



**Technological Educational Institute
of Thessaloniki**



Department of Information Technology

**SPARQL, the query language for RDF
(Bachelor's Thesis)**

Written by:

Fotini Bogiatzi

fotinibogiatzi@gmail.com

**Thesis Supervisor:
Dr. Euclid Keramopoulos**

euclid@it.teithe.gr

Thessaloniki 2011

Prologue

This Bachelor's thesis titled "SPARQL, the query language for RDF" was written by Fotini Bogiatzi during the academic semester of 2010-2011 while studying in the Department of Information Technology, at the Technological Educational Institute of Thessaloniki, Greece. The supervisor for this thesis was Dr. Euclid Keramopoulos.

Abstract

The goal of this thesis is to provide an introduction for people new to the Semantic Web, about the query language SPARQL. The Semantic Web is the Web of meaning, where information is no longer only displayed by machines, but also understood and processed by them. The goal of Semantic Web is to have unambiguous information, reusability of data and distributed knowledge. To realize the Semantic Web goal, data on the Web needs to be represented in a simple data model that even machines can understand. This model is the Resource Description Framework (RDF), a graph model written in triples. The RDF language describes resources. Ontologies are needed to describe concepts that the RDF data will build upon. Ontologies are designed to create class hierarchies, abstract concepts, etc. The Web Ontology Language (OWL) is used to create ontologies. With ontologies and data, RDF datasets are born, existing all over the Web, available for sharing, reusability, and distribution of information. The RDF data needs to be queried, and therefore a query language is needed. In the Semantic Web vision, this language is called SPARQL Query Language for RDF. This thesis analyzes SPARQL, its syntax, its constructs and provides examples of its use.

Περίληψη

Ο στόχος της παρούσας πτυχιακής είναι η δημιουργία μιας ολοκληρωμένης εισαγωγής στη γλώσσα αιτημάτων SPARQL. Στοχεύει σε άτομα που δεν έχουν γνώσεις για τον Σημασιολογικό Ιστό και τις τεχνολογίες του. Ο Σημασιολογικός Ιστός είναι επέκταση του Ιστού όπου προσδίδει νόημα στα δεδομένα. Με τον Σημασιολογικό Ιστό η πληροφορία δεν παρουσιάζεται απλά από τις μηχανές, αλλά μπορεί να κατανοηθεί και να επεξεργαστεί από αυτές. Ο στόχος του Σημασιολογικού Ιστού είναι η δημιουργία ξεκάθαρης και σαφής πληροφορίας, η επαναχρησιμοποίηση των δεδομένων και η εύκολη κατανομή γνώσης. Για να επιτευχθεί ο στόχος του Σημασιολογικού Ιστού, πρέπει τα δεδομένα που υπάρχουν στον Ιστό να αναπαρασταθούν με τη μορφή ενός απλού μοντέλου δεδομένων. Με αυτόν τον τρόπο ακόμα και οι μηχανές μπορούν να κατανοήσουν τα δεδομένα. Το μοντέλο αυτό των δεδομένων ονομάζεται Resource Description Framework (RDF), και είναι ένα μοντέλο γράφου. Η γλώσσα RDF είναι μια γλώσσα αναπαράστασης της πληροφορίας σχετικά με πόρους στον Ιστό. Οντολογίες προσδίδουνε μετα-πληροφορία στα ήδη υπάρχοντα δεδομένα. Οι οντολογίες σχεδιάστηκαν για την δημιουργία ιεραρχιών κλάσεων, αφηρημένων εννοιών, κτλ. Οι οντολογίες δημιουργούνται με την γλώσσα Web Ontology Language (OWL). Με οντολογίες και δεδομένα, τα ΡΔΦ δατασεντς δημιουργούνται, τοποθετούνται στον Ιστό, και είναι έτοιμα για διανομή, κατανομή και επαναχρησιμοποίηση της πληροφορίας. Τα RDF δεδομένα χρειάζονται μια γλώσσα αιτημάτων. Στον Σημασιολογικό Ιστό, αυτή η γλώσσα ονομάζεται SPARQL. Η παρούσα πτυχιακή αναλύει την SPARQL, το συντακτικό της, και παρέχει παραδείγματα για την χρήση της.

Acknowledgements

I would like first to give thanks to my supervisor, Dr. Euclid Keramopoulos for his patience when I felt stressed, and for his help.

Furthermore, I would like to express my immense gratitude for Michal Nagy, for his support and fruitful discussions.

A shout-out goes to my friends for their support, and to my family for their patience.

Contents

Contents	8
List of Figures	13
1 Semantic Web	19
1.1 General	19
1.2 RDF	21
1.2.1 RDF Graph	21
1.2.2 URI	22
1.2.3 Syntax	24
1.2.3.1 Blank Nodes	26
1.2.4 RDF Containers	27
1.2.4.1 Unordered Container, Bag	28
1.2.4.2 Ordered Container, Sequence	28
1.2.4.3 Container of Alternatives, Alternative	28
1.2.5 RDF Collections	29
1.3 Notations	29
1.3.1 General	29
1.3.1.1 Conversion between formats	29
1.3.2 Notation 3	30
1.3.3 Turtle	31
1.3.4 N-Triples	32
1.3.5 RDF/XML	33
1.4 RDF Storages	34
1.4.1 Jena	35

1.4.2	Oracle Spatial RDF	35
1.4.3	Sesame	35
1.4.3.1	Summary	36
1.4.3.2	Namespaces	37
1.4.3.3	Contexts	37
1.4.3.4	Types	38
1.4.3.5	Explore	39
1.4.3.6	Query	40
1.4.3.7	Export	41
1.5	Epilogue	42
2	Ontologies	45
2.1	General	45
2.2	RDFS – RDF Schema	46
2.2.1	Classes	48
2.2.2	Properties	49
2.2.3	Container Classes and Properties	50
2.2.4	RDF Collections	52
2.3	OWL	54
2.3.1	Introduction to OWL Sub-languages	55
2.3.1.1	OWL Datatypes	56
2.3.2	OWL Lite	56
2.3.2.1	Features	57
2.3.2.2	Classes	57
2.3.2.3	Equality - Inequality Features	57
2.3.2.4	Property Characteristics	58
2.3.2.5	Property Restrictions	61
2.3.2.6	Restricted Cardinality	62
2.3.2.7	Extra Restrictions	63
2.3.3	OWL DL	64
2.3.4	OWL Full	64
2.4	Reasoners	65
2.4.1	Forward chaining reasoners	65

2.4.2	Backward chaining reasoners	66
2.4.3	Description Logic reasoners	66
2.5	Epilogue	66
3	Protégé	69
3.1	Introduction	69
3.2	Protégé Features	70
3.3	Comparison of Versions	72
3.4	Demonstration of Protégé	73
3.4.1	Classes	74
3.4.2	Properties	77
3.4.3	Instances - Individuals	79
3.4.4	Refactoring	80
3.4.5	Reasoning	80
3.4.6	OWLviz	81
3.4.7	Export	82
3.4.8	Plug-Ins	82
3.5	Epilogue	84
4	SPARQL	85
4.1	General	85
4.2	Syntax	86
4.2.1	Basic Query	87
4.2.1.1	Blank nodes in a query result	89
4.2.1.2	Guidelines and syntax	89
4.2.2	Graph Patterns	91
4.2.2.1	Basic Graph Patterns	91
4.2.2.2	Group Graph Patterns	91
4.2.2.3	Optional Graph Patterns	94
4.2.2.4	Alternative Graph Patterns	97
4.2.2.5	Patterns on Named Graphs	99
4.2.3	Solution Sequence Modifiers	100
4.2.3.1	ORDER BY	100

4.2.3.2	Projection	102
4.2.3.3	DISTINCT	102
4.2.3.4	REDUCED	103
4.2.3.5	LIMIT	104
4.2.3.6	OFFSET	106
4.2.4	Query Forms	106
4.3	Comparison with SQL and XQuery	111
4.4	Epilogue	112

Bibliography	115
---------------------	------------

List of Figures

1.1.1 The Semantic Web stack [10]	20
1.2.1 RDF graph representing one triple	21
1.2.2 RDF graph representing multiple triples	22
1.2.3 Subject - Predicate - Object statement	24
1.2.4 Pairwise disjointness	27
1.2.5 Example of a blank node [37]	28
1.3.1 The relationship between RDF notations	30
1.4.1 Starting screen of Sesame Workbench	36
1.4.2 Creating a new repository in Sesame	36
1.4.3 Persistent storage for our data	37
1.4.4 Summary of the new repository	37
1.4.5 Uploading of RDF data to repository	38
1.4.6 Existing namespaces in the repository	38
1.4.7 Contexts in the repository	39
1.4.8 Existing types in the repository	39
1.4.9 All the object properties	40
1.4.10 All the datatype properties	40
1.4.11 Datatype properties of an ontology with inferred axioms	41
1.4.12 All the symmetric properties	41
1.4.13 The empty query window in Sesame Workbench	42
1.4.14 Choosing the desirable format to export the RDF data	42
2.2.1 An RDF container of type <code>rdf:Bag</code>	51
2.2.2 An example of an Alternative [37]	52
2.3.1 OWL sub-languages	56

2.3.2 A visualization of the owl:AllDifferent construct	59
2.3.3 Illustration of the inverseOf construct	59
3.4.1 Protégé 4.0.2 welcome screen	74
3.4.2 Active Ontology screen	74
3.4.3 Entities tab in the family ontology	75
3.4.4 Adding a subclass	76
3.4.5 List of Classes of the family ontology	76
3.4.6 Choosing an equivalent class for the class Child	77
3.4.7 Creating cardinality restriction for the hasMother property	77
3.4.8 List of Object Properties in family ontology	78
3.4.9 List of Data Properties for family ontology	79
3.4.10 Individuals Tab with additional information	80
3.4.11 Object Property assertion for Individual "Sophia"	81
3.4.12 List of Individuals grouped by Class	82
3.4.13 Results from the reasoner	83
3.4.14 Inferred class hierarchy	83
3.4.15 Visualization of the class hierarchy	84
4.2.1 Query result	88
4.2.2 Individuals with their first name as "Peter"	92
4.2.3 All individuals under the age of 30	93
4.2.4 Results of the bound function	94
4.2.5 Results of !bound()	94
4.2.6 Parents with children and their friends	95
4.2.7 Parents with children that could possibly have friends	96
4.2.8 FILTER on OPTIONAL clause effect	97
4.2.9 People that belong to the Sibling class	98
4.2.10 Showing people who belong to the Brother or Sister class	98
4.2.11 Show the first names for those that belong in the Brother class	99
4.2.12 All the male individuals that have age	101
4.2.13 All the male individuals that have an age, ordered according to their age	101
4.2.14 Show possible grandparents in the dataset	102

4.2.15	The results are significantly decreased now, all of them unique	103
4.2.16	All the possible mothers without any modifier	104
4.2.17	All the mothers with the effect of the REDUCED modifier	104
4.2.18	People in the dataset under age 50	105
4.2.19	The three youngest people in the dataset	106
4.2.20	The two oldest individuals that are under 50	107
4.2.21	The CONSTRUCT result set	108
4.2.22	A new hasNiece property constructed	108
4.2.23	The family:hasNephew property created by the CONSTRUCT clause	109
4.2.24	The ASK query form returns "no"	110
4.2.25	The ASK query form returns "yes"	110
4.2.26	Results of the DESCRIBE query form	111

Introduction

The World Wide Web as we know it today is a Web of documents. The machines were designed to display the data located on the Web to humans, but are not able to understand the data. Therefore they cannot make use of it. The vision to have machines that process information developed and so Semantic Web was born. Semantic Web aims to make information so simple that even machines can understand it. Therefore, complex human language structures need to be explicitly defined in a simple way. This is the role of the Resource Description Framework (RDF), from which RDF data is created. Abstract concepts and class hierarchies need to be defined. Vocabularies are used for this purpose in order to describe data using those concepts. These vocabularies are called ontologies, and one of the possible languages to create them is the Web Ontology Language (OWL). There is also a need for a tool to query the RDF data, which is called SPARQL query language for RDF (recursively SPARQL). This thesis introduces the Semantic Web technology and related concepts. It also analyzes the technologies with examples, in order to show SPARQL and what it is capable of.

In Chapter 1, we introduce the Semantic Web and present its goals and benefits. We introduce the RDF language, a building block for the Semantic Web. RDF is a language describing resources in the form of a graph. This graph can represent statements about facts, real or abstract. Resources are identified on the Web by Uniform Resource Identifiers), which are unique and global and do not necessarily locate a website on the Web. In the same chapter we also describe the syntax for RDF, as well as RDF containers and collections. Notations are ways to write the RDF graph in a textual format. These are analyzed and shown with examples. Tools to store RDF data are needed, the so-called RDF storages. Sesame is an RDF storage that is analyzed with screenshots because it will be used to demonstrate the SPARQL queries. In Chapter 2, we give an introduction to ontologies. Ontologies are needed to describe abstract concepts in a domain model, to

create class hierarchies, define the structure of data, etc. We start with describing features that exist in RDF Schema, a simple descriptive language for the creation of ontologies. Furthermore we analyze and give examples for OWL features, since they build upon the RDF Schema features. OWL language is the language used to create ontologies, and it has three sublanguages, OWL Lite, OWL DL and OWL Full.

Chapter 3 demonstrates the ontology that was created for this thesis in Protégé. Protégé is an open-source platform to create ontologies and domain models. This tool can be used instead of manually writing OWL. First we present the features in the latest stable version of Protégé, and then we compare the newer with the older stable version. After that we demonstrate the usage of Protégé with the aid of screenshots.

Chapter 4 reaches the main point of this thesis, SPARQL. SPARQL is the query language for RDF. In the first section of the chapter we provide general information about SPARQL, from its creation to its different versions. Then we describe the syntax used, how to form a basic query, add constraints, etc. The examples are used in Sesame, and screenshots demonstrate the results. In the last section we compare SPARQL to SQL and XQuery.

Chapter 1

Semantic Web

1.1 General

The World Wide Web was created for humans, therefore most of the data on the Web today is only displayed by machines, not understood by them [RDF]. Developers started to think of a possible way to make these machines to perform tasks more intelligently. There are two approaches to this problem.

One way is to make the machines so intelligent that they can understand the complex human language. Artificial Intelligence is trying to solve this problem.

Another way is to simplify the description of the world so much, that even computers can understand the meaning. This is where Semantic Web steps in. The Semantic Web is the Web of data [35]. This is a vision for the Web that is being realized currently, in which data is defined and described in a simple way. In Figure 1.1.1, we can see the Semantic Web stack.

The Semantic Web realizes two goals - unambiguity and full descriptions [21]. One goal of the Semantic Web is to have unambiguous and context-independent information. The Resource Description Framework (RDF) was created for this reason, to simplify the complexity of the human language, structure it and make it unambiguous. The Universal Resource Identifiers (URI) exist to create unambiguous global references to entities. Ontologies create unambiguous concepts. Another goal is to have more meaning with the same data. By explicitly defining concepts and using reasoners on the ontologies, new data can be inferred. By generating new data from the reasoners we can have more meaning. Data designed to be used with the Semantic Web technology has to be in a

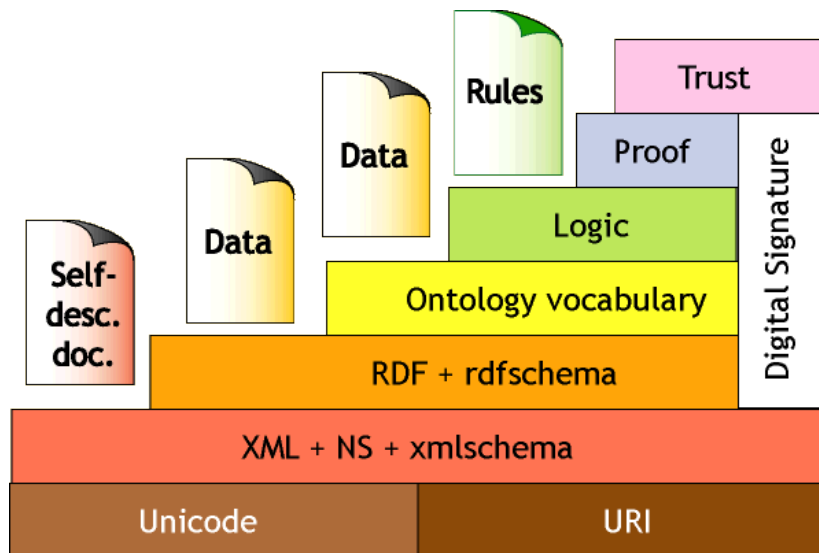


Figure 1.1.1: The Semantic Web stack [10]

certain format called RDF. RDF is a simple data model designed to describe concepts and express knowledge in a distributed, decentralized world [39] [40]. The World Wide Web Consortium (W3C) created the first recommendation for RDF in 1999 [28]. RDF is a general-purpose language to describe resources on the Web. A resource can be a person, a song, a video, a place, and more. The computers this way no longer process or present data without understanding what it is about, but with RDF defined data, machines can understand the meaning behind the data. This opens up new possibilities for "intelligent" applications that process information.

"One benefit of the RDF property-centric approach is that it allows anyone to extend the description of existing resources, one of the architectural principles of the Web" [31] [11].

There is no permission needed to create new concepts in RDF, and RDF applications can thus connect RDF files posted on various locations over the Internet and store them into a single knowledge base. Let us consider that Amazon creates an RDF document with prices of their books and their corresponding reviews, and BookDepository does the same using the same vocabulary. An intelligent Semantic Web application can now take the ontology, the RDF documents, and create a greater knowledge database by posting prices and reviews of both online sellers for each book, therefore creating a comparison.

Another benefit is that new information can be inferred from these documents. This is done by linking documents with their common vocabularies and by allowing each document

to use any vocabulary needed [13]. These vocabularies are called ontologies, and they will be discussed further in the thesis. For example if one creates a complete ontology about humans, there is no need for anyone to create another such ontology again. Whether future programmers want to create ontologies about books and authors, or athletes and sports, anything that is related to humans is already defined. The only thing needed to do is to use concepts and properties from the human ontology, referring to them by their URIs (Uniform Resource Identifier), also to be explained in section 2.2. Therefore RDF makes it easier to connect information spread across the Internet in distributed documents, and for knowledge to be structured and reused.

1.2 RDF

1.2.1 RDF Graph

RDF graph is a collection of RDF triples [26]. The graph shows information in a way that humans can immediately understand. An RDF graph visualizes one (Figure 1.2.1) or more (Figure 1.2.2) RDF triples. These graphs are directed and labelled. A triple is a statement written in RDF in order to describe an entity (real or abstract). It is in the format of subject-predicate-object. Every triple expresses a fact. The classic Entity-Relationship diagrams or Class diagrams are very similar as they are all designed to provide visual models for concepts [53].

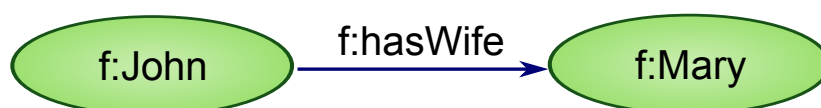


Figure 1.2.1: RDF graph representing one triple

The above example is an RDF graph representing one statement, stating the fact about a resource. Below we can see an example of an RDF graph representing multiple statements, therefore building an RDF dataset. The statements and syntax will be analyzed further below.

It is important to note that the arrow must always point from the subject to the object [34]. Using the RDF graph is a simple and easy way to understand and view the whole RDF document or its subset. Furthermore it makes it easier to see the relationships between

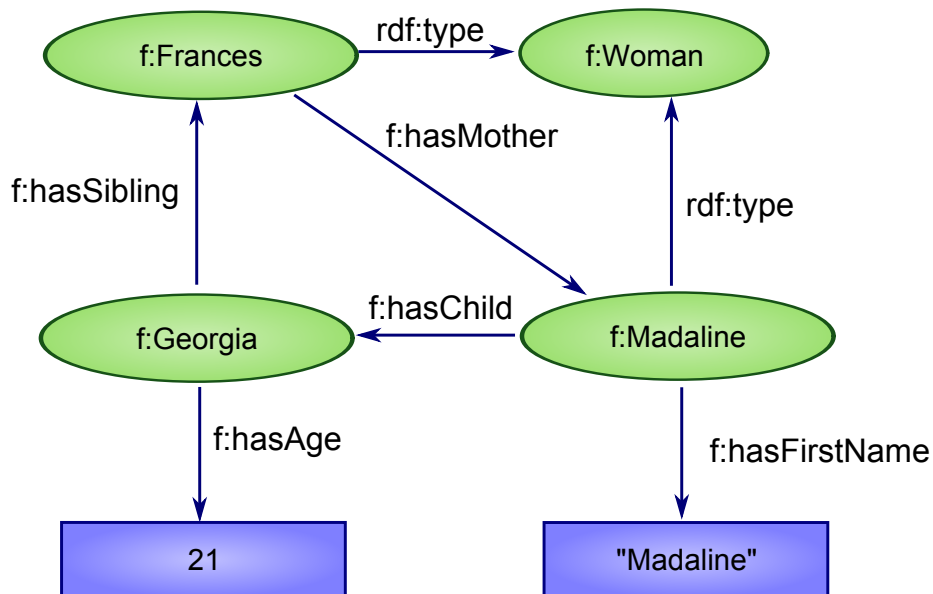


Figure 1.2.2: RDF graph representing multiple triples

resources and properties. Every arrow in the RDF graph represents a predicate of a triple. Every set of the arrow with the two ends on each side represent a statement. In an RDF document, there should be as many triples as there are arrows in the RDF graph.

1.2.2 URI

The Uniform Resource Identifier is a form of identification on the Web. Its purpose is to globally identify resources [39] [13]. URLs (Uniform Resource Locators) are a special type of URI. In general there is no need for the URI to pinpoint to an actual location to the Web. One shouldn't assume that it does, because URIs are used in Semantic Web to identify, not locate (URLs on the other hand have this requirement. There is no special authority that creates URIs; anyone can create one using any name they desire, using a few guidelines [12]. The name can be a reference to an actual location on the Web, or it can be an imaginary one. A URI may take different forms; it is not required to look like a website address. Even URNs (Uniform Resource Name), a subtype of URIs that identifies books, can be used [39]. There is no guarantee that something will exist on the Web, as will be further discussed below. If it is an actual location, then this means that the full description and specification of the corresponding resource can be found at that location.

In RDF, URIs are used to identify resources [55]. They contain the main URI and at the end may optionally have a fragment identifier. This is separated with the "#" character

with the rest of the URI. The fragment identifier is the name of the resource. These URIs in RDF can have Unicode characters, making it possible for many languages to exist. They are encoded in US-ASCII.

The URI `<http://www.example.com/family#Frances>` consists of the main URI `<http://www.example.com/family>` and the fragment identifier `Frances` which is the resource. The fragment identifiers are always separated by the rest of the URI with the pound sign `"#"`. For example, the URI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>` contains the W3C specification for RDF located on the Web. URIs are treated differently by RDF and by Web browsers [29]. RDF uses URIs to identify resources, while Web browsers use URIs to retrieve resources. RDF many times uses URIs to identify resources that do not exist on the Web, like abstract concepts of life. When two URIs are different, RDF considers them to be two different resources (unless otherwise specified) since they are lexically different.

What RDF does not assume about URIs [29]:

- RDF does not assume a URI points to an actual location on the Web and therefore can be retrieved.
- RDF does not assume that two URIs are the same when their domain URIs are common.

For example, it is not considered that the following two URIs are describing the same resource, because of their syntactic differences.

```
<http://www.example.com/family#Soap>
```

```
<http://www.example.com/family#Sophia>
```

Even though they both consist of the URI `<http://www.example.com/family>`, it is not assumed that they are about the same resource unless explicitly specified.

In order to simplify the creation of RDF documents as well as sharing and reading them, it is possible to have abbreviated URIs in a document. There are three different ways to write a URI:

1. The first way is to write the full URI: `<http://www.example.com/ontology/family>`. In this case whenever we want to refer to a resource in the ontology, it is done by writing the complete URI each time, followed by the character `"#"` and the

resource identifier. A URI that identifies the resource Georgia would look like `<http://www.example.com/ontology/family#Georgia>`.

2. Another way is shown in the following example. We have the BASE URI written once at the beginning of the document, and when it is needed to refer to it, we just use `<family>`. This is a relative URI. An example looks as following:

```
BASE <http://www.example.com/ontology/>
<family>
```

3. The third way is to abbreviate URIs by declaring a namespace. With the `@prefix` keyword we assign an arbitrary namespace to the domain URI. Whenever the namespace appears in the document, the reasoner or query processor will know which URI link it belongs to. An example of this approach is the following:

```
@prefix f: <http://www.example.com/ontology/>
f:family
```

1.2.3 Syntax

The foundation of RDF is a graph model for representing named properties and property values [28]. This graph, which represents RDF data, follows the model of subject-predicate-object statements (triples). With these triples we can describe resources in an easy to understand and structured way. The subject is always the resource in question; the predicate is the attribute of the said resource, and object is the value of that attribute. In Figure 1.2.3 we can see a visualization of this.

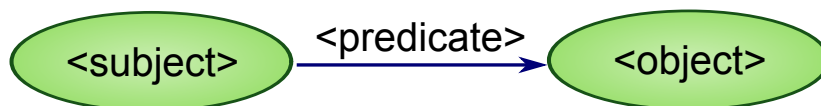


Figure 1.2.3: Subject - Predicate - Object statement

As mentioned above, the subject represents a resource described by the statement. This resource can be anything imaginable. For example it could be a webpage, an element of a webpage, or a whole website. But the extent of what a resource can be is not limited to the Web. It is possible to semantically describe a song, a person, a place, a book, even

an abstract meaning such as "like", as in the typical example "John likes Mary". When describing properties, the subject is the property being described (e.g., hasChild). In RDF documents these resources must always be expressed in the form of URIs [28].

