirements connected to every *?capability*, represented by the variable *?requirement*. The obligations (*?obligation*) and rights (*?right*) potentially linked to every *?requirement* are queried within the *OPTIONAL* environment, as not every requirement directly involves rights and vice versa. If the statements were not embedded within the *OPTIONAL* environment, the results would only include requirements that involve at least one *?right* instance and one *?obligation* instance simultaneously. For every obligation and right, the involved articles (*obligationArticle, rightArticle*) are also included.

```
SELECT ?capability ?requirement ?obligation ?obligationArticle ?right ?rightArticle WHERE{
  ?capability organization:partOfMaturityLevel organization:3Essential.
  ?capability jurisdiction:capabilityInvolvesRequirement ?requirement.
  OPTIONAL{?requirement jurisdiction:requirementInvolvesObligation ?obligation.
  ?obligation jurisdiction:obligationInvolvesArticle ?obligationArticle}
  OPTIONAL{?requirement jurisdiction:requirementInvolvesRight ?right.
  ?right jurisdiction:rightInvolvesArticle ?rightArticle.}
}ORDER BY ?capability
```

Listing 5.8: Implementation of SPARQL Query 7.

The results of Query 7 are too numerous to visualize them in one picture. Therefore, we focused on a specific capability, *LegalP33*. The selection of this instance is arbitrary, any capability can be similarly retrieved. To return only relations including this specific instance, we replace the *?capability* variable in the query with the URI of the instance (*organization:LegalP33*). The result is visualized in Figure 5.8. The description of *LegalP33* is as follows:

> "Systematically send notices and reminders, require and track confirmations. To manage exceptions, employees can look up their holds at any time. Communications tailored to recipient role (IT, RIM, employee)."

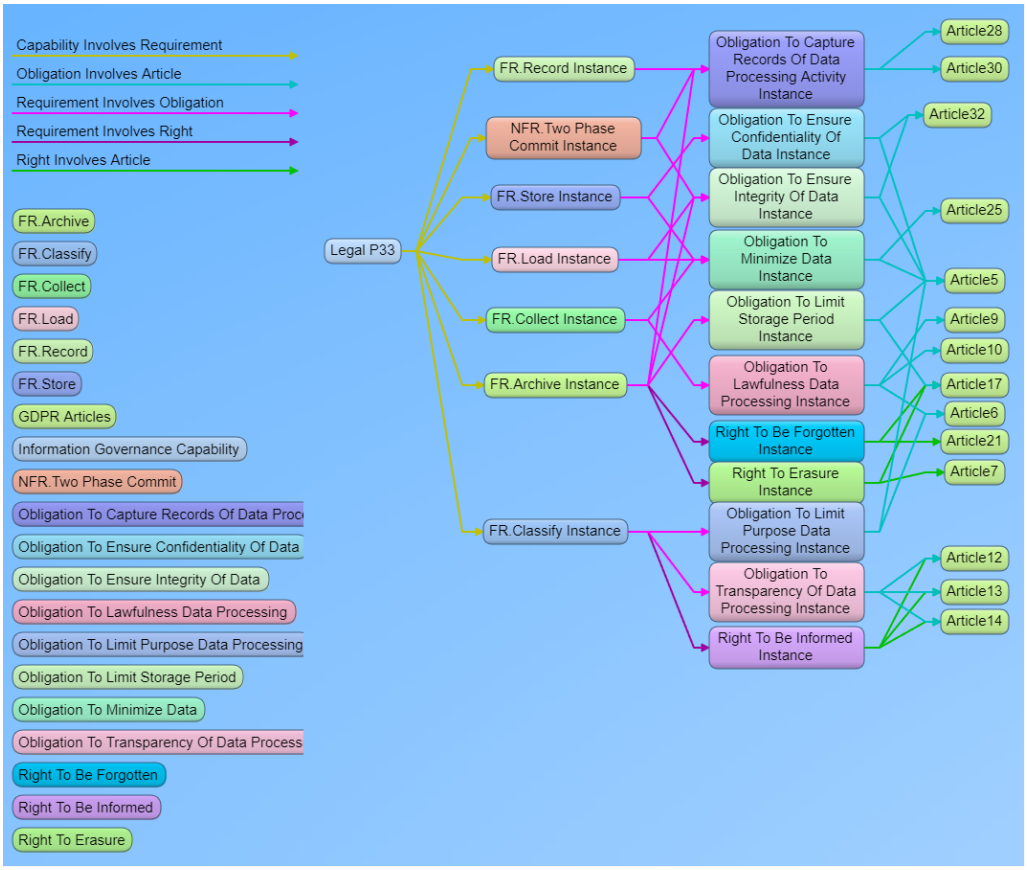


Figure 5.8: Gruff result visualization of Query 7 with the capability *LegalP33*.

Next we analyze requirements and involved capability in order to assess their relevance.

1. **FR.Record**, because tracking information is stored within records.
2. **FR.Store:**, because records need to be stored securely to ensure GDPR compliance.
3. **FR.Load:**, because requested information needs to be retrieved accurately and efficiently.
4. **FR.Collect:**, because collecting communication information is crucial for effective data management.
5. **FR.Archive:**, because archiving information is crucial for later processes such as audits.
6. **FR.Classify:**, because data also needs to be classified when collected to ensure correct data management and, therefore, correct tailoring to individual roles.
7. **NFR.TwoPhaseCommit:**, because transactional data integrity is crucial in the context of exception management and confirmation tracking.

Every requirement explicitly relates to rights and/or obligations, which are documented in the instance label and are not further discussed here.
The capabilities of the association CGOC are directly linked to governance-related processes because they focus on a technical interpretation of GDPR, whereas ARMA focuses on principles such as transparency and availability among others. Therefore, capabilities implemented by CGOC are directly linked to IG processes. Query 8 extends Query 7 by including the *?process* linked to every capability, the *?organizationUnit* to which they belong, and, more importantly, the potential *?risk* that a process exposes if not implemented correctly.

```
SELECT * WHERE{
  ?capability organization:partOfMaturityLevel organization:3Essential.
  ?capability organization:capabilityLinkedToProcess ?process.
  ?process organization:processLinkedToOrganizationUnit ?organizationUnit.
  ?process organization:description ?risk.
}ORDER BY ?capability
```

Listing 5.9: Implementation of SPARQL Query 8.

Because the resulting graph is too large to visualize in a single image, we focus only on the capabilities whose processes are connected to the Legal department. This is accomplished by replacing the *?organizationUnit* with the representative Legal instance (*organization:LegalInstance*). The resulting graph is shown in Figure 5.9.

Capabilities connected to processes are also crucial to our use case. SWRL rules S4-S8 discussed in Section 4.6 connect our two use cases - small enterprise and large enterprise - to the capabilities they need to fulfill according to GDPR. For large enterprises with more than 250 employees, every capability needs to be fulfilled. For enterprises with fewer than 250 employees, GDPR defines an exception in article 30 (5), stating that these enterprises are not required to keep record activities,

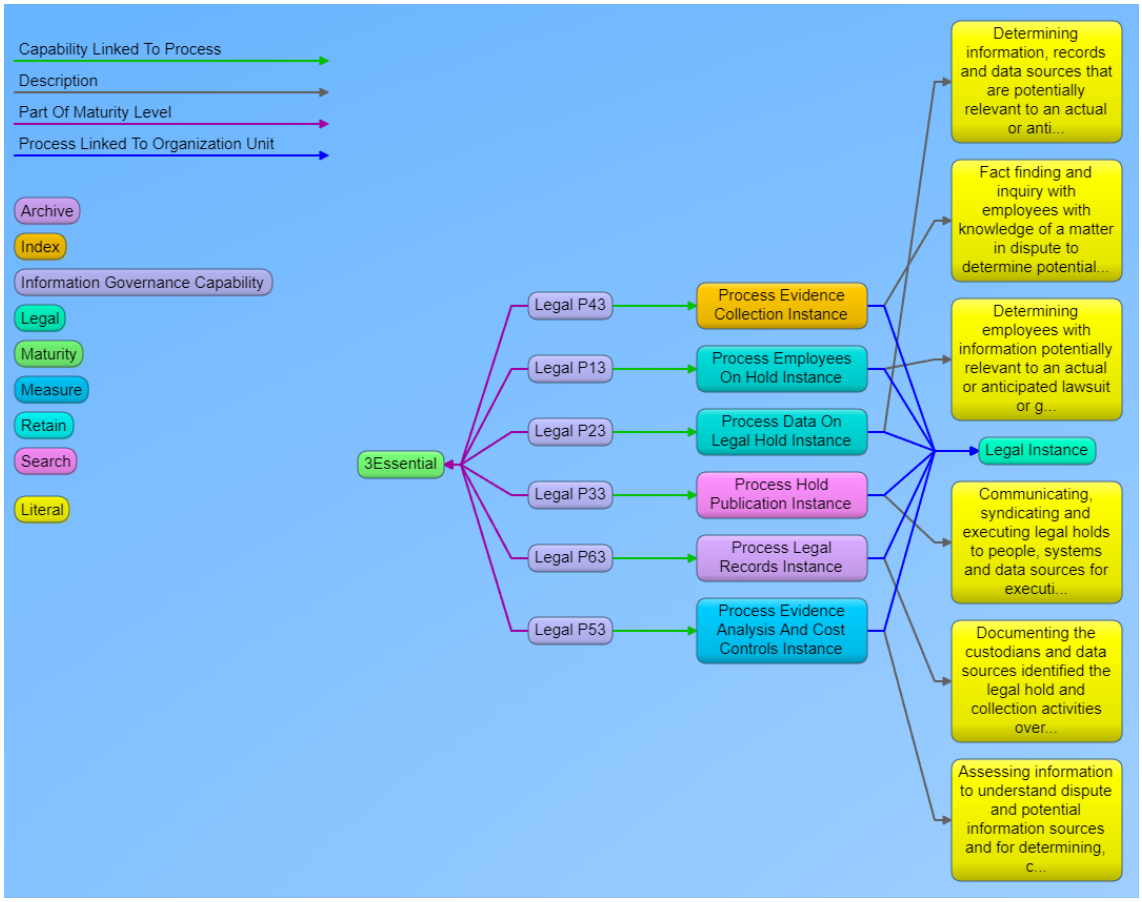


Figure 5.9: Gruff result visualization of Query 8 with legal capabilities connected to the *LegalInstance* organization.

and therefore do not need a RIM nor do they need to possess the capabilities connected to the RIM. Query 9 can be used to retrieve information about what capabilities each use case needs to fulfill. Here, the use case of the small enterprise is implemented. Because many capabilities are connected to both use cases, we focus on the CGOC capabilities by asking for the *?process* they are connected with. To retrieve the information about the large enterprise, simply change the variable *?smallEnterprise* to *?large_enterprise* and modify the first line to state the *?largeEnterprise* is *a organization:LargeEnterprise*. Figure 5.10 shows the resulting graph for the small enterprise, and Figure 5.11 shows the result for the large enterprise.

```
SELECT ?smallEnterprise ?capability WHERE{
  ?smallEnterprise a organization:SmallEnterprise.
  ?smallEnterprise igonto:enterpriseNeedsToFulfillCapability ?capability.
  ?capability organization:capabilityLinkedToProcess ?process.
}ORDER BY ?smallEnterprise
```

Listing 5.10: Implementation of SPARQL Query 9.