The predicate can be an attribute, property, state or relationship of the subject. This must always be a URI as well. The predicate is also known as the property of the triple [34].

The object can be a URI, or of primitive type. Primitive type means that it is a non-class type, for example a string, boolean, integer, etc. The object can be anything from the range of primitive types, from a string to a boolean property. This essentially means that it can either be a resource or a value, called a literal. These literals have the option to be accompanied by a URI, but this will be discussed further below.

The following is an example of these two cases, the first being a statement with a URI as an object, and the second is a statement with a literal as an object.

```
<http://www.example.com/family#Father>  
  <http://www.example.com/family#hasChild>  
    <http://www.example.com/family#Child> .
```

```
<http://www.example.com/family#Father>  
  <http://www.example.com/family#firstName> "Peter".
```

RDF documents that do not have same URIs cannot be matched or related with each other. When common URIs exist this means that the RDF documents are describing the same thing. This is how RDF tools can "understand" what is being described and know what it is about. For example, if one RDF document states the specifications of a certain motorcycle, then someone else could extend this description by using the same URI and simply adding more information. This is an essential part of Semantic Web vision, the reusability of data, distributed knowledge and using the Web as a knowledge base.

It is tedious and non-human friendly to write the full URIs every time a resource is described in an RDF document. For this reason, URIs can be shortened by using the @prefix keyword. The domain URIs are written once in the beginning of the document, and the rest of the document containing RDF triples is used with the appropriate and corresponding prefixes, shown in the example below.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix o: <http://www.example.com/my-ontology/olympic-games#> .
```

```
o:Game      rdf:type      rdfs:Class .
```

```
o:Athlete   rdf:type      rdfs:Class ;
            rdfs:subClassOf  o:Human .
```

In the above example we can see that the URI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>` was declared in the beginning of the document with the corresponding `rdf:` namespace. Whenever we see in the document a construct beginning with `rdf:`, this means that it is a resource belonging to the declared URI. Instead of writing the whole URI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` each time the resource is needed, we shorten it by simply typing `rdf:type`. Internally, it is always interpreted as `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`, and this is simply a help for human-readability.

An RDF document may need URIs for the RDF specification, RDFS, OWL, XSD, and the ontology itself. All these technologies will be explained further in this thesis.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

1.2.3.1 Blank Nodes

A blank node is a concept that represents a resource without a name. There is an infinite set of blank nodes, with no limitations as to how many can be used [34]. This makes them very useful for large knowledge bases, when the name of the resource does not matter. Blank nodes are also used to describe restrictions in an ontology (to be explained further in the thesis). Internally, a name is created to identify each blank node. But this generated name is not permanent and therefore should not be used as an identifier. The RDF specification does not mention how the blank nodes should be internally implemented, and therefore it does not matter. However, it should always be possible to determine if two blank nodes

are the same or not. The set of blank nodes with the set of all the URIs and the set of all literals are all disjoint with each other (Figure 1.2.4). This means that a blank node is neither a URI nor a literal.

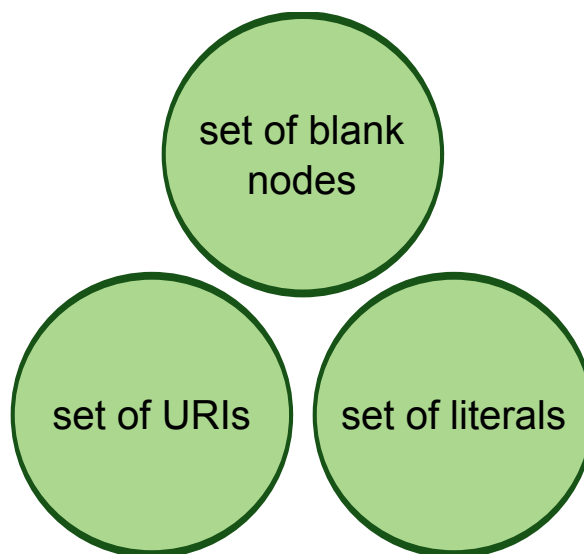


Figure 1.2.4: Pairwise disjointness

Blank nodes are usually written in ontologies in the form of `_:node342dcvx`. The label `node342dcvx` is the blank node's label. Blank nodes can also be written with square brackets, i.e. "[]". This is the abbreviated form. These brackets can be placed also at the end of the triple without changing the meaning. The following statements are all equivalent with each other.

```
[ ] family:hasChild family:Sophia .
[ family:hasChild family:Sophia ] .
_:node54xd3 family:hasChild family:Sophia .
```

The above statements all state that the individual Sophia has a parent, but we do not really care who this parent individual is. We can see a visualization of a blank node in Figure 1.2.5.

1.2.4 RDF Containers

RDF provides a means to encapsulate data. This allows a property value to refer to a single container as a resource. This has been created because sometimes it is needed to refer to a group of things as a property value. RDF containers are intended to describe a group of

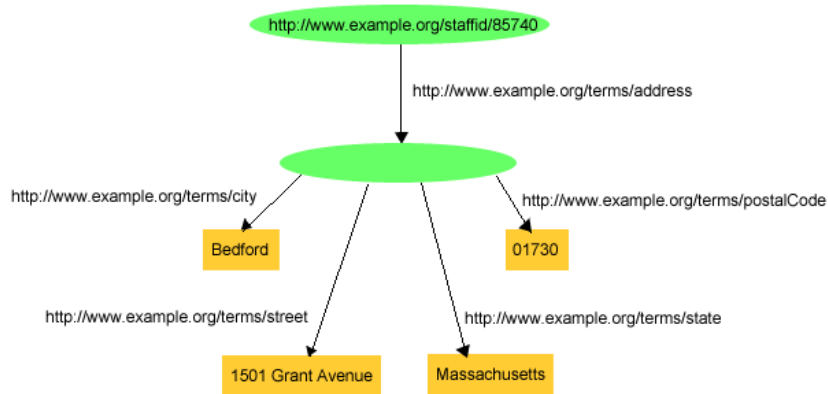


Figure 1.2.5: Example of a blank node [37]

objects. For example, in an ontology about a university, one may want to list the students enrolled in a course, to state that a book has been written by several authors, and many more use cases. The members of a container may be resources – blank nodes as well – or literals [23]. Containers are described in RDF with predefined types and properties. There are three predefined types of containers: bags, sequences and alternatives. In section 3.2.1 of the thesis we will discuss how to define these RDF containers, whereas this section is a theoretical introduction to them.

1.2.4.1 Unordered Container, Bag

A Bag is a resource which represents a group of resources or literals. A Bag can therefore be a value of a property. It is possible for a Bag to contain duplicate members. The member order does not matter.

1.2.4.2 Ordered Container, Sequence

A Sequence (Seq) is a resource that represents an ordered list of resources or literals. It is possible for a Sequence to include duplicate members. In Sequences the order of the members is important. For example, a Sequence might be used to describe a group that must be maintained in alphabetical order.

1.2.4.3 Container of Alternatives, Alternative

An Alternative (Alt) is a resource that represents a group of resources or literals that are alternatives to each other, usually for a single value of a property. The user can select only

one of the members. For example, an Alt might be used to describe alternative language translations for the title of a book, or to describe a list of alternative Internet sites at which a resource might be found. An application using a property whose value is an Alt container should be aware that it can choose any one of the members of the group as appropriate.

1.2.5 RDF Collections

RDF collections describe groups that can contain only the specified members, and none else. The difference is that a container only says that certain identified resources are members; it does not say that other members do not exist.

1.3 Notations

1.3.1 General

Any RDF data can be transformed and written in textual format, apart from being displayed as a graph. The textual format can be written in different interchangeable formats, depending on the preference. The two different views - graphical and textual - are equivalent. RDF is a data model that can be presented in various textual formats, including serialization. The syntax and therefore the encoding is different for each format. These formats will be further analyzed below. There are a few different ways to write an RDF document in textual format. These are the so-called notations, with small differences between each other. These are RDF/XML, Notation 3 (shortly N3), Turtle, and N-Triples (Figure 1.3.1). A popular notation is N3 [40]. It is much easier for humans to understand than RDF/XML. The latter is more often used for serialization and information processing applications, because XML parsers have already been developed and are well understood by developers.

1.3.1.1 Conversion between formats

Tools that convert one RDF data interchange format to another usually exist on the Web. Not every format can be converted, but in general it's easy to find a converter between RDF/XML and N3 or Turtle (e.g. Sswap [4]). In any case, the data encoded in the triples

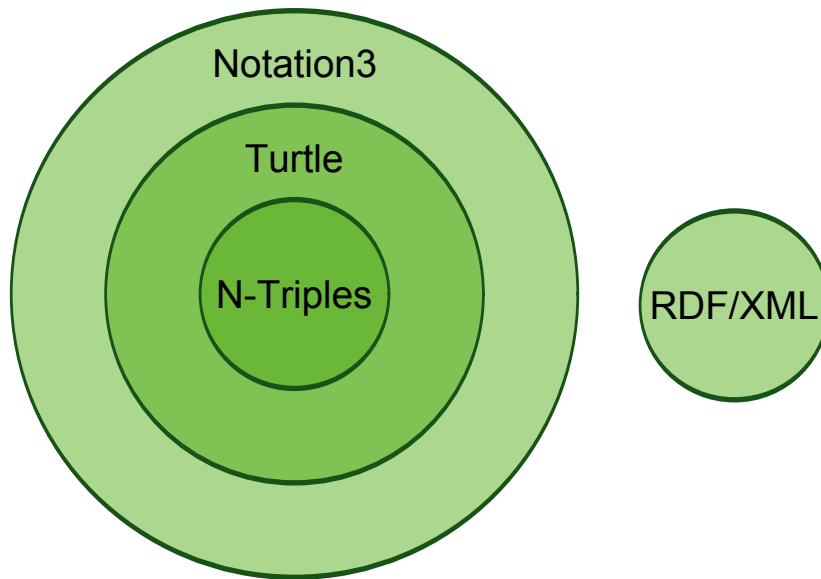


Figure 1.3.1: The relationship between RDF notations

is what matters, not the notation they are written in [5]. Each notation is simply a method to write the data in a textual format, according to each developer's needs.

1.3.2 Notation 3

Notation 3 or its abbreviated form N3 is the notation that is used often to write RDF documents. It is more human-readable than RDF/XML. It is designed for RDF graphs to be written in a compact and natural way [9].

Triples are written in the classical subject-predicate-object format, each element separated by the other by whitespace. Each triple must be terminated at the end with a dot ".". The keyword "a" can be used as an alternative to the `rdf:type` construct, and it is case-sensitive.

```
@prefix family: <http://www.example.com/family.owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

family:Sarah a family:Human , family:Female , owl:Thing ;
family:hasAge "19"^^xsd:int ;
family:hasSurname "Connor"^^xsd:string ;
family:firstName "Sarah"^^xsd:string ;
family:hasFriend family:Ange , family:Sarah .
```

The ";" symbol is used for triples that have common subjects, and the "," symbol is used for triples with common subject and predicate. This will be analyzed more in section 4.2.

1.3.3 Turtle

Turtle (TTL) stands for Terse Triple Language and is a subset of N3. Like its superset N3, Turtle is a natural way to write RDF triples, human-readable and easy to understand. Turtle is compatible with the other two formats that are used, N-Triples and Notation 3, as well as the SPARQL query language, mentioned further in this thesis. "Turtle is an extension of N-Triples carefully taking the most useful and appropriate things added from Notation 3 while keeping it in the RDF model" [8].

Comments are available for use by adding the character "#". The effect of this is valid till the end of the line [8].

The following do not exist in Turtle, but are included in N3 [7]:

1. { ... }
2. is of
3. paths like :a.:b.:c and :a^:b^:c
4. @keywords
5. => implies
6. = equivalence
7. @forAll
8. @forSome
9. <=

A few differences that exist in Turtle, but not in N3 [7]:

1. Decimal arbitrary length literals
2. Boolean literals

Below we will show an example of the same code as presented for N3, but written in Turtle:

```
@prefix family: <http://www.example.com/family.owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

family:Sarah a family:Human , family:Female , owl:Thing ;
family:hasAge "19"^^xsd:int ;
family:hasSurname "Connor"^^xsd:string ;
family:firstName "Sarah"^^xsd:string ;
family:hasFriend family:Ange , family:Sarah .
```

We can see that in this subgraph of the same example, N3 and Turtle look exactly the same. The notations in general have small differences between each other, but are not shown in this example.

1.3.4 N-Triples

N-Triples is a line-based format for encoding RDF graphs, written in plain text [32]. The character encoding is 7-bit US-ASCII, which limits the range of what is available to be used. N-Triples is a subset of Turtle. N-Triples should not be confused with Notation 3 which is a superset of Turtle [51]. N-Triples was intended to be a rigid subset of N3. Therefore, N3 tools are possible to be used to read and process the data written in N-Triples. For example, CWM (a popular reasoner, see section 2.4.1) can provide an output format of N-Triples when the tool is invoked using the command "cwm-ntriples" [32]. It is suggested to store N-Triples formatted data in files with a ".nt" suffix. This distinguishes the files from belonging to the N3 notation. N-Triples is used to state RDF test cases, and to define the connection between RDF/XML and the RDF abstract syntax. RDF/XML is recommended for applications to exchange RDF information, whereas N-Triples is more for human readability.

```
<http://www.example.com/ontology/family-inferred.owl>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.w3.org/2002/07/owl#Ontology> .
```

```

<http://www.example.com/family.owl#Sarah>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.example.com/family.owl#Human> .
<http://www.example.com/family.owl#Sarah>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.example.com/family.owl#Female> .
<http://www.example.com/family.owl#Sarah>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.w3.org/2002/07/owl#Thing> .
<http://www.example.com/family.owl#Sarah>
  <http://www.example.com/family.owl#hasAge>
    "19"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://www.example.com/family.owl#Sarah>
  <http://www.example.com/family.owl#hasSurname>
    "Connor"^^<http://www.w3.org/2001/XMLSchema#string> .
<http://www.example.com/family.owl#Sarah>
  <http://www.example.com/family.owl#firstName>
    "Sarah"^^<http://www.w3.org/2001/XMLSchema#string> .
<http://www.example.com/family.owl#Sarah>
  <http://www.example.com/family.owl#hasFriend>
    <http://www.example.com/family.owl#Ange> .
<http://www.example.com/family.owl#Sarah>
  <http://www.example.com/family.owl#hasFriend>
    <http://www.example.com/family.owl#Sarah> .

```

We can clearly see how such a notation differs from N3 or Turtle in terms of human-readability.

1.3.5 RDF/XML

The RDF/XML notation expresses the RDF model using XML. By using XML, RDF information can easily be exchanged between different types of computers, operating systems and application languages. Using XML is easy because there are more tools for XML that have already been developed even before Semantic Web erupted. This does not

mean that XML is better than other notations. For example there are parsers for N3, but there are not so many yet. The disadvantage of RDF/XML is that it is not human-readable like other notations are. An example of the RDF/XML notation is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:family="http://www.example.com/family.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <rdf:Description
    rdf:about="http://www.example.com/family.owl#Sarah">
    <rdf:type
      rdf:resource="http://www.example.com/family.owl#Human"/>
    <rdf:type
      rdf:resource="http://www.example.com/family.owl#Female"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <family:hasAge rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
    19</family:hasAge>
    <family:hasSurname rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Connor</family:hasSurname>
    <family:firstName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Sarah</family:firstName>
    <family:hasFriend rdf:resource="http://www.example.com/family.owl#Ange"/>
    <family:hasFriend rdf:resource="http://www.example.com/family.owl#Sarah"/>
  </rdf:Description>
```

1.4 RDF Storages

There are currently some tools for storing RDF data on the Web. These include:

- Jena and Joseki
- Oracle Spatial RDF
- Sesame

- Virtuoso Universal Server
- RDFLib

1.4.1 Jena

Jena was developed by Hewlett-Packard from the HP Labs Semantic Web Program and it is widely used. As stated in the Jena website, "Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL and includes a rule-based inference engine." [1]. Joseki is a HTTP engine allowing Jena storage to be web-accessible with SPARQL [21].

1.4.2 Oracle Spatial RDF

Oracle Spatial is based on the Spatial extension which was initially introduced for managing geographic data. Oracle Spatial does not support SPARQL, and therefore RDF triples are mapped to and stored in relational tables [21].

1.4.3 Sesame

Sesame is an open-source project developed by the Dutch software company Aduna [3]. "Sesame is a framework for storage, inferring and querying of RDF data". In many respects Sesame similar to Jena, but while Jena pays a lot of attention to reasoning, Sesame is optimized for storage and querying. Sesame is also often used by developers for RDF storage.

To use Sesame, it is possible to do so through a Console or through its Web application via Apache Tomcat. In this thesis, the Web application will be used for demonstration, called Sesame Workbench. Typing `http://localhost:8080/openrdf-workbench` in the browser after starting up Tomcat, we can see the following screen (Figure 1.4.1).

There are no user repositories at the moment. First we must create an empty repository. On the menu located on the left, we click on "New repository". The In Memory Store type must be chosen, and all we need to do is write an ID and a description for identification. We can see the procedure of this in Figure 1.4.2 and Figure 1.4.3.

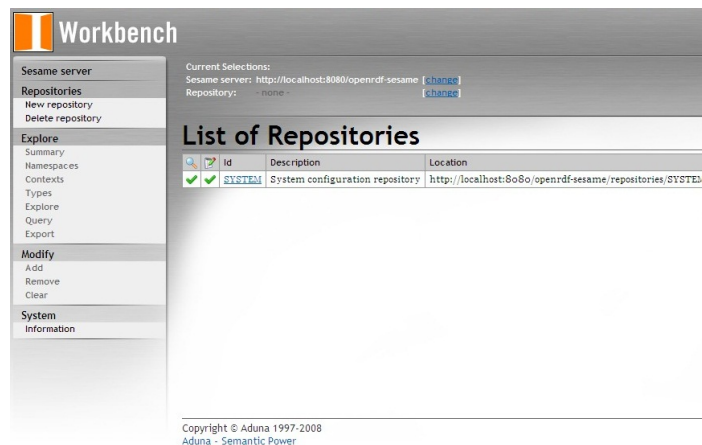


Figure 1.4.1: Starting screen of Sesame Workbench

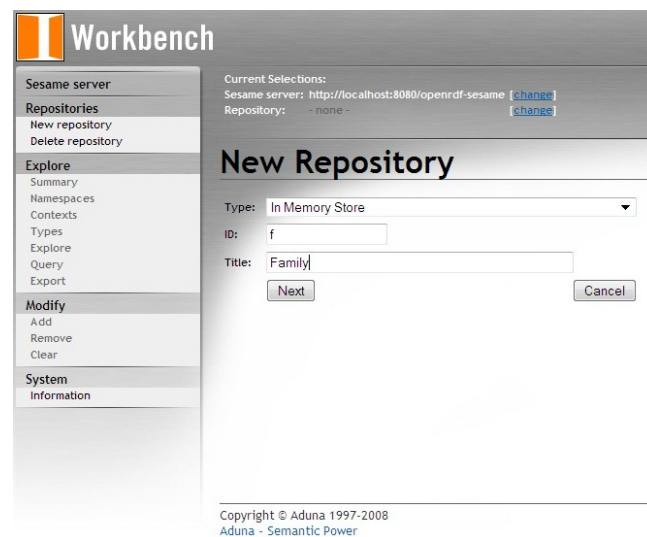


Figure 1.4.2: Creating a new repository in Sesame

1.4.3.1 Summary

After clicking on "Create", we see the following screen. We can see that the ID field is needed to identify the repository in Sesame. This screen can be found again if we click on "Explore->Summary" in the left-hand menu, shown in Figure 1.4.4.

The repository is still empty after creation, which can be easily seen by looking into the Types link on the menu. Therefore it needs to be filled with data from the ontology. This is done by clicking "Modify->Add" and simply uploading the owl file, in this case family2.owl. The rest will be automatically filled by Sesame, as shown in Figure 1.4.5.

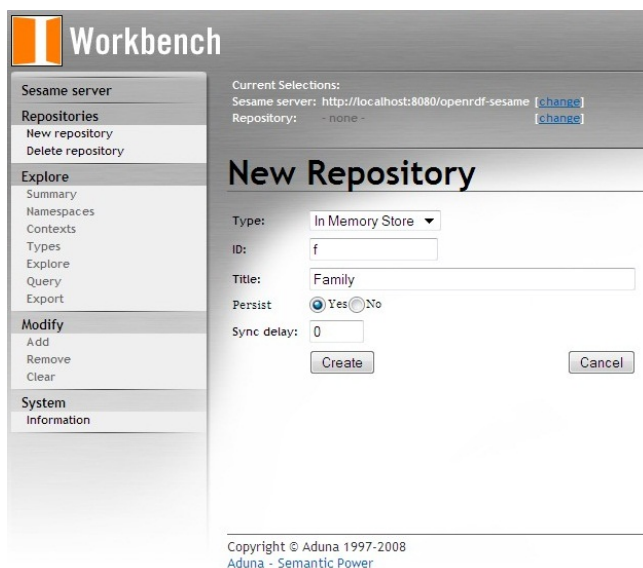


Figure 1.4.3: Persistent storage for our data

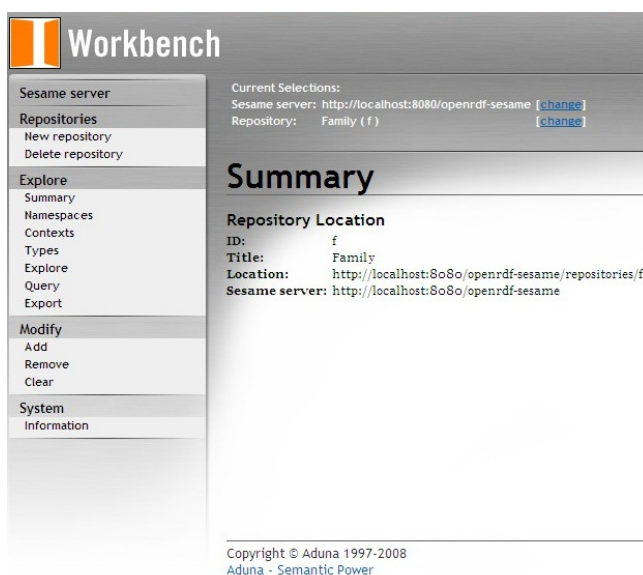


Figure 1.4.4: Summary of the new repository

1.4.3.2 Namespaces

Clicking on the link Namespaces we can see the namespaces that currently exist. We can see this in Figure 1.4.6. If desired, it is possible to add more.

1.4.3.3 Contexts

Contexts existing in the repository could be compared to tables in a relational table. In essence they are sections of data, divided logically. A repository should be able to support

Add RDF

Base URI:

Context:

Data format:

Location of the RDF data you wish to upload

RDF Data URL:

Select the file containing the RDF data you wish to upload

RDF Data File:

Enter the RDF data you wish to upload

RDF Content:

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 1.4.5: Uploading of RDF data to repository

Namespaces In Repository

Prefix:

Namespace:

Prefix	Namespace
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl2xml	http://www.w3.org/2006/12/owl2-xml#
family	http://www.example.com/family.owl#
xsd	http://www.w3.org/2001/XMLSchema#
owl	http://www.w3.org/2002/07/owl#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
dc	http://purl.org/dc/elements/1.1/

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 1.4.6: Existing namespaces in the repository

several contexts, which in turn are called named graphs. These named graphs can be used in SPARQL queries, instead of querying one large default graph. In this case we have only one context in the repository (Figure 1.4.7), and therefore will not analyze the matter further.

1.4.3.4 Types

By clicking on "Explore→Types", we can see a list of all the available concepts in the repository, as shown in the figure below (Figure 1.4.8).

By clicking on a type, we can explore everything that is related to it. For example, if we click on owl:ObjectProperty, we get the screen shown in Figure 1.4.9.

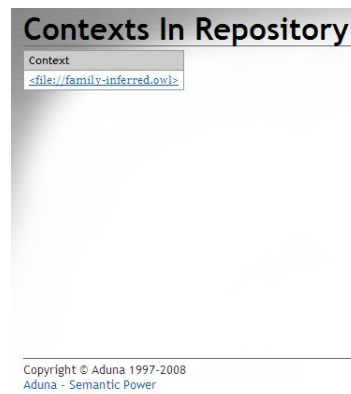


Figure 1.4.7: Contexts in the repository



Figure 1.4.8: Existing types in the repository

1.4.3.5 Explore

The search keyword to explore a certain construct, class or property is case-sensitive, and it matches only the complete string. It is not possible to insert a substring in hopes of finding the parent string. For example, to search for the blank node label `_:node15t2dbpp1x2`, we have to insert exactly that string. Inserting only `_:node` does not yield any results. In Figure 1.4.10, we can see three datatype properties. These are the currently created datatype properties in the family ontology.

Explore (owl:ObjectProperty)			
Subject	Predicate	Object	Context
family:hasChild	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasFather	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasFriend	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasHusband	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasMother	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasParent	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasSibling	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasSister	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasSpouse	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasWife	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
owl:topObjectProperty	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>
family:hasBrother	rdf:type	owl:ObjectProperty	<file://family-inferred.owl>

Resource:

Limit results:

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 1.4.9: All the object properties

Explore (owl:DatatypeProperty)			
Subject	Predicate	Object	Context
family:firstName	rdf:type	owl:DatatypeProperty	<file://family2.owl>
family:hasAge	rdf:type	owl:DatatypeProperty	<file://family2.owl>
family:hasSurname	rdf:type	owl:DatatypeProperty	<file://family2.owl>

Resource:

Limit results:

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 1.4.10: All the datatype properties

Below, in Figure 1.4.11, we can see the difference between an imported ontology with no inferred axioms, and an imported ontology including the inferred axioms. What has been added is the `owl:topDataProperty` construct. For the rest of the thesis the context of the inferred ontology will be used unless otherwise specified.

In Figure 1.4.12 below, we see a list of all the symmetric properties in the ontology.

1.4.3.6 Query

This is where all the SPARQL queries are performed and the results returned (Figure 1.4.13). The needed namespaces are already written in the beginning of the query window, but if we need to use more we can add them to the list. It is possible also to query using SeRQL,

Explore (owl:DatatypeProperty)

Subject	Predicate	Object	Context
family:hasAge	rdf:type	owl:DatatypeProperty	<file://family-inferred.owl>
family:hasSurname	rdf:type	owl:DatatypeProperty	<file://family-inferred.owl>
owl:topDataProperty	rdf:type	owl:DatatypeProperty	<file://family-inferred.owl>
family:firstName	rdf:type	owl:DatatypeProperty	<file://family-inferred.owl>

Resource:

Limit results:

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 1.4.11: Datatype properties of an ontology with inferred axioms

Explore (owl:SymmetricProperty)

Subject	Predicate	Object	Context
family:hasFriend	rdf:type	owl:SymmetricProperty	<file://family-inferred.owl>
family:hasSibling	rdf:type	owl:SymmetricProperty	<file://family-inferred.owl>
family:hasSpouse	rdf:type	owl:SymmetricProperty	<file://family-inferred.owl>

Resource:

Limit results:

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 1.4.12: All the symmetric properties

but this is not the topic of this thesis.

Queries with their results and their corresponding screenshots will be shown in section 4.2.2.

1.4.3.7 Export

By clicking on Export we can save the data as a different notation, like Turtle, N3 (in this case they are the same), N-Triples, etc (Figure 1.4.14). With this method we can compare the different notations and use whichever suits us according to the circumstance (Figure 1.4.14).

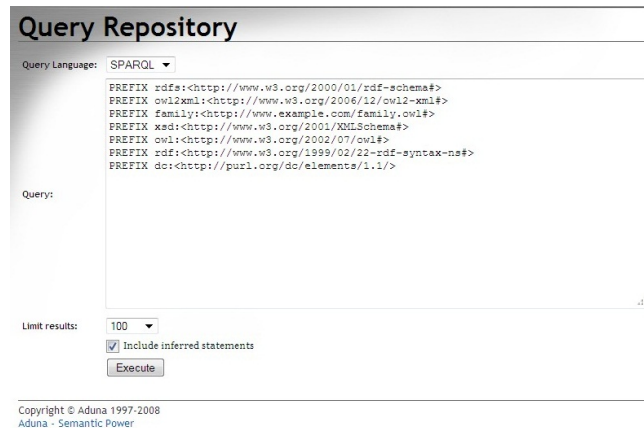


Figure 1.4.13: The empty query window in Sesame Workbench

Export Repository

Download format: Turtle Download

Limit results: 100 The results shown maybe truncated.

Subject	Predicate	Object	Context
family:Father	rdfs:subClassOf	family:Male	<file://family-inferred.owl>
family:Father	owl:disjointWith	family:Female	<file://family-inferred.owl>
family:Father	owl:disjointWith	family:Mother	<file://family-inferred.owl>
family:Father	owl:disjointWith	family:Sister	<file://family-inferred.owl>
family:Father	owl:disjointWith	family:Wife	<file://family-inferred.owl>
family:Female	rdfs:type	owl:Class	<file://family-inferred.owl>
family:Female	rdfs:subClassOf	family:Human	<file://family-inferred.owl>
family:Female	owl:disjointWith	family:Husband	<file://family-inferred.owl>
family:Female	owl:disjointWith	family:Male	<file://family-inferred.owl>
family:Human	rdfs:type	owl:Class	<file://family-inferred.owl>
family:Human	rdfs:subClassOf	owl:Thing	<file://family-inferred.owl>
family:Husband	rdfs:type	owl:Class	<file://family-inferred.owl>
family:Husband	rdfs:subClassOf	family:Male	<file://family-inferred.owl>
family:Husband	owl:disjointWith	family:Mother	<file://family-inferred.owl>
family:Husband	owl:disjointWith	family:Sister	<file://family-inferred.owl>
family:Husband	owl:disjointWith	family:Wife	<file://family-inferred.owl>
family:Male	rdfs:type	owl:Class	<file://family-inferred.owl>
family:Male	rdfs:subClassOf	family:Human	<file://family-inferred.owl>
family:Male	owl:disjointWith	family:Mother	<file://family-inferred.owl>
family:Male	owl:disjointWith	family:Sister	<file://family-inferred.owl>
family:Male	owl:disjointWith	family:Wife	<file://family-inferred.owl>
family:Mother	rdfs:type	owl:Class	<file://family-inferred.owl>
family:Mother	rdfs:subClassOf	family:Female	<file://family-inferred.owl>
family:Offspring	rdfs:type	owl:Class	<file://family-inferred.owl>
family:Offspring	rdfs:subClassOf	family:Human	<file://family-inferred.owl>
family:Parent	rdfs:type	owl:Class	<file://family-inferred.owl>
family:Parent	rdfs:subClassOf	family:Human	<file://family-inferred.owl>
family:Parent	rdfs:subClassOf	_:node1st8dcccaxz	<file://family-inferred.owl>
_:node1st8dcccaxz	rdfs:type	owl:Restriction	<file://family-inferred.owl>
_:node1st8dcccaxz	owl:isProperty	family:hasChild	<file://family-inferred.owl>
_:node1st8dcccaxz	owl:someValuesFrom	family:Child	<file://family-inferred.owl>

Figure 1.4.14: Choosing the desirable format to export the RDF data

1.5 Epilogue

In this chapter we started by giving an introduction for the Semantic Web, then proceeded to explain the technology of RDF. RDF is a method of representing resources in the form of a graph. URIs are used for global identification of resources. We can also create RDF

containers, such as a Bag, Sequence or Alternative. Notations are ways to write RDF in a textual format, equivalent to the graph. N3, Turtle, N-Triples and RDF/XML are all data interchange formats. RDF storages exist to store RDF data, and possibly query the data. A few popular RDF storages include Jena, Oracle Spatial RDF, and Sesame. The latter is used in this thesis to demonstrate examples of queries in Chapter 4. Data represented in RDF needs metadata, and this is what will be discussed in the following Chapter 2.

Chapter 2

Ontologies

2.1 General

An ontology is a core element of the Semantic Web. As stated by W3C, "An ontology defines the terms used to describe and represent an area of knowledge" [47]. Another definition of ontology is that "An ontology is an explicit specification of a conceptualization" [17]. An ontology describes the meaning of concepts and their relationships, and therefore is similar to a vocabulary of human language. In addition to defining and describing terms, ontologies also define the structure of knowledge within a certain domain (e.g. fishing domain, domain of medicine, transportation domain, etc.).

With ontologies, the vision of Semantic Web can be further realized through "intelligent" applications. These applications can come closer to the level of human conception by using domain-specific ontologies and semantically annotated data. Ontologies are meant to be used by applications and databases, but they are also used by people that need to define or share information about a specific area of knowledge (domain). They are also being used more and more by businesses and scientific communities in order to process domain knowledge, as well as share with others and reuse it.

Within the Semantic Web stack (Figure 1.1.1), an ontology is based on RDF. One thing that confuses Semantic Web newcomers is that an ontology is also an RDF document.

The reasons for developing an ontology include:

- To share a common understanding of the structure of information
- To offer re-usability of domain knowledge

- To explicitly state assumptions in a domain (e.g. a woman is a female person)
- To analyze the domain knowledge

In many cases, ontologies develop out of the need to define data in a domain for use in applications. In other words, ontologies are developed more as a means to an end, as opposed to them being the intended end product. Their use in describing information on the Web in an unambiguous way is one such example. Software agents, applications that are domain-independent, and methods used for problem-solving can use ontologies or knowledge bases as data. A knowledge base is in essence an ontology with its set of instances [25].

There are different degrees of ontologies and each is dependent on the structures they represent. For example, there are ontologies to describe metadata schemes (for example Dublin Core, to be explained further below), simple taxonomies and logical theories. As such, ontologies need to have a high degree of structure. In order to develop an ontology, logic-based language is used to provide "detailed, accurate, consistent, sound and meaningful distinctions can be made among the classes, properties and relations." [47] Support for decision making, understanding of speech and natural language, knowledge management, electronic commerce and intelligent databases are all intelligent applications that can have ontology tools with automated reasoning. This way ontologies are not only descriptions of knowledge bases, but also good providers for intelligent applications that aid humans.

2.2 RDFS – RDF Schema

Ontologies are important for applications that aim for cross-communication between various sources. Up until a few years ago, this was done with XML DTDs and XML Schemas. Even though they are sufficient for exchanging data with mutually agreed definitions, a reliable outcome cannot be guaranteed due to the lack of semantic assigning to those definitions. This happens even in the natural language, with words having similar meanings, or the meaning being dependent on the context. RDF Schema tries to solve this problem by "allowing simple semantics to be associated with identifiers" [47]. Class hierarchies are defined, and the same with property hierarchies. RDF Schema on its own might not be sufficient to create a complex ontology. An example of such an ontology is a musical

ontology, where a state "string quartet has exactly four musicians" is not definable by RDF Schema.

RDF Schemas must not be confused with XML Schemas. Even though both are Schemas, their usage and role is very different. RDF Schemas are descriptive, whereas XML Schemas are prescriptive. This essentially means that while XML Schemas provide strict guidelines and rules of how XML should be written, RDF Schemas simply enhance already defined descriptions. RDF Schema extends RDF semantically [31]. This means that RDF Schema provides more predicates in order to describe classes and properties in a more complete way. Classes in RDF Schema are treated like Object-Oriented programming classes. This essentially means that resources (a class or property) can be instances of classes, subclasses of other classes, and more.

A property is also an entity, an abstract concept of the real world. This means that it is not only possible to describe concrete objects, but also properties. Properties need to be described in order to provide complete information in the ontologies. Properties can have domains as well. The domain states the class of the property's subject. The range expresses the class of the property's value. As an example to illustrate this, we can use a property `hasChild`. If this property is defined as having domain `Parent` and range `Child`, then a statement "Peter `hasChild` Sophia" is valid only if individual Peter is of type `Parent` and at the same time individual Sophia is of type `Child`.

```
@prefix family: <http://www.example.com/family#> .
family:Frances family:hasChild family:Sophia .
family:Frances family:firstName "Frances" .
```

The property range of `hasChild` is of type `Human`, and the range of `firstName` is of type `String`. In both cases the domain is of type `Human`. All of the resources belong to the same ontology since they all have the same namespace name `family:`.

Every predicate that exists in RDF Schema is usually preceded by the namespace name `rdfs:`. We can create a different namespace, as long as the prefix namespace is connected to the proper URI, `<http://www.w3.org/2000/01/rdf-schema#>`. For example, if we want to define a resource as a literal, we will write it as following:

```
family:Parent rdfs:subClassOf family:Human .
```


2.2.1 Classes

In RDF Schema both classes and instances are resources. A class is a set of resources in RDF. Much like in Object-Oriented programming, a member of a class is called an instance. It is possible for a class to be an instance of itself. To define a resource as an instance of a predefined class, the property `rdf:type` is used. An example is shown below:

```
family: Sophia    rdf:type    family: Female .
family: Sophia    rdf:type    family: Wife .
```

rdfs:Resource This is the class of everything [31]. Everything described with RDF Schema and therefore, RDF, is a resource. This class is an instance of `rdfs:Class`.

rdfs:Class The group of RDF classes is a class in itself which is `rdfs:Class`. `rdfs:Class` is an instance of itself.

rdfs:Literal This is the class of all literal values, for example string, date, integer, boolean, and many more. `rdfs:Literal` is an instance of `rdfs:Class` and a subclass of `rdfs:Resource`. All literals have a lexical form that is a Unicode string [34]. A literal can be plain, or typed. Typed literals are those that belong to a specific predefined type. These can belong to XML Schema [36]. They are usually written with the namespace "xsd: ", e.g. `xsd:string`. Typed literals are accompanied by datatype URI references. Plain literals are strings that can optionally have a language tag, denoting the natural language of the text. The main difference is that with typed literals the applications know how to interpret and handle the literals, whereas with plain literals it is not possible.

Typed literal:

```
family: Sophia    family: firstName    "Sophia"^^xsd:string .
```

Plain literal:

```
family: Sophia    family: firstName    "Sophia" .
```

rdfs:Datatype This is the class of datatypes, as the name suggests. Every instance of `rdfs:Datatype` is a subclass of `rdfs:Literal`. The construct `rdfs:Datatype` is an instance as well as a subclass of `rdfs:Class`.

rdf:XMLLiteral The class of the XML literal values is `rdf:XMLLiteral`. It is an instance of `rdfs:Datatype`, ergo a subclass of `rdfs:Literal`.

rdf:Property The class `rdf:Property` is the RDF properties class. It is also an instance of `rdfs:Class`.

2.2.2 Properties

rdfs:domain This is an instance of the class `rdf:Property`. It is used to express that the domain of a certain property is a certain class. For example, the triple

```
family:hasChild    rdfs:domain    family:Parent .
```

states that whenever we create a statement that uses `hasChild` as the predicate, the subject of the statement must be an instance of the class `Parent`. We can state that the domain of the `rdfs:domain` property is the class `rdf:Property`. Which means that the resource being annotated with this property will always be an instance of the `rdf:Property` class.

rdfs:range The property `rdfs:range` is also an instance of the class `rdf:Property`. It is used to express that the range of a certain property is a certain class. For example, the triple

```
family:hasChild    rdfs:range    family:Child .
```

states that whenever we create a statement that uses `hasChild` as the predicate, the object of the statement must be an instance of the class `Child`. Therefore we can say that the range of the `rdfs:range` property is the class `rdfs:Class`.

rdf:type The property `rdf:type` is used to declare that a resource is an instance of a class. This property is also an instance of `rdf:Property`. For example, the statement

```
family:Sophia rdf:type family:Human .
```

means that `Human` is an instance of `rdfs:Class`, and that the resource `Sophia` is an instance of `Human`.

rdfs:subClassOf This property, an instance of `rdf:Property`, states that all instances of one class are also instances of another class. The domain and range of this property are both instances of `rdfs:Class`. The triple

```
family:Female rdfs:subClassOf family:Human .
```

means that `family:Female` and `family:Human` are instances of `rdfs:Class`, and that `f:Female` is a subclass of `family:Human`. The property `rdfs:subClassOf` is transitive. The meaning of this will be explained further on in this thesis.

rdfs:subPropertyOf This property declares that all resources that are related by a property are also related by another property. For example, the statement

```
family:hasMother rdfs:subPropertyOf family:hasParent .
```

tells us that `family:hasMother` and `family:hasParent` are instances of the `rdf:Property` class, and that `family:hasMother` is a sub-property of `family:hasParent`. The property `rdfs:subPropertyOf` is transitive.

rdfs:label: With the `rdfs:label` property a label is attached to the resource that is easily read by humans.

```
family:PeterD rdfs:label "Peter David" .
```

The domain of the `rdfs:label` property is an a resource, and the range is a literal.

rdfs:comment The `rdfs:comment` property attaches to the resource a comment about it, possibly describing it for quick recognition by a human.

```
family:PeterD rdfs:comment "Peter David, tall guy with glasses" .
```

The domain of the `rdfs:comment` property is a resource, and the range is a literal.

2.2.3 Container Classes and Properties

RDF containers are resources with the purpose to represent collections. The members in a container are either resources or literals. A blank node is also considered a member of the container. RDF has no real understanding of the differences of each container. These are

just intended to indicate the accepted understanding of the purpose of each container, and it is up to the developer to use them as intended.

rdfs:Container The `rdfs:Container` class is a superclass of the RDF Container classes.

rdf:Bag The `rdf:Bag` class is the class of RDF Bag containers. This class is a subclass of the `rdfs:Container` class. Syntactically it is not different than `rdf:Seq` and `rdf:Alt`. It is used to show the reader that the container is unordered. To state that a blank node or a resource with a URI is a container of type `rdf:Bag`, we use the `rdf:type` property. The resource then describes the group as one entity. An example is shown in Figure 2.2.1.

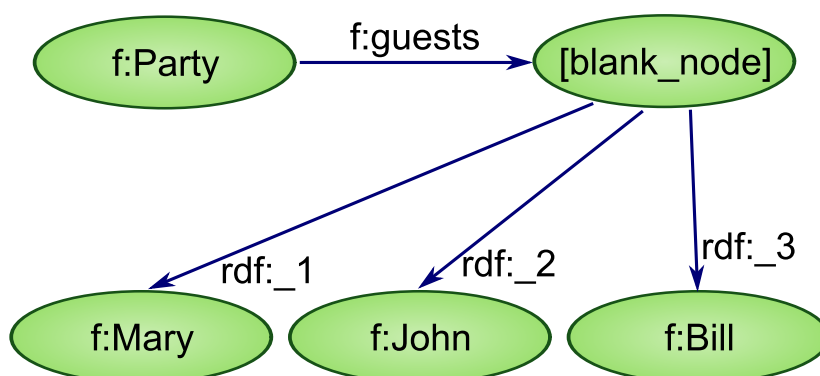


Figure 2.2.1: An RDF container of type `rdf:Bag`

rdf:Seq This is the class of RDF Sequence containers. It is also a subclass of `rdfs:Container`. It is syntactically the same as `rdf:Bag` and `rdf:Alt`. The difference is that it is used to show the human user that the numerical order in which the membership properties of the container is important. These membership properties are explained in the `rdfs:ContainerMembershipProperty` subsection.

rdf:Alt The class `rdf:Alt` is the class of RDF Alternative containers. This is also a subclass of `rdfs:Container`. As mentioned above, this class is syntactically also no different than `rdf:Bag` and `rdf:Seq`. The `rdf:Alt` class signals to the reader that processing will suggest selecting one of the members of the container. The first member of the container is the default choice. An example of an Alternative is shown in Figure 2.2.2.

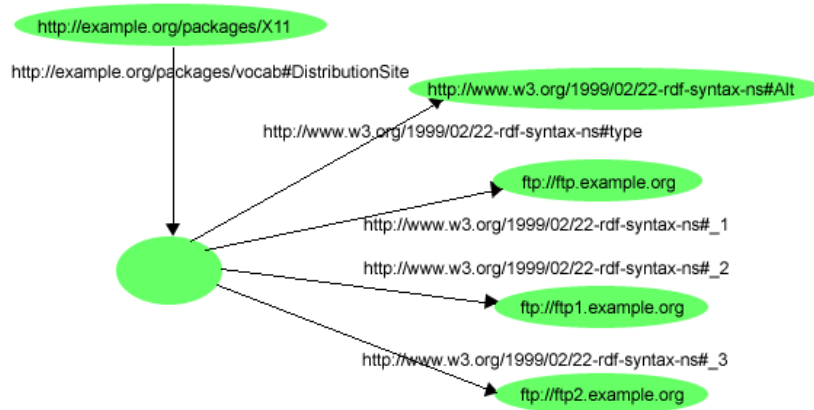


Figure 2.2.2: An example of an Alternative [37]

rdfs:ContainerMembershipProperty To describe the members of a container, we define a property of type `rdfs:ContainerMembershipProperty` for each corresponding member. The `rdfs:ContainerMembershipProperty` class has instances with names in the following manner:

`rdf:_1`, `rdf:_2`, `rdf:_3` , etc.

They are used to declare that a resource is a member of a container (Figure 2.2.1). Each instance of the `rdfs:ContainerMembershipProperty` class is a sub-property of the `rdfs:member` property. The container resource is the domain of the property, and the range is the member of the container. It is possible to have other properties that describe the container resource.

rdfs:member Every container membership property is a sub-property of `rdfs:member`. Therefore, `rdfs:member` is the super-property of all the container membership properties. The `rdfs:member` property is an instance of `rdf:Property`. The domain of `rdfs:member` is a resource, and the range is a resource as well. It states that a resource is a member of

2.2.4 RDF Collections

A collection represents a list of items. RDF Schema does not demand only one first element of a list. Nor does it require for a list to have a first element [31]. A visual representation will be shown further in section 2.3.2.3.

rdf:List This is an instance of `rdfs:Class` that can be used to build descriptions of lists and other list-like structures.

rdf:first This collection is an instance of `rdf:Property` that can be used to build descriptions of lists and other list-like structures. For example, `Ardf:firstB` declares that there is a first-element relationship between A and B. The domain of `rdf:first` is a list (`rdf:list`), and the range is a resource (`rdfs:Resource`). The resource B is the first element of the list that is represented by the resource A.

rdf:rest The `rdf:rest` collection is an instance of `rdf:Property`. It is used to create descriptions of lists. For example, `Ardf:restB` states that there is a rest-of-list relationship between A and B. Both the domain and range are `rdf:List`. The domain, in this case named A, is the the resource that represents the list. The range B is the resource that is part of the list, but not the first element.

rdf:nil This is an instance of `rdf:List` and it is used to represent an empty list-like structure. For example, `Ardf:restrdf:nil` states that A is the instance of an `rdf:List` that contains only one element, while the rest is an empty list. That one element can be declared with `rdf:first`.

An example ontology based on the Olympic Games was created for demonstrating RDF Schema. The subset code is in pure RDF Schema in order to show its syntactic rules and constraints, as well as its simplicity. The complete ontology can be found in Appendix B. We can see that it is not possible to distinguish between the different types of properties, to declare a property as an inverse of another, etc.

```
@prefix :           <http://www.example.com/olympics.owl#> .
@prefix rdfs:       <http://www.w3.org/2000/01/rdf-schema#> .
@prefix protege:    <http://protege.stanford.edu/plugins/owl/protege#> .
@prefix rdf:        <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

:Game rdf:type rdfs:Class .
:Human rdf:type rdfs:Class .
:Sport rdf:type rdfs:Class .
:Stadium rdf:type rdfs:Class .
:Athlete rdf:type rdfs:Class ; rdfs:subClassOf :Human .
```

```
:hasGender rdf:type rdf:Property ; rdfs:domain :Human
    ; rdfs:range rdfs:Literal .
:hasLevel rdf:type rdf:Property; rdfs:domain :Game
    ; rdfs:range rdfs:Literal .
:hasFirstName rdf:type rdf:Property; rdfs:domain :Human
    ; rdfs:range rdfs:Literal .
:playedIn rdf:type rdf:Property; rdfs:domain :Game
    ; rdfs:range :Stadium .
:hasStadiumName rdf:type rdf:Property; rdfs:domain :Stadium
    ; rdfs:range rdfs:Literal .
:inCity rdf:type rdf:Property; rdfs:domain :Stadium
    ; rdfs:range rdfs:Literal .
```

2.3 OWL

Ontologies for the Semantic Web can also be written in OWL, which stands for Web Ontology Language. Currently there are two versions of OWL, namely OWL version 1.0 and OWL version 2.0. OWL 2.0 became a W3C recommendation just recently and there are not many implementations of it. Therefore, whenever we refer to OWL, we mean OWL version 1.0, unless explicitly specified otherwise. OWL version 1.0 is a W3C (World Wide Web Consortium) standard, and became a formal W3C recommendation on February 10, 2004 [52]. OWL is derived from the DAML+OIL Web Ontology Language [6] [20]. DAML+OIL is a language that succeeds the languages DAML and OIL, and it combines features of both [49]. DAML stands for DARPA Agent Markup Language, and it was created by the Defence Advanced Research Projects Agency. OIL stands for Ontology Inference Layer or Ontology Interchange Language. The DAML project was terminated in 2006, with OWL already having taken its place. OWL is the latest development among the ontology languages, aimed to support and belong to the Semantic Web vision. This means that with OWL the unambiguous information on the Web is processed by computers. OWL ontologies can be distributed, as OWL allows ontologies to refer to concepts defined in other ontologies. Because of this, OWL is created for the Semantic Web as knowledge bases are built and information is structured [19].

OWL is a stronger and more expressive language than RDF Schema, with greater machine interpretability. It builds upon RDF Schema and extends its features. With this, OWL succeeds to add more flexibility and syntactic freedom. This language was designed to be interpreted by machines. It can be written in XML, since XML has already cemented the serialization format. This way the information written in OWL (and therefore XML) can easily be exchanged between different platforms and applications, rendering it fully versatile and widely-used. OWL has the ability to process machine interpretable information stored on the Web [44]. Therefore, it is designed for use by applications that process information and not only present it to humans. OWL extends the vocabulary of RDF [30] and adds possibilities to describe concepts, to create complex hierarchies and the relationships between classes and properties, and many more features that will be analyzed further below. With OWL we can state for example that two classes are disjoint with each other, meaning that there cannot be an instance belonging to those two classes simultaneously. Furthermore, we can state that a property is the inverse of another property, as well as many more capabilities for classes and properties. An important difference between an OWL and an RDF document is that OWL formally describes the semantics of abstract concepts, whereas RDF documents describe concrete entities based on those predefined concepts.

2.3.1 Introduction to OWL Sub-languages

OWL has three sub-languages - OWL Lite, OWL DL and OWL Full. They are in order of increasing complexity and decreasing completeness (Figure 2.3.1). Each language will be further analyzed and discussed below.