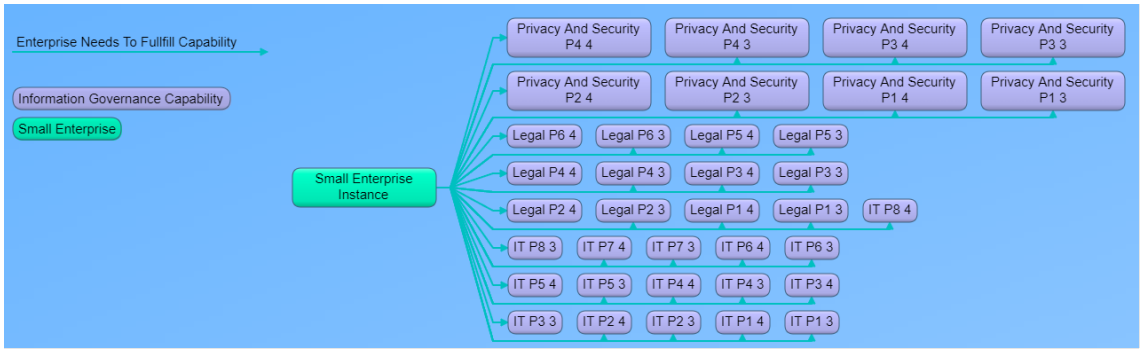


Figure 5.10: Gruff result visualization of Query 9 for small enterprises (1).

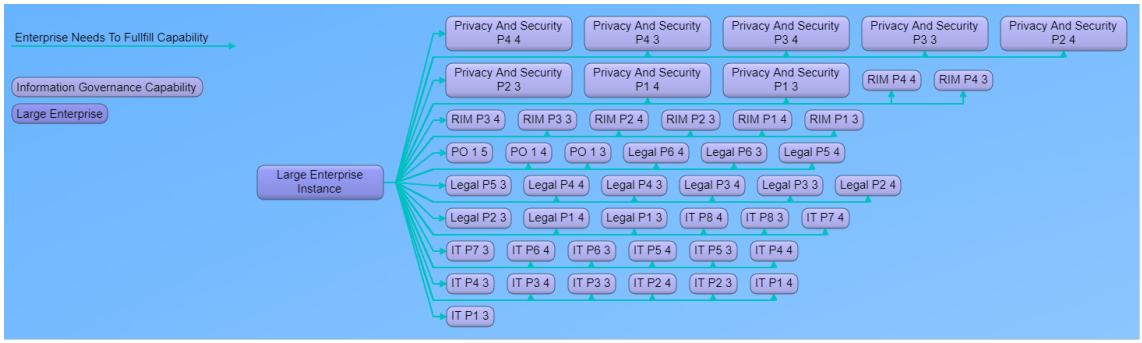


Figure 5.11: Gruff result visualization of Query 9 for large enterprises (2).

The connections of each capability to their corresponding organization unit can be determined as in Query 9, but this is excluded here due to the otherwise overwhelming size of the graph. Figure 5.12 shows that the large enterprise also includes all capabilities connected to RIM (*RIM P1 3,RIM P1 4,RIM P2 3,RIM P2 4,RIM P3 3,RIM P3 4,RIM P4 3,RIM P4 4*).

```
SELECT ?smallEnterprise ?capability ?process WHERE{
  ?smallEnterprise a organization:SmallEnterprise.
  ?smallEnterprise igonto:enterpriseDoesNotNeedCapability ?capability.
  ?capability organization:capabilityLinkedToProcess ?process.
}ORDER BY ?capability
```

Listing 5.11: Implementation of SPARQL Query 10.

The knowledge of the RIM capabilities that are not needed for small enterprises is not lost, however. As implemented in SWRL Rule S8, an explicit connection to these capabilities is inferred. This is crucial because, as already mentioned, ontologies act according to the open-world assumption. Without a relationship established between the small enterprise and the RIM capabilities, we would not be able to infer that these capabilities are not needed. Therefore, the small enterprise is connected via *enterpriseDoesNotNeedCapability* to explicitly express that these capabilities are not necessary. Query 10 shows the syntax to retrieve this information, including the connected processes, whereas Figure 5.12 visualizes the result.

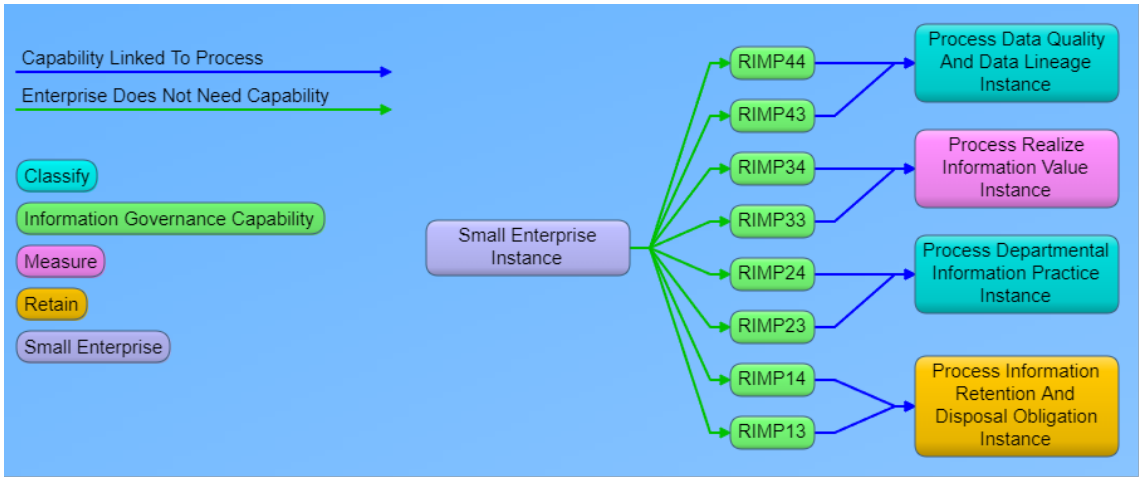


Figure 5.12: Gruff result visualization of Query 10 for small enterprises.

Now that the Functional Requirements (FR), Non-Functional Requirements (NFR), and Obligatory Requirements (OR) are specified, we can ask for a suitable solutions architecture, list associated software components, and the IG services (capabilities) they must provide. Based on SWRL Rules S9 and S10, the correct solutions are mapped to our use cases. Depending on whether the RIM and thus its capabilities are needed, different solution instances can be found. Query 11 retrieves this information, including the solution type (*?solutionType*). Solutions are classified by type, to indicate what problem the solutions solve. Solutions can be composed of other solutions to address multiple problem areas. Every instance in an ontology is automatically an *owl:NamedIndividual* when using OWL2 syntax, and its type is from a random annotation ID after materialization. Because this information is unnecessary, we filter it out by directly excluding *owl:namedIndividuals* and types that start with the "anon" string.
Figure 5.13 visualizes the solutions for small enterprises, whereas the solutions for large enterprises are shown in Figure 5.15. The visualization shows that the small enterprise is only connected to the *ECM Solution*, the *Archive Solution*, and a *Reference Model*. These solutions recursively include further implementation requirements. The *ECM Solution includes* a *Information Retrieval Solution*, and a *Content Repository Solution*. The *Content Repository Solution includes* a *Storage Service* and a *Content Object Catalog*, and *requires* an *Indexing Component* and a *Transformation Component*. The *Archive Solution implements Archive Management* and an *Archive Method*, and *requires* a *Transfer Service* and a *Records Plan*.

```
SELECT * WHERE {
  ?smallEnterprise a organization:SmallEnterprise.
  ?smallEnterprise igonto:needsSolution ?solution.
  ?solution rdf:type ?solutionType.
  FILTER (
    ISIRI(?solutionType) &&
    ?solutionType != owl:NamedIndividual &&
    ?solutionType != igso-arc:ArchitectureDesignDomain &&
    ?solutionType != igso-arc:Solution &&
    !STRSTARTS(STR(?solutionType), "anon") &&
    !STRSTARTS(STR(?solutionType), STR(igso-ddy:))
  )
  OPTIONAL {
    ?solution ?predicate ?solutionComponent.
    FILTER (?predicate IN (igso-arc:implementsArchiveManagement,
    igso-arc:implementsArchiveMethod, igso-arc:requiresRecordsPlan,
    igso-arc:requiresTransferService, igso-arc:includesContentRepositorySolution,
    igso-arc:includesInformationRetrievalSolution))
  }
  OPTIONAL {
    ?solutionComponent ?predicate2 ?component.
    FILTER (?predicate2 IN (igso-arc:includesContentObjectCatalog,
    igso-arc:rerquiresContentManagementComponent,
    igso-arc:requiresTransformationComponent, igso-arc:includesStorageService,
    igso-arc:requiresTransactionManagementComponent))
  }
}ORDER BY ?solution
```

Listing 5.12: Implementation of SPARQL Query 11a.

The query 11b for the large enterprise needs to be customized as it includes more and different solutions. We connected the large enterprise to the Alfresco governance solution to simulate an existing real-world ECM [Alf24a]. The resulting diagram is divided into Figure 5.15 and Figure 5.14 because the diagram is already too large for Gruff to export without visual errors. Furthermore, the graph could be expanded in certain components, but this would lead to the same visualization problems. For example, the *Content Repository Solution Instance* could be extended as shown in Figure 5.13.

```
SELECT * WHERE {
  ?LargeEnterprise a organization:LargeEnterprise.
  ?LargeEnterprise igonto:needsSolution ?solution.
  ?solution igso-arc:includesSolution ?solution2.
  ?solution igso-arc:requiresSolution ?solution3.
  ?solution igso-arc:consistsOfManagement ?management.
  OPTIONAL {
    ?solution2 igso-arc:includesComponent ?component1.
    ?component1 ?consistsOf ?artifact.
    OPTIONAL{
    ?artifact igso-data:consistsOfDatabase ?db.
    ?artifact igso-data:consistsOfSystem ?system.
    ?artifact igso-data:consistsOfRelationalSchema ?schema.
    }
  }
  ?solution3 igso-arc:consistsOfService ?service.
  ?solution3 igso-arc:includesSolution ?solution4.
  ?solution3 igso-arc:consistsOfComponent ?component2.
  ?solution3 igso-arc:includesComponent ?component3.
  ?component3 igso-arc:includesSolution ?solution5.
}ORDER BY ?solution
```

Listing 5.13: Implementation of SPARQL Query 11b.

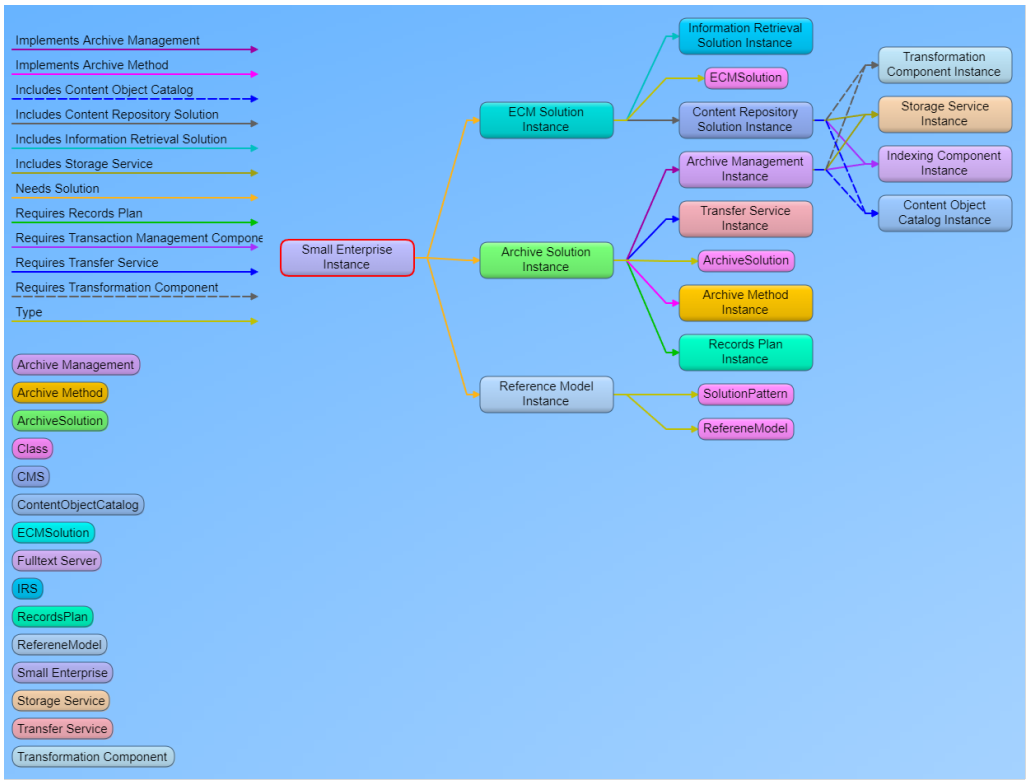


Figure 5.13: Gruff result visualization of Query 11a.