All classes are subclasses of `owl:Thing`, the root class. All classes are subclassed by `owl:Nothing`, the empty class. No instances can be members of `owl:Nothing`. Modellers use `owl:Thing` and `owl:Nothing` to assert facts about all or no instances [23]. A property is a relation between two entities that specifies class characteristics. There are two types of properties in OWL, object properties and datatype properties. For the object property, the property range is a class (individual to individual relation). A datatype property is when the range is a data type (individual to datatype relation). The domain in both cases is an object with a URI. Datatype properties are formulated using `owl:DatatypeProperty` type, whereas object properties are formulated using `owl:ObjectProperty`. Languages in the OWL

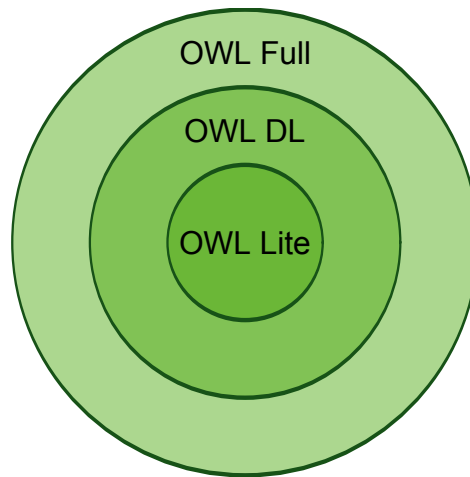


Figure 2.3.1: OWL sub-languages

family support various operations on classes such as union, intersection and complement. They also allow class enumeration, cardinality, and disjointness.

2.3.1.1 OWL Datatypes

Datatype properties may have as range RDF literals or simple types that are defined according to the XML Schema datatypes. Most of the built-in XML Schema datatypes are used in OWL [45]. These XML Schema datatypes, as well as `rdfs:Literal`, are part of the built-in OWL datatypes. All OWL reasoners are required to support the `xsd:integer` and `xsd:string` datatypes [45]. There might be a confusion among the datatypes and deciding which one is correct to use. Assigning a range value, we can see that `xsd:int` exists, but `xsd:integer` does as well. What is the difference? The datatype `xsd:int` represents 32-bit signed integers and is derived from the Long datatype. On the other hand, `xsd:integer` represents unbound signed integers and values may begin with an optional "+" or "-" sign. The latter is derived from the Decimal datatype.

2.3.2 OWL Lite

OWL Lite is designed to be the simplest of the three, thus having many constraints concerning classes, cardinality, etc. OWL Lite is easier to support by reasoning software. Reasoning an OWL Lite document is fast because of its simplicity. Every OWL Lite ontology is an OWL DL ontology.

Originally, OWL Lite was created to provide support for people who needed to define hierarchies of classes and simple constraints. For example, even though it supports cardinality constraints, the only values allowed are 0 or 1. The reason was to have a simpler way to provide tool support for documents written in OWL Lite. This in theory allows easy transformation for taxonomy systems.

The limitations on the syntax in OWL Lite are aimed to create a simple but useful subset of OWL features. OWL Lite contains at least basic capabilities for the creation of an ontology, as subclass hierarchy structure, optional or required properties, minimal cardinality, etc. Being less complex than OWL DL or OWL Full creates the possibility for efficient OWL Lite reasoners.

2.3.2.1 Features

First of all, OWL Lite contains also RDF Schema features, which were explained previously in the thesis. These include: `Class`, `rdfs:subClassOf`, `rdfs:Property`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`, and `Individual`. OWL Lite extends these features and adds more functionality. Equality and inequality features, property characteristics and restrictions, as well as cardinality with its restrictions.

2.3.2.2 Classes

In OWL Lite (and OWL DL) an individual can never be an OWL class as well, and the inverse is true too. The same applies to properties and data values. The construct `owl:class` is a subclass of `rdfs:class`.

2.3.2.3 Equality - Inequality Features

owl:equivalentClass: Instances can belong to more than one class, when the classes are stated to be equivalent. This allows the possibility to set synonymous classes. For example, the class `Offspring` can be stated to be equivalent to the class `Child`. From this, we can safely assume that any instance of `Offspring` is also an instance of `Child`, and vice versa.

owl:equivalentProperty It is possible to state two properties to be equivalent. This is helpful in order to create synonymous properties.

owl:sameAs This construct is used to state that two individuals are the same. This is used when we want to create different names for the same individual.

```
family:Vivian rdf:type owl:Thing , family:Female ;
              owl:sameAs family:Vivi .
```

owl:differentFrom Everything in RDF must be explicitly defined, therefore we must also define distinction among instances. With the construct `differentFrom` we can state two individuals to be different from each other. This helps the reasoner understand that Peter and PeterD are two different people. It might not otherwise be assumed. This construct also helps the reasoner to realize contradictions, especially where cardinality restraints are involved.

owl:AllDifferent This is used to define a whole group of distinct entities, e.g. a group of five different people. For example, we can state that all the individuals in the family ontology are distinct with each other. Instead of using the construct `differentFrom` for each individual with all the rest, we can use the construct `AllDifferent` (Figure 2.3.2).

```
_:node15sb4m6gcx44 a owl:AllDifferent ;
owl:distinctMembers _:node15sb4m6gcx45 .
_:node15sb4m6gcx45 rdf:first family:Sarah ;
rdf:rest _:node15sb4m6gcx46 .
_:node15sb4m6gcx46 rdf:first family:Frances ;
rdf:rest _:node15sb4m6gcx51.
_:node15sb4m6gcx51 rdf:first family:PeterD ;
rdf:rest rdf:nil .
```

2.3.2.4 Property Characteristics

owl:inverseOf A property X can be stated to be inverse of property Y. This means that if individuals A and B are connected with X (creating statement A X B), then B will be connected with A with the Y property (creating statement B Y A). For example, the property `hasParent` is the inverse property of `hasChild` (Figure 2.3.3). This means that if

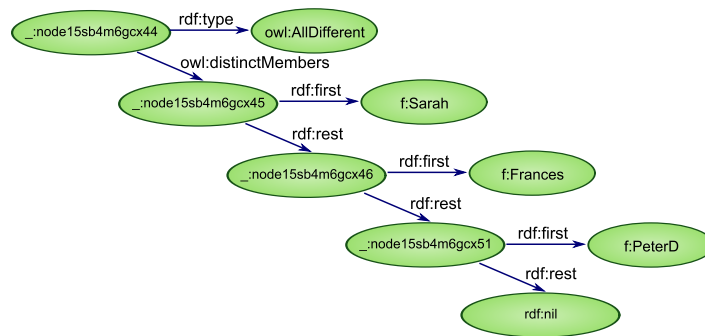


Figure 2.3.2: A visualization of the owl:AllDifferent construct

family:Sophia family:hasChild family:Ange, then the reasoner can assume that family:Ange family:hasParent family:Sophia .

```
family:hasChild a owl:ObjectProperty ;
rdfs:range family:Child ;
rdfs:domain family:Parent ;
owl:inverseOf family:hasParent .
```

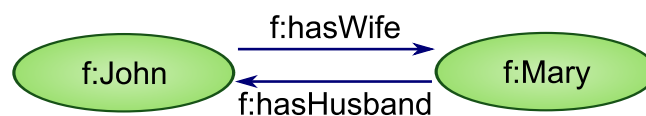


Figure 2.3.3: Illustration of the inverseOf construct

owl:TransitiveProperty A property can be stated to be transitive. This means that if a property X is transitive, then any instances of that property with common entities are transitive with each other. For example let us take the case where the property hasFriend is transitive. Then the triple family:George family:hasFriend family:Sarah is an instance of the transitive property, and family:Sarah family:hasFriend family:Mark is another instance of the transitive property with common entities. The reasoner can now deduce that family:George family:hasFriend family:Mark. One important thing to note is that OWL Lite (and OWL DL) do not allow transitive properties to have maximum cardinality of 1 [44]. Furthermore, OWL DL requires that no cardinality constraints are declared on the property, its super-properties, or even on its inverse [46].

```
family:hasFriend a owl:ObjectProperty , owl:TransitiveProperty ;
```

```
rdfs:domain family:Human ;
rdfs:range family:Human .
```

owl:SymmetricProperty A property may be stated to be symmetric. If a property X is symmetric, and the entities A and B form an instance of that property, then B and A will also be an instance of that property [26]. For example, the property hasSpouse is set to be symmetric, then we can safely state that if `f:Sophia` `f:hasSpouse` `f:Peter`, then `f:Peter` `f:hasSpouse` `f:Sophia`.

```
family:hasSpouse a owl:ObjectProperty , owl:SymmetricProperty ;
rdfs:range family:Parent ;
rdfs:domain family:Parent .
```

owl:FunctionalProperty If a property is functional, this means that it has a unique value. Every individual can have maximum one unique value. It is not required for it to have a value however. This is basically saying that the maximum cardinality will be 1, and the minimum will be 0. For example, hasFather could be a functional property, depending on the culture. This would mean that an individual may have only one father, but it is not mandatory. In the following example, hasFather is a functional object property.

```
family:hasFather a owl:ObjectProperty , owl:FunctionalProperty ;
rdfs:domain family:Child ;
rdfs:range family:Father ;
rdfs:subPropertyOf family:hasParent .
```

owl:InverseFunctionalProperty This construct is used to state that the inverse of a functional property is functional as well. We can therefore know that the inverse of functional property X has maximum cardinality of 1 too. In the following example, we state that hasHusband is a functional property as well as the inverse of the property hasWife.

```
family:hasHusband a owl:ObjectProperty , owl:FunctionalProperty
, owl:InverseFunctionalProperty ;
rdfs:range family:Husband ;
rdfs:domain family:Wife ;
rdfs:subPropertyOf family:hasSpouse ;
```

```
owl:inverseOf family:hasWife .
```

2.3.2.5 Property Restrictions

Apart from the property characteristics, there are also property restrictions which constrain the usage of properties even more depending on how they are used. These are used within the `owl:Restriction` construct. The `owl:onProperty` construct states that property being restricted. Most restrictions have a global range, meaning that they apply to all the instances of the corresponding property. The properties `owl:allValuesFrom` and `owl:someValuesFrom` have a local range as they depend also on the class.

owl:allValuesFrom This construct is used to state that all the values pertaining a specific property must be of a certain class. For example, if we want to say that a resource can be a `Parent` if it has human children, we can declare it in the following manner, written in Turtle:

```
family:Parent a owl:Class ;
rdfs:subClassOf family:Human , _:node15sb4m6gcx7 .

_:node15sb4m6gcx7 a owl:Restriction ;
owl:onProperty family:hasChild ;
owl:allValuesFrom family:Child .
```

The above example states that the class `Parent` is a subclass of the class `Human`, and of a blank node that serves as a "virtual class". It is an instance of `owl:Restriction`, and the properties of this restriction are `owl:onProperty` and `owl:allValuesFrom`. This virtual class is a class of all the individuals `X` for which this is true: If there are any statements that involve individual `X` as the subject and property `hasChild` as the predicate (statements of type `X hasChild *`), then the object in all of these statements must be of type `Child`. It does not mean that there must be statements involving `X` as the subject and `hasChild` as the predicate; there may have none. But if they exist, then the objects in all of them must be instances of class `Child`. If the resource belonging to the class `Parent` is connected to another resource of class `Dog` with the `hasChild` property, this means that there is inconsistency and the resource cannot be considered a `Parent`.

owl:someValuesFrom This restriction is similar to `owl:allValuesFrom`, but it is used to declare that only some values pertaining a property must be of a certain class. This is also a `minCardinality = 1` restriction. The property in question must have at least one instance of a certain class, but not all instances have to be of that class. It is written in Turtle as follows:

```
family:Sibling a owl:Class ;
rdfs:subClassOf family:Human , _:node15sb4m6gcx9 .

_:node15sb4m6gcx9 a owl:Restriction ;
owl:onProperty family:hasSibling ;
owl:someValuesFrom family:Sibling .
```

This means that for a person to be of type `Sibling`, it must be connected with at least one other instance of type `Sibling` with the `hasSibling` property. If the same person was connected with an instance of type `Baby` with the `hasSibling` property for example, it would still be considered correct, because it only states that some values from the `hasSibling` property must be of type `Sibling`, not all instances.

2.3.2.6 Restricted Cardinality

As mentioned previously in this thesis, OWL Lite has restricted cardinality, allowing only values of 0 and 1. These restrictions are known as local restrictions because they apply to properties of a certain class.

owl:minCardinality This stands for minimum cardinality. If the minimum cardinality on a property is 0, this just declares that the property is optional for a certain class. If the value is 1, it states that any instance of a specific class will be related to at least one individual with that property. Therefore, the reasoner can assume that the property must always have a value for instances of that class. To demonstrate, we can use the typical example of the property `hasChild` belonging to the class `Parent`. The property `hasChild` should have a `minCardinality` of 1, because a human cannot be a parent without having at least one child. It is not mandatory to be a parent though, so class `Human` can have a `minCardinality` of 0 on the `hasChild` property.

owl:maxCardinality This also applies to properties concerning a specific class. This restriction stands for maximum cardinality. If the maxCardinality of a property of a certain class is 1, then this means that any instance of that class will be related to one individual at most with that property. Stating maximum cardinality does not give any information for the reasoner about the minimum cardinality. If the maximum cardinality is 0, then instances of a class must not be related to any individuals by that property.

```
family:Child a owl:Class ; owl:equivalentClass family:Offspring
; rdfs:subClassOf family:Human , _:node15sb4m6gcx35 .
```

```
_:node15sb4m6gcx35 a owl:Restriction; owl:onProperty family:hasMother
; owl:onClass family:Mother
; owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger .
```

The above example states that there is a cardinality restriction on the class Child pertaining the hasMother property, with maxCardinality being 1. This means that each child cannot have more than one mother.

owl:cardinality This is useful when we want to say that minCardinality and maxCardinality are both of the same value. If a property's cardinality is 1, this means that the minimum cardinality is 1, and the maximum cardinality is 1. For example, the class Human has exactly one value for the property hasBirthMother. Knowing this, the reasoner can infer that if two different instances of class Mother are related to the same individual with the property hasBirthMother, then there must be a mistake or inconsistency.

2.3.2.7 Extra Restrictions

OWL Lite puts the restriction on the construct owl:intersectionOf that it must be used only on lists that have more than one member, and these members must be only class names. For example, stating that a Woman is the intersection of a Female and a Human with both belonging in a collection, is valid. OWL Lite also requires that the statements owl:allValuesFrom and owl:someValuesFrom have class names or datatype names as the object.

2.3.3 OWL DL

OWL DL extends OWL Lite, and adds maximum functionality and expressiveness of knowledge. DL stands for Description Logic. "A Description Logic (DL) models concepts, roles and individuals, and their relationships" [DL]. The goal of OWL DL is to provide maximum expressiveness but at the same time having completeness of reasoners. OWL DL has no restrictions on the language constructs, but these constructs have certain restrictions as to how they can be used. For example, transitive properties are not allowed to have cardinality restrictions. Every OWL DL ontology is an OWL Full ontology. OWL DL requires type separation (a class cannot be an individual or property at the same time, a property cannot also be an individual or class). Furthermore, OWL DL requires that properties are either object properties or datatype properties. As in OWL Lite, `owl:Class` is a subclass of `rdfs:Class`.

2.3.4 OWL Full

OWL Full provides maximum functionality and expressiveness just like OWL DL, but in the case of OWL Full there is complete syntactic freedom. It is the most complex language of the three, but it is the most powerful as well. There are no guarantees that every feature of OWL Full can be reasoned completely by reasoning software. Plus it takes longer time for a reasoner to complete its reasoning because of its complexity.

OWL Full is based on different semantics from OWL Lite or OWL DL, and is intended to be compatible with RDF Schema. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual, something not allowed in OWL DL or OWL Lite. RDF documents are in OWL Full, unless they are specifically constructed to be in OWL DL or Lite. This means that if a person finds an RDF document, they can assume that it is written in OWL Full unless specified otherwise. An OWL Full reasoner can therefore be used safely. Even if the document is stated to be in OWL DL, an OWL Full reasoner can be used because every legal OWL DL document is an OWL Full document.

In OWL Full, `rdfs:Class` and `owl:Class` are equivalent. Furthermore, object properties and datatype properties are no longer disjoint with each other as in OWL Lite and OWL DL, since data values can be treated as individuals.

The possibility to use the following constructs from OWL is added to OWL DL and

OWL Full, whereas it is forbidden in OWL Lite [46]:

owl:oneOf Classes can be described by enumeration of their individuals. For example, the class `Seasons` can be described by enumerating the individuals `Spring`, `Summer`, `Autumn`, `Winter`. The reasoner can deduce that the cardinality is 4 because there cannot be any more or less instances for this class.

owl:unionOf, **owl:complementOf**, **owl:intersectionOf** We can state for example that the class `Woman` is the intersection of the classes `Human` and `Female`.

owl:hasValue A property can have a requirement of having a certain individual as a value.

owl:disjointWith We can declare disjoint classes, e.g. an individual can never be an instance of `Woman` and `Man` simultaneously, so these classes can be stated as disjoint with each other.

minCardinality, **maxCardinality**, **cardinality** There is also no restriction on cardinality as in OWL Lite. We can have cardinality restrictions greater than 0 or 1.

2.4 Reasoners

Reasoners or rules engines are software tools that accept axioms or facts and produce logical consequences [54]. It is basically an inference engine. There are many types of reasoners, depending on the method of reasoning. Some of these are:

- forward chaining reasoners
- backward chaining reasoners
- description logic reasoners

2.4.1 Forward chaining reasoners

Forward chaining reasoners work as the name suggests, forwards, by starting with the data that has been given and by inference rules it can extract more data. This process goes

on until there is some break point or it reaches the end of the data. If there is a specific goal (for example to find out certain information), then the reasoner goes through the data until the goal is reached. Forward chaining is also called data-driven, because the data determines which rules will be used. One advantage of forward chaining reasoners compared to backward chaining reasoners is that since forward chaining reasoners start with data, new inferences can be triggered with the addition of new data. This gives them an edge for situations where dynamic content is a main aspect [50]. A popular forward chaining reasoner is CWM.

2.4.2 Backward chaining reasoners

Backward chaining reasoners are the opposite of forward chaining reasoners. They are also called goal-driven. This is because the reasoner starts with the goal and works backwards to see if there is data to support the goal, thus the goal determines which rules will be used [48].

2.4.3 Description Logic reasoners

Description Logic (DL) reasoners are reasoners that are made for OWL-DL documents. Some popular description logic reasoners include FaCT++, Pellet, RacerPro, CEL, and more. FaCT++ is based on C++ and is open-source software, whereas Pellet is Java-based and commercial software. RacerPro and CEL are Lisp-based.

2.5 Epilogue

Ontologies are needed in Semantic Web in order to describe abstract concepts and define class and property hierarchies. Ontologies are usually written in OWL, which is built on RDF Schema. RDF Schema semantically assigns definitions and is descriptive language (whereas XML Schema is prescriptive). RDF Schema is probably not sufficient enough to create a complex ontology, because it is not possible to have property or cardinality restrictions. OWL uses RDFS features and extends them by adding more functionality and possibilities. OWL has three sub-languages, OWL Lite, OWL DL, and OWL Full, each one being more complex than the previous. Reasoners are tools that perform reasoning, meaning that they infer and produce data using pre-defined rules. These ontologies can be

written in applications that are designed for this purpose. We will analyze one of these in the following chapter.

Chapter 3

Protégé

3.1 Introduction

Protégé is a free open-source platform developed by Stanford Center for Biomedical Informatics Research at the Stanford University School of Medicine. It provides tools to create ontologies and domain models. It is possible with Protégé to create, edit and visualize ontologies, as well as save them in various formats. Classes and properties in an ontology can be created, edited and visualized as well. Protégé supports the usage of various plug-ins as well as Java-based API (Application Programming Interface) [2].

The Stanford University currently supports three versions of Protégé, versions 3.4.5 Release, 4.0.2 Release, and 4.1 Beta. In this thesis the 4.0.2 Release version will be used. Although the 3.4.5 Release version might seem like a better option at the current moment, the goal of the thesis is to have a futuristic scope. The 3.4.5 Release version will continue to be supported for a while longer, but 4.0.2. Release version is the one that will become the future stable version. For this reason, the screen shots of the developed example ontology will be in the 4.0.2 Release version.

As stated in the OWL Web Ontology Language Guide [45], "An OWL ontology may include descriptions of classes, properties and their instances. Given such an ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics. These entailments may be based on a single document or multiple distributed documents that have been combined using defined OWL mechanisms".

3.2 Protégé Features

The main features of Protégé 4.0.2, taken from the Protégé Wiki page [27], are :

GUI Framework

- Configurable (persistent) layout of components
- Creation, import, and export of user configured tabs
- Multiple alternative views of the same ontology
- Tear-off and cloning of components
- Keyboard shortcuts
- Drag and drop support
- Lazy loading components/plugin-ins for improved speed and memory usage

API

- OWL API for OWL 2.0 provides efficient in-memory model
- Plug-in framework is OSGi compliant Equinox (easily extensible)
- Generic application framework is separated from OWL Editor Kit

Modularization

- Intelligent use of local/global repositories to handle import dependencies
- Loading of multiple ontologies into a single workspace
- Switching between ontologies dynamically
- UI hints for showing in which ontology statements are made
- Refactoring: merging ontologies and removal of redundant imports
- Refactoring: moving axioms between ontologies

Navigation

- History
- Global/local find
- Global usage
- Hyperlinking in editors

Refactoring Tools

- Renaming (including multiple entities)
- Handling disjoints/different
- Quick defined class creation
- Various transforms on restrictions (including covering)
- Conversion of IDs to labels
- Moving axioms between ontologies

Reasoning Support

- Inferred axioms show up in most standard views
- DL Query tab for testing arbitrary class expressions
- Direct interface to FaCT++ reasoner
- Direct interface to Pellet reasoner
- Reasoners are plug-ins

OWL Editing

- Consistent rendering of ontology entities, using URI fragments or annotation values
- OWL description parsing (also supports names in annotations)
- Built-in change support allowing compound changes and undo

- Autocompletion and expression history
- Syntax highlighting
- Auto creation of IDs/labels for new entities
- SWRL rules editing

Plug-ins

- Highly pluggable architecture with support for lots of different types of plugin including views, menu actions, reasoners, preferences, several manager hooks and more
- Auto-update for notification of new plugins and new versions
- Many plugins available including reasoners, matrices, scripting, clouds, existential tree, text mining, explanation, ontology processing, lint test framework, natural language generation and more

3.3 Comparison of Versions

A beginner may be confused as to which Protégé version should be used. There are significant differences between Protégé 3.4 and Protégé 4.0, thus the developer's choice should be considered carefully. Below there is an analysis and comparison of the two versions from different points of view.

Frames Support In Protégé 3.4 the editing of frames is supported through the Protégé-Frames editor. In version 4.0 there is no support as the Protégé-Frames editor has not been migrated yet.

OWL Support Version 3.4 supports OWL 1.0, whereas version 4.0 supports OWL 2.0. The latter version provides a framework purely for OWL language, while version 3.4 provides support for OWL and RDF Schema. There is no SPARQL support yet in Protégé 4.0, as well as no support for OWL Full. Protégé 4.0 has direct connection to Pellet, FaCT++ and other DL reasoners for optimum speed.

Plugins In both Protégé versions a large set of plugins is provided, developed by Stanford as well as external developers. In version 4.0 the plugin framework has been switched to OSGi, which allows any type of plugin extension.

User Interface In version 3.4, widgets are used for configuring the user interface, whereas in 4.0 version, plugins define all the user interface elements.

Multi-user Support In version 3.4, many users can edit the same ontology simultaneously via the client-server version of Protégé, whereas it is not possible in Protégé 4.0 yet.

Database Storage Model In Protégé 3.4, one can store ontologies in a database provided by the JDBC database back-end. This is not possible in version 4.0 as the database back-end has not been migrated yet.

3.4 Demonstration of Protégé

An example ontology has been created specifically to demonstrate Protégé and specifically version 4.0.2. This is based on the domain model known to everyone, for example a family domain. This ontology was created also to demonstrate property characteristics and restrictions in OWL, as well as cardinality constraints. The complete OWL document can be found in Appendix A. The goal of this example is to create a complete ontology. Several instances are provided in order to show examples of property characteristics and restrictions.

The main screen of Protégé is the Welcome screen, shown in Figure 3.4.1. We open the family2.owl file using the "Open OWL Ontology" button.

The first screen that opens by default is the "Active Ontology" tab (Figure 3.4.2). This serves as an overview of the whole ontology. Annotations about the ontology include information about the creator, language, publisher, contributors, and more. Metrics on the ontology can also be seen here, as well as axioms of each content (classes, object properties, and more).

The next tab on the Main menu is the Entities tab (Figure 3.4.3). Here one can navigate through all the classes, properties and individuals that belong to the ontology. This way everything is in one place.