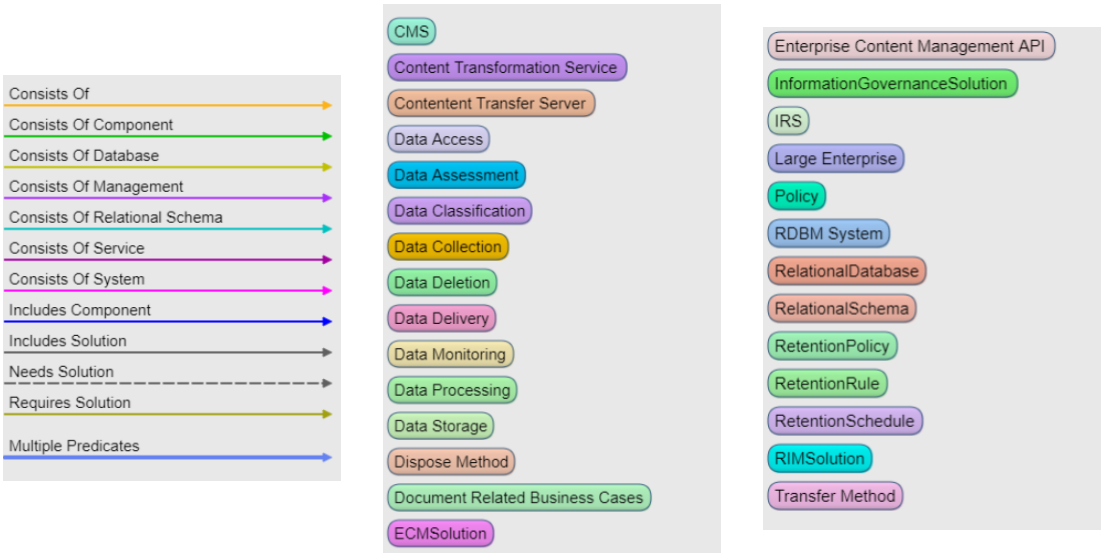


Figure 5.14: Gruff result visualization legend of Query 11b.

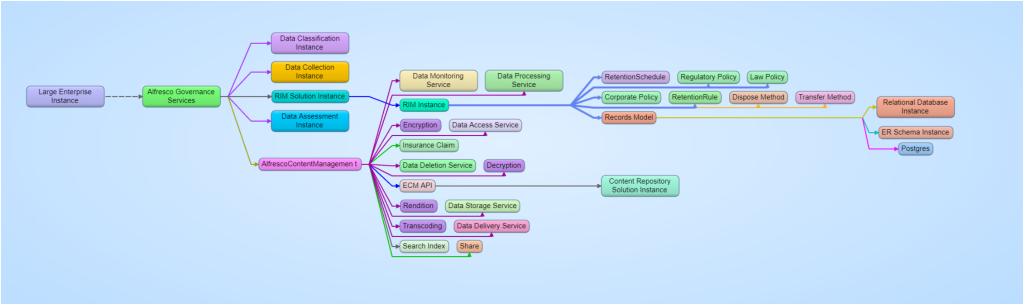


Figure 5.15: Gruff result visualization of Query 11b.

To compare the resulting graph, Figure 5.16 shows the high-level architecture of Alfresco. The *AlfrescoContentManagement* component consists of a *Share* Document Management Component (DMC) and a digital workspace, represented in the graph by the individual *Data Services*. The ReST API is represented by the *ECM API*, and *includes* a *Content Repository Solution*, which could be expanded similarly to Figure 5.13. The *Search Index* is a physical storage, where search indices for the *Search Services* are kept. Since the large enterprise includes records, a RIM solution is needed, which *consists of* a *Retention Schedule, Regulatory Policy, Law Policy, Corporate Policy, Retention Rule, Dispose Method, Transfer Method* and a *Records Model*. This model operates as a *Content Catalog* and *consists of Postgres*, an *ERSchema*, and a *Relational Database*. Postgres is a Relational Database Management System (RDBMS) that provides efficient data storage and querying capabilities for the *Records Model*. The *ERSchema* helps to structure and visualize relationships between Data entities. The *Relational Database* is a physical database that stores Data according to the ERSchema. To adapt content for various needs, the Alfresco solution includes *Content Transformation Services*, such as *Encryption, Decryption, Rendition, and Transcoding*. Furthermore, to utilize information effectively, certain *Data Management* processes are needed such as *Data Classification, Data Collection,* and *Data Assessment*.

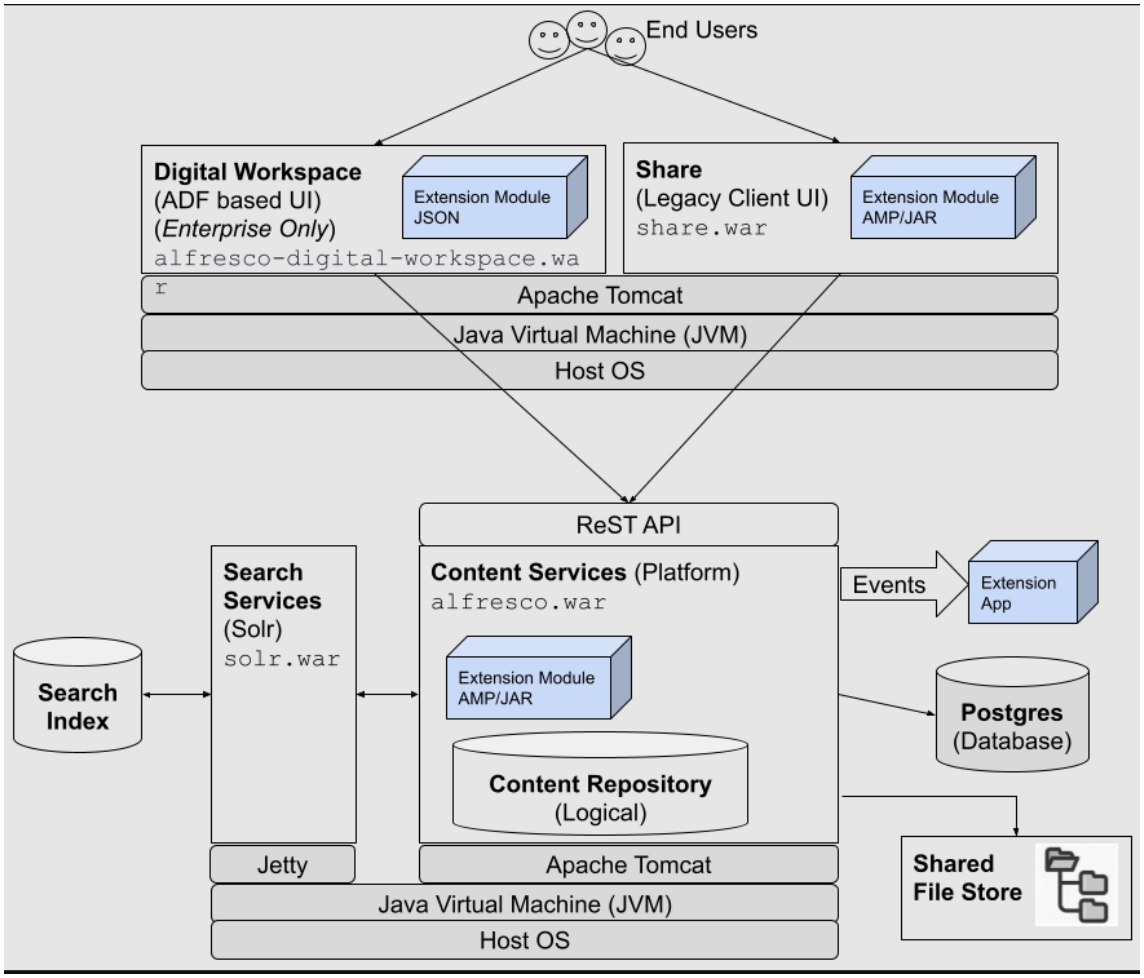


Figure 5.16: High-level architecture of Alfresco [Alf24b].

```
SELECT ?smallEnterprise ?negativePredicate ?solution ?article WHERE{
  ?smallEnterprise a organization:SmallEnterprise.
  ?smallEnterprise ?negativePredicate ?solution.
  OPTIONAL{?solution igonto:justifiedByArticle ?article.}
  FILTER((?negativePredicate IN (igonto:enterpriseDoesNotNeedSolution,
  igonto:enterpriseDoesNotNeedCapability,
  igonto:enterpriseDoesNotNeedOrganizationUnit))
    )
}ORDER BY ?solution
```

Listing 5.14: Implementation of SPARQL Query 12.

Applying the same reasoning as that in Query 10, we inferred negative relationships to solutions that the small enterprise does not need. In addition, the organization unit that is not needed is retrieved, including the *?article*, justifying why it is not needed. Query 12 and Figure 5.17 include the information for the small enterprise.

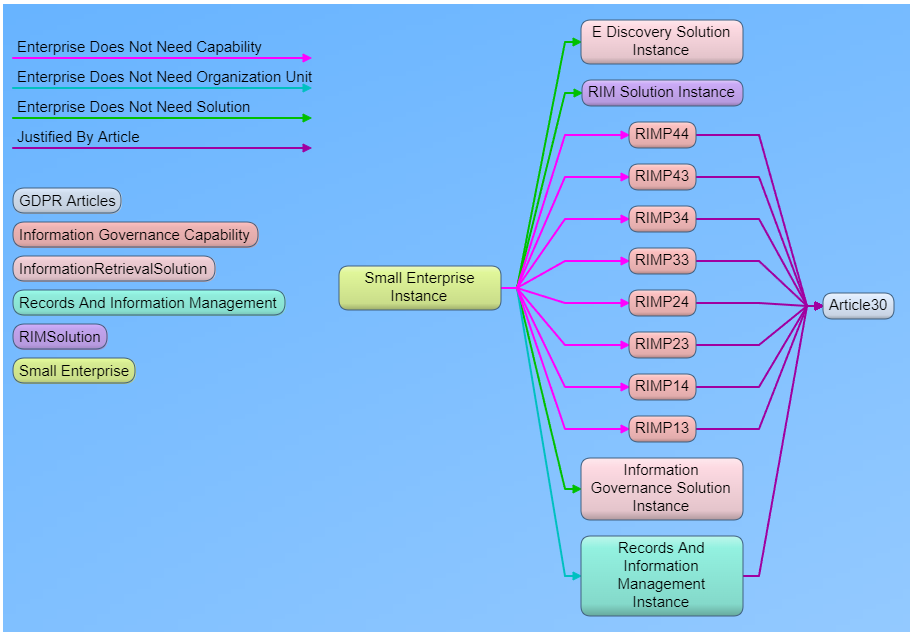


Figure 5.17: Gruff result visualization of Query 12 for small enterprises.

5.2 Performance

As previously mentioned, performance is not crucial for our prototype knowledge base. However, some performance measurements are taken in this section to establish a reference point and should be considered a sniff test. The focus is not on improving the performance of individual queries, but rather on identifying characteristics that influence them. Our test environment contains a virtual machine configured with 4 VCPUs, 16GB of RAM, and a 200GB disk. The measurements were carried out using the graph visualization tool Gruff [23a]. The application is running on a Docker container, which has been allocated 1GB of shared memory.
Figure 5.18 demonstrates the average performance on the y-axis and the query ID on the x-axis, according to the queries described in Section 5. Performance-relevant data for each query is listed beneath the bar chart. These include the number of query triples resulting from each query, involved classes, SPARQL functions, and the individual measurements leading to the average time. To have a performance reference within our tests, Query 0 retrieves all triples.

Analyzing the bar chart, the first conspicuousness is the performance of Q7. The obvious reason for that is the number of resulting triples for the query. Table 5.1 lists additional information about each query, including the number of resulting query triples, classes involved, SPARQL functions, average time, and standard deviation. While other queries do not exceed 48 triples, Q7 includes 19109 results, leading to an average time of 797.3 ms.
Because a sample of 10 measures is considered, we will take the formula for calculating the standard

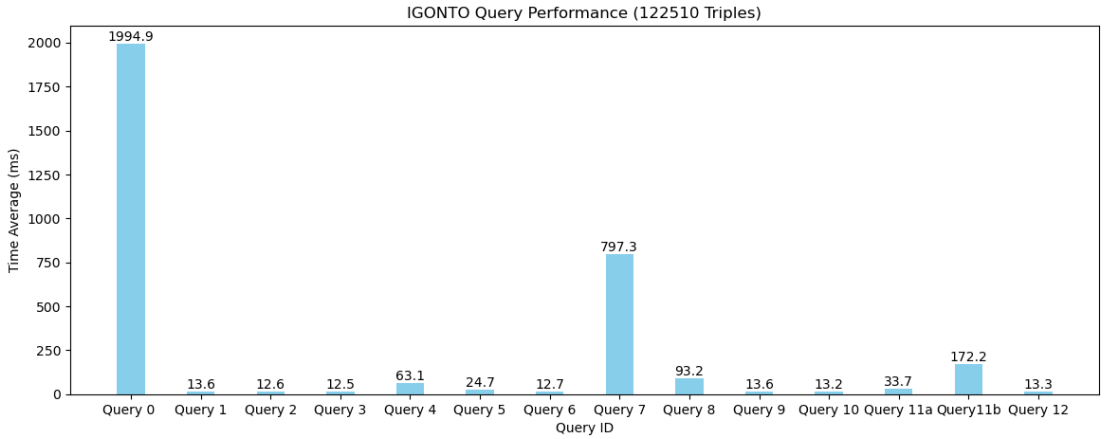


Figure 5.18: Performance overview for Queries 0-12 in IGONTO.

deviation of a sample, where s is the standard deviation, n is the number of measurements, x_i is the individual measure, and $\overline{x}$ is the average:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

The high standard deviation of Query 7 shows that the measurements of Query 7 are unstable. High standard deviations for Query 4, 5, and 8 can also be seen. This is explained by the individual measures within the queries that take 111-115 ms. Because they are always in that range and only appear sometimes, Gruff is presumed to be performing some operations, leading to a higher but constant query time. Gruff itself does not document further information, but it is noticeable that these anomalies appear more often in queries with more classes involved.

Query	**Query Triples**	**Involved Classes**	**SPARQL Functions**	**Average Time $\overline{x}$ (ms)**	**Standard Deviation s (ms)**
Query 0	122510	1133	-	1994.9	451.03
Query 1	30	2	1 x ORDER BY	13.6	2.32
Query 2	13	1	1 x ORDER BY	12.6	1.17
Query 3	15	2	1 x ORDER BY, 1 x UNION	12.5	1.58
Query 4	27	3	1 x ORDER BY, 1 x UNION	63.1	51.78
Query 5	48	2	1 x ORDER BY	24.7	31.2
Query 6	1	4	1 x ORDER BY	12.7	0.48
Query 7	19109	6	1 x ORDER BY, 2 x OPTIONAL	797.3	254.97
Query 8	26	3	1 x ORDER BY	93.2	35.09
Query 9	30	2	1 x ORDER BY	13.6	1.43

Continued on next page

Table 5.1 continued from previous page

Query	Query Triples	Involved Classes	SPARQL Functions	Average Time $\overline{x}$ (ms)	Standard Deviation s (ms)
Query 10	8	3	1 x ORDER BY	13.2	1.14
Query 11a	17	7	1 x ORDER BY, 2 x OPTIONAL, 1 x FILTER	33.70	44.29
Query 11b	2640	28	1 x ORDER BY, 2 x OPTIONAL	172.2	32.17
Query 12	10	5	1 x ORDER BY, 1 x FILTER	13.3	2.31

Table 5.1: IGONTO query deviation calculations.

Since operations like *ORDER BY* can have a large impact on queries, Figure 5.19 demonstrates the results without the *ORDER BY* function, while Table 5.2 provides additional information, similar to Table 5.1. Other functions like *FILTER* or *OPTIONAL* are necessary to achieve the same triple results.

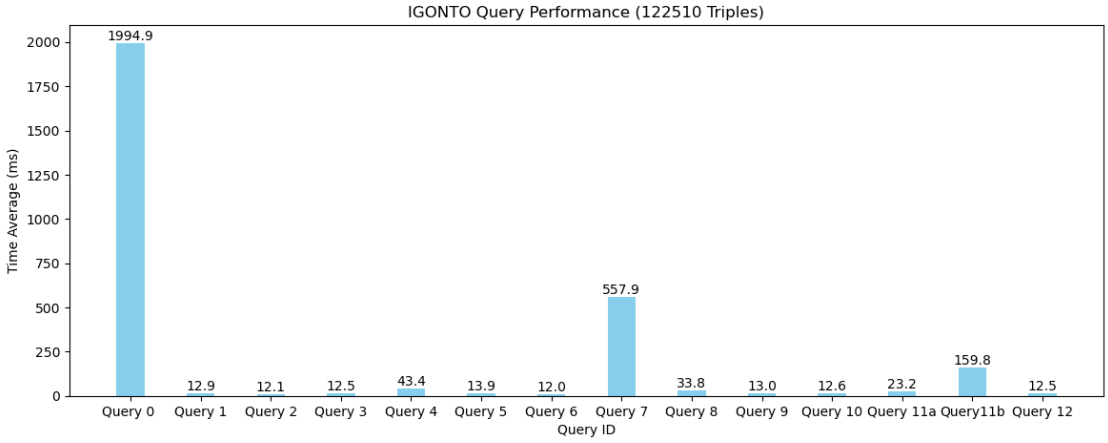


Figure 5.19: Performance overview for Queries 0-12 in IGONTO without ORDER BY.

Without the *ORDER BY* function, queries with a larger number of triples run significantly faster and most of them have a better standard deviation.
To test how the number of triples in the whole ontology influences performance, Queries 1-5 were performed only within their jurisdiction ontology, which contained only 16970 triples. The remaining queries require knowledge of multiple ontologies and cannot be performed in one sub-ontology.
Figure 5.20 illustrates the results. In these measures, the results do not differ much from the results of IGONTO. However, considering the standard deviation listed in Table 5.3, queries from the jurisdiction ontology appear to have fewer measure anomalies.

Query	Query Triples	Involved Classes	SPARQL Functions	Average Time $\overline{x}$ (ms)	Standard Deviation s (ms)
Query 0	122510	1133	-	1994.9	451.03
Query 1	30	2	-	12.9	1.85
Query 2	13	1	-	12.1	1.52
Query 3	15	2	1 x UNION	12.5	1.18
Query 4	27	3	1 x UNION	43.4	47.85
Query 5	48	2	-	13.9	1.64
Query 6	1	4	-	12.0	0.82
Query 7	19109	6	- 2 x OPTIONAL	557.9	72.17
Query 8	26	3	-	33.8	41.80
Query 9	30	2	-	13.0	1.94
Query 10	8	3	-	12.6	1.65
Query 11a	17	7	2 x OPTIONAL, 1 x FILTER	23.2	31.20
Query 11b	2640	28	2 x OPTIONAL	159.8	6.70
Query 12	10	5	1 x FILTER	12.5	0.97

Table 5.2: IGONTO query deviation calculations.

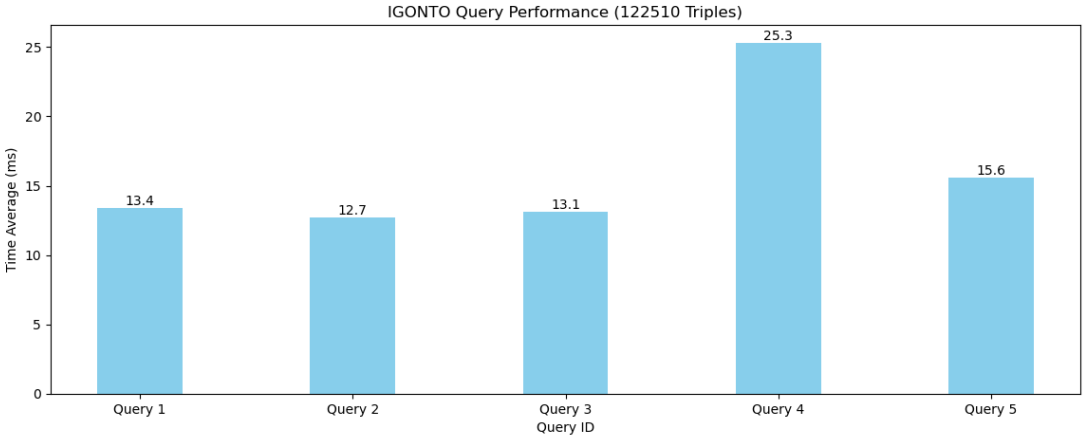


Figure 5.20: Performance overview for Queries 1-5 in Jurisdiction.

To facilitate the factor of how the number of involved classes influences the performance, Query 13 and Query 14 are defined as illustrated in Listing 13 and Listing 14. Query 13 retrieves all capabilities and their involved articles, whereas Query 14 retrieves multiple connections from the enterprise down to the records.

Query	Average Time $\overline{x}$ (ms)	Standard Deviation s (ms)
Query 1	13.40	1.27
Query 2	12.70	0.95
Query 3	13.10	2.33
Query 4	25.30	30.54
Query 5	15.60	3.06

Table 5.3: Jurisdiction query deviation calculations.

```
SELECT * WHERE {
    ?capability jurisdiction:capabilityInvolvesArticle ?article.
}
```

Listing 5.15: Implementation of SPARQL Query 13.

```
SELECT * WHERE {
    ?smallEnterprise a organization:SmallEnterprise.
    ?smallEnterprise jurisdiction:operatesInCountry ?country.
    ?country jurisdiction:isRegulatedByRegulation ?regulation.
    ?smallEnterprise organization:hasOrganizationUnit ?organizationUnit.
    ?organizationUnit a organization:IT.
    ?organizationUnit organization:createsData ?data.
    ?data organization:governedByRecord ?record.
    ?process organization:createsRecord ?record.
    ?rule organization:executesRecord ?record.
    ?policy organization:definesRule ?rule.
}
```

Listing 5.16: Implementation of SPARQL Query 14.