Welcome to Protégé

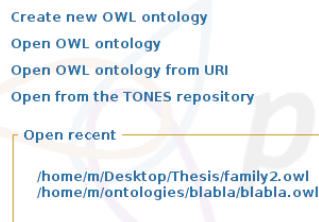


Figure 3.4.1: Protégé 4.0.2 welcome screen

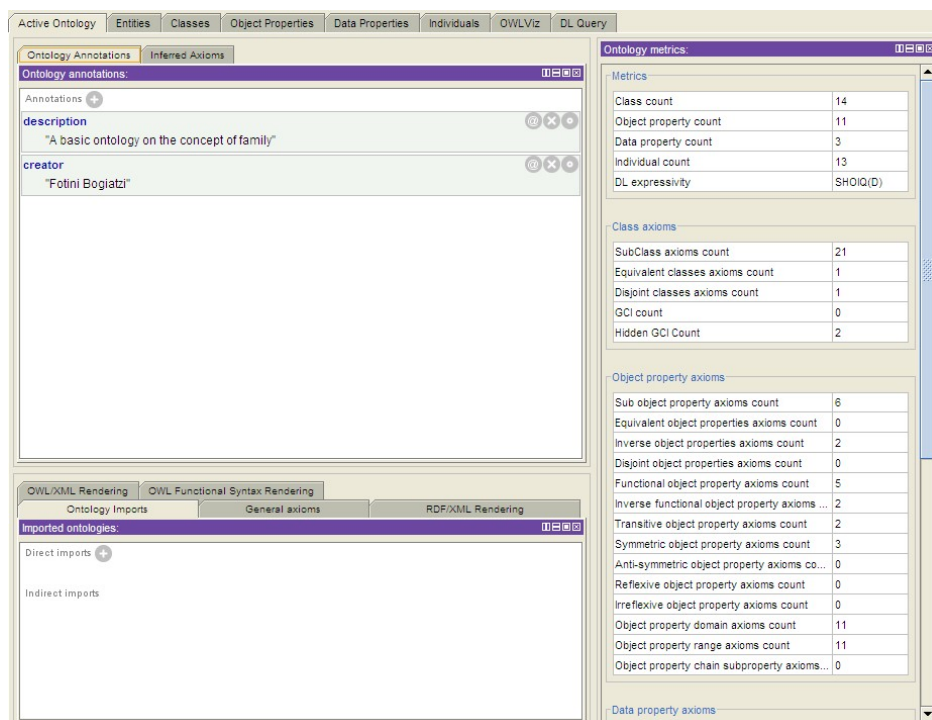


Figure 3.4.2: Active Ontology screen

3.4.1 Classes

One of the first things that someone should do when creating an ontology is to create the classes, therefore define a class hierarchy. This can be done in the Entities section, but it is possible to be done by clicking on the "Classes" tab located on the Main menu. This section contrary to the Entities tab contains information only about classes and their hierarchy. This brings us to the Class window. We can see the list of available classes and their members. In an empty ontology, only the superclass Thing will exist. To add a

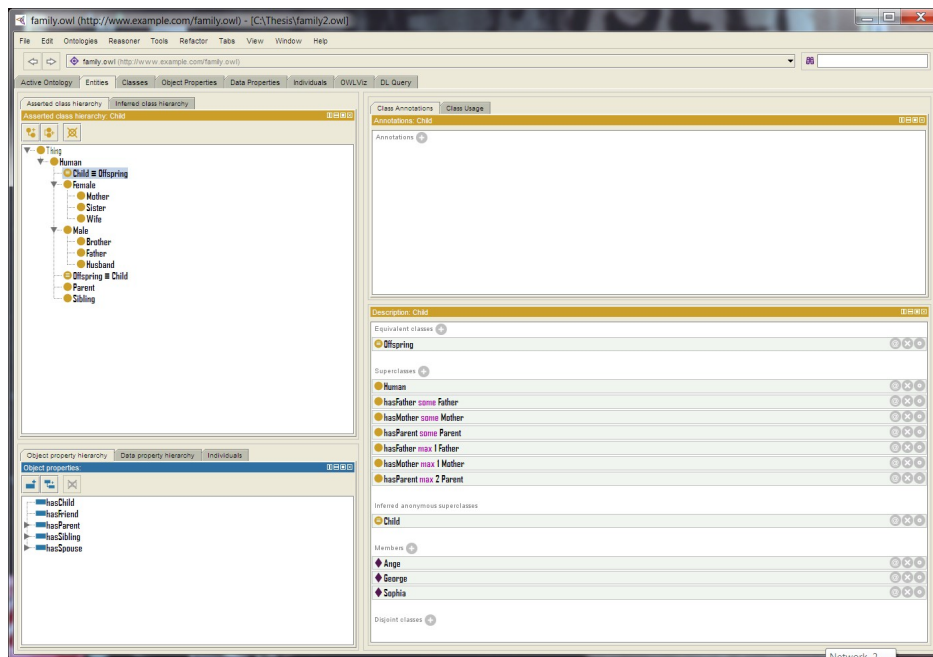


Figure 3.4.3: Entities tab in the family ontology

new subclass, we click on the "Add Subclass" button located far-left above class "Thing". From the same row of buttons the right button is for deleting a class. In Figure 3.4.4 we can see the dialog box that prompts the user to enter a class name. Directly underneath the textbox, the URI of the class name can be seen, with the fragment identifier being last. This is what separates the actual resource name from the rest of the URI, as mentioned previously.

In Figure 3.4.5 that is following, we can see the list of classes that exist in the Family ontology that was developed for the purpose of demonstration in the thesis.

Equivalent Classes To set equivalent classes in Protégé, the user must click on the "Equivalent Classes" setting that is on top at the Description box. Of course the proper class must be selected, and it must be done in the "Classes" tab. For example, if we want to set the classes "Child" and "Offspring" to be equivalent, we click on the "Add" button, and are presented with the screen shown below (Figure 3.4.6).

Cardinality Views In order to set cardinality restrictions in the ontology, the Cardinality View plug-in should be installed first of all. To enable viewing and usage, by clicking on "View → Class views → Cardinality View" we can set the corresponding box to be positioned anywhere we want in the Classes tab view. To create a new cardinality restriction, we click

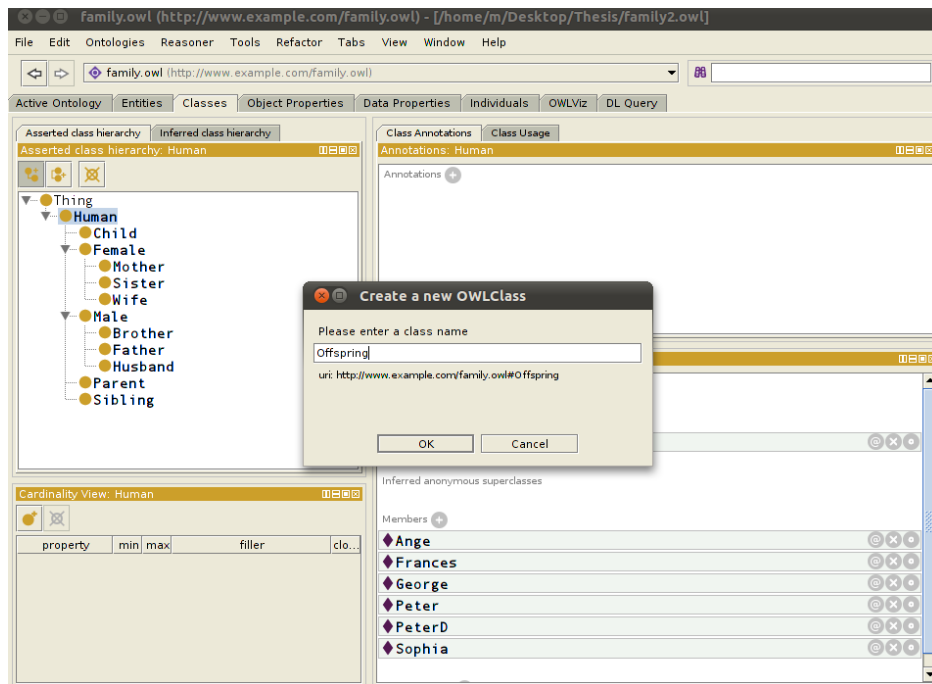


Figure 3.4.4: Adding a subclass

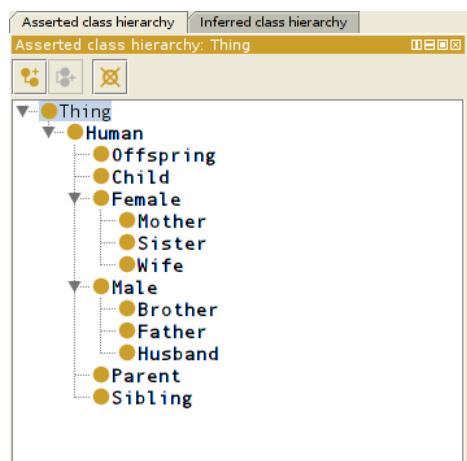


Figure 3.4.5: List of Classes of the family ontology

on "Add Row" which brings up the screen shown in Figure 3.4.7. In this case we are creating a cardinality restriction for the property `hasMother`. This ontology was designed for a family with biological parents, which means that the property `hasMother` is referring to the birth mother. Each child has only one biological mother or father, so we can state that the cardinality restriction for `hasMother` is exactly one. This is achieved in Protégé by setting minimum of 1 and maximum of 1.

We can see in Figure 3.4.7 that the classes `Offspring` and `Child` are indeed set to be

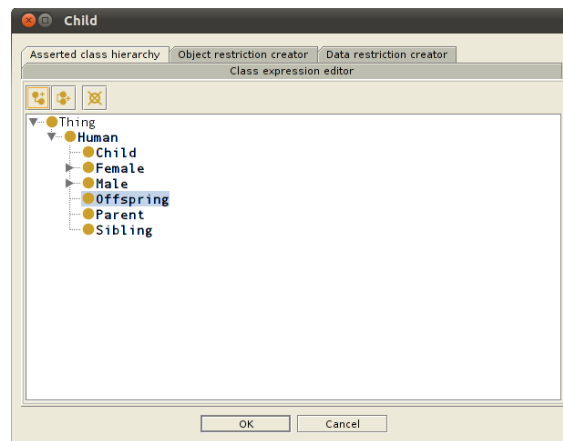


Figure 3.4.6: Choosing an equivalent class for the class Child

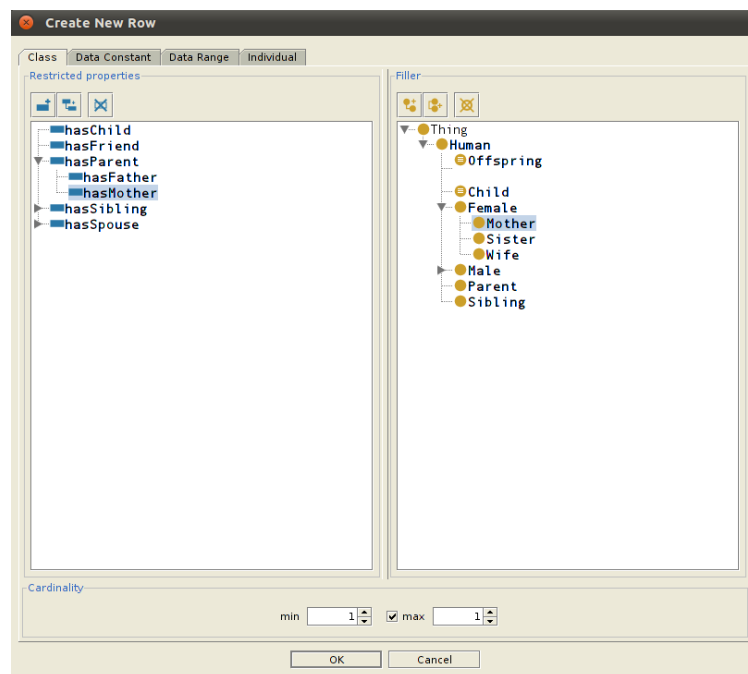


Figure 3.4.7: Creating cardinality restriction for the hasMother property

equivalent, due to the different circle preceding the class name. The three white lines indicate the equivalence.

3.4.2 Properties

Object Properties In order to define object properties, we click on the tab "Object Properties" located on the Main Menu. We can see that the screen is divided again in different sections. The Characteristics section is for setting special characteristics on the

properties. Therefore we can define symmetric properties, inverse functional properties, transitive properties and more. This is done simply by clicking in the appropriate box of the desirable characteristic. The reasoner can thus deduce certain information that it wouldn't be able to do otherwise. For example, unless otherwise stated, the reasoner cannot know that the property `hasFriend` is symmetric. Everything must be explicitly defined in order to have a complete ontology. Below, in Figure 3.4.8, we can see the screen of the Object Properties along with the list of properties that the Family ontology contains. Properties can have sub-properties as it was mentioned previously in the thesis, and this also can be seen. The reasoner can deduce that sub-properties contain at least the same characteristics as the super-property. Object properties in Protégé are identified and recognized by their distinctive blue color.

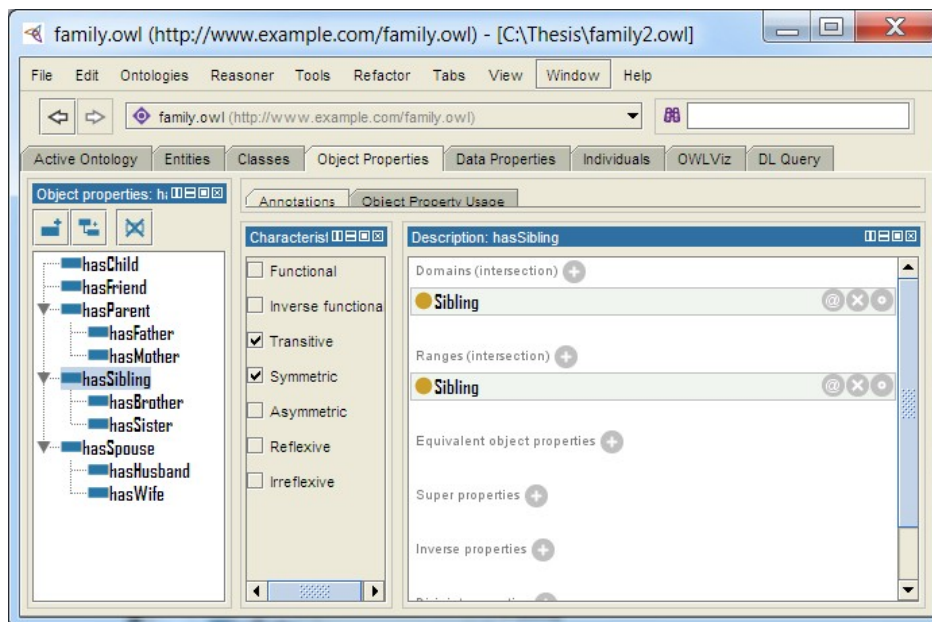


Figure 3.4.8: List of Object Properties in family ontology

Data Properties The data properties of each ontology can be seen by clicking on the "Data Properties" tab. Here we can see a similar screen as the "Object Properties" section. Data properties are defined and recognized in the ontology editor with the green color (Figure 3.4.9). Also, they have only one characteristic which is the functional characteristic. Sub-properties can also be defined here, although in this example there are none. The `firstName` and `hasSurname` properties can be defined as disjoint if we do not want the same name to be as a first name and surname. Decisions like these are arbitrary.

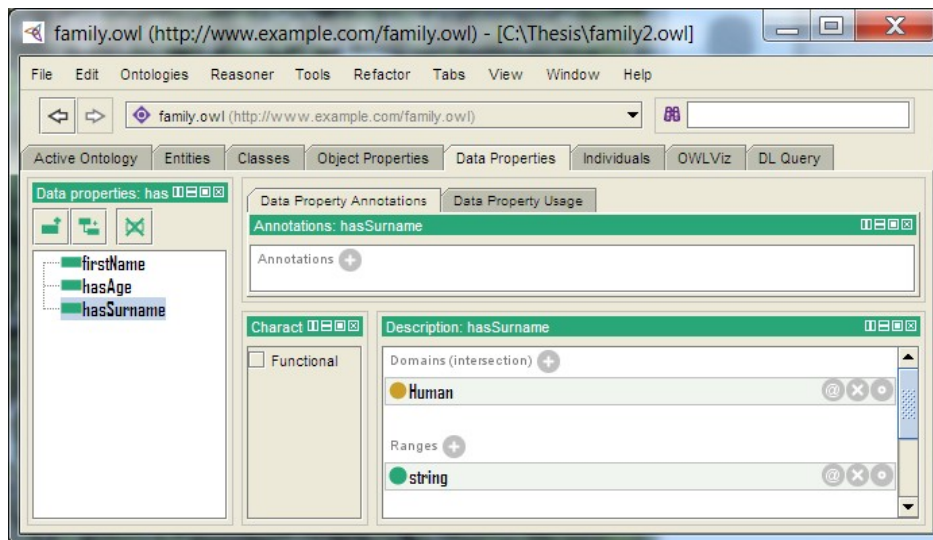


Figure 3.4.9: List of Data Properties for family ontology

3.4.3 Instances - Individuals

Individuals are class instances created to make the ontology more concrete. They are the resources. The list of these individuals can be found in the "Individuals" tab on the Main Menu. To create a new individual, in this case a human part of the family ontology, we simply click on the "Add individual" located in the Individuals section box. If we want to delete one, we click on the "Delete individual(s)". The description of each individual can be seen also in the "Description" section, as well as the "Property assertions" section. From these two sections we can see the summary of each individual in the ontology, which classes it belongs to and which properties it is associated with. An example of this is seen in Figure 3.4.10 about the individual named Sophia.

To assert a new property for an individual, we click on the "Add" button on the appropriate type of property we desire. For example, to set Sophia's mother, by clicking on the plus sign in the "Object property assertions" part we are met with the following screen (Figure 3.4.11).

We can be sure that these are object properties from the blue color.

In large ontologies with many individuals per class, a user may want to view the list of those individuals grouped by their corresponding classes. This is a quick way to see all the classes that contain instances, and how many instances each class has. This is done by clicking on the "Individuals By Class" tab. The result of the currently used family ontology is shown below in Figure 3.4.12.

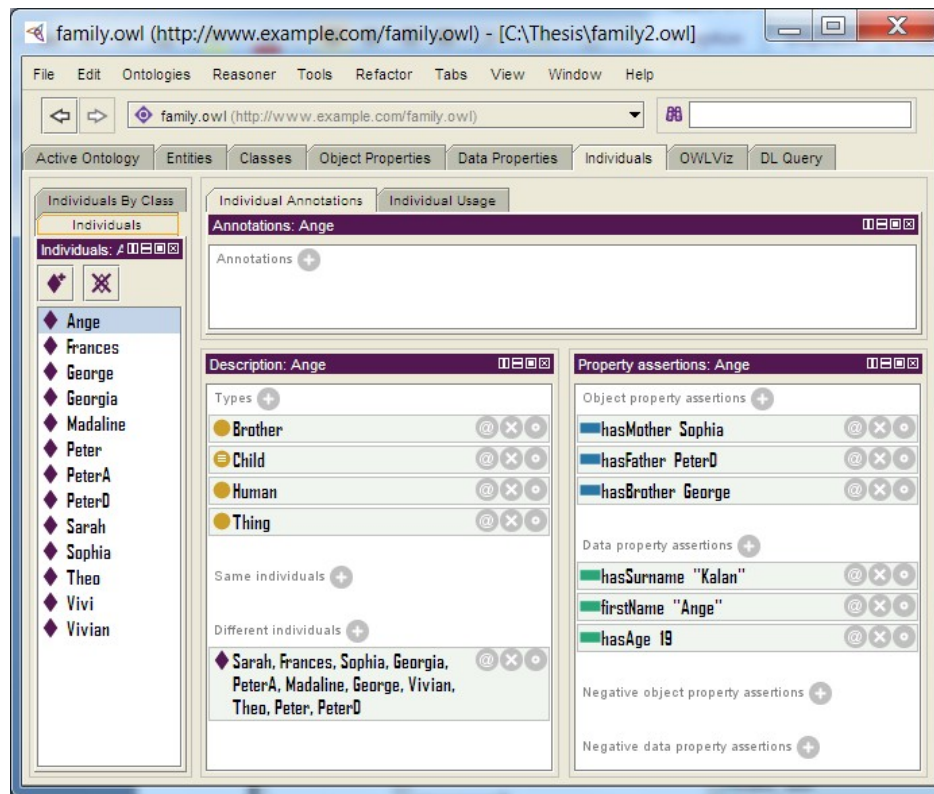


Figure 3.4.10: Individuals Tab with additional information

3.4.4 Refactoring

To change the name of an entity, we click on "Refactor" located on the Title menu and from there on "Change entity URI". There is an option to change either a specific entity name, or even all the entities with the same URI. If there is a large ontology, there it is possible to "Change multiple entity URIs". In this case there is no distinction between separate URIs. This means that if a class and an object property contain the same name, they will both be changed. For example, the class *Sister* and the object property *hasSister*.

3.4.5 Reasoning

Reasoners are integrated into the Protégé environment with plugins. Any reasoner can be used, each one has its advantages and disadvantages. Pellet is written in Java, and FaCT++ is written in C++. To enable the reasoner, it simply needs to be clicked on the drop down menu of "Reasoner". Immediately the reasoner deduces any information that has not been explicitly defined. This can be seen by the differently colored background in the parts that were reasoned, as well as the dotted border. For example, in Figure 3.4.13

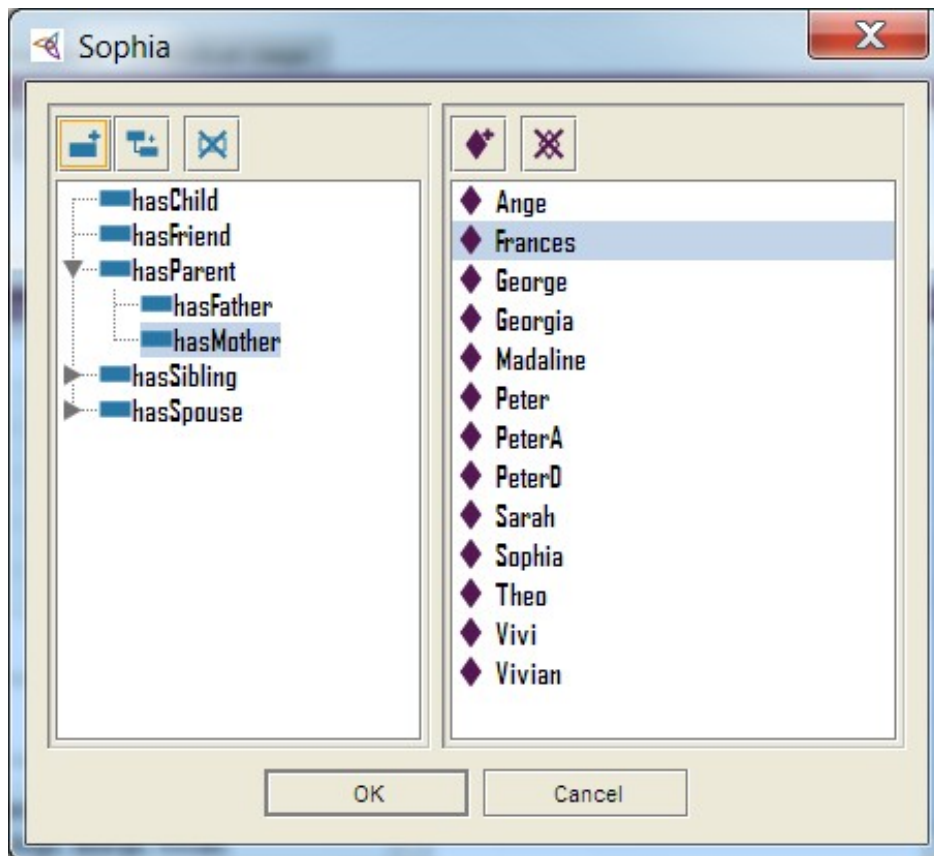


Figure 3.4.11: Object Property assertion for Individual "Sophia"

below, we can see that the reasoner deduced that the domain of the property `hasMother` can also be `Offspring`, and that the range can be a `Parent` as well.

In the tab "Inferred class hierarchy" located in the Entities or Classes section, we can see that the reasoner has deduced also that the class `Nothing` is a subclass of class `Thing` (Figure 3.4.14).

3.4.6 OWLViz

The OWLViz tab is for visualizing the hierarchy of classes in the form of a graph (Figure 3.4.15). They can also be incrementally navigated, something very helpful for large ontologies or ontologies based on complicated domain models. The asserted class hierarchy and the inferred class hierarchy can be compared this way as well. The graph can easily be exported and saved as an image file. This is done by clicking the "Export to Image" button on the OWLViz toolbar.

It may happen that at first errors will occur upon clicking the "OWLViz" tab, it is usually

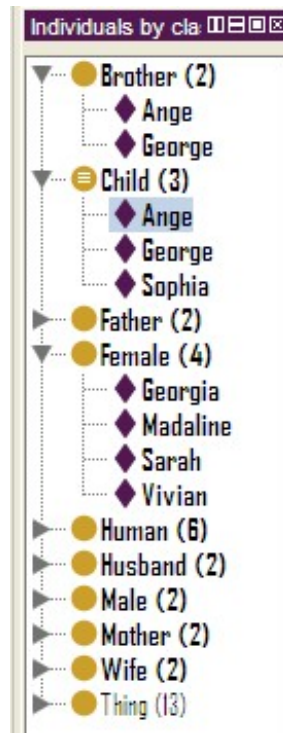


Figure 3.4.12: List of Individuals grouped by Class

an occurrence when GraphViz is not installed, or when the GraphViz DOT application is not found in the path. Simply correcting the path to the application in the preferences fixes the problem.

3.4.7 Export

To save the ontology, all that is needed is to click on "File-> Save". The extension of the file will be .owl. This file now can be imported into RDF storage, query processor, etc. However, in order to include also the inferred axioms created by the reasoner, it is needed to click on "File->Export inferred axioms as ontology". This is done after the reasoner (e.g. Pellet) has been activated in Protégé. Otherwise, Sesame will not be able to detect the inferred information during SPARQL queries (more information in Chapter 5).

3.4.8 Plug-Ins

There are some interesting plug-ins available for Protégé, including :

OWL2Query OWL2Query is a conjunctive query and metaquery engine and visualization plug-in. It facilitates creation of queries using SPARQL or graph-based syntax,

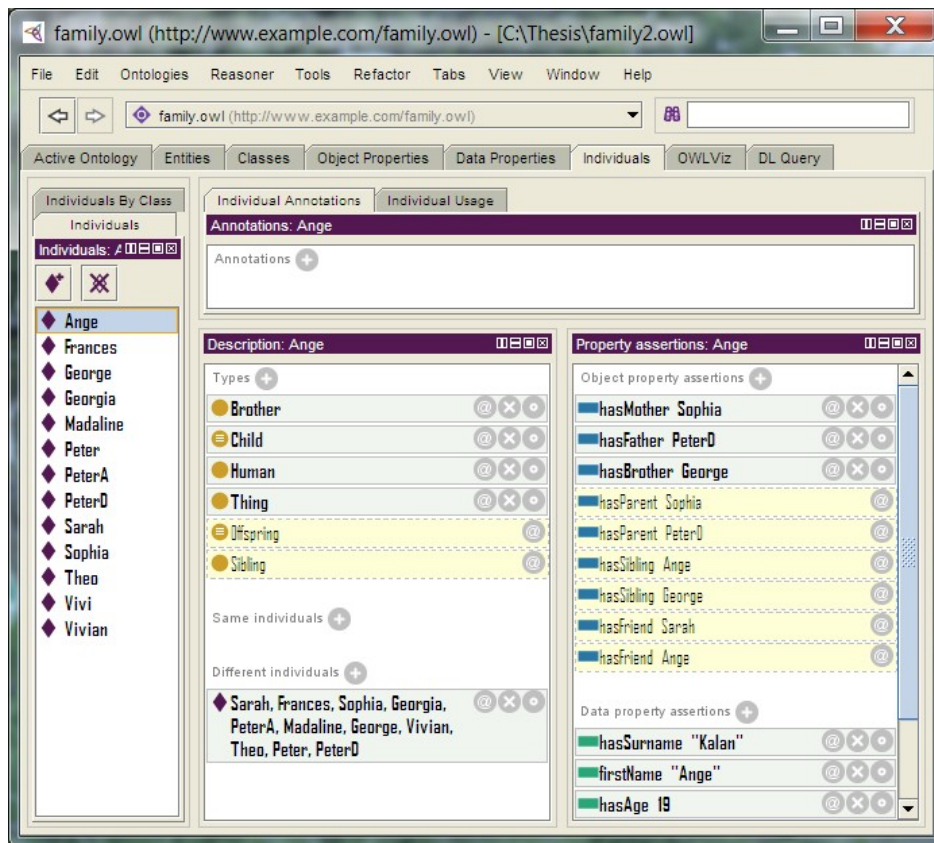


Figure 3.4.13: Results from the reasoner

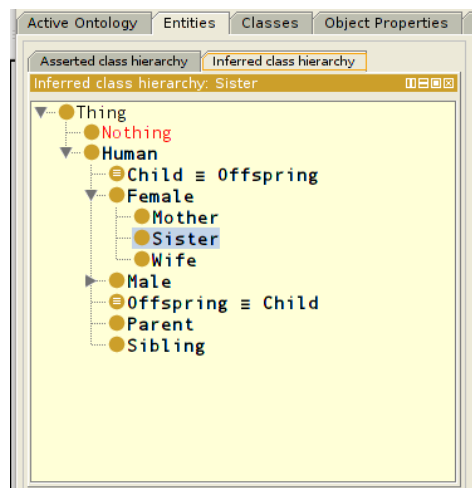


Figure 3.4.14: Inferred class hierarchy

and evaluates them using any OWL API-compliant reasoner [22]. This plug-in is compatible with the 4.1 Beta version.

Annotation Template View A view that can be configured to show a set of standard

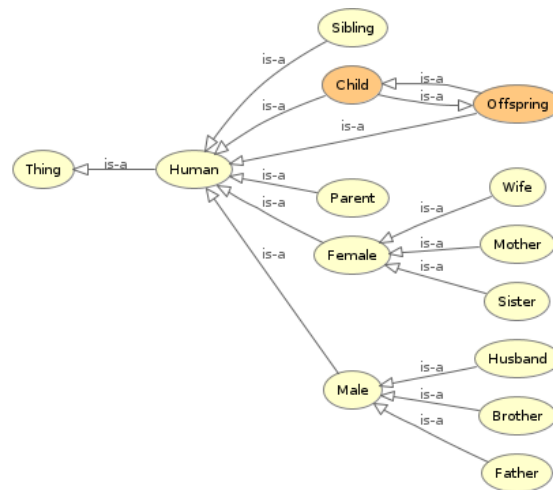


Figure 3.4.15: Visualization of the class hierarchy

annotation fields for every class/property/individual. This makes it quicker to annotate as they are always in the same order, editable inline and easy to navigate [14].

Cardinality View An experimental alternative to the conditions widget that shows restrictions in relation to their cardinality.

Cloud Views This plug-in helps the developer and contributors to visualize the ontology as a tag cloud, which is a set of related tags.

OWLViz Enables class hierarchies in an OWL ontology to be viewed and incrementally navigated, allowing comparison of the asserted class hierarchy and the inferred class hierarchy.

Changes Tab Allows users to track and annotate changes to Protégé ontologies.

3.5 Epilogue

Chapter 3 describes the platform that provides tools to create ontologies and domain models, Protégé. A comparison of versions is given for those that are not sure where to start from. A demonstration of the creation of an ontology is also provided using step-by-step examples and screenshots of the process. In the following chapter, we will talk about the query language for RDF.

Chapter 4

SPARQL

4.1 General

SPARQL is a recursive acronym that stands for SPARQL Protocol and RDF Query Language. As is suggested by the name, SPARQL is both a protocol and an RDF query language. The SPARQL query language for RDF is a specification for querying RDF data, done by pattern matching. SPARQL queries RDF data (datasets) which is in the form of graphs [15]. The SPARQL protocol for RDF is a specification for invoking SPARQL queries remotely, and receiving the results. "It specifies a simple interface that can be supported via HTTP or SOAP that a client can use to issue SPARQL queries against some endpoint" [15]. SPARQL is considered the query language of Semantic Web and is part of the Semantic Web vision. There are currently two versions of SPARQL, 1.0 and 1.1. On 15 January 2008, SPARQL 1.0 became an official W3C Recommendation [41]. For the rest of this thesis, whenever we refer to SPARQL we mean version 1.0. Furthermore, we mean the SPARQL query language for RDF and the SPARQL protocol for RDF will not be analyzed. Both the SPARQL query language for RDF and the SPARQL protocol for RDF are products of the RDF Data Access Working Group of W3C.

SPARQL 1.0 includes:

1. SPARQL 1.0 Query Language
2. SPARQL 1.0 Protocol
3. SPARQL Results XML Format

SPARQL 1.1 is currently in development, and includes:

1. Updated 1.1 versions of SPARQL Query and SPARQL Protocol
2. SPARQL 1.1 Update
3. SPARQL 1.1 Uniform HTTP Protocol for Managing RDF Graphs
4. SPARQL 1.1 Service Descriptions
5. SPARQL 1.1 Entailments
6. SPARQL 1.1 Basic Federated Query

With SPARQL it is possible to query unknown relationships, transform RDF data from one ontology to another, query structured as well as semi-structured data, and unite different databases into a single query [16].

4.2 Syntax

As mentioned previously, SPARQL queries data by matching patterns. This means that subgraphs of the RDF dataset are defined, and if such subgraphs exist (if a substitution for the variable is found) then they appear in the query result set. The only difference is that the subgraphs in the query may contain variables.

SPARQL query results can return in various formats, i.e. RDF, XML, HTML and JSON. A SPARQL query may contain [16]:

1. Prefix declarations, to abbreviate URIs
2. Result clause, to declare what should be returned and shown in the result set
3. Dataset definition, to state what RDF graphs are being queried
4. Query pattern, to specify the query
5. Query modifiers, to modify the query results in quantity and order

An example of a query is the following ('#' means one-line comment):

```
# prefix declarations
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX f: <http://www.example.com/ontology/family#>
```

```
# result clause
SELECT ?mother

# dataset definition
FROM <http://www.example.com/ontology/family>

# query pattern
WHERE {
    ?mother rdf:type f:Mother
}

# query modifiers
ORDER BY ...
```

4.2.1 Basic Query

At a glance, SPARQL is syntactically like SQL. This feature makes it easy for SQL users to learn how to use SPARQL. Like SQL, SPARQL also consists of two mandatory clauses (SELECT and WHERE) and one optional (FROM). The SELECT clause defines the variables that should appear in the query results. The FROM clause states the RDF graphs that will be queried. The WHERE clause declares the query pattern that should match against the RDF data.

The triple in the following example query is a basic graph pattern, consisting of a triple pattern with one variable in place of the object. For the following queries of the whole chapter, we will use the RDF dataset shown in Appendix A. The queries will be written and shown in Sesame.

RDF data:

```
family:Wife rdf:type owl:Class .
```

SPARQL query:

```
SELECT ?class
WHERE
```



```
{  
  family:Wife rdf:type ?class .  
}
```

The variable could have any character or name (e.g. ?x). The query returns only one solution, shown below in Figure 4.2.1. If we want to select all the variables that appear in a query, we can also write "SELECT *".

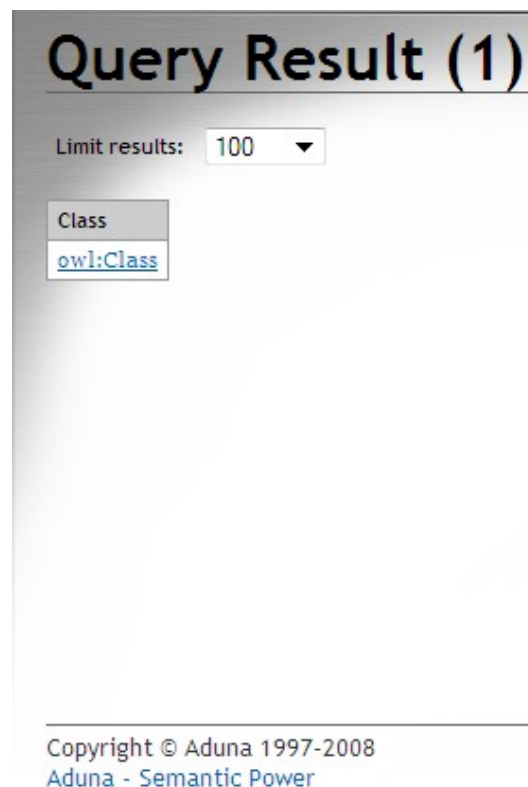


Figure 4.2.1: Query result

The result of the query is called a solution sequence. The sequence depends on the way of the pattern matching. A query may have none, one or multiple solutions. Each solution presents an alternative of how the query pattern could match the data. A solution sequence is a list of solutions, which is not necessarily ordered. This is done by the solution sequence modifiers (explained further below in section 4.2.3). In a result set, we can see all the possible bindings. A binding is a pair matching a variable to an RDF term. This RDF term can be a URI, a literal, etc. In the result set shown in Figure 4.2.1, the variable Class is bound to the RDF term owl:Class.

When there is a literal in the query pattern, the pattern matching varies depending on the type of literal, further analyzed below.

Language Tags When a literal is accompanied by a language tag, then the query pattern must contain the language tag as well, otherwise a solution cannot be found. The literal "Sophia" is not the same as "Sophia"@en.

Numeric Types A typed literal representing a number (e.g. xsd:integer) can be shortened to only its numeric value. In a query asking who is aged 42 years old, instead of writing:

```
?person f:hasAge "42"^^xsd:integer .
```

we can make the query simpler and write:

```
?person f:hasAge 42 .
```

The same is possible for a typed literal representing a string (xsd:string). When the string contains whitespace, it should be enclosed in double quotes.

For arbitrary datatypes, the full URI must be written. A solution will be found if such exists because of the lexical matching. As stated in the W3C document for SPARQL, "The query processor does not have to have any understanding of the values in the space of the datatype" [33].

4.2.1.1 Blank nodes in a query result

A query result may contain blank nodes as well. These are written as "_:" followed by the corresponding label. The scope of the blank nodes are local to the result set. It does not mean that the same blank node label will be returned as the existing label in the dataset, and it should not be expected. We can safely assume that two blank nodes are different if they have different labels from each other in the result set.

4.2.1.2 Guidelines and syntax

To abbreviate a URI, instead of writing @prefix which is correct to some notations, we use the keyword PREFIX: .

For literals, double quotes (" ") or single quotes (' ') may be used with an optional language tag (e.g., "@en"). If it is a typed literal, then it can be followed by the datatype

URI, starting with "^^". If there is a need to have quotation marks in the literal itself, then SPARQL allows this with the usage of three single (or double) quotation marks.

Variables in the query should be preceded by the question mark "?" or the dollar sign "\$". These characters are not part of the variable name and they indicate to the query processor that the label is a variable. Query variables have a global scope for each query.

Blank nodes in a query are not references to blank nodes in the data. They serve to pinpoint variables that cannot be distinguished. They can be written with their label (e.g., `_:54xwd`), or with their abbreviated form, `[]`. We cannot use the same blank node label in two different graph patterns of the same query. There are two possible ways to write blank nodes in their abbreviated form:

```
[ ] f:hasAge "30" .
```

```
[ f:hasAge "30" ] .
```

Both cases are the same as the statement:

```
_:5e5 f:hasAge "30" .
```

It is possible to truncate common concepts in a query, for example if a subject is used more than once in the graph pattern. This is done by using the ";" symbol and writing the subject only in the first triple. These are called predicate-object lists. For example, instead of writing:

```
f:Sarah f:hasFriend f:Ange .
```

```
f:Sarah f:hasAge "19" .
```

It is possible to write:

```
f:Sarah f:hasFriend f:Ange ;
```

```
f:hasAge "19" .
```

It is possible to have object lists, when multiple triples share the subject and the predicate. In this case the usage is with the comma ",". For example:

```
f:Madaline f:hasSibling f: Sophia .
```

```
f:Madaline f:hasSibling f: PeterA .
```

We can write:

```
f:Madaline  f:hasSibling  f:Sophia , f:PeterA .
```

4.2.2 Graph Patterns

SPARQL deals with graph pattern matching [33]. Simple as well as complex graph patterns can be used. The various types of graph patterns according to the specification are:

1. Basic Graph Patterns
2. Group Graph Patterns
3. Optional Graph Patterns
4. Alternative Graph Patterns
5. Patterns on Named Graphs

4.2.2.1 Basic Graph Patterns

Basic graph patterns are those that are a set of triple patterns. Using these basic graph patterns we can group them, add a filter, etc. The set of triple patterns must match the query in order for results to show.

4.2.2.2 Group Graph Patterns

Group graph patterns are used with braces `{ }`, inside which the triple patterns are enclosed. It is possible to have a group of one main basic graph patterns, or to break up the group by using an empty group pattern in between the triples patterns. A constraint placed on the group (keyword `FILTER`) has a scope on the whole group. It does not matter where the constraint is placed (whether it is before the triple patterns, in between them or at the bottom). The keyword `FILTER` has a variety of functions. The `FILTER` function `regex()` matches plain literals that are not accompanied by language tags. The format is `regex(string, pattern)`.

```
SELECT ?surname
WHERE { ?x family:hasSurname ?surname .
       ?x family:firstName ?name
```

```

    FILTER regex (?name, "Peter")
  }

```

The result is shown in Figure 4.2.2.



Figure 4.2.2: Individuals with their first name as "Peter"

The regular expression matching is case-sensitive. To make it case-insensitive, we use the flag "i" [18]. The format is `regex (string, pattern, flags)`. The query below:

```

SELECT ?surname
WHERE {
  ?x family:hasSurname ?surname .
  ?x family:firstName ?name
  FILTER regex (?name, "peter" , "i")
}

```

produces the same result set as in Figure 4.2.2.

FILTER can restrict also numeric values by using arithmetic expressions. The function `regex` is not used here. The result of the example below is shown in Figure 4.2.3.

```

SELECT ?name ?age
WHERE {
  ?name family:hasAge ?age
  FILTER (?age < 30)
}

```

Name	Age
familv:Ange	"19"^^xsd:int
familv:George	"26"^^xsd:int
familv:Sarah	"19"^^xsd:int
familv:Vivi	"20"^^xsd:int
familv:Vivian	"20"^^xsd:int

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.3: All individuals under the age of 30

Another function used with FILTER is `bound()`. The format is `bound(?var)` and returns true if the variable is bound to a value. Even if the variable has a value NaN (Not a Number), it is considered bound. For example, let us query all the people that have an age. The results are shown in Figure 4.2.4.

```
SELECT ?name ?age
WHERE { ?x family:firstName ?name .
        OPTIONAL { ?x family:hasAge ?age } .
        FILTER (bound(?age))
}
```

We can also use Negation to query if a graph pattern is not expressed by asking if the variable is not bound. The negation of `bound()` is `!bound()`. An example is shown below, showing all the people that do not have an age. The result is shown in Figure 4.2.5.

```
SELECT ?name ?age
WHERE { ?x family:firstName ?name .
        OPTIONAL { ?x family:hasAge ?age } .
        FILTER (!bound(?age))
}
```

It is possible to query if two RDF terms are defined as equal. We do this by using the RDFterm-equal operator. This is expressed with the operator `"="`. Logical-or and

Query Result (9)

Limit results: 100 ▾

Name	Age
"Ange"^^xsd:string	"19"^^xsd:int
"Frances"^^xsd:string	"79"^^xsd:int
"George"^^xsd:string	"26"^^xsd:int
"Peter"^^xsd:string	"87"^^xsd:int
"Peter"^^xsd:string	"60"^^xsd:int
"Sarah"^^xsd:string	"19"^^xsd:int
"Sophia"^^xsd:string	"57"^^xsd:int
"Vivian"^^xsd:string	"20"^^xsd:int
"Vivian"^^xsd:string	"20"^^xsd:int

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.4: Results of the bound function

Query Result (3)

Limit results: 100 ▾

Name	Age
"Georgia"^^xsd:string	
"Georgia"^^xsd:string	
"Theo"^^xsd:string	

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.5: Results of !bound()

logical-and can also be used in SPARQL, using the operators "||" and "&&". Using the `isBlank()` function on `FILTER`, we can also query if an RDF term is a blank node.

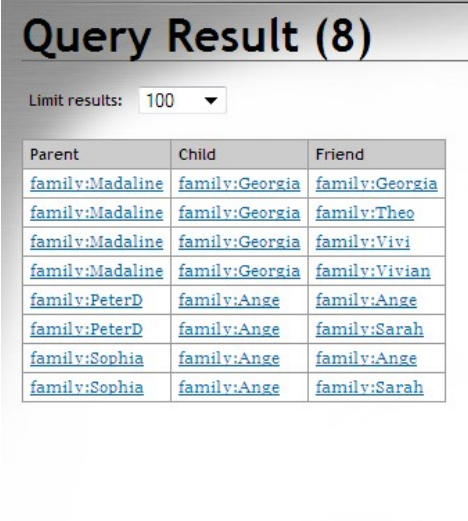
4.2.2.3 Optional Graph Patterns

The patterns that are queried demand that the entire patterns match in order for a solution to appear in the query result set. As mentioned previously, every variable is bound to an RDF

term in each solution. However, not all data is structured and complete. SPARQL excels by querying semi-structured data. This is done by employing the OPTIONAL keyword. With this keyword, solutions are not eliminated if the optional part does not match. This is especially helpful when semi-structured data exists with partial information. To demonstrate this, let us consider the following example.

```
SELECT ?parent ?child ?friend
WHERE { ?parent family:hasChild ?child .
        ?child family:hasFriend ?friend
}
```

The query above consists of a basic graph pattern, which in turn consists of two triple patterns. The query translates to: "Show me all the parents with their children that have friends". The solution sequence is shown below in Figure 4.2.6.



Query Result (8)

Limit results: 100

Parent	Child	Friend
family:Madaline	family:Georgia	family:Georgia
family:Madaline	family:Georgia	family:Theo
family:Madaline	family:Georgia	family:Vivi
family:Madaline	family:Georgia	family:Vivian
family:PeterD	family:Ange	family:Ange
family:PeterD	family:Ange	family:Sarah
family:Sophia	family:Ange	family:Ange
family:Sophia	family:Ange	family:Sarah

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.6: Parents with children and their friends

All the variables are bound to the solutions, because it is a requirement for basic graph patterns. However, if we add the OPTIONAL keyword:

```
SELECT ?parent ?child ?friend
WHERE { ?parent family:hasChild ?child .
OPTIONAL { ?child family:hasFriend ?friend }
}
```


}

We get a very different result (Figure 4.2.7):

Query Result (15)

Limit results: 100

Parent	Child	Friend
family:Frances	family:Madaline	
family:Frances	family:PeterA	
family:Frances	family:Sophia	
family:Madaline	family:Georgia	family:Georgia
family:Madaline	family:Georgia	family:Theo
family:Madaline	family:Georgia	family:Vivi
family:Madaline	family:Georgia	family:Vivian
family:Peter	family:Madaline	
family:Peter	family:Sophia	
family:PeterD	family:Ange	family:Ange
family:PeterD	family:Ange	family:Sarah
family:PeterD	family:George	
family:Sophia	family:Ange	family:Ange
family:Sophia	family:Ange	family:Sarah
family:Sophia	family:George	

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.7: Parents with children that could possibly have friends

We can observe in Figure 4.2.7 that not all variables are bound to the solutions, and therefore even children without friends were included in the results. This query translates to: "Show me all the parents and their children, and if the children have friends show them as well". We can also see some cases where a child is a friend of himself. This is the case because the property `f:hasFriend` has been set in the ontology to be both inverse and transitive. Constraints on OPTIONAL Constraints can be placed on an optional graph pattern as well. For example, we could apply an age filter on the persons that have age information about them. This way all the humans will show up in the results, but for those that are younger than 30 years old (and have an age), their age will show up as well (Figure 4.2.8).

```

SELECT ?person ?age
WHERE { ?person rdf:type family:Human .
OPTIONAL { ?person family:hasAge ?age . FILTER (?age < 30)
}
}

```

Query Result (13)

Limit results: 100 ▾

Person	Age
family:Ange	"19"^^xsd:int
family:Frances	
family:George	"26"^^xsd:int
family:Georgia	
family:Madaline	
family:Peter	
family:PeterA	
family:PeterD	
family:Sarah	"19"^^xsd:int
family:Sophia	
family:Theo	
family:Vivi	"20"^^xsd:int
family:Vivian	"20"^^xsd:int

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.8: FILTER on OPTIONAL clause effect

Among the solutions that don't have the age variable bound, are those that didn't have any age, along with those that are older than the applied filter (30 years old).

4.2.2.4 Alternative Graph Patterns

Let us take the case where we would want to see all the people that belong to the Brother or Sister class. Without UNION, we would do it by querying instances of the Sibling class only, as shown below in Figure 4.2.9.

```
SELECT ?sibling
WHERE { ?sibling rdf:type family:Sibling }
```

This method, however, is not flexible. We cannot know immediately if the person belongs to the Brother or Sister class, and we cannot use any constraints on those separate classes. With UNION however, this is possible. Results of the following query are shown in Figure 4.2.10.

```
SELECT ?sister ?brother
WHERE { {?sister rdf:type family:Sister } UNION
{ ?brother rdf:type family:Brother } }
```

Now that we have separated the graph patterns into two alternatives, we can for example place additional patterns, filters or constraints depending on what we want. For example

Query Result (5)

Limit results: 100 ▾

Sibling
family:Ange
family:George
family:Madaline
family:PeterA
family:Sophia

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.9: People that belong to the Sibling class

Query Result (5)

Limit results: 100 ▾

Sister	Brother
family:Madaline	
family:Sophia	
	family:Ange
	family:George
	family:PeterA

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.10: Showing people who belong to the Brother or Sister class

below we ask for: "Show me all the people that belong to the Sister class, all the people that belong to the Brother class, and for the latter show me their first names" (Figure 4.2.11).

```
SELECT ?sister ?brother ?firstName
WHERE { {?sister rdf:type family:Sister } UNION
{ ?brother rdf:type family:Brother .
```

```
?brother family:firstName ?firstName } }
```

Sister	Brother	FirstName
family:Madaline		
family:Sophia		
	family:Ange	"Ange"
	family:George	"George"

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.11: Show the first names for those that belong in the Brother class

4.2.2.5 Patterns on Named Graphs

An RDF dataset represents a collection of graphs, which in turn contains one default graph, and optional named graphs. The default graph does not have a name. IRIs (Internationalized Resource Identifiers) are used to identify the named graphs. Each FROM clause is accompanied by an IRI that points to a graph. A difference between URIs and IRIs is that URIs are US-ASCII encoded, whereas IRIs are UTF-8 encoded. An example of a query with named graphs is shown below.

```
SELECT ?var
FROM NAMED <urlOfGraphA>
FROM NAMED <urlOfGraphB>
WHERE {
  GRAPH <urlOfGraphA> { graph patterns from A }
  GRAPH <urlOfGraphB> { graph patterns from B }
}
```

If the queried dataset contains the graphs, we do not need to write the FROM NAMED clause. This happens only when we want to query the named graphs remotely.

4.2.3 Solution Sequence Modifiers

The solutions that are found in a result set are not ordered. These solutions form a sequence (collection) and this sequence can be modified according to the our needs and desires of the outcome. These modifiers are:

ORDER BY This modifier is used to put the solutions in order, according to a variable.

Projection A subset of the solution sequence is shown only.

DISTINCT Every solution in the sequence is unique with this modifier.

REDUCED Permits any solutions that are not unique to be eliminated.

OFFSET It can be declared where the solutions should start from.

LIMIT We limit the number of solutions shown.

These clauses will be further analyzed below.

4.2.3.1 ORDER BY

With the ORDER BY modifier, we can order the solutions in the solution sequence according to a variable. Optionally, we can include an order modifier (ASC() or DESC ()), which will indicate an ascending or descending order. The result set of the following query is shown in Figure 4.2.12.

```
SELECT ?name ?age
WHERE { ?name rdf:type family:Male .
       ?name family:hasAge ?age
}
```

This solution sequence is ordered automatically by the name, but not the age. If we were to desire ordering by age, we would do the following (results are shown in Figure 4.2.13):

Query Result (4)

Limit results: 100 ▼

Name	Age
family:Ange	"19"^^xsd:int
family:George	"26"^^xsd:int
family:Peter	"87"^^xsd:int
family:PeterD	"60"^^xsd:int

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.12: All the male individuals that have age

```

SELECT ?name ?age
WHERE { ?name rdf:type family:Male .
?name family:hasAge ?age }
ORDER BY ?age

```

Query Result (4)

Limit results: 100 ▼

Name	Age
family:Ange	"19"^^xsd:int
family:George	"26"^^xsd:int
family:PeterD	"60"^^xsd:int
family:Peter	"87"^^xsd:int

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.13: All the male individuals that have an age, ordered according to their age

Strings and literals are not the only RDF terms that are ordered by this modifier. URIs may be ordered as well, by comparing them as plain literals. Numeric values are ordered with the "<" operator. There is also an order between RDF terms. Unbound results are

sorted first, then blank nodes (blank nodes are not ordered among themselves), URIs, RDF literals (plain literals are sorted first, and typed literals follow).

4.2.3.2 Projection

Projection is possible by using SELECT and giving only the variables that we would like to see in the solution sequence (only the variables in SELECT will appear).

4.2.3.3 DISTINCT

It is possible that duplicate solutions may be present in the solution sequence, as for example in Figure 4.2.14.

```
SELECT ?grandparent
WHERE {
  ?grandparent family:hasChild ?parent .
  ?parent family:hasChild ?child .
}
```



Figure 4.2.14: Show possible grandparents in the dataset

The solution sequence contains non-unique results. This can be changed with the DISTINCT modifier. No duplicates will be included in the result set with this keyword (Figure 4.2.15).

```
SELECT DISTINCT ?grandparent
```

```
WHERE { ?grandparent family:hasChild ?parent .
?parent family:hasChild ?child .
}
```

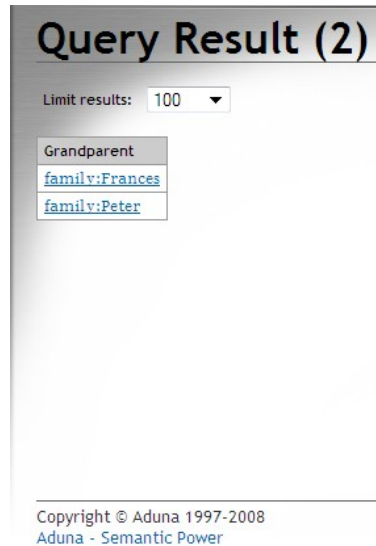


Figure 4.2.15: The results are significantly decreased now, all of them unique

4.2.3.4 REDUCED

With the REDUCED modifier, non-unique results are simply permitted to be eliminated. It cannot be predicted as to how many non-unique results will appear. The REDUCED modifier produces results that are between the results of DISTINCT, and results when using no such keyword at all. For example, in the family dataset the same mother appears more than once in the results. The result set of the following query is shown in Figure 4.2.16.

```
SELECT ?mother
WHERE { ?child family:hasMother ?mother .
}
```

Adding the REDUCED modifier to the query, we get the following results (Figure 4.2.17):

```
SELECT REDUCED ?mother
WHERE { ?child family:hasMother ?mother .
}
```


Query Result (6)

Limit results: 100 ▾

Mother
family: Sophia
family: Sophia
family: Madaline
family: Frances
family: Frances
family: Frances

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.16: All the possible mothers without any modifier

Query Result (3)

Limit results: 100 ▾

Mother
family: Sophia
family: Madaline
family: Frances

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.17: All the mothers with the effect of the REDUCED modifier

In this case we see that the effect is the same as we had applied DISTINCT. This is valid. It could have also been possible for the individual Frances to show up two or three times in the solution sequence.

4.2.3.5 LIMIT

The LIMIT clause limits how many solutions can be present in the results. For example if we have a large dataset, and want to see only a few results, or if we are interested only

in the top five, etc. If LIMIT is set to zero, then no results are returned. In the following query we ask for all the people in the dataset that are under 50 years old, and the results are shown in Figure 4.2.18.

```
SELECT ?person ?age
WHERE { ?person family:hasAge ?age
FILTER (?age < 50 )
}
ORDER BY ?age
```

Person	Age
family:Ange	"19"^^xsd:int
family:Sarah	"19"^^xsd:int
family:Vivi	"20"^^xsd:int
family:Vivian	"20"^^xsd:int
family:George	"26"^^xsd:int

Limit results: 100

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.18: People in the dataset under age 50

Applying the LIMIT clause, we can show only the first three results for example. In the following query we ask to see the three youngest people out of those who are under 50, therefore also the youngest in the whole dataset (Figure 4.2.19).

```
SELECT ?person ?age
WHERE { ?person family:hasAge ?age
FILTER (?age < 50 )
}
ORDER BY ?age
LIMIT 3
```

Person	Age
family:Anee	"19"^^xsd:int
family:Sarah	"19"^^xsd:int
family:Vivi	"20"^^xsd:int

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.19: The three youngest people in the dataset

4.2.3.6 OFFSET

The OFFSET clause prints only those solutions that appear after the specified number of solutions [33]. If OFFSET should be used with the modifier LIMIT to specify a certain subset of the solutions, then also ORDER BY should be used in order for it to be useful. Following the previous example and using OFFSET, we get as a result the two oldest individuals that are under 50 (Figure 4.2.20).

```
SELECT ?person ?age
WHERE { ?person family:hasAge ?age
FILTER (?age < 50 )
      }
ORDER BY ?age
LIMIT 3
OFFSET 3
```

4.2.4 Query Forms

SPARQL consists of query forms that form result sets or RDF graphs depending on the solutions. These query forms are SELECT, CONSTRUCT, ASK and DESCRIBE. The SPARQL Query Results XML Format can be used to serialize the results from a SELECT

Person	Age
family:Vivian	$20^{xsd:int}$
family:George	$26^{xsd:int}$

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.20: The two oldest individuals that are under 50

query or ASK query [42]. The SELECT query form returns a solution sequence of variable bindings to RDF terms. The CONSTRUCT query form returns a graph that we specify in the query. This is done by substituting the triple patterns with the solutions and adding all the triple patterns together (with UNION) to form a graph. The triples are not ordered, therefore it is possible to apply a solution modifier to order the resulting graph. The result of our following CONSTRUCT is shown in Figure 4.2.21.

```
CONSTRUCT { ?person family:hasAge ?age }
WHERE { ?person family:hasAge ?age
}
```

We can see from the previous example that we construct the statement (in the CONSTRUCT clause), and in the WHERE clause we provide the query pattern. In this example we just queried all the people that have an age. The result can be downloaded in any notation desired.

In the following example, we can see a more complex example where we construct a new object property for the family ontology. This way it is possible to create new data by combining the existing data, one of the goals of Semantic Web. In this example we create a new object property family:hasNiece that can be written in any notation (Figure 4.2.22).

```
CONSTRUCT { ?person family:hasNiece ?niece }
```

Query Result (9)

Download format:

Limit results:

Subject	Predicate	Object
family:Ange	family:hasAge	"19"^^xsd:int
family:Frances	family:hasAge	"79"^^xsd:int
family:George	family:hasAge	"26"^^xsd:int
family:Peter	family:hasAge	"87"^^xsd:int
family:PeterD	family:hasAge	"60"^^xsd:int
family:Sarah	family:hasAge	"19"^^xsd:int
family:Sophia	family:hasAge	"57"^^xsd:int
family:Vivi	family:hasAge	"20"^^xsd:int
family:Yvian	family:hasAge	"20"^^xsd:int

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.21: The CONSTRUCT result set

```
WHERE { ?person family:hasSibling ?parent.
        ?parent family:hasChild ?niece.
        ?niece rdf:type family:Female.
}
```

Query Result (3)

Download format:

Limit results:

Subject	Predicate	Object
family:Madaline	family:hasNiece	family:Georgia
family:PeterA	family:hasNiece	family:Georgia
family:Sophia	family:hasNiece	family:Georgia

Copyright © Aduna 1997-2008
Aduna - Semantic Power

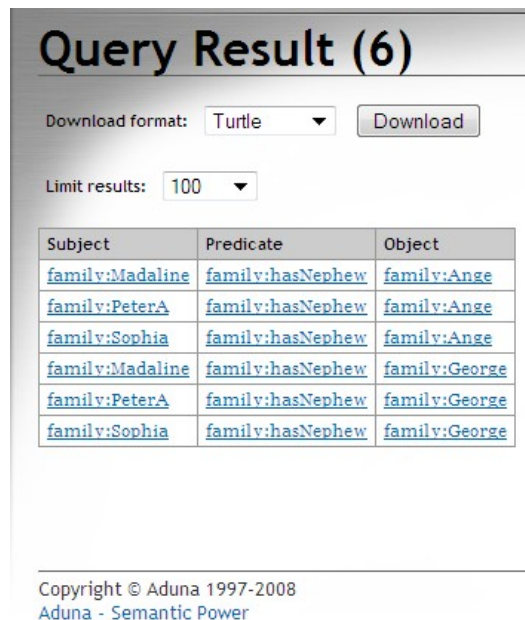
Figure 4.2.22: A new hasNiece property constructed

Even though the individual Madaline is a mother to the individual Georgia, the solution still shows up in the results because the `family:hasSibling` property is set to be both symmetric and transitive, and therefore it becomes a reflexive property.

Similar results can be observed for the `family:hasNephew` property, constructed in the example below (Figure 4.2.23).

```
CONSTRUCT { ?person family:hasNephew ?nephew }
```

```
WHERE { ?person family:hasSibling ?parent.
        ?parent family:hasChild ?nephew.
        ?nephew rdf:type family:Male.
      }
```



The screenshot shows a web interface titled "Query Result (6)". It includes a "Download format" dropdown set to "Turtle" and a "Download" button. Below that is a "Limit results" dropdown set to "100". The main content is a table with three columns: "Subject", "Predicate", and "Object". The table contains six rows of results, each with a subject (family:Madaline, family:PeterA, or family:Sophia), the predicate family:hasNephew, and an object (family:Ange or family:George). At the bottom, there is a copyright notice: "Copyright © Aduna 1997-2008 Aduna - Semantic Power".

Subject	Predicate	Object
family:Madaline	family:hasNephew	family:Ange
family:PeterA	family:hasNephew	family:Ange
family:Sophia	family:hasNephew	family:Ange
family:Madaline	family:hasNephew	family:George
family:PeterA	family:hasNephew	family:George
family:Sophia	family:hasNephew	family:George

Figure 4.2.23: The family:hasNephew property created by the CONSTRUCT clause

The ASK query form is used to answer a "yes" or "no" question about the query pattern. If the query pattern has a solution, the result will be "yes". For example, if we query any siblings for the individual Georgia, we get the following result, shown in Figure 4.2.24:

```
ASK { ?person family:hasSister family:Georgia }
```

Let us take an example that will return "yes" (Figure 4.2.25). In this example we query if the individual Madaline has any child.

```
ASK { family:Madaline family:hasChild ?x }
```

The answer given is simply "yes", we cannot know just from this simple query if the individual Madaline has one child or more. The DESCRIBE query form returns an RDF graph that contains data about the resource queried. The syntax "DESCRIBE * " is for querying information about all the variables available in a query. In the following example,

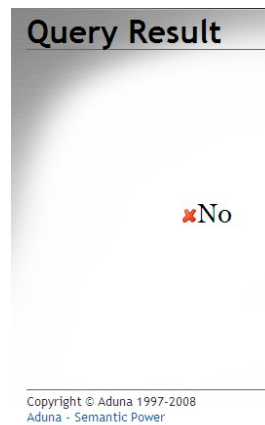


Figure 4.2.24: The ASK query form returns "no"

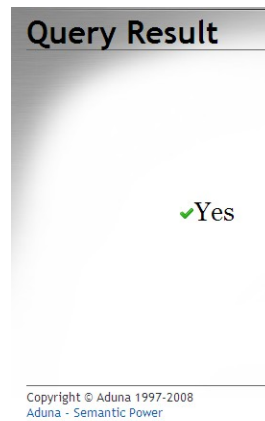


Figure 4.2.25: The ASK query form returns "yes"

we query all the information that is concerning the individual that has a first name "Ange" (Figure 4.2.26).

```
DESCRIBE ?x
WHERE { ?x family:firstName "Ange"}
```

All the object properties as well as the datatype properties that are in relation with this individual appear in the results. If instead of "DESCRIBE ?x " we had written "DESCRIBE * ", we would have gotten the same results because there is only one variable in the query.

Query Result (26)

Download format:

Limit results:

Subject	Predicate	Object
familv:Ange	rdf:type	familv:Human
familv:Ange	rdf:type	familv:Brother
familv:Ange	rdf:type	familv:Child
familv:Ange	rdf:type	familv:Male
familv:Ange	rdf:type	familv:Offspring
familv:Ange	rdf:type	familv:Sibling
familv:Ange	rdf:type	owl:Thing
familv:Ange	familv:hasAge	"19"^^xsd:int
familv:Ange	familv:firstName	"Ange"
familv:Ange	familv:hasSurname	"Kalan"
familv:Ange	familv:hasSibling	familv:Ange
familv:Ange	familv:hasFriend	familv:Ange
familv:Ange	familv:hasSibling	familv:George
familv:Ange	familv:hasBrother	familv:George
familv:Ange	familv:hasParent	familv:PeterD
familv:Ange	familv:hasFather	familv:PeterD
familv:Ange	familv:hasFriend	familv:Sarah
familv:Ange	familv:hasParent	familv:Sophia
familv:Ange	familv:hasMother	familv:Sophia
familv:Ange	familv:hasSibling	familv:Ange
familv:Ange	familv:hasFriend	familv:Ange
familv:George	familv:hasSibling	familv:Ange
familv:PeterD	familv:hasChild	familv:Ange
familv:Sarah	familv:hasFriend	familv:Ange
familv:Sophia	familv:hasChild	familv:Ange
_node15t8dccc3ax37	rdf:first	familv:Ange

Copyright © Aduna 1997-2008
Aduna - Semantic Power

Figure 4.2.26: Results of the DESCRIBE query form

4.3 Comparison with SQL and XQuery

As with any fairly new language, people seek to compare it to other older languages, especially when similarities appear prominent. SPARQL can be compared to SQL and XQuery, who perform a similar task but for completely different data. Below are some advantages and disadvantages of SPARQL [15].

One main advantage is that SPARQL is created purely for the purpose to query RDF data. SPARQL looks and behaves like RDF, and there is no need for mapping of the RDF data to perform a query. Furthermore, there is no need to specify relationships between data that is structured differently, because the relationships have a fixed size and all the data needed in a query is in one graph. Therefore, explicit join syntax is not required in SPARQL [24]. SPARQL supports querying semi-structured data, with unreliable structure. There is a possibility to query unknown relationships, and the OPTIONAL keyword can

be comparable to the SQL left joins. Also, in a single query it can be possible to query unrelated sources of data with SPARQL. This is possible because RDF represents data as binary relations, and therefore data can be easily and quickly mapped to RDF. Another benefit of SPARQL is that queries can be performed on multiple data sources that exist on the Web, without a need for one single document. This is possible because of the URI identification of named graphs. Named graphs with a default graph comprise an RDF dataset, which is queried with SPARQL.

A drawback for SPARQL is that it is considerably a young language, and therefore there are not many data storages that support direct querying. SQL for example has comparatively more. Furthermore, XQuery has the benefit of an explicit processing model, and SQL is backed up by many years of optimization research. However, with research, contributions and implementations, SPARQL is likely to improve. Also, SPARQL does not support an easy way to query transitive relations within a graph, or hierarchical structures, whereas XQuery's axes are much more powerful in comparison [15]. Nested queries are not supported yet by default in SPARQL [38]. Aggregate functions, e.g. COUNT, SUM, MIN, MAX, are also not supported by default in SPARQL, but there are several implementations created by developers that extend the features [43]. GROUP BY is also not currently supported by default, as well as INTERSECT. However, the W3C SPARQL Working Group is continuing to update and design features for SPARQL, in order to create more possibilities for queries as in SQL [38].

4.4 Epilogue

SPARQL is a popular query language for RDF, a recursive acronym. On the surface it looks like SQL, but their philosophies are different. A SPARQL query may contain prefix declarations, result clause, possibly dataset definition, query pattern, and optionally query modifiers. Queries are written and shown in Sesame with appropriate screenshots. At the end a small comparison is made between SPARQL and SQL.

Conclusion

The Semantic Web vision comprises of distributed, reusable knowledge in a decentralized world. Furthermore, the data must be unambiguous and context-independent. To achieve this, we need to represent the data using RDF, reference the resources using URIs, and describe abstract concepts with ontologies. RDF aims to solve the problem of ambiguity for complex language structures by using triples to express facts. These triples can be written in many different formats, equivalent to each other. Some are human-readable, e.g. N3 and Turtle, and others are designed for easy serialization between applications, e.g. RDF/XML. URIs provide global identification of resources. Ontologies are needed to describe concepts, class hierarchies, etc. They give more information about the data and about how it can be used. RDF Schema is a language that is used to create ontologies, but it is simple. OWL builds on RDF Schema and provides stronger descriptions and constraints to the ontology. Tools exist to accommodate the creation of ontologies, one such tool being Protégé. Ontologies with their data can be stored into data storages. A storage used and demonstrated for this thesis was Sesame. RDF data can be queried with the SPARQL query language for RDF. This query language at a glance looks like SQL, but the queries look and behave like RDF. New data can also be constructed using the CONSTRUCT query form of SPARQL, and then serialized. "Intelligent" applications can accept queries as input and provide the resulted information as output. If developers continue to implement the Semantic Web vision and build upon optimizing its technologies, then the Web in the future will be much more accommodating, easy to use, and intuitive.

Bibliography

- [1] Jena - a semantic web framework for java. URL <http://jena.sourceforge.net/>.
- [2] Protege home page. URL <http://protege.stanford.edu/>.
- [3] Openrdf home page. URL <http://www.openrdf.org/>.
- [4] Rdf graph format converter. URL <http://sswap.info/format-converter.jsp>.
- [5] Sparql tutorial - data formats. URL <http://jena.sourceforge.net/ARQ/Tutorial/data.html>.
- [6] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, second edition, 2008.
- [7] Dave Beckett. Turtle - terse rdf triple language, December 2006. URL <http://www.dajobe.org/2004/01/turtle/2006-12-04/>.
- [8] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language: W3c team submission 28 march 2011. W3c team submission, W3C, March 2011. URL <http://www.w3.org/TeamSubmission/turtle/>.
- [9] Tim Berners-Lee. Notation 3, . URL <http://www.w3.org/DesignIssues/Notation3>.
- [10] Tim Berners-Lee. World wide web consortium architecture, . URL <http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html>.
- [11] Tim Berners-Lee. What the semantic web can represent, September 1998. URL <http://www.w3.org/DesignIssues/RDFnot.html>.
- [12] Tim Berners-Lee. Why rdf model is different from the xml model, September 1998. URL <http://www.w3.org/DesignIssues/RDF-XML.html>.

- [13] Tim Berners-Lee, Nigel Shadbolt, and Wendy Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006. URL http://eprints.ecs.soton.ac.uk/12614/1/Semantic_Web_Revisted.pdf.
- [14] Nick Drummond and Timothy Redmond. Annotation template view, March 2010. URL http://protegewiki.stanford.edu/wiki/Annotation_Template_View.
- [15] Lee Feigenbaum. Sparql faq, February 2008. URL <http://www.thefigtrees.net/lee/sw/sparql-faq>.
- [16] Lee Feigenbaum and Eric Prud'hommeaux. Sparql by example: A tutorial. URL <http://www.cambridgesemantics.com/2008/09/sparql-by-example/>.
- [17] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. URL <http://tomgruber.org/writing/ontolingua-kaj-1993.pdf>.
- [18] John Hebel, Matthew Fisher, Ryan Blace, and Andrew Perez-Lopez. *Semantic Web Programming*. Wiley Publishing, Inc., 2009.
- [19] Jim Hendler. Frequently asked questions on w3c's web ontology language (owl), May 2008. URL <http://www.w3.org/2003/08/owlfaq.html.en>.
- [20] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 2003. URL <http://www.comlab.ox.ac.uk/people/ian.horrocks/Publications/download/2003/HoPH03a.pdf>.
- [21] Artem Katasonov. Storing and querying rdf data, 2009. URL <http://users.jyu.fi/~akataso/itks544/Lecture1.pdf>.
- [22] Petr Kremen and Bogdan Kostov. Owl2query, March 2011. URL <http://protegewiki.stanford.edu/wiki/OWL2Query>.
- [23] Lee W. Lacy. *OWL: Representing Information Using the Web Ontology Language*. Trafford Publishing, 2005.
- [24] Jim Melton. Sql, xquery, and sparql: What's wrong with this picture?, March 2006. URL <http://www.w3.org/2006/Talks/0301-melton-query-langs.pdf>.

- [25] Peter V. Mikhaleiko. The benefits of the web ontology language in web applications, August 2003. URL <http://www.techrepublic.com/article/the-benefits-of-the-web-ontology-language-in-web-applications/5060266>.
- [26] Jeffrey T. Pollock. *Semantic Web for Dummies*. Wiley Publishing, Inc., March 2009.
- [27] Protege Wiki. Protege 4.0 features, June 2009. URL <http://protegewiki.stanford.edu/wiki/Protege4Features>.
- [28] RDF Core Working Group. Resource description framework (rdf) model and syntax specification: W3c recommendation 22 february 1999. W3C recommendation, W3C, February 1999. URL <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>.
- [29] RDF Core Working Group. Rdf primer: W3c recommendation 10 february 2004. W3C recommendation, W3C, February 2004. URL <http://www.w3.org/TR/rdf-primer/>.
- [30] RDF Core Working Group. Rdf semantics: W3c recommendation 10 february 2004. W3C recommendation, W3C, February 2004. URL <http://www.w3.org/TR/2004/REC-rdf-mt-20040210/>.
- [31] RDF Core Working Group. Rdf vocabulary description language 1.0 rdf schema: W3c recommendation 10 february 2004. W3c recommendation, W3C, February 2004. URL <http://www.w3.org/TR/rdf-schema/>.
- [32] RDF Core Working Group. Rdf test cases: W3c recommendation 10 february 2004. W3C recommendation, W3C, February 2004. URL <http://www.w3.org/TR/rdf-testcases/>.
- [33] RDF Data Access Working Group. Sparql query language for rdf: W3c recommendation 15 january 2008. W3c recommendation, W3C, January 2008. URL <http://www.w3.org/TR/rdf-sparql-query/>.
- [34] RDFcore Working Group. Resource description framework (rdf) concepts and abstract syntax: W3c recommendation 10 february 2004. W3c recommendation, W3C, February 2004. URL <http://www.w3.org/TR/rdf-concepts/>.
- [35] Eric Prud'hommeaux Sandro Hawke, Ivan Herman and Ralph Swick. W3c semantic web activity. URL <http://www.w3.org/2001/sw/>.

- [36] Semantic Web Best Practices and Deployment Working Group. Xml schema datatypes in rdf and owl: W3c working group note 14 march 2006. W3c working group note, W3C, March 2006. URL <http://www.w3.org/TR/swbp-xsch-datatypes/>.
- [37] Semantic Web Interest Group. Rdf primer - turtle version: W3c note in development. W3c working group note, W3C, 2007. URL <http://www.w3.org/2007/02/turtle/primer/>.
- [38] SPARQL Working Group. Sparql new features and rationale: W3c working draft 2 july 2009. W3c working draft, W3C, July 2009. URL <http://www.w3.org/TR/sparql-features/>.
- [39] Joshua Tauberer. Quick intro to rdf. URL <http://www.rdfabout.com/quickintro.xpd>.
- [40] Joshua Tauberer. rdf:about, 2005. URL <http://www.rdfabout.com/intro/?section=1>.
- [41] W3C. Semantic web activity news, January 2008. URL http://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation.
- [42] W3C RDF Data Access Working Group. Sparql query results xml format: W3c recommendation 15 january 2008. W3C recommendation, W3C, January 2008 2008. URL <http://www.w3.org/TR/rdf-sparql-XMLres/>.
- [43] W3C Wiki. Sparql/extensions/aggregates, September 2008. URL <http://www.w3.org/wiki/SPARQL/Extensions/Aggregates>.
- [44] Web Ontology Working Group. Owl web ontology language overview: W3c recommendation 10 february 2004. W3c recommendation, W3C, February 2004. URL <http://www.w3.org/TR/owl-features/>.
- [45] Web Ontology Working Group. Owl web ontology language guide: W3c recommendation 10 february 2004. W3c recommendation, W3C, February 2004. URL <http://www.w3.org/TR/owl-guide/>.
- [46] Web Ontology Working Group. Owl web ontology language reference: W3c recommendation 10 february 2004. Technical report, W3C, February 2004. URL <http://www.w3.org/TR/owl-ref/>.

- [47] Web Ontology Working Group. Owl web ontology language use cases and requirements: W3c recommendation 10 february 2004. W3C recommendation, W3C, 2004. URL <http://www.w3.org/TR/webont-req/>.
- [48] Wikipedia, the free encyclopedia. Backward chaining, . URL http://en.wikipedia.org/wiki/Backward_chaining.
- [49] Wikipedia, the free encyclopedia. Daml+oil, . URL <http://en.wikipedia.org/wiki/DAML%2B0IL>.
- [50] Wikipedia, the free encyclopedia. Forward chaining, . URL http://en.wikipedia.org/wiki/Forward_chaining.
- [51] Wikipedia, the free encyclopedia. N-triples, . URL <http://en.wikipedia.org/wiki/N-Triples>.
- [52] Wikipedia, the free encyclopedia. Web ontology language, . URL http://en.wikipedia.org/wiki/Web_Ontology_Language.
- [53] Wikipedia, the free encyclopedia. Resource description framework, . URL http://en.wikipedia.org/wiki/Resource_Description_Framework.
- [54] Wikipedia, the free encyclopedia. Semantic reasoner, . URL http://en.wikipedia.org/wiki/Semantic_reasoner.
- [55] Wikipedia, the free encyclopedia. Uniform resource identifier, . URL http://en.wikipedia.org/wiki/Uniform_Resource_Identifier.

Appendices

Appendix A: Family ontology

This appendix contains the whole RDF dataset developed for this thesis. Fragments of the code were used in examples and for the demonstration of OWL, SPARQL, Protégé, etc. The data is written in Turtle and annotates the family ontology.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .
@prefix family: <http://www.example.com/family.owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://www.example.com/ontology/family-inferred.owl> a owl:Ontology ;
dc:creator "Fotini Bogiatzi"^^xsd:string ;
dc:description "A basic ontology on the concept of family" .

dc:description a owl:AnnotationProperty .

dc:creator a owl:AnnotationProperty .

family:hasBrother a owl:ObjectProperty ;
rdfs:range family:Brother ;
rdfs:domain family:Sibling ;
rdfs:subPropertyOf family:hasSibling .
```

```
family:hasChild a owl:ObjectProperty ;  
rdfs:range family:Child ;  
rdfs:domain family:Parent ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
family:hasFather a owl:ObjectProperty , owl:FunctionalProperty ;  
rdfs:domain family:Child ;  
rdfs:range family:Father ;  
rdfs:subPropertyOf family:hasParent .
```

```
family:hasFriend a owl:ObjectProperty , owl:SymmetricProperty  
    , owl:TransitiveProperty ;  
rdfs:domain family:Human ;  
rdfs:range family:Human ;  
owl:inverseOf family:hasFriend ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
family:hasHusband a owl:ObjectProperty , owl:AsymmetricProperty  
    , owl:FunctionalProperty , owl:InverseFunctionalProperty  
    , owl:IrreflexiveProperty ;  
rdfs:range family:Husband ;  
rdfs:domain family:Wife ;  
rdfs:subPropertyOf family:hasSpouse .
```

```
family:hasMother a owl:ObjectProperty , owl:FunctionalProperty ;  
rdfs:domain family:Child ;  
rdfs:range family:Mother ;  
rdfs:subPropertyOf family:hasParent .
```

```
family:hasParent a owl:ObjectProperty ;  
rdfs:domain family:Child ;  
rdfs:range family:Parent ;
```

```
owl:inverseOf family:hasChild ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
family:hasSibling a owl:ObjectProperty , owl:SymmetricProperty  
    , owl:TransitiveProperty ;  
rdfs:range family:Sibling ;  
rdfs:domain family:Sibling ;  
owl:inverseOf family:hasSibling ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
family:hasSister a owl:ObjectProperty ;  
rdfs:domain family:Sibling ;  
rdfs:range family:Sister ;  
rdfs:subPropertyOf family:hasSibling .
```

```
family:hasSpouse a owl:ObjectProperty , owl:FunctionalProperty  
    , owl:InverseFunctionalProperty , owl:SymmetricProperty ;  
rdfs:range family:Parent ;  
rdfs:domain family:Parent ;  
owl:inverseOf family:hasSpouse ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
family:hasWife a owl:ObjectProperty , owl:AsymmetricProperty  
    , owl:FunctionalProperty , owl:InverseFunctionalProperty  
    , owl:IrreflexiveProperty ;  
rdfs:domain family:Husband ;  
rdfs:range family:Wife ;  
owl:inverseOf family:hasHusband ;  
rdfs:subPropertyOf family:hasSpouse .
```

```
owl:topObjectProperty a owl:ObjectProperty .
```

```
family:firstName a owl:DatatypeProperty ;  
rdfs:domain family:Human ;  
rdfs:range xsd:string ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
family:hasAge a owl:DatatypeProperty ;  
rdfs:domain family:Human ;  
rdfs:range xsd:integer ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
family:hasSurname a owl:DatatypeProperty ;  
rdfs:domain family:Human ;  
rdfs:range xsd:string ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
owl:topDataProperty a owl:DatatypeProperty .
```

```
family:Brother a owl:Class ;  
rdfs:subClassOf family:Male ;  
owl:disjointWith family:Female , family:Mother  
    , family:Sister , family:Wife .
```

```
family:Child a owl:Class ;  
owl:equivalentClass family:Offspring ;  
rdfs:subClassOf family:Human , _:node15t8dcc3ax22 .
```

```
_:node15t8dcc3ax22 a owl:Restriction ;  
owl:onProperty family:hasParent ;  
owl:someValuesFrom family:Parent .
```

```
family:Child rdfs:subClassOf _:node15t8dcc3ax23 .
```

```
_ :node15t8dcc3ax23 a owl:Restriction ;
owl:onProperty family:hasFather ;
owl:someValuesFrom family:Father .

family:Child rdfs:subClassOf _ :node15t8dcc3ax24 .

_ :node15t8dcc3ax24 a owl:Restriction ;
owl:onProperty family:hasFather ;
owl:onClass family:Father ;
owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger .

family:Child rdfs:subClassOf _ :node15t8dcc3ax25 .

_ :node15t8dcc3ax25 a owl:Restriction ;
owl:onProperty family:hasMother ;
owl:someValuesFrom family:Mother .

family:Child rdfs:subClassOf _ :node15t8dcc3ax26 .

_ :node15t8dcc3ax26 a owl:Restriction ;
owl:onProperty family:hasParent ;
owl:onClass family:Parent ;
owl:maxQualifiedCardinality "2"^^xsd:nonNegativeInteger .

family:Child rdfs:subClassOf _ :node15t8dcc3ax27 .

_ :node15t8dcc3ax27 a owl:Restriction ;
owl:onProperty family:hasMother ;
owl:onClass family:Mother ;
owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger .

family:Father a owl:Class ;
```

```
rdfs:subClassOf family:Male ;  
owl:disjointWith family:Female , family:Mother , family:Sister  
    , family:Wife .
```

```
family:Female a owl:Class ;  
rdfs:subClassOf family:Human ;  
owl:disjointWith family:Husband , family:Male .
```

```
family:Human a owl:Class ;  
rdfs:subClassOf owl:Thing .
```

```
family:Husband a owl:Class ;  
rdfs:subClassOf family:Male ;  
owl:disjointWith family:Mother , family:Sister , family:Wife .
```

```
family:Male a owl:Class ;  
rdfs:subClassOf family:Human ;  
owl:disjointWith family:Mother , family:Sister , family:Wife .
```

```
family:Mother a owl:Class ;  
rdfs:subClassOf family:Female .
```

```
family:Offspring a owl:Class ;  
rdfs:subClassOf family:Human .
```

```
family:Parent a owl:Class ;  
rdfs:subClassOf family:Human , _:node15t8dcc3ax28 .
```

```
_:node15t8dcc3ax28 a owl:Restriction ;  
owl:onProperty family:hasChild ;  
owl:someValuesFrom family:Child .
```

```
family:Sibling a owl:Class ;
rdfs:subClassOf family:Human , _:node15t8dcc3ax29 .

_:node15t8dcc3ax29 a owl:Restriction ;
owl:onProperty family:hasSibling ;
owl:someValuesFrom family:Sibling .

family:Sister a owl:Class ;
rdfs:subClassOf family:Female .

family:Wife a owl:Class ;
rdfs:subClassOf family:Female .

owl:Thing a owl:Class .

family:Ange a family:Human , family:Brother , family:Child , family:Male
, family:Offspring , family:Sibling , owl:Thing ;
family:hasAge "19"^^xsd:int ;
family:firstName "Ange" ;
family:hasSurname "Kalan" ;
family:hasSibling family:Ange ;
family:hasFriend family:Ange ;
family:hasSibling family:George ;
family:hasBrother family:George ;
family:hasParent family:PeterD ;
family:hasFather family:PeterD ;
family:hasFriend family:Sarah ;
family:hasParent family:Sophia ;
family:hasMother family:Sophia .

family:Frances a family:Mother , family:Female , family:Human
, family:Parent, family:Wife , owl:Thing ;
```



```

family:hasAge "79"^^xsd:int ;
family:hasSurname "Smith" ;
family:firstName "Frances" ;
family:hasChild family:Madaline ;
family:hasHusband family:Peter ;
family:hasSpouse family:Peter ;
family:hasChild family:PeterA , family:Sophia .

```

```

family:George a family:Sibling , family:Brother , family:Child
    , family:Human , family:Male , family:Offspring , owl:Thing ;
family:hasAge "26"^^xsd:int ;
family:firstName "George" ;
family:hasSurname "Kalan" ;
family:hasSibling family:Ange , family:George ;
family:hasParent family:PeterD ;
family:hasFather family:PeterD ;
family:hasParent family:Sophia ;
family:hasMother family:Sophia .

```

```

family:Georgia a owl:Thing , family:Child , family:Female
    , family:Human , family:Offspring ;
family:firstName "Georgia"^^xsd:string ;
family:hasFriend family:Georgia ;
family:hasParent family:Madaline ;
family:hasMother family:Madaline ;
family:hasFriend family:Theo , family:Vivi , family:Vivian .

```

```

family:Madaline a family:Female , family:Child , family:Human
    , family:Mother , family:Offspring , family:Parent
    , family:Sibling , family:Sister , owl:Thing ;
family:firstName "Georgia"^^xsd:string ;
family:hasMother family:Frances ;

```

family:hasParent family:Frances ;
family:hasChild family:Georgia ;
family:hasSibling family:Madaline ;
family:hasFather family:Peter ;
family:hasParent family:Peter ;
family:hasBrother family:PeterA ;
family:hasSibling family:PeterA , family:Sophia ;
family:hasSister family:Sophia .

family:Peter a family:Human , family:Father , family:Husband
 , family:Male , family:Parent , owl:Thing ;
family:hasAge "87"^^xsd:int ;
family:hasSurname "Smith" ;
family:firstName "Peter" ;
family:hasWife family:Frances ;
family:hasSpouse family:Frances ;
family:hasChild family:Madaline , family:Sophia .

family:PeterA a family:Male , family:Brother , family:Child
 , family:Human , family:Offspring , family:Sibling , owl:Thing ;
family:hasMother family:Frances ;
family:hasParent family:Frances ;
family:hasSister family:Madaline ;
family:hasSibling family:Madaline , family:PeterA , family:Sophia .

family:PeterD a family:Male , family:Father , family:Human , family:Husband
 , family:Parent , owl:Thing ;
family:hasAge "60"^^xsd:int ;
family:firstName "Peter" ;
family:hasSurname "Kalan" ;
family:hasChild family:Ange , family:George ;
family:hasSpouse family:Sophia ;

family:hasWife family: Sophia .

family: Sarah a family: Human , family: Female , owl: Thing ;
 family: hasAge "19"^^xsd: int ;
 family: hasSurname "Connor"^^xsd: string ;
 family: firstName "Sarah"^^xsd: string ;
 family: hasFriend family: Ange , family: Sarah .

family: Sophia a family: Sibling , family: Child , family: Female
 , family: Human , family: Mother , family: Offspring
 , family: Parent , family: Sister , family: Wife , owl: Thing ;
 family: hasAge "57"^^xsd: int ;
 family: firstName "Sophia" ;
 family: hasSurname "Kalan" ;
 family: hasChild family: Ange ;
 family: hasMother family: Frances ;
 family: hasParent family: Frances ;
 family: hasChild family: George ;
 family: hasSibling family: Madaline ;
 family: hasParent family: Peter ;
 family: hasFather family: Peter ;
 family: hasSibling family: PeterA ;
 family: hasSpouse family: PeterD ;
 family: hasHusband family: PeterD ;
 family: hasSibling family: Sophia .

family: Theo a family: Human , family: Male , owl: Thing ;
 family: firstName "Theo"^^xsd: string ;
 family: hasFriend family: Georgia , family: Theo
 , family: Vivi , family: Vivian .

family: Vivi a owl: Thing , family: Female , family: Human ;

```
family:hasAge "20"^^xsd:int ;
family:firstName "Vivian"^^xsd:string ;
family:hasFriend family:Georgia , family:Theo
    , family:Vivi , family:Vivian .

family:Vivian a owl:Thing , family:Female , family:Human ;
family:hasAge "20"^^xsd:int ;
family:firstName "Vivian"^^xsd:string ;
family:hasFriend family:Georgia , family:Theo ;
owl:sameAs family:Vivi ;
family:hasFriend family:Vivi , family:Vivian .

_:node15t8dcc3ax30 a owl:AllDifferent ;
owl:distinctMembers _:node15t8dcc3ax31 .

_:node15t8dcc3ax31 rdf:first family:Sarah ;
rdf:rest _:node15t8dcc3ax32 .

_:node15t8dcc3ax32 rdf:first family:Sophia ;
rdf:rest _:node15t8dcc3ax33 .

_:node15t8dcc3ax33 rdf:first family:Frances ;
rdf:rest _:node15t8dcc3ax34 .

_:node15t8dcc3ax34 rdf:first family:PeterA ;
rdf:rest _:node15t8dcc3ax35 .

_:node15t8dcc3ax35 rdf:first family:Georgia ;
rdf:rest _:node15t8dcc3ax36 .

_:node15t8dcc3ax36 rdf:first family:Madaline ;
rdf:rest _:node15t8dcc3ax37 .
```

```
_:node15t8dcc3ax37 rdf:first family:Ange ;  
rdf:rest _:node15t8dcc3ax38 .
```

```
_:node15t8dcc3ax38 rdf:first family:George ;  
rdf:rest _:node15t8dcc3ax39 .
```

```
_:node15t8dcc3ax39 rdf:first family:Vivian ;  
rdf:rest _:node15t8dcc3ax40 .
```

```
_:node15t8dcc3ax40 rdf:first family:Theo ;  
rdf:rest _:node15t8dcc3ax41 .
```

```
_:node15t8dcc3ax41 rdf:first family:PeterD ;  
rdf:rest _:node15t8dcc3ax42 .
```

```
_:node15t8dcc3ax42 rdf:first family:Peter ;  
rdf:rest rdf:nil .
```

Appendix B: Olympic Games ontology in RDF Schema

This appendix presents the code for the ontology created on Olympic Games. It is written in RDF Schema using the N3 notation.

```
@prefix :          <http://www.example.com/olympics.owl#> .
@prefix rdfs:      <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:       <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

```
:Game
    a      rdfs:Class .
```

```
:Participation
    a      rdfs:Class .
```

```
:Human
    a      rdfs:Class .
```

```
:FemaleSport
    a      rdfs:Class ;
    rdfs:subClassOf :Sport .
```

```
:MaleSport
    a      rdfs:Class ;
    rdfs:subClassOf :Sport .
```

```
:Sport
    a      rdfs:Class .
```

```
:Stadium
    a      rdfs:Class .
```

:Athlete

```
a      rdfs:Class ;  
rdfs:subClassOf :Human .
```

:Trainer

```
a      rdfs:Class ;  
rdfs:subClassOf :Human .
```

:participationOfAthlete

```
a      rdf:Property ;  
rdfs:domain :Participation ;  
rdfs:range :Athlete .
```

:hasQualification

```
a      rdf:Property ;  
rdfs:domain :Participation ;  
rdfs:range rdfs:Literal .
```

:hasBirthdate

```
a      rdf:Property ;  
rdfs:domain :Human ;  
rdfs:range rdfs:Literal .
```

:hasGender

```
a      rdf:Property ;  
rdfs:domain :Human ;  
rdfs:range rdfs:Literal .
```

:hasSpectatorCapacity

```
a      rdf:Property ;  
rdfs:domain :Stadium ;
```

```
    rdfs:range rdfs:Literal .
```

```
:participatesIn
```

```
    a      rdf:Property ;  
    rdfs:domain :Athlete ;  
    rdfs:range :Participation .
```

```
:trained_by
```

```
    a      rdf:Property ;  
    rdfs:domain :Athlete ;  
    rdfs:range :Trainer .
```

```
:originCountry
```

```
    a      rdf:Property ;  
    rdfs:domain :Human ;  
    rdfs:range rdfs:Literal .
```

```
:hasLevel
```

```
    a      rdf:Property ;  
    rdfs:domain :Game ;  
    rdfs:range rdfs:Literal .
```

```
:hasScore
```

```
    a      rdf:Property ;  
    rdfs:domain :Participation ;  
    rdfs:range rdfs:Literal .
```

```
:hasFirstName
```

```
    a      rdf:Property ;  
    rdfs:domain :Human ;  
    rdfs:range rdfs:Literal .
```


:playedIn

```
a      rdf:Property ;
rdfs:domain :Game ;
rdfs:range :Stadium .
```

:hasStadiumName

```
a      rdf:Property ;
rdfs:domain :Stadium ;
rdfs:range rdfs:Literal .
```

:consistsOfParticipation

```
a      rdf:Property ;
rdfs:domain :Game ;
rdfs:range :Participation .
```

:inCity

```
a      rdf:Property ;
rdfs:domain :Stadium ;
rdfs:range rdfs:Literal .
```

:inGame

```
a      rdf:Property ;
rdfs:domain :Participation ;
rdfs:range :Game .
```

:hasSurname

```
a      rdf:Property ;
rdfs:domain :Human ;
rdfs:range rdfs:Literal .
```

```
:trains
    a      rdf:Property ;
    rdfs:domain :Trainer ;
    rdfs:range :Athlete .
```

Appendix C: Olympic Games ontology in OWL

This appendix presents the code for the ontology created on Olympic Games. It is written in OWL using the Turtle notation.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .
@prefix olympics: <http://www.example.com/olympics.owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://www.example.com/ontology/olympic-games-inferred.owl> a owl:Ontology ;
rdfs:comment "An example ontology based on a subset of Olympic Games."@en .

olympics:aboutSport a owl:ObjectProperty ;
rdfs:domain olympics:Game ;
rdfs:range olympics:Sport ;
rdfs:subPropertyOf owl:topObjectProperty .

olympics:consistsOfParticipation a owl:ObjectProperty ;
rdfs:domain olympics:Game ;
rdfs:range olympics:Participation ;
rdfs:subPropertyOf owl:topObjectProperty .

olympics:inGame a owl:ObjectProperty ;
rdfs:range olympics:Game ;
rdfs:domain olympics:Participation ;
owl:inverseOf olympics:consistsOfParticipation ;
rdfs:subPropertyOf owl:topObjectProperty .

olympics:participatesIn a owl:ObjectProperty ;
rdfs:domain olympics:Athlete ;
```

```
rdfs:range olympics:Participation ;  
owl:inverseOf olympics:participationOfAthlete ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
olympics:participationOfAthlete a owl:ObjectProperty ;  
rdfs:range olympics:Athlete ;  
rdfs:domain olympics:Participation ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
olympics:playedIn a owl:ObjectProperty ;  
rdfs:domain olympics:Game ;  
rdfs:range olympics:Stadium ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
olympics:trained_by a owl:ObjectProperty ;  
rdfs:domain olympics:Athlete ;  
rdfs:range olympics:Trainer ;  
owl:inverseOf olympics:trains ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
olympics:trains a owl:ObjectProperty ;  
rdfs:range olympics:Athlete ;  
rdfs:domain olympics:Trainer ;  
rdfs:subPropertyOf owl:topObjectProperty .
```

```
owl:topObjectProperty a owl:ObjectProperty .
```

```
olympics:hasBirthdate a owl:DatatypeProperty ;  
rdfs:domain olympics:Human ;  
rdfs:range xsd:date ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
olympics:hasFirstName a owl:DatatypeProperty ;  
rdfs:domain olympics:Human ;  
rdfs:range xsd:string ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
olympics:hasGender a owl:DatatypeProperty ;  
rdfs:domain olympics:Human ;  
rdfs:range xsd:string ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
olympics:hasLevel a owl:DatatypeProperty ;  
rdfs:domain olympics:Game ;  
rdfs:range xsd:string ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
olympics:hasQualification a owl:DatatypeProperty ;  
rdfs:domain olympics:Participation ;  
rdfs:range xsd:string ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
olympics:hasScore a owl:DatatypeProperty , owl:FunctionalProperty ;  
rdfs:domain olympics:Participation ;  
rdfs:range xsd:float ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
olympics:hasSpectatorCapacity a owl:DatatypeProperty ;  
rdfs:domain olympics:Stadium ;  
rdfs:range xsd:int ;  
rdfs:subPropertyOf owl:topDataProperty .
```

```
olympics:hasStadiumName a owl:DatatypeProperty ;  
rdfs:domain olympics:Stadium ;
```

```
rdfs:range xsd:string ;
rdfs:subPropertyOf owl:topDataProperty .

olympics:hasSurname a owl:DatatypeProperty ;
rdfs:domain olympics:Human ;
rdfs:range xsd:string ;
rdfs:subPropertyOf owl:topDataProperty .

olympics:inCity a owl:DatatypeProperty ;
rdfs:domain olympics:Stadium ;
rdfs:range xsd:string ;
rdfs:subPropertyOf owl:topDataProperty .

olympics:originCountry a owl:DatatypeProperty ;
rdfs:domain olympics:Human ;
rdfs:range xsd:string ;
rdfs:subPropertyOf owl:topDataProperty .

olympics:sportName a owl:DatatypeProperty ;
rdfs:domain olympics:Sport ;
rdfs:range xsd:string ;
rdfs:subPropertyOf owl:topDataProperty .

owl:topDataProperty a owl:DatatypeProperty .

olympics:Athlete a owl:Class ;
rdfs:subClassOf olympics:Human .

olympics:FemaleSport a owl:Class ;
rdfs:subClassOf olympics:Sport .

olympics:Game a owl:Class ;
```

```
rdfs:subClassOf owl:Thing .

olympics:Human a owl:Class ;
rdfs:subClassOf owl:Thing .

olympics:MaleSport a owl:Class ;
rdfs:subClassOf olympics:Sport .

olympics:Participation a owl:Class ;
rdfs:subClassOf owl:Thing .

olympics:Sport a owl:Class ;
rdfs:subClassOf owl:Thing .

olympics:Stadium a owl:Class ;
rdfs:subClassOf owl:Thing .

olympics:Trainer a owl:Class ;
rdfs:subClassOf olympics:Human .

owl:Thing a owl:Class .

olympics:AthleteJan_Zelezny a owl:Thing , olympics:Athlete , olympics:Human ;
olympics:hasBirthdate "1966-07-23"^^xsd:date ;
olympics:originCountry "Czechoslovakia"^^xsd:string ;
olympics:hasFirstName "Jan"^^xsd:string ;
olympics:hasSurname "Zelezny"^^xsd:string ;
olympics:participatesIn olympics:Participation_Zelezny_Olympics ;
olympics:trained_by olympics:TrainerAlexander_Makarov .

olympics:AthleteSavva_Lika a owl:Thing , olympics:Athlete , olympics:Human
, olympics:Participation ;
```

olympics:hasBirthdate "1970-06-27"^^xsd:date ;
olympics:hasGender "Female"^^xsd:string ;
olympics:originCountry "Greece"^^xsd:string ;
olympics:hasSurname "Lika"^^xsd:string ;
olympics:hasFirstName "Savva"^^xsd:string ;
olympics:inGame olympics:GameNationals2009 ;
olympics:participatesIn olympics:Participation_Lika_Nationals ;
olympics:trained_by olympics:TrainerYannis_Peristeris .

olympics:GameNationals2009 a owl:Thing , olympics:Game ;
olympics:hasLevel "Semi-Final"^^xsd:string ;
olympics:consistsOfParticipation olympics:AthleteSavva_Lika
 , olympics:Participation_Lika_Nationals ;
olympics:aboutSport olympics:SportFemaleJavelin ;
olympics:playedIn olympics:StadiumKaftanzoglio .

olympics:GameOlympics2004 a owl:Thing , olympics:Game ;
olympics:hasLevel "Final"^^xsd:string ;
olympics:consistsOfParticipation olympics:Participation_Zelezny_Olympics ;
olympics:aboutSport olympics:SportMaleJavelin ;
olympics:playedIn olympics:StadiumKaftanzoglio .

olympics:Participation_Lika_Nationals a olympics:Participation , owl:Thing ;
olympics:hasScore "58.44"^^xsd:float ;
olympics:hasQualification "Q"^^xsd:string ;
olympics:participationOfAthlete olympics:AthleteSavva_Lika ;
olympics:inGame olympics:GameNationals2009 .

olympics:Participation_Zelezny_Olympics a olympics:Participation , owl:Thing ;
olympics:hasScore "90.17"^^xsd:float ;
olympics:hasQualification "Q"^^xsd:string ;
olympics:participationOfAthlete olympics:AthleteJan_Zelezny ;

olympics:inGame olympics:GameOlympics2004 .

olympics:SportFemaleJavelin a owl:Thing , olympics:FemaleSport , olympics:Sport ;
olympics:sportName "Javelin"^^xsd:string .

olympics:SportMaleJavelin a owl:Thing , olympics:MaleSport , olympics:Sport ;
olympics:sportName "Javelin"^^xsd:string .

olympics:StadiumKaftanzoglio a olympics:Stadium , owl:Thing ;
olympics:hasSpectatorCapacity "28200"^^xsd:int ;
olympics:hasStadiumName "Kaftanzoglio Stadium"^^xsd:string ;
olympics:inCity "Thessaloniki"^^xsd:string .

olympics:TrainerAlexander_Makarov a olympics:Trainer , olympics:Human
 , owl:Thing ;
olympics:hasBirthdate "1973-03-19"^^xsd:date ;
olympics:hasFirstName "Alexander"^^xsd:string ;
olympics:hasGender "Male"^^xsd:string ;
olympics:originCountry "Russia"^^xsd:string ;
olympics:hasSurname "Zelezny"^^xsd:string ;
olympics:trains olympics:AthleteJan_Zelezny .

olympics:TrainerYannis_Peristeris a owl:Thing , olympics:Human
 , olympics:Trainer ;
olympics:hasBirthdate "1977-07-12"^^xsd:date ;
olympics:originCountry "Greece"^^xsd:string ;
olympics:hasGender "Male"^^xsd:string ;
olympics:hasSurname "Peristeris"^^xsd:string ;
olympics:hasFirstName "Yannis"^^xsd:string ;
olympics:trains olympics:AthleteSavva_Lika .