Table 5.4 summarizes the results of both queries. Both queries were performed on IGONTO. The two main differences are the number of resulting triples and the number of involved classes. Query 13 involves 423 triples and 2 classes, whereas Query 14 involves 176 triples and 9 classes. Even though Query 13 involves more than double the number of triples as Query 14, its performance is 23.1 ms better on average. Query 14 also involves more aforementioned performance jumps up to 126 ms. Even when only considering the better measures, the performance is notably worse. This is also represented in the slightly worse standard deviation of Query 13. Because the lower-ranged measures of Query 13 are better, whereas the jumps are similar to the ones of Query 14, the average is smaller, and therefore its deviation is higher.

The insights gained from this performance analysis lead to the conclusion that the volume of resulting triples and the extent of class involvement are crucial factors for performance, whereas the number of classes involved outweighs the number of resulting triples. For queries with a larger number of triples, additional query functionality results in noticeable performance degradation. The total amount of triples in the ontology appears to increase individual performance jumps and thus reduce stability, but not directly query time. Since the cause of the performance jumps is unknown

Query	Query Triples	Involved Classes	SPARQL Functions	Average Time $\overline{x}$ (ms)	Standard Deviation s (ms)
Query 13	423	2	-	57	48.70
Query 14	176	9	-	80.1	46.41

Table 5.4: Query 13 and Query 14 deviation calculations.

and only what factors could lead to further jumps are known, the queries could be tested outside of Gruff in the next increment of the prototype. Another possible environment is the RDFLib framework mentioned in Section 5.4.

5.3 Validation

The validation of IGONTO is achieved with the implementation of SHACL validation schemas, as described in Section 3.3. Based on the implemented structure regarding object properties, described in Section 3.1.2, we implemented property shapes for each class and relationship within their sub-ontology. The validation is performed in Stardog [23d], an online RDF-Graph database. Stardog enables an existing database to be easily validated [Sta24]. This is simply achieved by uploading a SHACL validation script in an empty editor. By pressing the "Get Validation Report" button, the knowledge graph is validated against all the constraints implemented within the schema.

SHACL and its basic building blocks were described in the 3.3 section. This section focuses on the semantic schema and how SHACL forms are implemented based on the IGONTO structure.
Each relationship in the sub-domains is implemented with DL and therefore validated within the schema. Figure 5.21 shows an example of how a connection is implemented in Protegé and expressed with DL. As mentioned in the legend 4.1, the cardinality is assigned to a specific Protégé keyword. Table 5.5 maps all used graph cardinalities to the corresponding Protégé, DL, and SHACL constraint descriptions.

Cardinality	Protégé Keyword	DL Axiom	SHACL Constraints
0...	min 0	$(\geqslant nR.C)^{\mathcal{I}}, n = 0$	sh:minCount 0
1...	some	$(\exists R.C)^{\mathcal{I}}$	sh:minCount 1
1	exactly 1	$((\geqslant nR.C)^{\mathcal{I}} \sqcap (\leqslant nR.C)^{\mathcal{I}}), n = 1$	sh:minCount 1, sh:maxCount 1

Table 5.5: Used axioms translation.

Using these building blocks, we can systematically construct our SHACL shapes. Figure 5.22 shows a schematic approach regarding the development of SHACL shapes. Each validation script starts with a listing of all prefixes. These contain the standard prefixes such as owl, rdfs, and the prefixes of the validated *domains*. The ID of *a NodeShape* consists of its corresponding *domain* prefix, its *SubjectClass* URI without the namespace, and the suffix "Shape". Afterwards, the *targetClass* is declared, and its *SuperClass*, since we also want to validate the hierarchy. In addition, the DL

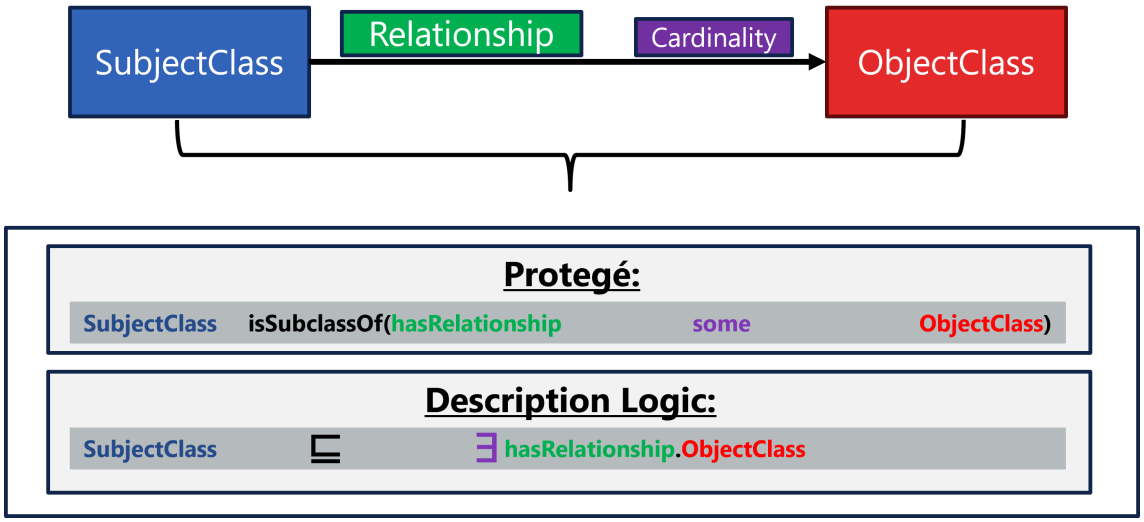


Figure 5.21: SHACL Shapes building schema (1).

declared within the *SuperClassShape* will be inherited by its subclasses, enabling a reduced and transparent validation. Afterwards, the involved properties are listed.
The uniqueness of our relationships enables the implementation of unique *PropertyShapes*. This ensures a correct validation of every relationship tailored to its specific requirements. The *PropertyShape* ID is formed by the *domain, URI* and the "Shape" suffix, similar to the *NodeShape*. Afterward, we declare the *path*, such as the object property validated by this shape. Since every relationship has only one *ObjectClass* range implemented, its specification is not necessary but is included here for the sake of completeness. Considering that object properties are also implemented hierarchically, the corresponding *SuperProperty* is also included. As mentioned in Section 3.3.2, every relationship is semantically grouped to infer more knowledge about the object property. At the end, the constraints are listed according to the mapping listed in Table 5.5.

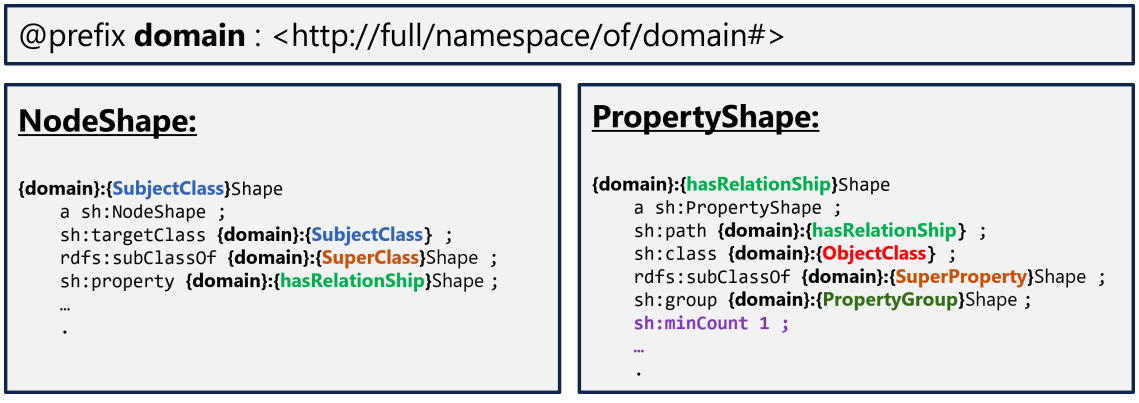


Figure 5.22: SHACL Shapes building schema (2).

Based on our structural implementation approach, the creation of such validation scripts could be automated with further development. This would save development time, as SHACL schemas still need to be manually implemented. Cimminio et. al. [CFG20] propose *Aestra*, a tool to automate SHACL shape generation. Their covered SHACL restrictions would theoretically also

include all important restrictions presented here (*sh:NodeShape, sh:PropertyShape, sh:targetClass, sh:property, sh:path, sh:class, sh:minCount, sh:maxCount*). The only constraint that is not covered due to practitioner-required restrictions is the *sh:group* constraint. Although grouping properties infers additional knowledge, it is not necessary for validation. Because PropertyShapes needs to be included in the knowledge graph to make use of their grouping semantics, the group association can be handled separately outside of the validation. The PropertyShapes can be generated first for validation purposes and then manually added to include them in the knowledge graph.
This work only implements a prototype. It does not utilize the full potential of SHACL and further limitations may follow. It may not be possible to cover the additional constraints, so shape generation would again require manual assistance. Nevertheless, Aestra covers 60% of all SHACL constraints, and the idea can be considered for further increments of IGONTO to save manual effort to some extent. In this way, automating SHACL shape generation offers potential for IGONTO.

5.4 Further Development

With the development of a prototype, questions arise regarding future work and its reuse for being integrated into further research projects. Creating prototypes that are not just throwaway models but a basis for new ones that form a final and functioning product is one of the main characteristics of evolutionary prototyping. In the latter approach, the idea of the prototype is continuously refined and evolved based on feedback and requirements until the lessons learned can be translated into the final product [Boe88]. Evolutionary prototyping reduces risks (which is very important for an expert system that relies on compliance) [Som11] and allows more flexibility and adaptability in design, as changes can be iteratively incorporated [PB14].
Therefore, the ability to reuse or work further with a prototype is also a quality aspect and therefore part of this chapter. This section describes how this prototype can be integrated into an independent application and how certain design decisions are implemented to help in developing an application.

5.4.1 Toward an IGONTO-Based Expert System Solution

Given that IGONTO is a prototype for an IG consulting (expert) system, there is a definitive need to develop a bespoke application or user interface for interacting with this knowledge base. This requirement became evident during the development of the prototype. For instance, to visualize the query results, AllegroGraph was used in conjunction with Gruff. AllegroGraph is a scalable RDF Graph Database that supports big data analysis. Gruff, a visualization application, can utilize AllegroGraph as a database and visualize query results, allowing interaction with nodes in various ways [23a]. While AllegroGraph enables querying with reasoning, it cannot incorporate the SWRL rules implemented in IGONTO. Therefore, to observe all inferences, the ontology must be materialized, thus saving all reasoning triples in the RDF graph before uploading to AllegroGraph. To address this issue, we explored Stardog, another RDF-Graph database with visualization and querying capabilities. Stardog facilitates reasoning not only within its application but also via its HTTP Endpoint. However, its visualization capabilities are limited, offering only a “Query builder” that restricts query options like FILTER, and the reasoning process is slow. Stardog itself

acknowledges that it "performs reasoning in a lazy and late-binding fashion: it does not materialize inferences; rather, the reasoning is performed during the query time" [23d]. This experience, made during the development of IGONTO, illustrates the challenge of finding services that satisfy all the requirements of ontology, and knowledge graph development. Moreover, there are reasons to consider implementing a proprietary front-end system for interfacing with the ontology/ knowledge graph that is hosted online, locally on-premise, or both online and on-premise:

1. **Extensibility and Specification:**
   IGONTO, being an expert system for a specific domain, may require specific features not available by third-party services. Developing a front-end application ensures independence and facilitates the implementation of specialized features as needed.

2. **Integration:**
   Another advantage is the potential for integration into larger and already existing systems. When integrating IGONTO into larger systems, reliance on third-party services can be limiting. The ability to integrate into a broader system can be enhanced by implementing features tailored to individual requirements and needs.

3. **Cost:**
   While the initial development of a custom application involves costs, it can be more economical in the long term. This is because most third-party services involve ongoing expenses, and it is unlikely that a single service will meet all of IGONTO's requirements, potentially leading to additional charges for multiple services.

4. **Useability:**

   A critical aspect of this discussion centers on user interaction. Information governance represents a complex domain that is often unfamiliar to users seeking compliance guidance. These users may lack foundational knowledge about ontologies. Even when they possess a basic understanding of ontology interaction and SPARQL query formulation, they probably will not grasp all regulatory requirements. Therefore, despite predefined SPARQL queries offering guidance on various use cases, users could still encounter problems when selecting proper queries that accurately represent their situation. Two potential solutions exist. The first involves the engagement of a domain expert to navigate user interaction through this process. However, this approach incurs ongoing costs. Alternatively, developing a user interface that enables intuitive interaction with the ontology and the knowledge base could represent a better solution. This interface would ideally bridge the gap between the user's compliance needs and the technical requirements to manage the knowledge base environment, thereby enhancing usability while reducing long-term company expenses.

Considering these points, a design concept will be presented that illustrates how a potential application might appear, how certain design decisions from the ontology will aid in the development of this application, and how some potentially desired features will enhance the system's quality.

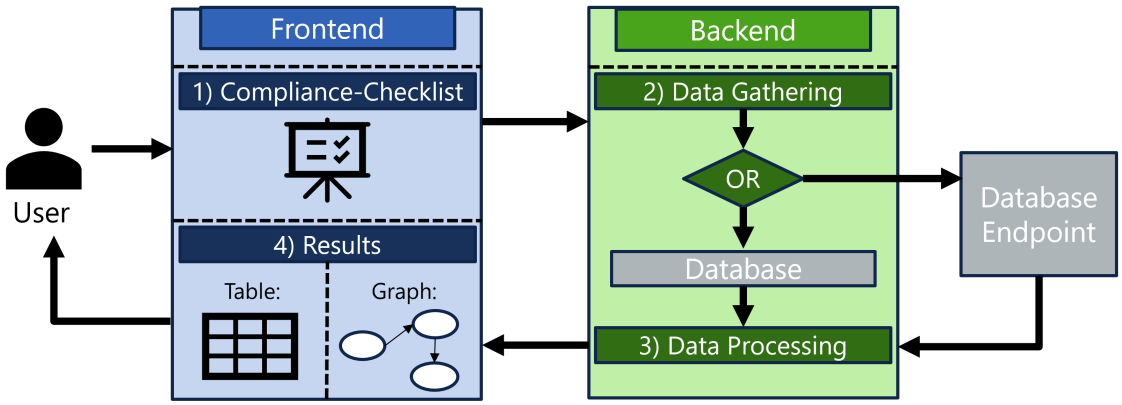


Figure 5.23: Proposed system design for the IGONTO application.

5.4.2 Application Design

This section introduces a preliminary design proposal for the IGONTO application. Figure 5.23 illustrates the proposed system architecture, comprising a front-end for user interaction and a back-end for data acquisition and processing. This architecture enables the front-end to effectively utilize processed data. The application's workflow is segmented into four subprocesses: Compliance Checklist, Data Gathering, Data Processing, and Results Presentation.

Compliance Checklist

The Compliance Checklist serves as the user's initial interaction point. Its primary function is to identify the user's specific use case, thereby guiding the subsequent data acquisition process. This is achieved through a series of predefined questions tailored to ascertain the precise use case. Key questions include the following:

1. **What is the location of the company?**
   This question aims to categorize the company within the appropriate jurisdiction, establishing the regulatory framework applicable to the user.
2. **How many employees does the company have?**
   The size of the company is an important factor. For instance, companies with less than 250 employees have to comply with fewer regulations than large companies (see 5.1).
3. **Does the company deal with sensitive data?**
   Handling sensitive data like Personally Identifiable Information (PII) data needs more caution and therefore influences the regulatory requirements.
4. **How often is data processed/accessed?**
   The regularity of data processing impacts the compliance measures needed. There is a greater likelihood that additional measures will be necessary when data are accessed more frequently.

This section introduces a preliminary design proposal for the IGONTO application. Figure 5.10 illustrates the proposed system architecture, comprising a front-end for user interaction and a back-end for data acquisition and processing. This architecture enables the front-end to effectively utilize processed data. The application's workflow has four subprocesses: Compliance Checklist, Data Gathering, Data Processing, and Results Presentation. These cover only example questions. A comprehensive checklist would implement additional queries to cover all potential use cases. In addition, questions may vary based on the regulations of the company's location. For example, U.S. regulations, not covered in this prototype, may necessitate different questions. To limit the possibilities of combination in the back-end, it would be necessary to contain answers. Suggested limitations for each question may include the following:

1. Implement a drop-down list of countries, similar to other common user applications.
2. Allow the user to input an integer representing the company's employee count. Implementing corresponding ranges in the back-end should be straightforward.
3. Offer a data type checklist a company can potentially deal with, where the user can select multiple options.
4. Include single choices such as "once a year", "once a month", which are tailored to different compliance requirements based on processing frequency.

Upon completion, the user's responses are forwarded to the back-end for further processing.

Data Gathering

With all questions answered the back-end can determine the correct use case by analyzing the combinations answered from the checklist. Based on the user responses provided, the back-end either picks predefined SPARQL queries or dynamically generates queries. These queries are then executed against the ontology to retrieve the necessary results. If the database is hosted externally, it can be accessed via its Endpoint. A possible Database endpoint could be any online RDF Graph Database. For example, stardog also provides a comprehensive HTTP API documentation [Sta23]. Alternatively, the database can be integrated directly within the back-end and queried locally using RDFLib [23c], which is a Python framework designed to work with ontologies. RDFLib allows data serialization into graphs, enabling the execution of local SPARQL queries and yielding the same results. Additionally, RDFLib supports reasoning and the pre-preparation of queries, which can enhance overall performance. Either way, both methodologies lead to the same results.

Data Processing

After receiving the query results, the data needs to be processed for easy utilization by the front-end. These results can include various formats, such as CSV, JSON, and XML. The documentation of Stardog or RDFLib details what formats are supported. Listing 5.4.2 illustrates an example in JSON. This data now requires processing and preparation for the front-end to use it effortlessly.

```
    {
        "predicate": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#enterpriseDoesNotNeed"
        },
        "small_enterprise": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/organization#
SmallEnterpriseInstance"
        },
        "object": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#RIMSOlutionInstance"
        }
    },
    {
        "predicate": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/jurisdiction#operatesInCountry"
        },
        "small_enterprise": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/organization#
SmallEnterpriseInstance"
        },
        "object": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/jurisdiction#Germany"
        }
    },
```

Listing 5.17: Example of a SPARQL result without PropertyGroups.

Listing 5.4.2 presents a code snippet from an example query that retrieves every connection from the instance “small_enterprise”. The properties are stored in the variable ’predicate’, and the objects connected to the small enterprise are represented by the variable ’object’. This result is then processed as needed. However, additional context is required to interpret this code meaningfully. This is where PropertyGroups become useful.
In the first triple, the predicate “enterpriseDoesNotNeed” is an inference and a negative object property. This connection explicitly indicates that small enterprises do not require a “RIMSolutionInstance”. The second triple is a static property, stating that the small enterprises operate in Germany. Without further modifications, further interpretations regarding this result would be difficult, including determining whether an object property represents an inference or if it should represent a negative object property. The JSON file simply lists both results as properties.
This scenario exemplifies the utility of ObjectProperties defined in Section 3.3.2 By grouping object properties, we infer additional knowledge used by the application to introduce enhanced

features. The potential uses of these in the front-end are discussed later in Section 5.4.2. While it is theoretically possible to store all object properties in a hardcoded dictionary, the organization of data should ideally be managed by the ontology rather than the application itself. By implementing PropertyGroups, they can also be queried, enhancing the richness of the results.

```
      {
        "predicate": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#enterpriseDoesNotNeed"
        },
        "small_enterprise": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/organization#
SmallEnterpriseInstance"
        },
        "objectGroup": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#
InferenceObjectPropertyGroup"
        },
        "object": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#RIMSOlutionInstance"
        }
      },
      {
        "predicate": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#enterpriseDoesNotNeed"
        },
        "small_enterprise": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/organization#
SmallEnterpriseInstance"
        },
        "objectGroup": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#
NegativeObjectPropertyGroup"
        },
        "object": {
          "type": "uri",
          "value": "http://www.semanticweb.org/igonto/igonto#RIMSOlutionInstance"
        }
      },
```

Listing 5.18: Example of a SPARQL result with PropertyGroups.

One drawback of this method is that if an object property is grouped into two PropertyGroups, the overall information connected to the object will be duplicated because the variable “objectGroup” contains two values. However, the issue can be solved by concatenating both values into one string using the “GROUP_CONCAT” operation from SPARQL. The downside of this method is that afterwards, the string needs to be separated again. Additionally, there are not many object properties involved in a single subject, which makes performance issues unlikely. The overall knowledge acquired from these PropertyGroups outweighs the drawbacks they present.
When data are processed and meaningful results are obtained, they will be returned to the front-end for further processing.

Result Representation

When the results are processed in a way that will be easy to deal with, they should be presented in a user-friendly way to answer the questions about compliance and implementation needs. One suggestion is to present the results in tabular format and as an interactive RDF graph. The tabular representation provides quick and direct information, whereas the graph should visualize the result for clarity. The idea is to allow user interaction with the graph, such as expanding existing nodes for deeper searches and allowing the possibility to visually distinguish between certain connections. A design concept is illustrated in Figure 5.24. The aforementioned PropertyGroups can now be actively visualized in the graph and filtered using the selection options in the top right corner. The user can now choose to view inferences, negative relationships, only normal relationships, or any other combination of these three options. This approach enables the easy representation of more knowledge, allowing the user to precisely understand, for instance, what they do not need to do by selecting only negative object PropertyGroups.

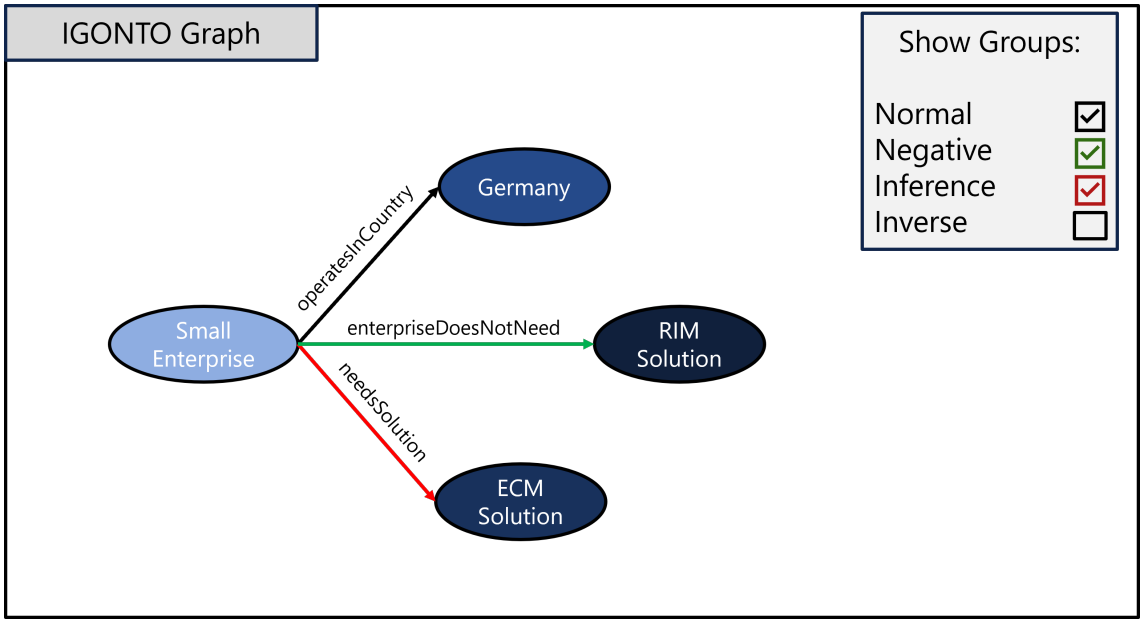


Figure 5.24: Graph visualization design concept for the IGONTO application.

Another use of an Object PropertyGroup, namely the “InverseObjectPropertyGroup” allows users to expand nodes and automatically retrieve the correct hierarchy, enhancing understandability. Meanwhile, the application uses the same generic query, with the exception being that the subject is changed (namely, the selected instance that the user wishes to expand). Accordingly, the query can be implemented as a single reusable query that asks for all connections with their corresponding PropertyGorup, representing all objects connected by this relationship, including the inverse relationship (“the way back”). Afterward, the application determines which relationship between the subject and object is an inverse property and decides, based on the user’s selection and desired hierarchy (e.g. top-down, bottom-up, or both ways), how the data is displayed. This approach results in an automatically generated tree structure without requiring the application to be aware of every relationship and object. Without this method, the application would be limited to only predefined SPARQL queries, restricting the user’s ability to interact with the graph as they wish.

5.4.3 Agile Development

While the previous section focused on integrating the IGONTO prototype into an independent application, this section shifts its attention to how the further development of this prototype can be managed within an agile team. The division of IGONTO into several independent sub-ontologies, outlined in Section 3.1.1, is crucial to agile development. Additionally, this approach exploits the capability to store RDF graphs into different named graphs. A named graph is a set of triples with a specific name, based on a certain ontology. Named graphs facilitate the organization of RDF data into different subsets. These subsets can be increments from each other or from different viewpoints.
Figure 5.25 illustrates how the prototype can be developed by different teams. Each sub-ontology can be developed by an independent person or department. The various lines represent different branches in the development process. The overall ontology can be saved in the “Master” branch. Whenever a pull request is created, the corresponding graph can be validated and, if successful, saved as a named graph in the database. The graphs can be named based on the development branch and the pull request number to uniquely identify each graph and trace every change. Upon merging into the master branch, a new master graph can be stored in the database as a named graph with an increment of the version number. The integration of a new pull request or merge into a new named graph can be automated using workflow files, similar to the validation of every ontology.
The exact implementation depends on the database used. When using Stardog, its HTTP API enables the direct integration of newly named graphs. With RDFlib, these named graphs need to be stored in the *Dataset* object provided by RDFlib. Several named graphs can be integrated within this dataset.

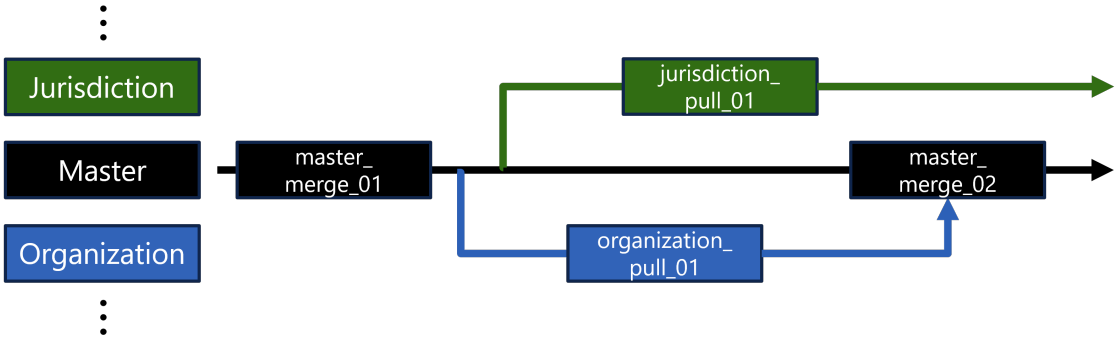


Figure 5.25: Agile development of IGONTO.

6 Conclusion

The work of this thesis includes the development of an ontology on information governance, IGONTO, capturing valuable knowledge found in the Information Governance (IG) domain. Part of the work focused on implementing of an IG Knowledge Graph a prototype based on IGONTO. The work of Mega [IGONTO23] was used as a foundation, where the IG domain was analyzed and its concepts identified.
We divided the ontology into several sub-ontologies, namely *Jurisdiction*, *Organization*, *Information*, *Lifecycle*, and *Implementation*. To use the correct reasoning approach, we parameterized the reasoning aspects, analyzed the IGONTO domain needs, and decided to use a logic-based approach with Description Logic (DL) with the extension of Semantic Web Rule Language (SWRL) rules.
The effectiveness of the ontology is demonstrated using two different use cases, representing small and large companies, to demonstrate its adaptability in providing compliance solutions following GDPR requirements. These use cases were evaluated using SPARQL queries, with the results visualized using AllegroGraph and Gruff.
To ensure consistency, SHACL validation schemas were implemented for each sub-ontology, where IGONTO's implemented structure provides a systematic approach, enabling potential automation in future development stages.
Looking forward, the work proposes an application architecture for IGONTO that aims to simplify user interaction with the ontology. This architecture leverages PropertyGroups to add additional meaning to relationships and improve the user's ability to interpret and visualize these relationships.

7 Outlook

The IGONTO prototype has demonstrated considerable potential but still needs some improvements to be fully operational. This chapter outlines some improvements to achieve compliance, including refinement 7.1 and the use case extension 7.2. Additionally, the possible improvements to show the potential of this prototype explained in Section 5.4 are summarized to provide a comprehensive outlook.

7.1 Refinement

The structure of the prototype is already robust, including the division of ontologies, the description logic, the SWRL rules, and the validation. The focus of refinement involves the connection between requirements, rights, obligations, and GDPR articles.

As explained in Section 4.1 obligations, rights, and capabilities are linked to GDPR articles. Figure 7.1 illustrates why and where refinement should be performed. The *Obligation To Ensure Accuracy Of Data Instance* involves *Article 5* and *Article 16*, as well as a citation from the articles explaining why the relationship exists in the first place. Sometimes, like in *Article 5*, the citations are straightforward and directly include keywords ("accuracy") with their explanation. Here, the involvement of *Article 5* in this obligation is clear. At other times, out-of-the-box explanations are necessary to connect the articles. For instance, *Article 16* states that,

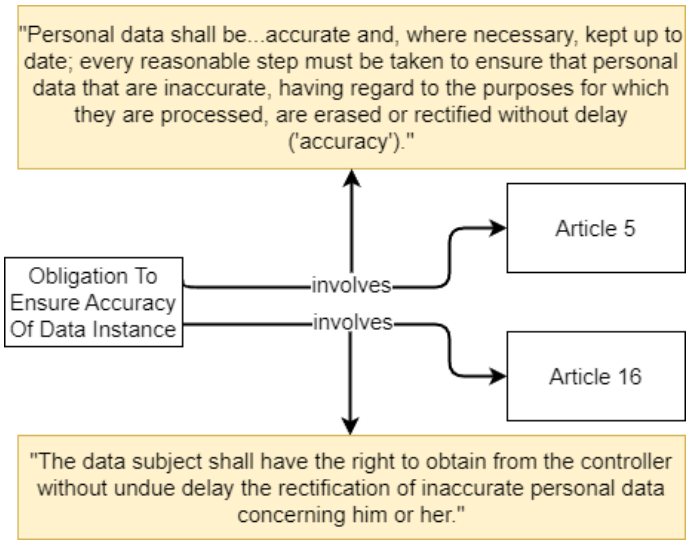


Figure 7.1: Refinement reason example.

> "The data subject shall have the right to obtain from the controller without undue delay the rectification of inaccurate personal data concerning him or her."

Here, the connection to the obligation is not immediately apparent. This article obligates controllers to correct inaccuracies in personal data. However, this requires that the data be kept accurate. Otherwise, rectification will be challenging or could lead to errors within their data, potentially resulting in compliance violations.
This demonstrates that it is often not clear which articles are relevant to obligations, rights, or capabilities. Although it was not a requirement for this prototype to be compliant, to show that the idea of the whole process works, the need for refinement should still be mentioned. IGONTO is an expert system that requires a high degree of accuracy to ensure compliance, and therefore, it should be revised by (data compliance) experts.

7.2 Use Case Extension

The use case extension also needs to be completed. In this prototype, we focused on the size of enterprises to demonstrate the validity of the idea. There are additional use cases and parameters that play a role in compliance, some of which have been identified but not implemented and are briefly mentioned in Section 5.4.2.

The first additional use case is the type of data the enterprise processes. If the data is sensitive, such as PII data, more security, and therefore, stricter regulations need to be followed. The other use case is the frequency of data processing. These parameters also have to be added to the ontology, including the right connections and inferences. Although we did not directly implement these use cases, the concept has already been implemented. For example, in Figure 4.5, the sensitive data are already implemented as a type of control data. The frequency of data access can be implemented by attributes or by adding a class that represents the concept of date, including the start date, end date, and other subclasses required to implement a process cycle.
Besides the already identified but not implemented use cases, there could be more use cases needed to ensure compliance. As mentioned earlier, IGONTO is an expert system and should be revised by experts, not only regarding article refinement but also regarding use case extension.

8 Related Work

In “An Ontology-based Knowledge Management Model on Information Governance”, Cataldo Mega [IGONTO23], analyzes the Information Governance (IG) domain and suggests several models. These models describe the concepts of an enterprise and its connection to IG. Mega’s analysis includes IG domain sources such as associations describing compliance capabilities, ISO standards, regulations, Enterprise Information Management (EIM) standards, and service vendors. The synthesis of these sources forms a harmonized IG taxonomy, which is divided and implemented in multiple IG knowledge models. These models represent enterprise concepts from different perspectives, and their concepts are used for the sub-ontologies within IGONTO. These include the Organizational Model (ORG), Information Model (INF), System Model (SYS), Component Model (CPT), Lifecycle Model (LCS), and the Platform Model (PLT), where each model concept is part of IGONTO.

Ling Tian et. al. [TZW+22] in “Knowledge graph and knowledge reasoning: A systematic review” analyzed several reasoning approaches. These include reasoning based on logic rules, reasoning based on logic, statistics, graph structure, representation learning, and neural networks. From their summaries of the advantages and disadvantages, we extracted these statements and turned them into reasoning parameters. After analyzing the IGONTO requirements and comparing them with the reasoning parameters, we identified a suitable reasoning solution for our prototype.

Since we chose to use description logic as a reasoning method, we referred to Claudia d’Amato et. al’s. [dFE05] and Ian Horrocks et. al’s. [HKS06] theoretical definitions of the description logic used in OWL 2. We synthesized both definition languages to ensure they aligned and drew conclusions on how these theorems are defined in OWL 2 keywords.

For the implementation side, we gained important insights from the official OWL 2 documentation [23b], including its implemented description logic [W3C12]. Using the SHACL framework documentation [23e] we implemented validation schemas. These schemas validate not only the ontology structure but also its inferences. Additionally, with the SHACL PropertyGroups, we implemented further knowledge for the relationships within IGONTO, enabling more efficient development of future applications built around this prototype.

With the help of the triple-store browser AllegroGraph, including the integrated tool Gruff [23a], we effectively visualized our results. The online RDF-Database Stardog [23d] facilitated easy and fast ontology validation, including the capability for dynamic reasoning queries for inference testing.

9 Acknowledgement

I would like to thank Prof. Dr.-Ing. habil. Bernhard Mitschang for the opportunity to complete my master's thesis at the Institute for Parallel and Distributed Systems.

I'm extremely grateful to Dipl. Phys. Cataldo Mega for his competent and friendly support during the preparation of this master thesis. Thank you for the data sets provided, the major contribution to the Implementation Ontology, the proofreading, and the valuable suggestions that enriched this work.

Bibliography

[23a] *About Franz - AllegroGraph*. https://allegrograph.com/about-franz/. 2023 (cit. on pp. 18, 96, 103, 117).

[23b] *OWL Web Ontology Language Reference*. https://www.w3.org/TR/owl-ref/. 2023 (cit. on pp. 39, 117).

[23c] *RDFLib Documentation*. https://rdflib.readthedocs.io/en/stable/index.html. 2023 (cit. on p. 106).

[23d] *Stardog Latest Documentation*. https://docs.stardog.com/. 2023 (cit. on pp. 18, 101, 104, 117).

[23e] *W3C Shapes Constraint Language (SHACL) Vocabulary*. https://www.w3.org/ns/shacl.ttl. 2023 (cit. on pp. 37, 38, 117).

[Alf24a] Alfresco. *Alfresco Content Services Documentation*. 2024. URL: https://docs.alfresco.com/content-services/latest/ (cit. on p. 92).

[Alf24b] Alfresco Software, Inc. *Software Architecture*. https://docs.alfresco.com/content-services/latest/develop/software-architecture/. 2024 (cit. on p. 95).

[Ben10] W. Benedon. "History of Records and Information Management". In: (2010), pp. 2133–2141. DOI: 10.1081/E-ELIS4 (cit. on p. 51).

[Bjo75] L. A. Bjork. "Generalized Audit Trail Requirements and Concepts for Data Base Applications". In: *IBM Syst. J.* 14 (1975), pp. 229–245. DOI: 10.1147/sj.143.0229 (cit. on p. 55).

[Boe88] B. W. Boehm. "A spiral model of software development and enhancement". In: *Computer* 21.5 (1988), pp. 61–72. DOI: 10.1109/2.59 (cit. on p. 103).

[BP08] S. G. Blethyn, C. Y. Parker. "Data dictionary". In: *World Statistics Pocketbook 2007* (2008). DOI: 10.1016/B978-0-7506-1038-4.50013-0 (cit. on p. 55).

[CBK04] J. Callahan, C. Bastos, D. Keyes. "The Evolution of IT Governance at NB Power". In: (2004), pp. 343–356. DOI: 10.4018/978-1-59140-140-7.CH013 (cit. on p. 50).

[CFG20] A. Cimmino, A. Fernández-Izquierdo, R. García-Castro. "Astrea: Automatic Generation of SHACL Shapes from Ontologies". In: *The Semantic Web* 12123 (2020), pp. 497–513. DOI: 10.1007/978-3-030-49461-2_29 (cit. on p. 102).

[CJX20] X. Chen, S. Jia, Y. Xiang. "A review: knowledge reasoning over knowledge graph". In: *Expert Systems With Applications* 141 (2020), p. 112948. DOI: 10.1016/j.eswa.2019.112948 (cit. on p. 24).

[dFE05] C. d'Amato, N. Fanizzi, F. Esposito. "A Dissimilarity Measure for the ALC Description Logic." In: *SWAP*. 2005 (cit. on pp. 29, 117).

[DL96] G. De Giacomo, M. Lenzerini. “TBox and ABox reasoning in expressive description logics”. In: *KR* 96 (1996), pp. 316–327 (cit. on p. 29).

[Fit90] M. Fitting. “First-order logic and automated theorem proving”. In: (1990). DOI: 10.1007/978-1-4684-0357-2 (cit. on p. 24).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8 (cit. on p. 64).

[Fra13] P. C. Franks. “Records and Information Management”. In: *Internal Controls Toolkit* (2013). DOI: 10.1002/9781119554424.ch9 (cit. on p. 54).

[Hei21] I. Heine. *3 Years Later: An Analysis of GDPR Enforcement*. 2021. URL: https://www.csis.org/blogs/strategic-technologies-blog/3-years-later-analysis-gdpr-enforcement (cit. on p. 27).

[HH13] A. Haider, W. Haider. “Improving engineering asset lifecycle data quality: Setting the rules”. In: *2013 Proceedings of PICMET ’13: Technology Management in the IT-Driven Services (PICMET)* (2013), pp. 1200–1206 (cit. on p. 50).

[HKS06] I. Horrocks, O. Kutz, U. Sattler. “The even more irresistible SROIQ”. In: Jan. 2006, pp. 57–67 (cit. on pp. 29, 31, 117).

[HOD10] S. Hassanpour, M. O’Connor, A. K. Das. “A Software Tool for Visualizing, Managing and Eliciting SWRL Rules”. In: (2010), pp. 381–385. DOI: 10.1007/978-3-642-13489-0_28 (cit. on p. 77).

[HPBT05] I. Horrocks, P. Patel-Schneider, S. Bechhofer, D. Tsarkov. “OWL rules: A proposal and prototype implementation”. In: *J. Web Semant.* 3 (2005), pp. 23–40. DOI: 10.1016/J.WEBSEM.2005.05.003 (cit. on p. 74).

[HS05] I. Horrocks, U. Sattler. “A Tableaux Decision Procedure for SHOIQ”. In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-19)*. Morgan Kaufmann. Los Altos, 2005, pp. – (cit. on p. 31).

[IGONTO23] C. Mega. “An Ontology-based Knowledge Management Model on Information Governance.” In: *Proceedings ofthe 25th International Conference on Enterprise Information Systems (ICEIS 2023) - Volume 2, pages 168-178* (2023). DOI: 10.5220/0011985000003467 (cit. on pp. 17, 18, 45, 113, 117).

[KBM+11] S. Köhler, S. Bauer, C. J. Mungall, G. Carletti, C. L. Smith, P. N. Schofield, G. V. Gkoutos, P. N. Robinson. “Improving Ontologies by Automatic Reasoning and Evaluation of Logical Definitions”. In: *BMC Bioinformatics* (2011). DOI: 10.1186/1471-2105-12-418 (cit. on p. 23).

[Lam14] J. Lambrechts. “Information Lifecycle Governance (ILG)”. In: *Journal of Telecommunications and the Digital Economy* (2014). DOI: 10.18080/JTDE.V2N3.284 (cit. on p. 52).

[Mil07] C. Milne. “Taxonomy development: assessing the merits of contextual classification”. In: *Records Management Journal* 17 (2007), pp. 7–16. DOI: 10.1108/09565690710730660 (cit. on p. 55).

[Nd20] A. Nikolov, M. d'Aquin. "Uncovering Semantic Bias in Neural Network Models Using a Knowledge Graph". In: *Proceedings of the 29th ACM International Conference on Information Knowledge Management* (2020). DOI: 10.1145/3340531.3412009 (cit. on p. 26).

[OAS19] OASIS TOSCA TC. *Topology and Orchestration Specification for Cloud Applications Version 1.3*. OASIS Standard. Nov. 2019. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.pdf (cit. on p. 70).

[PB14] R. Pressman, D. Bruce R. Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014. ISBN: 9780078022128. URL: https://books.google.de/books?id=i8NmnAEACAAJ (cit. on p. 103).

[PK21] P. Pareti, G. Konstantinidis. "A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs". In: (Dec. 2021). DOI: https://doi.org/10.1007/978-3-030-95481-9_6 (cit. on p. 35).

[Pro23] Protégé Project. *Protégé*. https://protegeproject.github.io/protege/. 2023 (cit. on p. 18).

[Pur06] W. C. Purdy. "Inexpressiveness of First-Order Fragments". In: *The Australasian Journal of Logic* 4 (2006). DOI: 10.26686/AJL.V4I0.1777 (cit. on p. 28).

[QZC09] Y. Qing, L. Zhu, W. Chen. "Research on ontology matching method based on description logics reasoning mechanism". In: *2009 International Conference on Web Information Systems and Mining* (2009). DOI: 10.1109/wism.2009.50 (cit. on p. 23).

[ROM11] A. Rivolli, J. P. Orlando, D. A. Moreira. "An Analysis of Rules-based Systems to Improve SWRL Tools". In: (2011), pp. 191–194. DOI: 10.5220/0003439901910194 (cit. on p. 77).

[SBW19] F. A. Setiawan, E. K. Budiardjo, W. C. Wibowo. "ByNowLife: A Novel Framework for OWL and Bayesian Network Integration". In: *Information* (2019). DOI: 10.3390/info10030095 (cit. on p. 23).

[SGSH16] L. T. Slater, G. V. Gkoutos, P. N. Schofield, R. Hoehndorf. "Using AberOWL for Fast and Scalable Reasoning Over BioPortal Ontologies". In: *Journal of Biomedical Semantics* (2016). DOI: 10.1186/s13326-016-0090-0 (cit. on p. 23).

[SH20] S. Stephen, T. Hahmann. "Model-Finding for Externally Verifying FOL Ontologies: A Study of Spatial Ontologies". In: (2020), pp. 233–248. DOI: 10.3233/faia200675 (cit. on p. 28).

[Sir21] K. B. Sirait. "The Interrelation of Information Technology Governance and Enterprise Risk Management to The Organization's Performance: A Review of Empirical Literature". In: *RSF Conference Series: Business, Management and Social Sciences* (2021). DOI: 10.31098/bmss.v1i5.450 (cit. on p. 50).

[SL14] M. Simard, D. Laberge. "Governance Representations in Temporary Organization: A Case of Governance Sensemaking". In: *Procedia Technology* 16 (2014), pp. 967–978. DOI: 10.1016/J.PROTCY.2014.10.050 (cit. on p. 50).

[Som11] I. Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011. ISBN: 9780137053469. URL: https://books.google.de/books?id=l0egcQAACAAJ (cit. on p. 103).

[Sta23] Stardog Union. *Stardog HTTP Documentation*. 2023. URL: https://stardog-union.github.io/http-docs/ (cit. on p. 106).

[Sta24] Stardog. *Studio SHACL Released*. https://www.stardog.com/blog/studio-shacl-released/. 2024 (cit. on p. 101).

[Tra08] L. A. Tran. "A Semantic Web Primer". In: *Library Hi Tech* (2008). DOI: 10.1108/07378830810903409 (cit. on p. 23).

[TZW+22] L. Tian, X. Zhou, Y.-P. Wu, W.-T. Zhou, J.-H. Zhang, T.-S. Zhang. "Knowledge graph and knowledge reasoning: A systematic review". In: *Journal of Electronic Science and Technology* 20.2 (2022), p. 100159. ISSN: 1674-862X. DOI: https://doi.org/10.1016/j.jnlest.2022.100159. URL: https://www.sciencedirect.com/science/article/pii/S1674862X2200012X (cit. on pp. 24, 117).

[VMT13] R. Virgilio, A. Maccioni, R. Torlone. "Converting relational to graph databases". In: June 2013. DOI: 10.1145/2484425.2484426 (cit. on p. 26).

[W3C12] W3C OWL Working Group. *OWL 2 Web Ontology Language: Direct Semantics*. 2012. URL: https://www.w3.org/TR/owl2-direct-semantics/#Introduction (cit. on pp. 31, 33, 117).

[Wor12] World Wide Web Consortium (W3C). *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. https://www.w3.org/TR/xmlschema-2/. 2012 (cit. on p. 23).

[Wor21] World Wide Web Consortium. *DCAT 3.0: An RDF Vocabulary Designed to Facilitate Interoperability Between Data Catalogs Published on the Web*. 2021. URL: https://www.w3.org/TR/vocab-dcat-3/ (cit. on p. 23).

All links were last followed on January 30, 2024.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature