Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2007-40

## Masterarbeit

im Studiengang "Angewandte Informatik"

# OWLQ – An OWL-based Query Language for OWL Data

Jochen Kemnade

am Institut für

Informatik

AG Datenbanken & Informationssysteme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

15. November 2007

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

| | |
|---|---|
| Tel. | +49 (5 51) 39-1 44 14 |
| Fax | +49 (5 51) 39-1 44 15 |
| Email | office@informatik.uni-goettingen.de |
| WWW | www.informatik.uni-goettingen.de |

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 15. November 2007

Master's Thesis

# OWLQ – An OWL-based Query Language for OWL Data

Jochen Kemnade

November 15, 2007

supervised by Prof. Dr. Wolfgang May
Databases and Information Systems Group
Institute for Computer Science
University of Göttingen

## Abstract

This thesis deals with the OWL-based, semantic query language OWLQ which defines a query ontology based on classes, variables and constraints and whose queries are completely stated in RDF. Hence it is possible to handle the components of the query, i.e., definitions of relevant classes and constraints, like any other Semantic Web resource constituting the facilities to specify metadata about queries, reason about them with regards to their satisfiability and containment in each other and publish them as Semantic Web data.

Besides describing the theoretical concepts of OWLQ queries, this thesis addresses the implementation of an application that supports their analysis and execution based upon the Jena framework.

## Zusammenfassung

Diese Arbeit behandelt die OWL-basierte, semantische Anfragesprache OWLQ. Diese definiert eine Query-Ontologie, die auf Klassen, Variablen und Bedingungen basiert. OWLQ-Anfragen werden komplett in RDF dargestellt, somit können die Komponenten einer Anfrage, also ihre relevanten Klassendefinitionen und Bedingungen als Ressourcen des Semantischen Webs behandelt werden. Dadurch ist es möglich, die Anfragen mit Metadaten zu beschreiben, sie hinsichtlich ihrer Erfüllbarkeit und ihres gegenseitigen Enthaltenseins zu untersuchen und sie als Objekte des Semantischen Webs zu publizieren.

Neben der Beschreibung der theoretischen Konzepte von OWLQ-Anfragen beschreibt diese Arbeit die Implementierung einer auf dem Jena-Framework basierenden Applikation zu deren Analyse und Ausführung.

# Contents

# 1 Introduction

"You don't need to know everything. You need to know where to look it up." Apart from this famous citation mostly being used with reference to human beings, it is also an important concept of the Semantic Web. An ontology that is available through one location on the Internet may contain statements about a resource that is published in another place and may specify relations that connect that resource to yet another resource from yet another location.

To obtain information from this sort of distributed information database, queries can be stated against one or more ontologies, whose combined information is used to answer the query. However, the queries written in the established query languages are merely syntactic entities which cannot be described in a semantic context.

OWLQ uses a semantic approach to stating queries. It defines an ontology that specifies the elements of a query such as classes, variables and constraints. OWLQ queries entirely consist of RDF resources and statements about them and are RDF resources themselves, which can again be published as Semantic Web data and referenced in other ontologies' statements. As OWLQ queries are addressable via their URI, an ontology may also include statements about the relation of two queries to one another.

The ability to refer to a query's every particular element allows for its analysis on a semantic level which makes it possible to examine for example the satisfiability of both a whole query or a single class or constraint. Being able to compare the structural elements of different OWLQ queries from a semantic point of view even provides the facilities for a heuristic approach to the examination of query containment.

To be able to semantically examine a query with regards to its satisfiability and containment in other queries does not only provide a way to know whether it can have any results at all or if its results can be computed out of the results of another query, but it also serves for the examination of the completeness and validity of both the query itself and of the ontology it refers to.

This thesis is subdivided into six chapters, the first of which is this introduction. Chapter 2 provides some basic information about Semantic Web ontologies and queries and how they can be dealt with in Java. The 3rd chapter starts with introducing the OWLQ query language and continues with theoretical details about what is possible with regards to executing and reasoning about queries. Chapter 4 describes the implementation of a Java package, that enables its importing application to use the features presented before like the execution of OWLQ queries and the analysis regarding their satisfiability and containment. In Chapter 5, the reference implementation of an OWLQ servlet is explained to show, how the package can be utilized in an application. Finally, the 6th chapter concludes the thesis summarizing its results.

# 2 Foundations

This chapter deals with presenting the basic concepts, on which this thesis is based.

In the first section, a short overview is given about the Resource Description Framework and the Web Ontology Language, followed by explanations about the SPARQL Protocol and RDF Query Language. The presentation of those languages is followed by the introduction of the Jena framework and one of its plugins, namely the Pellet reasoner.

## 2.1 RDF and OWL

The Resource Description Framework (RDF) is a formal language, developed for the purpose of making data and metadata available on the World Wide Web. RDF consists of statements, which describe a resource, referred to as the *subject* and which are triples in the form `subject predicate object`. A statement's subject is a resource, which can be either anonymous or identified via a Unified Resource Identifier (URI). The *predicate*, which describes a relationship between the subject and the *object*, is a resource as well and the object is either a resource or a literal such as a number or a string.

RDF can be represented by a directed graph structure as well as in different textual representations. The most common textual formats are RDF/XML and N3. RDF/XML is commonly used when it comes to providing and transferring RDF data via the Internet, whereas the N3 format exactly reflects the subject predicate object syntax mentioned above, making it more intuitive for humans to read.

The Web Ontology Language (OWL) can be seen as an extension to RDF. OWL introduces new vocabulary and thereby makes it possible to make more expressive statements including semantic aspects.

```
 1  @prefix :  <http://family#>.
 2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
 3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
 4  @prefix owl: <http://www.w3.org/2002/07/owl#>.
 5
 6  :Person a owl:Class.
 7  :Female a owl:Class; rdfs:subClassOf :Person.
 8  :Male a owl:Class; rdfs:subClassOf :Person; owl:disjointWith :Female.
 9  :hasChild a rdf:Property; rdfs:range :Person; owl:inverseOf
       :hasParent.
10  :age a owl:FunctionalProperty.
11  :name a owl:FunctionalProperty.
12
13  :kate a :Female; :name "Kate"; :age 62; :hasChild :john, :sue, :frank.
14  :john a :Male; :name "John"; :age 35; :hasChild :alice, :bob.
15  :sue :name "Sue".
```

*Figure 2.1: An example RDF document*

Figure 2.1 shows an example family ontology in N3, defining the classes `:Person`, `:Female` and `:Male` and the properties `:hasChild`, `:age` and `:name` in Lines 6–11. Lines 13–15 introduce the individuals `:kate`, `:john`, `:sue`, `:frank`, `:alice` and `:bob` and make statements about them such as some parent child relationships, names and ages.

## 2.2 SPARQL

To obtain information from a given RDF file, there are several query languages, the SPARQL Protocol and RDF Query Language being the most sophisticated of them.

The example query in Figure 2.2 shows, how SPARQL can be used to extract information from the example family ontology. In Line 2, the query's result variables are defined. The query's constraints, which define, how the variables should be related, are specified in Lines 4–6 by again using triples. If this query is executed against the example family ontology in Figure 2.1, it yields the tuples ("Kate", "Sue") and ("Kate", "John") as results. The rest of the ontology's

```
PREFIX fam:  <http://family#>
SELECT  ?ParentName ?ChildName
WHERE
  { ?Parent fam:hasChild :Child;
            fam:name ?ParentName.
    ?Child  fam:name ?ChildName.
  }
```

*Figure 2.2: A SPARQL query that lists parents and their children*

`fam:hasChild` relationships, such as `:kate :hasChild :frank`, are not listed, due to the fact, that the query explicitly asks for the persons' names. As the names are only given for `:kate`, `:john` and `:sue`, the other members of the `:Person` class are not taken into account.

## 2.3 The Jena Framework

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[3]. It supports the input and output of RDF data in the established formats (RDF/XML, N3, N-Triples). Internally, this data is kept inside models, which can be processed via several functions that the Jena API[2] provides. These include functions for modifying and extracting data from these models and stating queries.

Jena's model interface is defined in `com.hp.hpl.jena.rdf.model.Model`. An RDF model is a set of statements. Methods are provided for creating resources, properties and literals and the statements which link them, for adding statements to and removing them from a model, for querying a model and set operations for combining models[2]. Those general models are just capable of keeping RDF data and do not have any inference support. This feature is among others offered by the sub-interface `com.hp.hpl.jena.ontology.OntModel`, which incorporates a base model with an optional reasoner. When it comes to reasoning about models, Jena offers a built-in OWL reasoner as well as the possibility to plug in virtually any external DIG[1] reasoner.

Queries can be stated against those models by using the ARQ package. ARQ is a query engine for Jena, that supports the SPARQL Protocol and RDF Query Language. Its main class is `com.hp.hpl.jena.query.Query`, instances of which are created either declaratively from query strings or programmatically by setting its result variables and constraints using the query object's respective methods.

A query's result is returned as a `com.hp.hpl.jena.query.ResultSet`, which is a sub-interface of `java.util.Iterator` and also part of the ARQ package. A `ResultSet` object can be either output using the `ResultSetFormatter` class or processed by inquiring the particular entries.

## 2.4 Pellet

Pellet is a powerful OWL reasoner, which supports the full expressivity of OWL-DL and most of the features proposed in OWL 1.1. Pellet provides functionalities to validate ontology species, check consistency of ontologies, classify the taxonomy, check entailments and answer a subset of RDQL queries (known as ABox queries in DL terminology) [4]. One of the main advantages that Pellet has over the Jena framework's internal reasoner is it being faster when it comes to bigger and more complicated ontologies. Pellet is the recommended reasoner to be used with Jena.

# 3 Analysis and Design

This chapter gives an overview of the concepts upon which the OWLQ query engine was realized.

Its first section is a brief description of the work-in-progress query language OWLQ. The rest of this chapter shows, how OWLQ queries can be executed and analyzed with respect to their satisfiability and containment in each other, along with some more advanced features of OWLQ. These descriptions also give an outline on which features the implementation should have along with theoretical explanations on how they are intended to work.

Finally, the last section lists the complete OWLQ ontology for easy reference.

## 3.1 An Introduction to OWLQ

The OWLQ query language was developed out of the idea that one should be able to handle queries as semantic web data, thus making one able to reason about them just like about any other (meta)data ontology.

In contrast to SPARQL, OWLQ is completely based on OWL. The entire query consists of triples, so it can be represented in common RDF notations such as RDF/XML or N3 and published as Semantic Web resources on the Internet. Additionally, that makes it possible to make statements about OWLQ queries as their URIs can be used to reference them in other ontologies just like any other resource. The complete OWLQ ontology is shown in Section 3.7.

Apart from being syntactically different, there are also some semantic differences between OWLQ and SPARQL. First of all, OWLQ offers facilities to define classes and data ranges, generally being referred to as scopes, within a query. Both classes and data ranges resemble their OWL equivalents and are additionally able to scope variables. In OWLQ, variables are not treated as syntactic elements of a query, but they are also semantic objects that belong to the OWL class `owlq:Variable`. Therefore, they can have properties and relationships with other resources and can be utilized for the analysis of a query on a semantic level.

An OWLQ variable can either directly range over a class or a data range, or it can be defined as the literal value of another variable's property. If `:P` is a variable, that ranges over the class `:Person`, one could define another variable `:Name` to refer to the name of the respective person assigned to `:P`.

A requested relation between two variables is expressed in a constraint, that requires one variable to have a property with the other variable as its value. For example, one could have the variables `:Country` and `:City` range over all countries and cities respectively and a constraint could require, that `:City` is the capital of `:Country`.

The query that is shown in Figure 3.1 is an OWLQ equivalent to the family SPARQL query (Figure 2.2). The query's root definition ranges over the Lines 6–9. Line 7 resembles the `SELECT` statement in SPARQL and defines the query's result variables, constituting the projection part of the query. Line 8 introduces classes, which will be defined in a subsequent part of the query and Line 9 refers to the constraint, that the result tuples have to satisfy and which will also be defined later.

In this example query, there are two OWLQ class definitions. The `:Parent` class which scopes the variable `:P` is defined as something that has a child which is an element of the `:Person` class. The class `:Child` ranges over the variable `:C` and is defined as everything, that has a `:Person` as its parent. Additionally, for both `:P` and `:C` the `owlq:VariableDefinition` construct is used to assign the result elements' names to the variables `:ParentName` and `:ChildName` respectively.

The desired relation between the variables `:P` and `:C` is stated in Lines 25–27 as an `owlq:Constraint` that limits the result set to those tuples that satisfy the statement `:P fam:hasChild :C`. As the query is equivalent to the SPARQL query from Figure 2.2, executing it also yields the tuples (”Kate”, ”Sue”) and (”Kate”, ”John”).

```
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix fam: <http://family#>.
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :ParentName, :ChildName;
  owlq:definesClass   :Parent, :Child;
  owlq:hasConstraint  :c1 ].

:Parent a owlq:Class; owlq:scopesVariable :P;
  owl:equivalentClass [ a owl:Restriction;
                        owl:onProperty fam:hasChild;
                        owl:someValuesFrom fam:Person ].
:P owlq:hasVariableDefinition [ owlq:onProperty fam:name;
                                owlq:toVariable :ParentName ].

:Child a owlq:Class owlq:scopesVariable :C;
  owl:equivalentClass [ a owl:Restriction;
                        owl:onProperty fam:hasParent;
                        owl:someValuesFrom fam:Person ].
:C owlq:hasVariableDefinition [ owlq:onProperty fam:name;
                                owlq:toVariable :ChildName ].

:c1 a owlq:Constraint; owlq:onVariable :P;
                       owlq:onProperty fam:hasChild;
                       owlq:equalsVariable :C.
```

*Figure 3.1: An OWLQ query that lists parents and their children*

As already mentioned, besides an `owlq:Class`, an `owlq:Variable` can also range over an `owlq:DataRange`, which is a subclass of `owl:DataRange`. It is used if a variable is to range over a literal data type such as a string or a numeric interval. This variable can then be referenced in a constraint.

```
 1 @prefix owl: <http://www.w3.org/2002/07/owl#>.
 2 @prefix owl11: <http://www.w3.org/2006/12/owl11#>.
 3 @prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
 4 @prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
 5 @prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
 6 @prefix : <ex:pl#>.
 7
 8 [ a owlq:Query; owlq:resultVariable :CityName, :CityPop;
 9   owlq:definesClass mon:City; owlq:hasConstraint :c1 ].
10
11 :AtLeast1M a owlq:DataRange; owlq:scopesVariable :CityPop;
12            owl11:onDataRange xsd:int; owl11:minInclusive 1000000.
13
14 :X owlq:rangesOver mon:City;
15    owlq:hasVariableDefinition [ owlq:onProperty mon:name;
16                                 owlq:toVariable :CityName ].
17
18 :c1 a owlq:Constraint;
19     owlq:onVariable :X;
20     owlq:onProperty mon:population;
21     owlq:equalsVariable :CityPop.
```

*Figure 3.2: An OWLQ query that lists cities with a big population*

The query shown in Figure 3.2[1] lists cities, that have a population of at least one million. The scope `:AtLeast1M`, which ranges over `xsd:int` values greater or equal than 1.000.000, is defined in Lines 11–12 and scopes the variable `:CityPop`. This variable is then referenced in the constraint `:c1` in Line 21 in the same way as a variable that ranges over an `owlq:Class`.

[1] All of this thesis' queries that use geographical facts are based on the MONDIAL database, which can be found at `http://www.dbis.informatik.uni-goettingen.de/Mondial/`

## 3.2 Reducing OWLQ to OWL and SPARQL

Jena does not offer the necessary facilities to handle OWLQ directly. But from Section 3.1, one can already infer that every OWLQ query can be partitioned into an OWL and a SPARQL portion. As Jena is meant for operating with OWL and SPARQL, this partitioning is the only step to be taken to make Jena capable of executing OWLQ queries by adding their OWL part to the knowledge base and evaluating their SPARQL part against it.

### 3.2.1 Extracting the OWL Class Definitions

There were two possible approaches for extracting the OWL portion from the query. The first one would consist of taking all elements of type `owlq:Class` and `owlq:DataRange` into account and removing all the statements about them that are specific to OWLQ. That would include all statements whose predicates belong to the `owlq` namespace, i.e. the property `owlq:scopesVariable`. Those predicates are exactly what distinguishes an `owlq:Class` from an `owl:Class` and an `owlq:DataRange` from an `owl:DataRange` so the plain OWL class and data range definitions would remain.

However, a slightly different approach was taken for the OWLQ engine's implementation. Every statement is taken into account and is added to the OWL portion's ontology if neither its predicate nor its object are from the `owlq` namespace. It will instantly become clear, that this has the same effect as the above-mentioned approach. Every OWL query consists of the query element, scope definitions and constraints. As for the class and data range definitions, the effect is just the same as above, the `owlq:scopesVariable` predicates are ignored. For the `owlq:Query` and `owlq:Constraint` elements, all the statements about them have predicates from the `owlq` namespace, which are removed as well. Of course, the query and constraint resources themselves are also removed, because their objects belong to the `owlq` namespace as per definition.

A special treatment is needed for members of the `owlq:DataRange` class. As every statement `:X a owlq:DataRange` would be lost because of its object belonging to the `owlq` namespace, for the OWL part, it is replaced by `:X a owl:DataRange`.

Figure 3.3 shows the class definitions that are extracted from the query given in Figure 3.1, that can now be added to the knowledge base.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix fam: <http://family#>.
@prefix : <ex:pl#>.
:Child owl:equivalentClass [
  a owl:Restriction; owl:onProperty fam:hasParent;
  owl:someValuesFrom fam:Person ].
:Parent owl:equivalentClass [
  a owl:Restriction;  owl:onProperty fam:hasChild;
  owl:someValuesFrom fam:Person ].
```

*Figure 3.3: The OWL class definitions generated out of an OWL query*

### 3.2.2 Extracting the SPARQL Query

Because of an OWLQ query being RDF data itself, an OWLQ query can be used as a knowledge base that can be queried using SPARQL. The SPARQL query

```
PREFIX owlq: <http://www.semwebtech.org/languages/2006/owlq#>
SELECT ?S ?CV ?V1 ?P ?V2
WHERE
  { {?S a owlq:Scope; owlq:scopesVariable ?CV}
    UNION
    { ?V1 owlq:hasVariableDefinition
        [ owlq:onProperty ?P; owlq:toVariable ?V2 ] }
    UNION
    {?S a owlq:Constraint;  owlq:onVariable ?V1;
        owlq:onProperty ?P; owlq:equalsVariable ?V2 ] }
  }
```

*Figure 3.4: Extracting variables and constraints from an OWLQ query*

listed in Figure 3.4 returns the necessary information for building a SPARQL query out of an OWLQ query[7]. For the example family query (Figure 3.1), the result would look like shown in Figure 3.5.

| S | CV | V1 | P | V2 |
|---|---|---|---|---|
| :Parent | :P | | | |
| :Child | :C | | | |
| | | :P | fam:name | :ParentName |
| | | :C | fam:name | :ChildName |
| :c1 | | :P | fam:hasChild | :C |

*Figure 3.5: Example result of the information extraction query*

Each row of the result set represents either information about classes and the variables they scope (upper part of the `UNION`), information on one variable being defined on a property of another one (the middle part), or information about the query's constraints (lower part of the `UNION`). For the former, there is always a scope (`S`) and its scoped variable (`CV`). For the variable definitions, the entry consists of the variable `V2` defined by the property `P` on the other variable `V1`.

For the constraints, the result is always the constraint's name (`S`), its affected variable (`V1`) and the property `P`, whose object is to be the variable `V2`.

From these results, one can easily generate a SPARQL query, which looks like the one in Figure 3.6.

```
PREFIX  : <ex:pl#>
PREFIX  fam: <http://family#>
SELECT  ?ChildName ?ParentName
WHERE
  { ?P  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  :Parent;
        fam:name              ?ParentName.
    ?C  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  :Child;
        fam:name              ?ChildName.
    ?P  fam:hasChild          ?C.
  }
```

*Figure 3.6: A SPARQL query generated out of an OWLQ query*

## 3.3 Advanced Features of OWLQ

By means of the features described so far, one can realize all conjunctive queries including filters, that can be stated using SPARQL, with the exception of `UNION` and `OPTIONAL` constructs. This section describes two of the OWLQ query language's features, that SPARQL does not provide, namely negated constraints and closures.

### 3.3.1 Negated Constraints

One feature that sets OWLQ apart from SPARQL is the ability to express negated constraints. Those allow for constraints, that filter all tuples $(X, Y)$, where $X$ is not related to $Y$ by a property $p$. First of all, strictly speaking, this formulation is not correct. In fact, negated constraints filter tuples $(X, Y)$, about which it is not *known*, that $X$ is related to $Y$ by a property $p$, which is slightly different but much more useful. The following example will show the difference.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :Country, :NonNeighbor;
  owlq:definesClass mon:Country;
  owlq:hasConstraint :c1 ].

mon:Country a owlq:Class;
            owlq:scopesVariable :Country, :NonNeighbor.

:c1 a owlq:NegatedConstraint; owlq:onVariable :Country;
    owlq:onProperty mon:neighbor;
    owlq:equalsVariable :NonNeighbor.
```

*Figure 3.7: An OWLQ query with a negated constraint*

The OWLQ query in Figure 3.7 asks for all the tuples of two countries, that are not neighbors of one another, or rather about which it is *not said*, that they *are* neighbors.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://www.semwebtech.de/mondial/10/meta#>.

:Country  a owl:Class.
:neighbor a owl:ObjectProperty, owl:SymmetricProperty;
          rdfs:domain :Country;
          rdfs:range  :Country.

<http://www.semwebtech.de/mondial/10/countries/E/> :neighbor
    <http://www.semwebtech.de/mondial/10/countries/P/>,
    <http://www.semwebtech.de/mondial/10/countries/F/>.
```

*Figure 3.8: An excerpt from the MONDIAL database*

Figure 3.8 shows an excerpt of the MONDIAL database, containing the countries Spain, Portugal and France and defining a symmetric property `:neighbor` between two countries. From this ontology, Pellet can infer, that Spain is a neighbor of Portugal and France and, due to the symmetry, both Portugal and France are also neighbors of Spain.

Pellet uses Open World Assumption (OWA), that means, all triples that are not given are assumed just not to be known. In contrast, Closed World Assumption (CWA) means, that all triples that are not given are false. So, as in the above ontology, Portugal is not stated to be a neighbor of France, CWA would mean, that there is no neighborship relation between them. However, as Pellet is using OWA, it will not infer that Portugal and France are not neighboring, because there *could* be a neighborship relation between them, which is just not given in the ontology.

To have the tuples (Portugal, France) and (France, Portugal) listed in the result set, OWLQ makes use of a SPARQL facility, that allows for the selection of unsaid statements using a combination of `OPTIONAL` and `FILTER` constructs. The resulting SPARQL query for the OWLQ query in Figure 3.7 is shown in Figure 3.9.

```
 1  PREFIX mon: <http://www.semwebtech.de/mondial/10/meta#>.
 2  SELECT ?Country ?NonNeighbor
 3  WHERE
 4    { ?Country a mon:Country.
 5      ?NonNeighbor a mon:Country.
 6      OPTIONAL
 7        { ?Country mon:neighbor ?NeighborOfCountry.
 8          FILTER ( ?NeighborOfCountry = ?NonNeighbor ) }
 9      FILTER ( ! BOUND (?NeighborOfCountry) )
10    }
```

*Figure 3.9: A negated constraint expressed in SPARQL*

Again, the variables are introduced up to Line 5. Lines 6–9 correspond to the negated constraint. By Line 5, the answer is `mon:Country` × `mon:Country`. The `OPTIONAL` statement binds the auxiliary variable `?NeighborOfCountry` to those countries that *are* neighbors of `?Country` and filters those entries, where `?NeighborOfCountry` is `?NonNeighbor`. The intermediate result set is shown below.

| ?Country | ?NonNeighbor | (?NeighborOfCountry) |
|---|---|---|
| Spain | Spain | |
| Spain<br>Spain | Portugal<br>France | Portugal<br>France |
| Portugal | Portugal | |
| Portugal | Spain | Spain |
| France<br>Portugal | France<br>France | |
| France | Spain | Spain |
| France | Portugal | |

The last FILTER statement in Line 9 filters all entries, where the auxiliary variable ?NeighborOfCountry is not bound (the non-grayed), which results in exactly those pairs of countries, which are not neighbors of each other. The query's final result is shown in Figure 3.10.

| ?Country | ?NonNeighbor |
|---|---|
| Spain | Spain |
| Portugal | Portugal |
| France | France |
| Portugal | France |
| France | Portugal |

*Figure 3.10: Results of the negated constraint query*

As this result still contains the pairs where both countries are identical, it is not exactly what was intended by the query. To additionally filter those pairs, another negated constraint can be introduced, which is shown in Figure 3.11.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix : <ex:pl#>.

:c2 a owlq:NegatedConstraint;
    owlq:onVariable :Country;
    owlq:onProperty owl:sameAs;
    owlq:equalsVariable :NonNeighbor.
```

*Figure 3.11: Filtering identical values for two variables*

The resulting SPARQL query is similar to the one in Figure 3.9 and is shown in Figure 3.12. Up to Line 10, it resembles the other query and the second negated constraint is reflected in Lines 11–14.

```
 1  PREFIX mon: <http://www.semwebtech.de/mondial/10/meta#>.
 2  PREFIX owl: <http://www.w3.org/2002/07/owl#>
 3  SELECT ?Country ?NonNeighbor
 4  WHERE
 5    { ?Country a mon:Country.
 6      ?NonNeighbor a mon:Country.
 7      OPTIONAL
 8        { ?Country mon:neighbor ?NeighborOfCountry.
 9          FILTER ( ?NeighborOfCountry = ?NonNeighbor ) }
10      FILTER ( ! BOUND (?NeighborOfCountry) )
11      OPTIONAL
12        { ?Country owl:sameAs ?SameAsCountry.
13          FILTER ( ?SameAsCountry = ?NonNeighbor ) }
14      FILTER ( ! BOUND (?SameAsCountry) )
15    }
```

*Figure 3.12: A negated constraint expressed in SPARQL*

| ?Country | ?NonNeighbor | (?NeighborOfCountry) | (?SameAsCountry) |
|---|---|---|---|
| Spain | Spain | | Spain |
| Spain | Portugal | Portugal | |
| Spain | France | France | |
| Portugal | Portugal | | Portugal |
| Portugal | Spain | Spain | |
| France | France | | France |
| Portugal | France | | |
| France | Spain | Spain | |
| France | Portugal | | |

The `FILTER` statements in Lines 10 and 14 seen together filter those entries in the result set, where neither `?NeighborOfCountry` nor `?SameAsCountry` are bound. The results are the tuples (Portugal, France) and (France, Portugal).

### 3.3.2 Closure of Predicates

Another feature that OWLQ offers in contrast to SPARQL is the ability to close predicates. The query in Figure 3.13 defines an `owlq:Class` of countries, which

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :Ctry;
  owlq:definesClass :MemberOfNineOrganizations ].

:MemberOfNineOrganizations a owlq:Class;
  owlq:scopesVariable :Ctry;
  owl:intersectionOf (
    mon:Country
    [ a owl:Restriction;
      owl:onProperty mon:isMember;
      owl:cardinality 9 ]
  ).
```

*Figure 3.13: A query for countries which are in nine organizations*

are members of nine organizations. But once again, the Open World Assumption will cause Pellet not to find any results for this query, as it will not find a country, that satisfies the cardinality restriction. If no country is explicitly stated to be a member of nine organizations, Pellet would assume, that for each country, about which nine `mon:hasMember` statements are given, there could be more membership statements, which are just not contained in the knowledge base. Also, if the knowledge base contains an existential assertion about a country, that it is a member of at least nine organizations, that would not lead Pellet to infer, that it belongs to the class `:MemberOfNineOrganizations`, as it could also be a member of ten or more organizations.

The solution is to close the predicate `mon:isMember` by extending the query by the statement `mon:isMember a owlq:ClosedPredicate`, which means, that

Pellet should assume Closed World for all statements `:X mon:isMember :Y`, so all statements that are not given are assumed not to hold. This is accomplished by classifying all the knowledge base's individuals by to how many organizations they have to belong at least to satisfy the ontology.

Let $P$ an `owlq:ClosedPredicate`. The restriction $\geq n.P$ denotes the class of all individuals $x$, about which it is known, that they have at least $n$ predicates $P$, be it explicit statements of the form $P(x,y)$ or $P^{-}(y,x)$ or existential assertions like $\exists^{\geq n} y.P(x,y)$. Applied to the given example, where $P$ is `mon:isMember`, $\geq n.P$ would be the class of individuals `:X`, about which it is known from the knowledge base, that there are at least $n$ statements of the form `:X mon:isMember :Org`. Pellet is able to tell, which individuals are in the respective classes, as each of them is an `owl:Restriction` with an `owl:minCardinality` of $n$, which contains the same individuals regardless of whether open or closed world is assumed.

For $n = 1$, the class $\geq n.P$ contains all individuals, which are assured to have any filler of predicate $P$ at all. For $n = 2$, the set is reduced to those individuals who are known have two such fillers of $P$, again either explicitly stated or existentially assured. Now the relative complement of $\geq 2.P$ in $\geq 1.P$ is those individuals about which it is known, that they have at least one but not necessarily two or more fillers of predicate $P$. By now incrementing $n$ by one at a time, each of the individuals in $\geq 1.P$ belongs to $\geq n.P$ up to a certain $n$. The process is finished, when for each individual such an $n$ has been found.

The next step is to create cardinality restrictions $n.P$ and for each individual, that belongs to $\geq n.P$ up to a given $n$, but not to $\geq (n+1).P$, to add to the knowledge base, that it is a member of $n.P$. Thereafter, the knowledge base contains that an individual belongs to the class $n.P$, if it is guaranteed to have $n$ fillers for predicates $P$. For the given example, that means, that for $P =$ `mon:isMember`, $9.P$ equals the restriction on the property `mon:isMember` with a cardinality of 9 in Lines 14–16 and therefore, Pellet is able to answer the query.

Figure 3.14 shows another example for predicate closure. The individual `:john` is said to have a son `:bob` and to be in the class `:HasDaughters`, which states, that he has one daughter at least. Additionally, also `:mary` is defined to have `:bob` as her son. When now an OWLQ query is stated against that database, that defines `fam:hasChild` as an `owlq:ClosedPredicate`, the ontology's individuals will be categorized according to the procedure that has just been explained.[2]

[2] The following steps assume, that `fam:hasSon` and `fam:hasDaughter` are defined as subproperties of `fam:hasChild` with the ranges `fam:Male` and `fam:Female` respectively.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix fam: <http://family#>.

:HasDaughters a owl:Class;
  owl:equivalentClass [ a owl:Restriction;
                        owl:onProperty fam:hasDaughter;
                        owl:minCardinality 1 ].

:john a :HasDaughters.
      fam:hasSon :bob;
:mary fam:hasSon :bob.
```

*Figure 3.14: Organizations with at most 3 countries, which are in the EU*

At first, the database is queried for all individuals with at least one child, which resembles $\geq n.P$ for $n = 1$ and $P =$ `fam:hasChild`, yielding the set {`:john`, `:mary`}, as both of them are stated to have the child `:bob`. Additionally, Pellet will infer, that `:john` has even at least two children. On one hand, that is `:bob` and on the other hand, he belongs to the `:HasDaughters` class, so he must have at least one daughter. Pellet will also infer, that this daughter cannot be `:bob`, as he is a `fam:Male`. However, `:bob` will not be in the set, as he is not known to have any children at all.

Thereafter, $n$ will be incremented to 2 so the new set contains all individuals, which have at least two children. As already mentioned, `:john` will be contained in this set. He will also be the only element of the set, as `:mary` is only known to have one child. So for the individual `:mary` the greatest $n$ has be found, such that it is contained in $\geq n.P$, but not in $\geq (n+1).P$, to be 1. That information is used to categorize `:mary` as an individual, who has one child and make it a member of $1.P$ for $P =$ `fam:hasChild`. In the next step, $n$ will be 3 and the set $\geq 3.P$ will be found to be empty, which leads to `:john` being a $2.P$. Now all individuals have been categorized in the respective cardinality restriction classes and thus, the closure of `fam:hasChild` is complete.

By further examining the query in which a closed predicate occurs, the procedure can be abbreviated. For a closed predicate $P$, let $n_{P,begin}$ the least and $n_{P,end}$ the greatest $n$, for which the restrictions have to be computed. For every restriction $R_i$ on the property $P$, that the query contains, let $C_l(R_i)$ the least value of $c$, such that a member of $c.P$ is contained in the restriction. For a restriction with an `owl:cardinality` of $c$, $C_l(R_i) = c$, whereas for a restriction with an `owl:maxCardinality` of $c$, $C_l(R_i) = 0$. Now $n_{P,begin}$ can be set to the minimum of relevant cardinalities for all restrictions $\min(C_l(R_i))$.

A similar statement can be made about $n_{P,end}$. For every restriction $R_i$ on the property $P$, that the query contains, let $C_g(R_i)$ the *greatest* value of $c$, such that a member of $c.P$ is contained in the restriction. For both a restriction with an `owl:cardinality` of $c$ and a restriction with an `owl:maxCardinality` of $c$, $C_l(R_i) = c$. Now $n_{P,end}$ can be set to the *maximum* of relevant cardinalities for all restrictions $\max(C_g(R_i))$. The restriction classes $n.P$ have to be created and computed only for $n_{P,begin} \leq n \leq n_{P,end}$.

Even further optimizations can be made to the procedure, if for a restrictions on $P$, `owl11:onClass` is set.

```
:OrganizationWithAtMost3EUCountries a owl:Class;
  owl:intersectionOf (mon:Organization
                      [ a owl:Restriction;
                        owl:onProperty mon:hasMember;
                        owl:maxCardinality 3;
                        owl11:onClass :EUCountry ] ).

mon:hasMember a owlq:ClosedPredicate.
```

*Figure 3.15: Organizations with at most 3 countries, which are in the EU*

Figure 3.15 shows a class definition for all organizations, which have at least 3 members, which belong to the European Union. By now, all individuals were categorized in the created restrictions $n.P$, which can principally be seen to have `owl11:onClass owl:Thing`. For this example, that were those individuals, who have any $n$ members. The restrictions on a property $P$ can be aggregated according to their `owl11:onClass` property. Now, instead of creating the more general $n.P$ restrictions, for each class $C$, that is referenced in a group of restrictions' `owl11:onClass` attribute, the restrictions $n.P.C$ are created. Of course, for those restrictions, where no explicit `owl11:onClass` property is given, setting $C$ to `owl:Thing` leads to $n.P.C \equiv n.P$.

### 3.3.3 Closure of Classes

Besides the ability to close predicates, also classes can be closed in OWLQ. The query in Figure 3.16 defines the Scandinavians `:A` through `:H` and states that a Scandinavian is either a Dane, a Swede or a Norwegian and defines `:A` and `:F` as Danes and `:C`, `:G` and `:H` as Swedes. When asking for the members of `:Norwegian`, the expected answer would be `:B`, `:D` and `:E`. But yet again, Pellet's OWA makes it unable to infer those memberships. The fact, that `:B`, `:D` and `:E` are not known to be Swedes or Danes, does not lead to it assuming, that this does not hold.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owl11: <http://www.w3.org/2006/12/owl11#> .
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#> .
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :N;
  owlq:definesClass :Scandinavian, :Dane, :Swede, :Norwegian ].

:Scandinavian owl:oneOf (:A :B :C :D :E :F :G :H);
  owl11:disjointUnionOf ( :Dane :Swede :Norwegian ).

owl:AllDifferent owl:distinctMembers (:A :B :C :D :E :F :G :H).

:A a :Dane.
:C a :Swede.
:F a :Dane.
:G a :Swede.
:H a :Swede.

:Norwegian a owlq:Class; owlq:scopesVariable :N.
```

*Figure 3.16: A query to infer, who is a Norwegian*

By closing the classes using the statements `:Dane a owlq:ClosedClass.` and `:Swede a owlq:ClosedClass` respectively, it can be stated, that there are no other members than those, about which this can be told from the knowledge base.

For every closed class `:C`, the results of querying the knowledge base for all its elements can be set as the object of the class' `owl:oneOf` property. Applied to the given example, that would lead to the additional facts `:Dane owl:oneOf (:A :F)` and `:Swede owl:oneOf (:C :G :H)` being added. Now, due to `:Scandinavian` being defined as the disjoint union of `:Dane`, `:Swede` and `:Norwegian`, Pellet can infer that the remaining members of `:Scandinavian` must belong to the class `:Norwegian`.

## 3.4 Query Compilation

Queries are not always written in an efficient way. Consider the example OWLQ query listed in Figure 3.17.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix owl11: <http://www.w3.org/2006/12/owl11#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :X, :Y;
  owlq:definesClass :BigCountry, :BigMountain;
  owlq:hasConstraint :c1 ].

:AtLeast2K a owl:DataRange;
  owl11:onDataRange xsd:int; owl11:minInclusive 2000.
:AtLeast100K a owl:DataRange;
  owl11:onDataRange xsd:int; owl11:minInclusive 100000.
:BigMountain owlq:scopesVariable :X;
  owl:intersectionOf (mon:Mountain
    [ a owl:Restriction; owl:onProperty mon:height;
      owl:someValuesFrom :AtLeast2K ] ).
:BigCountry owlq:scopesVariable :Y;
  owl:intersectionOf (mon:Country
    [ a owl:Restriction; owl:onProperty mon:area;
      owl:someValuesFrom :AtLeast100K ] ).

:c1 a owlq:Constraint; owlq:onVariable :X;
  owlq:onProperty mon:locatedIn;
  owlq:equalsVariable :Y.
```

*Figure 3.17: An OWLQ query that lists big mountains and their countries*

This query defines four scopes, two of them being linked to result variables. The variable `:Y` ranges over all elements of the class `:BigCountry`. A `:BigCountry` is a country whose area value is at least 100.000 km$^2$. The variable `:X` is bound to all resources of type `:BigMountain`, which contains all mountains that have a height of at least 2000 meters. In the end, the constraint `:c1` restricts the result set to those tuples that satisfy the statement `:X mon:locatedIn :Y`, that is all big mountains that are located in big countries.

In the final step, the evaluation of the constraint `:c1`, the reasoner has to iterate over all the possible combinations of assignments to `:X` and `:Y` to check, whether they satisfy the constraint. Executing the query against only the European part of the MONDIAL database, the reasoner has to check 16 different assignments of `:X` against 18 values of `:Y`, which makes 288 possible combinations. For the explanation why this is anything but optimal, the results of the query are shown in Figure 3.18.

| X | Y |
|---|---|
| Bjelucha | Russia |
| Elbrus | Russia |
| Galdhoeppig | Norway |
| Glittertind | Norway |
| Jostedalsbre | Norway |
| Kasbek | Russia |
| Kebnekaise | Sweden |
| Korab | Serbia and Montenegro |
| Montblanc | France |
| Oeraefajoekull | Iceland |
| Portefjaellen | Sweden |
| Sarektjokko | Sweden |
| Snoehetta | Norway |
| Zugspitze | Germany |

*Figure 3.18: Results of the big mountains query*

Considering the MONDIAL database, one can observe, that there are big mountains, that do not appear in the list, namely the “Großglockner” (3797 m) and the “Jezerce” (2694 m). In spite of them being higher than 2000 meters, of course, they are not contained in the result because of the countries they are located in being too small, namely Austria (83850 km$^2$) and Albania (28750 km$^2$).

### 3.4.1 Creating Result Classes out of Constraints

The set of relevant mountains comprises exactly those, which are located in big countries, that is those elements out of the `:BigMountain` class who have a property `mon:locatedIn` with an object out of the `:BigCountry` class, which is exactly the purpose of the constraint `:c1` in the query from Figure 3.17. As the other elements from the `:BigMountain` class are not relevant for the result, a new class can be defined as an intersection of `:BigMountain` and everything that is located in a `:BigCountry`. Its class definition is shown in Figure 3.19.

```
:X-BigMountainlocatedInBigCountry a owl:Class;
  owl:intersectionOf (:BigMountain [ a owl:Restriction;
    owl:onProperty mon:locatedIn;
    owl:someValuesFrom :BigCountry ]
  ).
```

*Figure 3.19: Restricting the `:BigMountain` class to the relevant elements*

As the members of the original `:BigMountain` class, which are not in the newly-generated one, are not relevant for the result sets anyway, for the compiled query, the variable `X` can be stated to range only over the new class `:X-BigMountainlocatedInBigCountry`. The final query is shown in Figure 3.20.

Unfortunately, the generation of those restriction classes cannot be applied to both sides of the constraint. If both the `:BigMountain` and the `:BigCountry` class were refined using restrictions on the constraint’s property, the resulting classes would look like shown in Figure 3.21.

```
@prefix :        <ex:pl#>.
@prefix owl11:   <http://www.w3.org/2006/12/owl11#>.
@prefix owl:     <http://www.w3.org/2002/07/owl#>.
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#>.
@prefix mon:     <http://www.semwebtech.de/mondial/10/meta#>.
@prefix owlq:    <http://www.semwebtech.org/languages/2006/owlq#>.

[ a owlq:Query;
  owlq:resultVariable :X, :Y;
  owlq:definesClass :BigMountain, :BigCountry,
      :X-BigMountainlocatedInBigCountry;
  owlq:hasConstraint :c1 ].

:AtLeast2K a owl:DataRange;
  owl11:minInclusive 2000;
  owl11:onDataRange xsd:int.
:AtLeast100K a owl:DataRange;
  owl11:minInclusive 100000;
  owl11:onDataRange xsd:int.
:BigCountry
  owl:intersectionOf (mon:Country [ a owl:Restriction;
    owl:onProperty mon:area;
    owl:someValuesFrom :AtLeast100K ] ).
:BigMountain
  owl:intersectionOf (mon:Mountain [ a owl:Restriction;
    owl:onProperty mon:height;
    owl:someValuesFrom :AtLeast2K ] ).
:X-BigMountainlocatedInBigCountry owlq:scopesVariable :X;
  a owl:Class;
  owl:intersectionOf (:BigMountain [ a owl:Restriction;
    owl:onProperty mon:locatedIn;
    owl:someValuesFrom :BigCountry ] ).

:c1 a owlq:Constraint;
  owlq:equalsVariable :Y;
  owlq:onProperty mon:locatedIn;
  owlq:onVariable :X.
```

*Figure 3.20: The compiled version of the big mountains OWLQ query*

```
:X-BigMountainlocatedInBigCountry a owl:Class;
  owl:intersectionOf (:BigMountain [ a owl:Restriction;
    owl:onProperty mon:locatedIn;
    owl:someValuesFrom :Y-BigCountryinverseOfLocadedInBigMountain ] ).

mon:inverseOfLocatedIn owl:inverseOf mon:locatedIn.

:Y-BigCountryinverseOfLocadedInBigMountain a owl:Class;
  owl:intersectionOf (:BigCountry [ a owl:Restriction;
    owl:onProperty mon:inverseOfLocatedIn;
    owl:someValuesFrom :X-BigMountainlocatedInBigCountry ] ).
```

*Figure 3.21: Restricting both the `:BigMountain` and the `:BigCountry` classes*

Taking a closer look at those classes, one will find, that there is a cyclic dependency of the classes on one another. Both of them are intersections containing an `owl:someValuesFrom` restriction with the respective other class as its target. So as both classes do not have any explicit members, Pellet will come to the conclusion, that they are both empty. For each tuple (`:X`, `Y`), where `:X a :BigMountain` and `:Y a :BigCountry`, `:X` also belongs to the class `:X-BigMountainlocatedInBigCountry`, if and only if `:Y` is already known to be a `:Y-BigCountryinverseOfLocadedInBigMountain` and vice versa. So the class restriction may only be applied to either variable.

The decision on which of a constraint's variable is to be restricted by such a class depends on which variable belongs to the query's result variables. For example, a query could define two variables `:P` and `:C` to range over the class `fam:Person` and a constraint requiring `:P fam:hasChild :C`. If the query had the single result variable `:P`, the restriction class would be defined as the intersection of `fam:Person` and a restriction on the property `fam:hasChild` with `owl:someValuesFrom fam:Person` and would scope the variable `:P`. However if `:C` was the only result variable, the restriction class would be created for `:C` as the intersection of `fam:Person` and a restriction on `fam:hasParent` (the inverse of `fam:hasChild`) with `owl:someValuesFrom fam:Person`.

If both of a constraint's variables belong to the query's result variables, it makes no difference for which of them the restriction class is created. However, generally the restriction class is to be created for the variable, that is 'closer' to one of a query's result variables in terms of the minimal number of constraints that create a link between the variables.

Again, there is a special case that involves the use of an `owlq:DataRange` as one of a constraint's variable's scope. Such a query is shown in Figure 3.22. For

```
@prefix owl11: <http://www.w3.org/2006/12/owl11#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :City, :CityPop;
  owlq:definesClass :City;
  owlq:hasConstraint :c1 ].

:AtLeast1M owlq:scopesVariable :CityPop;
  a owlq:DataRange;
  owl11:onDataRange xsd:int;
  owl11:minInclusive 1000000.
mon:City owlq:scopesVariable :City;

:c1 a owlq:Constraint; owlq:onVariable :City;
  owlq:onProperty mon:population;
  owlq:equalsVariable :CityPop.
```

*Figure 3.22: A query that uses an owlq:DataRange in a constraint*

the result of that query, only those members of `mon:City` are to be taken into consideration, that have a population of at least one million. Consequently, the class can be restricted to `mon:CitypopulationAtLeast1M` analogously to what was shown in the previous example.

For the opposite direction, obviously it does not make sense to restrict the set of numbers starting from 1.000.000 to those, that are populations of a `mon:City`. Therefore, if one of a constraint's variables ranges over an `owlq:DataRange`, the restriction class is to be created for the other variable. For this example, the resulting class definition is shown in Figure 3.23.

```
mon:CitypopulationAtLeast1M a owl:Class;
  owl:intersectionOf (mon:City [ a owl:Restriction;
    owl:onProperty mon:population;
    owl:someValuesFrom :AtLeast1M
  ] ).
```

*Figure 3.23: The generated class of cities with at least a million inhabitants*

### 3.4.2 Removing Obsolete Constraints

After having compiled the query with respect to restricting the variables' scopes depending on the constraints on the variables, there are cases, in which one or more of the constraints have become obsolete.

```
@prefix : <ex:pl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix fam: <http://family#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#> .

[ a owlq:Query; owlq:definesClass fam:Person;
  owlq:resultVariable :Grandparent; owlq:hasConstraint :c1, :c2 ].

fam:Person a owlq:Class;
         owlq:scopesVariable :Person, :Parent, :Grandparent.

:c1 owlq:onVariable :Grandparent;
    owlq:onProperty fam:hasChild;
    owlq:equalsVariable :Parent.
:c2 owlq:onVariable :Parent;
    owlq:onProperty fam:hasChild;
    owlq:equalsVariable :Person.
```

*Figure 3.24: A query for grandparents*

The query shown in Figure 3.24 defines three variables `:Person`, `:Parent` and `:Grandparent` all of which range over the class `fam:Person`. The constraints

`:c1` and `:c2` state the necessary `fam:hasChild` relationships. The restriction classes resulting from compiling this query are shown in Figure 3.25.

```
1  @prefix :        <ex:pl#>.
2  @prefix owl:     <http://www.w3.org/2002/07/owl#>.
3  @prefix fam:     <http://family#>.
4
5  :Parent-PersonhasChildPerson a owl:Class;
6    owl:intersectionOf (fam:Person
7                        [ a owl:Restriction;
8                          owl:onProperty fam:hasChild;
9                          owl:someValuesFrom fam:Person ]
10                      ) .
11
12 :Grandparent-PersonhasChildPerson a owl:Class ;
13   owl:intersectionOf (fam:Person
14                       [ a owl:Restriction;
15                         owl:onProperty fam:hasChild;
16                         owl:someValuesFrom
                               :Parent-PersonhasChildPerson ]
17                      ) .
```

*Figure 3.25: Restriction classes for the grandparents query*

Taking a closer look at the query's root definition, one can see, that its only result variable is `:Grandparent`. The other variables `:Parent` and `:Person` are just required to exist at all but their specific assignments are not relevant for the result. As the relationships about who is whose child are irrelevant, both constraints `:c1` and `:c2` just have an existential nature and only affect the variable `:Grandparent`.

Considering again the generated classes in figure 3.25, one can observe, that the requirement of the plain existence of values for `:Parent` and `:Person` is exactly, what is expressed by the generated classes' `owl:someValuesFrom` restrictions in Lines 9 and 16. So as the purpose of the original query's constraints `:c1` and `:c2` is completely encoded into those classes, the constraints can be removed from the query's compiled version of the query. The query's final compiled version is listed in Figure 3.26.

```
@prefix : <ex:pl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix fam: <http://family#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#> .

[ a owlq:Query; owlq:definesClass fam:Person,
    :Parent-PersonhasChildPerson, :Grandparent-PersonhasChildPerson;
  owlq:resultVariable :Grandparent ].

fam:Person a owlq:Class;
      owlq:scopesVariable :Person.
:Parent-PersonhasChildPerson a owlq:Class ;
      owlq:scopesVariable :Parent ;
      owl:intersectionOf (fam:Person [ a owl:Restriction ;
                  owl:onProperty fam:hasChild ;
                  owl:someValuesFrom fam:Person
                ]) .
:Grandparent-PersonhasChildPerson a owlq:Class ;
      owlq:scopesVariable :Grandparent ;
      owl:intersectionOf (fam:Person [ a owl:Restriction ;
                  owl:onProperty fam:hasChild ;
                  owl:someValuesFrom :Parent-PersonhasChildPerson
                ]) .
```

*Figure 3.26: Final compiled version of the grandparents query*

When extracting the SPARQL portion of a compiled query with removed constraints, it is necessary to set the query `DISTINCT`, as otherwise, it might contain the same result multiple times. If for example the `DISTINCT` is omitted in the SPARQL query, that is generated from the OWLQ query listed in Figure 3.26, it would compute the Cartesian product of `:Grandparent-PersonhasChildPerson`, `:Parent-PersonhasChildPerson` and `:Person`. As `:Grandparent` is the only result variable, if there are $m$ possible values for `:Parent` and $n$ for `:Person`, each value for `:Grandparent` would occur $m \cdot n$ times in the result set.

In the most cases however, all of a query's constraints are necessary and cannot be removed. If the grandparents query for example had the result variables `:Grandparent` and `:Person`, both of the constraints would be needed, because they do not only express an existential qualification on one variable, but the actual relation between the two result variables is important, so neither of the constraints `:c1` or `:c2` can be removed without changing the query. Altogether, a constraint can be removed, if it is not part in chain of constraints that link two result variables.

## 3.5 Evaluating a Query's Satisfiability

When reasoning about a query, an important aspect is its satisfiability. Being able to tell if a query is satisfiable can help to find errors in the query itself as well as in the ontology that it refers to.

Telling whether a query is satisfiable is not a matter of telling whether it has any results when executing it against a given database, but whether it *can* have any results at all. That is why the analysis of a query's satisfiability only takes the metadata into account, leaving the real individuals aside.

A simple example for an unsatisfiable OWLQ query is given in Figure 3.27. The query asks for all tuples of (`:Person`, `:Child`), where `:Child` is a child of `:Person`. As the variable `:Person` ranges over the class `:Childless`, which is a restricted not to have any children, the result of stating the query against an arbitrary database would be empty. To make the engine able to draw this conclusion, we use again the SPARQL extraction query from Figure 3.4. The result for the unsatisfiable example query is shown in Figure 3.28. The query is satisfiable if and only if there can be an assignment to all of its variables that satisfies all of its constraints. That means, first, for each variable $v$, its scope $S(v)$ must be satisfiable, which means, that it can have any members at all. An example for an unsatisfiable class is shown in Figure 3.29.

The class `:ChildlessParent` is defined as those individuals, that have some children (Lines 6–8) as well as none at all (Lines 9–11). If one of a query's result variables ranges over that class or one of its subclasses, the query would not be satisfiable.

Besides all of the query's result variables' scopes, its constraints must be satisfiable as well. An `owlq:Constraint` requests, that for two variables $v_1$ and $v_2$ and a property $p$, the statement $v_1\, p\, v_2$ holds. The constraint is satisfiable, if and only if a member of the subject variable's scope ($S(v_1)$) can have a property $p$ with a member of the object variable's scope ($S(v_2)$) as its target. In detail,

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix fam: <http://family#>.
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :Person, :Child;
  owlq:definesClass :Childless, :Child;
  owlq:hasConstraint :c1 ].

:Childless owlq:scopesVariable :Person;
  a owlq:Class;
  owl:equivalentClass [ a owl:Restriction;
                        owl:onProperty fam:hasChild;
                        owl:cardinality 0 ].
:Child owlq:scopesVariable :Child;
  a owlq:Class.

:c1 a owlq:Constraint; owlq:onVariable :Person;
  owlq:onProperty fam:hasChild;
  owlq:equalsVariable :Child.
```

*Figure 3.27: An unsatisfiable OWLQ query*

| S | CV | P | V1 | V2 |
|---|---|---|---|---|
| :Child | :Child | | | |
| :Childless | :Person | | | |
| :c1 | | fam:hasChild | :Person | :Child |

*Figure 3.28: Information about the unsatisfiable query from Figure 3.27*

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix fam: <http://family#>.
@prefix : <ex:pl#>.

:ChildlessParent owl:intersectionOf (
  [ a owl:Restriction;
    owl:onProperty fam:hasChild;
    owl:someValuesFrom fam:Person ]
  [ a owl:Restriction;
    owl:onProperty fam:hasChild;
    owl:cardinality 0 ] ).
```

*Figure 3.29: An unsatisfiable OWL class*

that means, that $S(v_1)$ is a equal to or a subclass of the domain of $p$ and $S(v_2)$ is equal to or a subclass of its range. Furthermore, neither must $S(v_1)$ be restricted to not have a cardinality of 0 on the property $p$, nor must $S(v_2)$ contain a zero cardinality restriction on the inverse of $p$. A constraint's satisfiability can be examined by using the same auxiliary classes like in the process of query compilation, that was explained in Section 3.4. By creating the classes $S(v_1)' = S(v_1) \sqcap \exists p.S(v_2)$ and $S(v_2)' = S(v_2) \sqcap \exists p^{-}.S(v_1)$ and checking whether both of them can have any members. If at least one of them is equal to `owl:Nothing`, that is it cannot have members, the constraint is not satisfiable.

For the constraint `:c1` in the example query (see Figures 3.27 and 3.28), $p$ is `fam:hasChild`, $S(v_1)$ is `:Childless` and $S(v_2)$ is `:Child`. According to the previously explained procedure, the classes $S(v_1)'$ and $S(v_2)'$ would be created, $S(v_1)'$ being defined as `:Childless` $\sqcap \exists$`fam:hasChild.:Child`, which is exactly the class `:ChildlessParent` from Figure 3.29. As this class is already known to be equivalent to `owl:Nothing`, the whole constraint would be found to be unsatisfiable.

So far, this process takes care of a query's result variables' scopes and every individual constraint, but there are also queries, which are not satisfiable due to their combination of multiple constraints.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix fam: <http://family#>.
@prefix : <ex:pl#>.

[ a owlq:Query;
  owlq:resultVariable :A, :B;
  owlq:definesClass :Person;
  owlq:hasConstraint :c1, :c2 ].

:Person owlq:scopesVariable :A, :B;
  a owlq:Class;
  owl:equivalentClass fam:Person.

:c1 a owlq:Constraint; owlq:onVariable :A;
  owlq:onProperty fam:hasChild;
  owlq:equalsVariable :B.
:c2 a owlq:Constraint; owlq:onVariable :B;
  owlq:onProperty fam:hasChild;
  owlq:equalsVariable :A.
```

*Figure 3.30: Another unsatisfiable OWLQ query*

Figure 3.30 shows a query like that. Both of its variables `:A` and `:B` range over the satisfiable class `:Person` and each of the constraints is satisfiable too, because for both of them, $S(v_1)'$ would be `:Person` $\sqcap\, \exists$`fam:hasChild.:Person`, the people, who have one child at least, and $S(v_2)'$ would be the people, who are children of somebody else. What makes the query unsatisfiable in the end is the combination, requesting those tuples (`:A,:B`) where `:A` has a child `:B` and `:B` has a child `:A`.[3]

The query's unsatisfiability can be proved by contradiction. Assuming, that it is satisfiable, one tries to assign an individual to each of the variables and add a relation for each constraint. If the resulting ontology is deemed inconsistent, the query is not satisfiable. For the query from Figure 3.30, the results of the SPARQL extraction query are presented in Figure 3.31.

[3]This requires an ontology about family relationships, that defines `fam:hasChild` as an anti-symmetric property.

| S | CV | P | V1 | V2 |
|---|---|---|---|---|
| :Person | :A | | | |
| :Person | :B | | | |
| :c1 | | fam:hasChild | :A | :B |
| :c2 | | fam:hasChild | :B | :A |

*Figure 3.31: Information about the unsatisfiable query from Figure 3.30*

The resulting proof ontology is shown Figure 3.32. A reasoner can be utilized to show, that this ontology is not consistent, which means, that the query is unsatisfiable.

```
@prefix fam: <http://family#>.
@prefix : <ex:pl#>.

:A a :Person; fam:hasChild :B.
:B a :Person; fam:hasChild :A.
```

*Figure 3.32: Proving the unsatisfiability of the query from Figure 3.30*

## 3.6 Query Containment

Another statement that can be made about a query, is whether it is contained in another query. Let $q_1$ and $q_2$ queries against a database instance $D$ with the results $q_i(D), i \in \{1,2\}$. The query $q_1$ is 'contained' in $q_2$, if and only if $q_1(D) \subseteq q_2(D)$ holds for an arbitrary $D$. The examination of containment of conjunctive queries is NP-complete [5], so this section only describes a heuristic approach that checks a sufficient criterion for query containment. This approach is based on comparing the queries' variables and constraints and allows for examining containment by the means of the facilities that OWLQ provides.

Let $q_1$ and $q_2$ queries with result variables $V_{q_1,i}$ and $V_{q_2,j}$ respectively. Each of those variables depends on an `owlq:Class`. A variable depends on a class, if it either directly ranges over that class or, in the case of a variable with a literal value, if it originates from a variable definition or constraint on another variable, that ranges over the class. Let for a variable $V_{q_n,s}$, $C(V_{q_n,s})$ the class it is defined by. A necessary condition for $q_1 \subseteq q_2$ is, that for every $V_{q_1,i}$, there is a $V_{q_2,j}$ with $C(V_{q_1,i}) \sqsubseteq C(V_{q_2,j})$. For every literal-valued variable $V_{q_n,s}$ that is defined by an `owlq:VariableDefinition` or referenced by an `owlq:Constraint`, let $r(V_{q_n,s})$ denote the property of the definition or constraint. In this case, besides the foregoing condition, that there has to be a subset relation of the classes they depend on, also $\forall V_{q_1,i}\, \exists V_{q_2,j} \,.\, r(V_{q_1,i}) = r(V_{q_2,j})$ must hold.

```
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
@prefix : <ex:pl#>.

[ a owlq:Query; owlq:resultVariable :City1, :Population1;
  owlq:definesClass mon:City ].

mon:City a owlq:Class; owlq:scopesVariable :C.
:C owlq:hasVariableDefinition
     [ owlq:onProperty mon:name; owlq:toVariable :City1 ];
   owlq:hasVariableDefinition
     [ owlq:onProperty mon:population; owlq:toVariable :Population1 ].
```

```
@prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>.
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
@prefix : <ex:pl#>.

[ a owlq:Query; owlq:resultVariable :City2;
  owlq:definesClass mon:City ].

mon:City a owlq:Class; owlq:scopesVariable :C.
:C owlq:hasVariableDefinition
     [ owlq:onProperty mon:name; owlq:toVariable :City2 ].
```

*Figure 3.33: Query containment checking by variables*

Figure 3.33 shows two queries that are equal in their selection parts but differ in their projection portions. Obviously, the first query is not contained in the second one, as it also lists the cities' populations along with their names. This information is obtained by iterating over the first query's result variables (`:City1` and `:Population1`) and comparing their classes to all of the second query's variables' classes. First, `:City1` is considered, whose class is `mon:City`. For the second query, there is only one variable (`:City2`), whose class is also `mon:City`. Comparing those classes, one finds that `mon:City` $\sqsubseteq$ `mon:City`, so the first condition is met. As the variable `:City1` is results from a (literal-valued) variable definition on another variable, now it has to be checked, whether $r($`:City1`$) = r($`:City2`$)$. Both variables are defined on the property `mon:name`, so this condition is also true, so one can infer that `:City1` is contained in `:City2`.

For the other variable, `:Population1`, the same steps are taken. Again, there is only one variable from the second query to compare it to, namely `:City2` and again, both variables' classes are the same, so again the first condition is met. For second condition however, $r($`:Population1`$) = r($`:City2`$)$ does not hold, so the first query's variable `:Population1` does not have a counterpart in the second query. This leads to the conclusion that the first query can not be contained in the second one.

If the variable containment for two queries was checked to hold, the next step is the comparison of their constraints. For example, the example family query on Page 17 has one constraint `c1`, which requires `:P` to have a property `fam:hasChild` with the object `:C`. As the variables `:P` and `:C` range over the classes `:Parent` and `:Child`, the constraint can be expressed via `:Parent fam:hasChild :Child`. Generally spoken, every constraint can be expressed by $r(C1, C2)$, where $C1$ and $C2$ are the classes its variables range over and $r$ is its property. Now, for two queries $q_1$ and $q_2$, if all of $q_2$'s constraints are contained in $q_1$'s constraints, $q_1$ is a sub-query of $q_2$. The constraints have to be compared 'backwards' because $q_1$ can only be contained in $q_2$, if its constraints are at least equally restrictive.

Let $\alpha$ and $\beta$ constraints, that can be described by $r_1(C_1, D_1)$ and $r_2(C_2, D_2)$ respectively. Let $\alpha \subseteq \beta$, if $\alpha$ is less or equally restrictive as $\beta$. Figure 3.34 shows an example, in which, of course, `:alpha` is less restrictive than `:beta`, as `:alpha` only requires `:Y` to be a child of `:X`, whereas `:beta` requires `:Y` to be a *son* of `:X`.

```
:alpha a owlq:Constraint;
       owlq:onVariable :X;
       owlq:onProperty fam:hasChild;
       owlq:toVariable :Y.
```

```
:beta a owlq:Constraint;
      owlq:onVariable :X;
      owlq:onProperty fam:hasSon;
      owlq:toVariable :Y.
```

*Figure 3.34: An example for constraint containment*

In the case of non-negated constraints, a sufficient criterion that leads to $\alpha \subseteq \beta$ consists again of three criteria:

- $\forall x_1 : C_1(x_1) \rightarrow C_2(x_1)$ or $C_1 \sqsubseteq C_2$
  ($\beta$'s affected variable ranges over at least the same set as $\alpha$'s)
- $\forall x_2 : D_2(x_2) \rightarrow D_1(x_2)$ or $D_2 \sqsubseteq D_1$
  ($\beta$'s targeted variable ranges over at most the same set as $\alpha$'s)
- $\forall x_1, x_2 : r_2(x_1, x_2) \rightarrow r_1(x_1, x_2)$
  ($\beta$ restricts on a property at least as specific as $\alpha$'s)

The approach used by the OWLQ engine is assuming that $\alpha \subseteq \beta$ and trying to prove the opposite by contradiction. This proof involves creating two resources $x_1 \in C_1$ and $x_2 \in D_2$ with $r_2(x_1, x_2)$ and checking, whether adding the negations of all of the above criteria to the knowledge base leads to an inconsistency. The complete formula, whose satisfiability leads to the conclusion $\alpha \not\subseteq \beta$, is shown below.

$$(x_1 \in C_1 \ \wedge \ x_2 \in D_2 \ \wedge \ x_1 \, r_2 \, x_2) \wedge \big(x_1 \notin C_2 \ \vee \ x_2 \notin D_1 \ \vee \ \neg \, (x_1 \, r_1 \, x_2)\big)$$

If this formula is not satisfiable, $\alpha \subseteq \beta$ holds. Now, if for all of $q_2$'s constraints $\alpha_i$, there is a constraint $\beta_j$ in $q_1$ so that $\alpha_i \subseteq \beta_j$, under the premise that the variables' containment is given, then $q_1(D) \subseteq q_2(D)$ can be guaranteed.

For negated constraints, the first criterion for $\alpha \subseteq \beta$ is the same as for non-negated ones, but the second and third ones are inverse:

- $\forall x_1 : C_1(x_1) \rightarrow C_2(x_1)$
  ($\beta$'s affected variable ranges over at least the same set as $\alpha$'s)
- $\forall x_2 : D_1(x_2) \rightarrow D_2(x_2)$
  ($\beta$'s targeted variable ranges over at least the same set as $\alpha$'s)
- $\forall x_1, x_2 : r_1(x_1, x_2) \rightarrow r_2(x_1, x_2)$
  ($\beta$ restricts on a property at most as specific as $\alpha$'s)

As for the properties, if in the example, `:alpha` and `:beta` were negated, this time it were `:beta`, which were less restrictive that `:alpha`, because `:beta` would only require `:Y` not to be a son of `:X`, whereas `:alpha` would require `:Y` not to be a *child* of `:X` at all.

A similar example can be used for the second criterion. If for example both $\alpha$ and $\beta$ were negated constraints of the form `X: fam:hasChild :Y`, where for $\alpha$, `:Y` ranges over `fam:Son` and for $\beta$, `:Y` ranges over `fam:Child`, $\alpha \subseteq \beta$ would hold, as $\alpha$ would only exclude those pairs of `:X` and `:Y`, where `:Y` is a son of `:X` and in contrast, $\beta$ would exclude the pairs, where `:Y` is just a child of `:X`. This leads to the containment formula for $\alpha \not\subseteq \beta$ in the case of them being negated constraints, looks as follows:

$$(x_1 \in C_1 \ \wedge \ x_2 \in D_1 \ \wedge \ x_1 \, r_1 \, x_2) \wedge \big(x_1 \notin C_2 \ \vee \ x_2 \notin D_2 \ \vee \ \neg\,(x_1 \, r_2 \, x_2)\big).$$

## 3.7 The OWLQ Ontology

The complete OWLQ ontology is listed in this section. Figure 3.35 shows its class definitions and its predicates including their domains, ranges and cardinalities are given in Figure 3.36.

```
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix owl11:   <http://www.w3.org/2006/12/owl11#> .
@prefix :        <http://www.semwebtech.org/languages/2006/owlq#> .

:Query a owl:Class.
       rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :resultVariable;
                         owl:minCardinality 1 ].

:Scope     a owl:Class;
           owl11:disjointUnionOf (:Class :DataRange).

:Class     a owl:Class;
           rdfs:subClassOf owl:Class.

:DataRange a owl:Class;
           rdfs:subClassOf owl:DataRange.

:Variable  a owl:Class.
  rdfs:subClassOf [ a owl:Restriction;
                    owl:onProperty :varDefinedBy;
                    owl:cardinality 1 ].

:VariableDefinition a owl:Class;
  rdfs:subClassOf [ a owl:Restriction; owl:onProperty :onProperty;
                    owl:allValuesFrom owl:DatatypeProperty ],
                  [ a owl:Restriction; owl:onProperty :onProperty;
                    owl:cardinality 1 ],
                  [ a owl:Restriction; owl:onProperty :toVariable;
                    owl:cardinality 1 ].

:Constraint a owl:Class;
  rdfs:subClassOf [ a owl:Restriction; owl:onProperty :onVariable;
                    owl:cardinality 1 ],
                  [ a owl:Restriction; owl:onProperty :onProperty;
                    owl:cardinality 1 ],
                  [ a owl:Restriction; owl:onProperty :equalsVariable;
                    owl:cardinality 1 ].

:NegatedConstraint a owl:Class;
                   rdfs:subClassOf :Constraint.

:ClosedClass       a owl:Class;
                   rdfs:subClassOf :Class.

:ClosedPredicate   a owl:Class;
                   rdfs:subClassOf rdf:Property.
```

*Figure 3.35: The OWLQ ontology: OWLQ classes*

```
:definesClass       a rdf:Property;
                    rdfs:domain :Query; rdfs:range :Class.

:hasConstraint      a rdf:Property;
                    rdfs:domain :Query; rdfs:range :Constraint.

:usesVariable       a rdf:Property;
                    rdfs:domain :Query; rdfs:range :Variable.

:resultVariable     a rdf:Property;
                    rdfs:subPropertyOf :usesVariable.

:varDefinedBy       a owl:FunctionalProperty;
                    rdfs:domain :Variable.

:rangesOver         rdfs:subPropertyOf :varDefinedBy;
                    rdfs:range :Scope.

:varDefinedByVarDef rdfs:subPropertyOf :varDefinedBy;
                    rdfs:range :VariableDefinition.

:scopesVariable     a owl:InverseFunctionalProperty;
                    owl:inverseOf :rangesOver.

:hasVariableDefinition a rdf:Property;
  rdfs:domain :Variable; rdfs:range :VariableDefinition.

:hasMandatoryVariableDefinition a rdf:Property;
  rdfs:subPropertyOf :hasVariableDefinition.

:hasOptionalVariableDefinition a rdf:Property;
  rdfs:subPropertyOf :hasVariableDefinition.

:toVariable a owl:FunctionalProperty;
            owl:inverseOf :varDefinedByVarDef.
            rdfs:domain   :VariableDefinition; rdfs:range :Variable.

:onProperty a owl:FunctionalProperty;
            rdfs:domain [ owl:unionOf (:Constraint
                                       :VariableDefinition) ] .
            rdfs:range  rdf:Property.

:onVariable a owl:FunctionalProperty;
            rdfs:domain :Constraint; rdfs:range :Variable.

:equalsVariable a owl:FunctionalProperty;
                rdfs:domain :Constraint; rdfs:range :Variable.
```

*Figure 3.36: The OWLQ ontology (cont'd): OWLQ predicates*

# 4 Implementing the OWLQ Engine

While in the previous chapter, the theoretical background of the engine's features was presented, this chapter describes the engine's actual implementation. Figure 4.1 outlines the basic package structure and shows the implementation's classes and how they are connected.

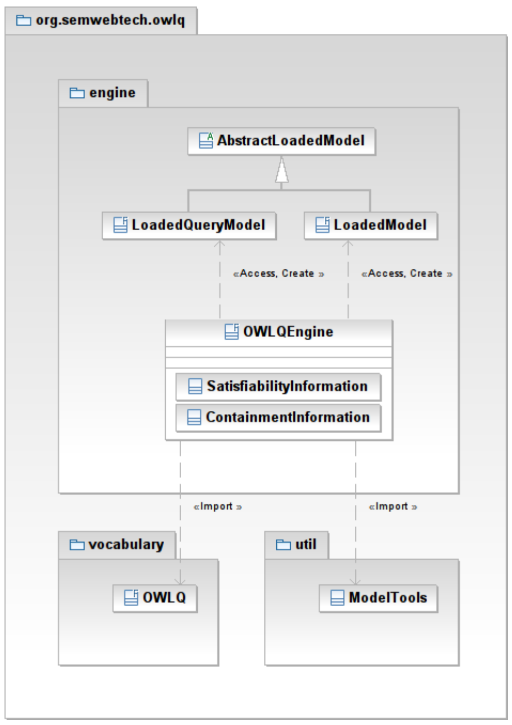


*Figure 4.1: The package structure of the OWLQ engine implementation*

## 4.1 Managing of Models and Queries

For applications that want to make use of the OWLQ package, the `OWLQEngine` is the contact point. It offers methods to add, remove and retrieve ontologies and queries and to execute and analyze queries. To support the handling of multiple ontologies and queries, the OWLQ engine keeps two lists each of which being a typed `java.util.LinkedList`. Those lists' elements are of type `LoadedModel` and `LoadedQueryModel` respectively, which are classes that extend the functionality of Jena's `Model` class. Those classes make the engine able to transparently operate on data and queries given as either objects of Jena's `Model` class or string representations in N3 or RDF/XML.

Both those classes extend the abstract class `AbstractLoadedModel`, which is a structure that basically offers access to an ontology's `Model` and `String` representations. It offers two constructor methods to create an instance from either a `Model` or a `String` (that is e.g. the RDF/XML oder N3 representation of the model). To create the respective other representation, the constructors use the functions `modelToString()` and `stringToModel()` from the `ModelTools` class. Furthermore, the `AbstractLoadedModel` class offers some methods that return information about the model such as its name, its string representation in several languages and whether it was created from a `String` or a `Model`.

For data ontologies, there is the `LoadedModel` class, that extends its abstract super-class by the possibility to query only its metadata, for the extraction of which it uses the `extractMetadata()` method from the `ModelTools` class.

Queries are held inside the `LoadedQueryModel` structure, which also extends the `AbstractLoadedModel` class. In addition to its inherited functions, it has several methods which are needed for the analysis and execution of queries and the extraction and retrieval of the queries' OWL and SPARQL portions among others.

A common task would be the registration of one or more ontologies and a query, followed by asking for the results of the query, which would be done as it is shown in Figure 4.2.

The first three lines add two models and a query, each of them being a `Model` or string read from a file[1]. For the purpose of registering models, the engine

[1]The methods `readFileIntoString()` and `readFileIntoModel()` are not part of the OWLQ engine but are supposed to be implemented in the application itself.

```
1  OWLQEngine.addModel(readFileIntoModel("family-facts.n3"), null);
2  OWLQEngine.addModel(readFileIntoString("family.n3"),
       "Family ontology");
3  OWLQEngine.addQuery(readFileIntoString("family-query.owlq"), null);
4
5  System.out.println("The query's OWL part is");
6  OWLQEngine.getOwlPart(0).write(System.out, "N3");
7
8  System.out.println("The query's SPARQL part is");
9  System.out.println(OWLQEngine.getSparqlPart(0)));
10
11 System.out.println("Results for "+OWLQEngine.getQueryName(0)+":");
12 OWLQEngine.outputQueryResults(0, System.out);
```

*Figure 4.2: Executing an OWLQ query from an external application*

offers the method `addModel()`. The first argument is either a `Model` (see Line 1) or a model string (see Line 2), whose language is tried to be inferred by the `guessLanguage()` function of the `ModelTools` class. The second argument is a name for the model, which will be generated if `null`. Queries are added to the engine via its method `addQuery()`, which takes the query string and an optional name as its arguments.

In Line 6, a model of the query's OWL portion is asked for and output. The method `getOwlPart()` is also overloaded. It can take an `int`, which denotes the index of the model of interest, and return a `Model` or it takes the `int` and a `String` stating the desired language such as *"N3"* or *"RDF/XML"* and returns a model string.

The SPARQL part is queried in Line 9 via the method `getSparqlPart()`, whose `int` argument is the index of the relevant query.

Both of these two methods return the respective portion of the *compiled* query. They are basically useful for testing purposes, as one can also directly execute an OWLQ query like in Line 12 via the functions `getQueryResults()` and `outputQueryResults()`. The process of executing a query will be described in detail in Section 4.3.

## 4.2 Compiling Queries

For most of the operations that the OWLQ engine is able to perform on queries, it uses their compiled versions. The advantages of using those queries and the theoretical compilation process were discussed in Section 3.4 and the engine's respective function is shown in Figure 4.3.

```
 1  private Model compileQuery() {
 2    Model newQuery = ModelFactory.createDefaultModel();
 3    Model oldQuery = getModel();
 4    newQuery.add(oldQuery);
 5    newQuery.setNsPrefixes(getModel());
 6    newQuery.setNsPrefix("rdf", RDF.getURI());
 7
 8    ResultSet constraintInformation =
        getConstraintInformationForDynamicClasses(newQuery);
 9    if (constraintInformation.hasNext()) {
10      HashMap<Resource, OntClass> specializedClasses = new
          HashMap<Resource, OntClass>();
11      while (constraintInformation.hasNext()) {
12        processConstraint(newQuery, specializedClasses,
            constraintInformation.nextSolution());
13      }
14    }
15    removeObsoleteConstraints(newQuery,
        ResultSetFactory.copyResults(constraintInformation));
16    return newQuery;
17  }
```

*Figure 4.3: The method* `compileQuery()`

This function is responsible for generating the compiled version of the query. For this purpose, first it creates a copy of the original query's model. Line 8 is a call to the function `getConstraintInformationForDynamicClasses()`, that extracts some information about the query's constraints. This function will be described below. If the query has any constraints, the function instantiates a `HashMap`, whose purpose will also become clear later. The generation of the

OWL restriction classes out of the query's constraints and the modification of the query happens in Line 12, where the method `processConstraint()` is called for each of the query's constraints. Finally, those of the query's constraints which have become obsolete in the compiled version are removed in Line 15.

The `getConstraintInformationForDynamicClasses()` function executes the SPARQL select query, which is shown in Figure 4.4, against the OWLQ query model. This query's purpose is to extract for each of a query's constraints its affected variables, the classes or data ranges that scope those variables, the constraint's property and the `owlq:Query` resource, that the constraint belongs to.

```
PREFIX  owlq: <http://www.semwebtech.org/languages/2006/owlq#>
SELECT ?Query ?Constraint ?Var1 ?Var1Scope ?Property ?Var2 ?Var2Scope
WHERE
  { ?Var1Scope  owlq:scopesVariable  ?Var1.
    ?Var2Scope  owlq:scopesVariable  ?Var2.
    ?Query      owlq:hasConstraint   ?Constraint
    ?Constraint owlq:onVariable      ?Var1;
                owlq:onProperty      ?Property;
                owlq:equalsVariable  ?Var2 .
    OPTIONAL
    { ?Constraint a ?ConstraintType .
       FILTER ( ?ConstraintType = owlq:NegatedConstraint )
    }
    FILTER ( ! bound(?ConstraintType) )
  }
```

*Figure 4.4: Extracting information about constraints (SPARQL version)*

### 4.2.1 Processing the Constraints into Class Definitions

The function `processConstraint()` extracts the resources from each of the result set's entries and prepares them for further processing. The method is shown in Figure 4.5.

From the given `QuerySolution` object, the method extracts the query in which the constraint occurs, the variable resources that the constraint refers to, the scopes, that those variables range over and the constraint's property. The scopes are then looked up in the `specializedClasses` map, which for every class that was generated as a restriction out of a constraint, contains a mapping of the affected variable to the new class. This ensures, that if multiple constraints state a restriction on the same variable, after having worked off the first of them, the second one restricts the class that was created when processing the first one and so on, so that all of the restrictions are merged in the last class. For both variables, if the map contains an entry for the variable, which means, that the associated class was already restricted, this new class is assigned to the `var1Scope` and `var2Scope` fields respectively. This leads to the effect, that the two fields are assigned the most specialized scope that the variables range over.

As the next step, the method `hopsToNextResultVariable()` checks for both of the constraint's variables, how many intermediate constraints the shortest link between the variable and the next of the query's result variables consists of. This is achieved by doing a breadth-first search starting from each of the result variables to the variables they are connected to by constraints. In the next step, the search is started from those variables and so on. The number of iterations before the variable `var1` or `var2` is reached, is returned.

Still, this method has just some preparatory tasks. The real process of OWL class generation is delegated to the method `createRestrictionClass()` in Lines 25 and 27, respectively. As already mentioned in Section 3.4, that restriction class is to be generated for the variable that is 'closer' to one of the query's result variables. Regardless of that, if one of the constraint's variables ranges over an `owlq:DataRange`, the engine should restrict the other scope. Therefore, in Line 24, the engine checks, which variable's scope is to be restricted and has the according restriction class created. The method `definesDatatype()` just looks up the given `Resource` in all ontologies' metadata and returns `true` if it is an `owl:DataRange`.

```java
private void processConstraint(Model newQuery,
    HashMap<Resource, OntClass> specializedClasses,
    QuerySolution constraintInformation) {
  OntModel pelletModel = ModelFactory.createOntologyModel(
      PelletReasonerFactory.THE_SPEC);
  pelletModel.add(getOwlPart(false));
  pelletModel.add(OWLQEngine.getAllModelMetadata());
  Resource query = constraintInformation.getResource("Query");
  Resource var1 = constraintInformation.getResource("Var1");
  Resource var2 = constraintInformation.getResource("Var2");
  Resource var1Scope = constraintInformation.getResource("Var1Scope");
  Resource var2Scope = constraintInformation.getResource("Var2Scope");
  OntResource propertyRes = pelletModel.getOntResource(
      constraintInformation.getResource("Property"));
  if (propertyRes == null || !propertyRes.isProperty()) {
    return;
  }
  OntProperty property = propertyRes.asProperty();
  if (specializedClasses.containsKey(var1)) {
    var1Scope = specializedClasses.get(var1);
  }
  if (specializedClasses.containsKey(var2)) {
    var2Scope = specializedClasses.get(var2);
  }
  int hopsToNextResultVariableFromVar1 =
      hopsToNextResultVariable(newQuery, query, var1);
  int hopsToNextResultVariableFromVar2 =
      hopsToNextResultVariable(newQuery, query, var2);

  if (hopsToNextResultVariableFromVar2 <
      hopsToNextResultVariableFromVar1 &&
      !definesDatatype(var2Scope)) {
    createRestrictionClass(newQuery, var2Scope, var2, var1Scope, var1,
         property, true, query, specializedClasses);
  } else {
    createRestrictionClass(newQuery, var1Scope, var1, var2Scope, var2,
         property, false, query, specializedClasses);
  }
}
```

*Figure 4.5: Checking whether a constraint can restrict a class*

The method `createRestrictionClass` (see Page 61) uses a temporary ontology model `tmpModel`, in which the classes are generated. As one can already see in Figure 4.5, the function is used to create the restricted class for either of the variables that occur in a constraint, depending on its boolean parameter `inverse`. If set to false, it derives a class from `classToRestrict`, that has a property `theProperty`, whose target value is a member of `targetClass`. If set to true, the resulting class is an intersection of `classToRestrict` and anything that has a property, which is the inverse of `theProperty`, with a value from `targetClass`. For given variables $a \in C_1$ and $b \in C_2$ and a property $r$, that means, `inverse` set to `false` results in the new class $C' \equiv C_1 \sqcap \exists\, r.C_2$, whereas when it is set to `true`, the function generates the class $C' \equiv C_1 \sqcap \exists\, r^-.C_2$.

The restriction ($\exists\, p.C_2$, where $p$ is $r$ or $r^-$) is made in Line 14 and the final intersection class ($C' \equiv C_1 \sqcap \exists\, p.C_2$) is created in Line 17. The generated class would be useless if it was describing the same class that it restricts, so this is checked in Line 19. If the classes are not equivalent, the new class is added to the compiled query's model in Line 20 and 21. To make use of that class, now, the query has to be modified to have the concerned variable range over that class instead of `classToRestrict`. This is done by the function `adaptQuery()`, which is called in Line 22. Finally, in Line 23, the new class is added to the `specializedClasses` map.

The method `adaptQuery()` is used to have the newly-generated restriction class scope the variable instead of the original one by removing the statement `:classToRestrict owlq:scopesVariable :scopedVariable` or its equivalent `:scopedVariable owlq:rangesOver :classToRestrict` from the query and adding `:dynamicClass owlq:scopesVariable :scopedVariable` instead. Additionally, the function also updates all of the query's previously-generated restriction classes that contain an `owl:someValuesFrom :classToRestrict` to refer to the new class (`:dynamicClass`) instead.

```
private void createRestrictionClass(Model newQuery,
    Resource targetClass, Resource targetVariable,
    OntProperty theProperty, boolean inverse, Resource theQuery,
    HashMap<Resource, OntClass> specializedClasses) {
  OntModel tmpModel = ModelFactory.createOntologyModel(
      PelletReasonerFactory.THE_SPEC);
  SomeValuesFromRestriction dynamicRestriction;
  String className;
  OntProperty property;
  if (!inverse || theProperty.isSymmetricProperty()) {
    property = theProperty;
  } else {
    if ((property = theProperty.getInverse()) == null) {
      property =
          tmpModel.createOntProperty(theProperty.getNameSpace() +
          "inverseOf" + property.getLocalName());
      property.setInverseOf(theProperty);
    }
  }
  dynamicRestriction = tmpModel.createSomeValuesFromRestriction(null,
      property, targetClass);
  className = className = scopedVariable.getURI() + "-" +
      classToRestrict.getLocalName() + property.getLocalName() +
      targetClass.getLocalName();
  RDFList intersectionList = tmpModel.createList(new RDFNode[] {
      classToRestrict, dynamicRestriction });
  OntClass dynamicClass = tmpModel.createIntersectionClass(className,
      intersectionList);
  dynamicClass.addRDFType(OWLQ.Class);
  if (!dynamicClass.hasEquivalentClass(classToRestrict)){
    newQuery.add(tmpModel.getBaseModel());
    newQuery.add(theQuery, OWLQ.definesClass, dynamicClass);
    adaptQuery(newQuery, scopedVariable, classToRestrict,
        dynamicClass, targetVariable);
    specializedClasses.put(scopedVariable, dynamicClass);
  }
}
```

*Figure 4.6: Generating the restricted classes*

### 4.2.2 Deleting the Unneeded Constraints

And now for the last function in the process of query compilation. Figure 4.7 shows the method `removeObsoleteConstraints()` that is responsible for removing those of a query's constraints that have become obsolete during the query's compilation. The approach, that is utilized to find those constraints was described in Section 3.4.2.

The method has a `Model` of the query's result variables and constraints generated by the method `findObsoleteConstraints()` in Line 8. This model contains resources for each of the query's variables and constraints. The result variables belong to the class `<aux://ResultVariable>` and for each constraint `:c` that is involved in chain of constraints that constitute a connection between two variables `:v1` and `:v2`, the statements `:c <aux://linksFrom> :v1` and `:c <aux://linksTo> :v2` are added to the model.

Within Lines 10–23, those constraints, which occur in a link of two (not necessarily distinct) result variables, are retrieved from the model and added to the list `neededConstraintList`. Now the method iterated over all of the query's constraints. For each one, that is not contained in the `neededConstraintList`, its definition and reference in the query are deleted. If a constraint was removed from the query, the boolean field `obsoleteConstraintRemoved` of the `LoadedQueryModel` is set to true, so that the resulting SPARQL query can be set `DISTINCT` later on.

After going through those steps, the compiled query is ready and stored inside a field of the `LoadedQueryModel` for further access.

```java
private void removeObsoleteConstraints(Model theQuery,
    ResultSet constraintInformation) {
  LinkedList<Resource> queryConstraints = new LinkedList<Resource>();
  OntModel query = ModelFactory.createOntologyModel(
      PelletReasonerFactory.THE_SPEC, theQuery);
  OntModel model = ModelFactory.createOntologyModel(
      PelletReasonerFactory.THE_SPEC);
  ObjectProperty linksFrom =
      model.createObjectProperty("aux://linksFrom");
  ObjectProperty linksTo =
      model.createObjectProperty("aux://linksTo");
  OntClass resultVariable = model.createClass("aux://ResultVariable");
  findObsoleteConstraints(constraintInformation, queryConstraints,
      query, model, linksFrom, linksTo, resultVariable);

  String neededConstraintsQueryString =
        "SELECT ?Constraint\n"
      + "WHERE\n"
      + "  { ?Constraint <" + linksFrom + "> ?Var1;\n"
      + "                <" + linksTo + ">   ?Var2.\n"
      + "    ?Var1 a <" + resultVariable + ">.\n"
      + "    ?Var2 a <" + resultVariable + ">.\n"
      + "  }";
  Query neededConstraintsQuery =
      QueryFactory.create(neededConstraintsQueryString);
  ResultSet neededConstraints = OWLQEngine.executeQuery(
      neededConstraintsQuery, model);
  LinkedList<Resource> neededConstraintList =
      new LinkedList<Resource>();
  while (neededConstraints.hasNext()) {
    neededConstraintList.add(
        neededConstraints.nextSolution().getResource("Constraint"));
  }
  for (Resource constraint : queryConstraints) {
    if (!neededConstraintList.contains(constraint)) {
      query.removeAll(constraint, null, null);
      query.removeAll(null, OWLQ.hasConstraint, constraint);
      obsoleteConstraintRemoved = true;
    }
  }
  model.close();
}
```

*Figure 4.7: Removing a query's obsolete constraints*

## 4.3 Query Execution

As one of the simple tasks, the OWLQ engine supports the execution of queries against RDF data. The result of an execution can be obtained through either the `getQueryResults()` method, which returns it as a `String` or through `outputQueryResults()`, which writes the result directly to the `OutputStream` that is given as a parameter. These functions are basically wrappers around the `getResult()` method of the `LoadedQueryModel` class, which is shown in Figure 4.8.

```java
protected ResultSet getResult() {
  if (OWLQEngine.cacheQueryResults && cachedResult != null) {
    return cachedResult;
  } else {
    ResultSet result;
    OntModel model = OWLQEngine.getModelOfAllModels();
    model.add(getOwlPart(true));
    Query qu = QueryFactory.create(getSparqlPart());
    closePredicates(model);
    closeClasses(model);

    if (OWLQEngine.cacheQueryResults) {
      cachedResult = OWLQEngine.executeQuery(qu, model);
      result = cachedResult;
      model.close();
    } else {
      QueryExecution qexec = QueryExecutionFactory.create(qu, model);
      result = qexec.execSelect();
    }
    return result;
  }
}
```

*Figure 4.8: The method* `getResult()`

The OWLQ engine supports the caching of results of queries. That speeds up repeatedly answering the same query as the results only have to be computed once and can be returned whenever needed. This method's drawback is the time that the engine needs to create an in-memory copy of the result set, so if the queries are predictable to be executed one time only, the engine performs better when the caching facility is disabled. This behavior can be set using the engine's `cacheQueryResults()` method that takes a `boolean` argument to enable and disable the result caching, respectively.

The `LoadedQueryModel` object has a field `cachedResult`, which contains the result of the query's last execution if the engine's result caching is enabled. In this case, the field is reset if there are changes to the loaded data ontology models. At first, the engine checks whether the caching of results is enabled and the `cachedResult` field already contains the needed information (Line 2) and, in this case, returns them. In the other case, beginning at Line 4, either caching is disabled or the results have not been computed yet.

To compute a query's result, at first, an ontology of all loaded models is created by the function `getModelOfAllModels()`. This model is then extended by the query's OWL part in Line 7. The boolean parameter to the `getOwlPart()` method states, whether or not the compiled query is to be returned (see Section 4.2). Line 8 creates a `Query` object from the query's SPARQL part. Both the methods `getOwlPart()` and `getSparqlPart()` are wrappers that check whether the OWL and SPARQL portions have already been computed and can be returned. If that is not the case, they call the functions `owlFromOwlq()` and `extractSparqlQuery()` respectively.

In the next step, if present, the query's closed classes and predicates are processed in Lines 9 and 10 respectively. The methods `closePredicates()` and `closeClasses()`, which accomplish these tasks, will be explained later.

The final steps depend on whether result caching is enabled or not. If caching is enabled, Line 13 executes the query against the model and assigns the execution's result to the `cachedResult` field. If caching is disabled, the result is just computed in Line 18. Finally, the result is returned in Line 20.

### 4.3.1 Extracting the OWL Class Definitions

The OWL class definitions of an OWLQ query are extracted by the method `owlFromOwlq()` as shown in Figure 4.9.

```java
private Model owlFromOwlq(Model model) {
  Model newModel = ModelFactory.createDefaultModel();
  newModel.setNsPrefixes(model);
  StmtIterator iter = model.listStatements();
  while (iter.hasNext()) {
    Statement statement = iter.nextStatement();
    RDFNode object = statement.getObject();
    if (object.toString().startsWith( OWLQ.getURI())) {
      if (object.equals(OWLQ.DataRange)) {
        newModel.add(statement.getSubject(), statement.getPredicate(),
             OWL.DataRange);
      } else if (object.equals(OWLQ.Class)) {
        newModel.add(statement.getSubject(), statement.getPredicate(),
             OWL.Class);
      }
      continue;
    }
    Property p = statement.getPredicate();
    if (p.getNameSpace().equals(OWLQ.getURI())) {
      continue;
    }
    newModel.add(statement);
  }
  return newModel;
}
```

*Figure 4.9: Extracting the OWL portion from an OWLQ query*

As already depicted in Section 3.2.1, the OWL part is extracted out of the OWLQ query by iterating over all of the query's statements and copying them into the OWL ontology if neither their predicate nor their object belong to the `owlq` namespace.

At first, the function first creates a new `Model` and then iterates over the query's statements in Lines 5–21. Inside the loop, first, the statement's object is considered (Lines 7–15). If it is an OWLQ resource the loop continues with the next statement. As special cases, if the object is `owlq:DataRange`, it is replaced by `owl:DataRange` (Line 10), and if the object is `owlq:Class`, is is replaced by `owl:Class` and the resulting statement is added to the new query. The same is done with the predicate in Lines 16–19. Finally, the remaining statements are added to the OWL model (Line 20), which is returned in Line 22.

### 4.3.2 Extracting the SPARQL Portion

The remaining part is the extraction of the SPARQL query from the original OWLQ query. This is done by the function `extractSparqlQuery()` shown in Figure 4.10, which does exactly what is described in Section 3.2.2.

```
 1 private String extractSparqlQuery(Model model) {
 2   Query sparqlQuery = new Query();
 3   sparqlQuery.setQuerySelectType();
 4   sparqlQuery.setPrefixMapping(model);
 5   ElementGroup elg = new ElementGroup();
 6   ResultSet results = extractInformationForSparqlExtraction(model);
 7   while (results.hasNext()) {
 8     QuerySolution solution = results.nextSolution();
 9     processResultEntry(elg, solution);
10   }
11   sparqlQuery.setQueryPattern(elg);
12
13   LinkedList<String> resultVariables = extractResultVariables(model);
14   for (String variable : resultVariables) {
15     sparqlQuery.addResultVar(variable);
16   }
17   if (obsoleteConstraintRemoved) {
18     sparqlQuery.setDistinct(true);
19   }
20   return sparqlQuery.toString();
21 }
```

*Figure 4.10: Building the SPARQL query*

The method creates a new `Query` object which is stated to become a select query and is configured to have the same namespaces as the original OWLQ query. The call in Line 6 executes the SPARQL query from Figure 3.4 against the OWLQ query, which returns a `ResultSet`, that contains all the needed information.

This information has to be processed into an `ElementGroup`, that contains the SPARQL query's statements. This processing is done in Lines 7–10. The while loop iterates over all entries in the result set and has them processed, in

Line 9, by the `processResultEntry()` function, which will be explained later. In Line 11, the query's pattern is set from the `ElementGroup`. This concludes the construction of the query's selection part.

What is missing is the projection part which is assembled in Lines 13–16. The `extractResultVariables()` function, which will also be described later, is used to extract the result variables from the query model. Those are then added to the list of the SPARQL query's result variables. If, during the query's compilation, there were constraints removed from the query, it is set `DISTINCT` to prevent the same result being output multiple times (Lines 17–19). Finally the new query is transformed to a `String` and returned in Line 20.

Consider again the example result set in Figure 3.5 and the explanations about the result set's structure on Page 21. As for the processing of the result entries (Line 9 of the `extractSparqlQuery()` function), obviously, there are three possible structures for each entry. Treating negative constraints separately, that makes four cases. Either, the row describes a class, which means, that `CV` is not `null` (case A). In the other case, there are again two possibilities. Either, the line has a value in the column `S`, which would mean, that it corresponds to a constraint, whose name is given in column `S` (case B.a). Or, if `S` is not set, the entry describes a variable definition (case B.b). As there are positive and negative constraints, this creates again two cases B.a+ and B.a-, respectively.

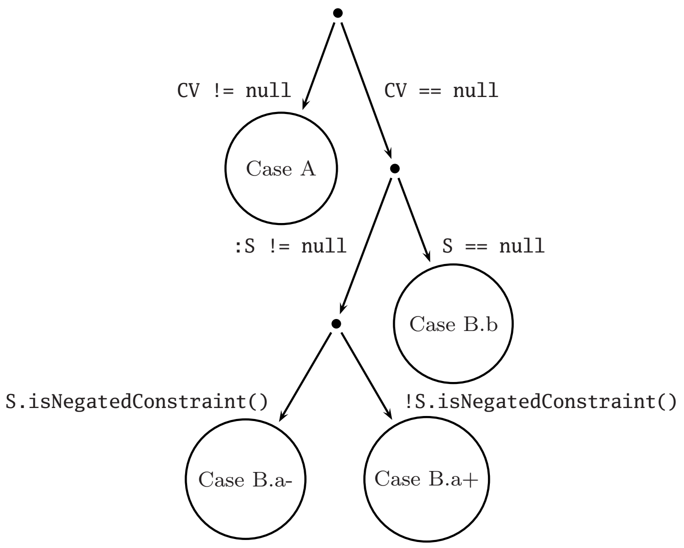


Figure 4.11 shows the function `processResultEntry()`, which is responsible for transforming the `QuerySolution` objects into triples for the SPARQL query.

First, it extracts the answer resources from every entry in Lines 2–6. Then it distinguishes the four cases explained above. For every result that satisfies case A (a class and a scoped variable), it adds a triple `CV a S`. For the case B.b (variable definition) and B.a+ (positive constraint), which result in the same SPARQL structure, it adds a triple `CV P V1`.

For the negated constraints, that is those triples, that fall in case B.a-, the

```java
private void processResultEntry(ElementGroup elg,
    QuerySolution entry) {
  Resource S = entry.getResource("S");
  Resource CV = entry.getResource("CV");
  Resource P = entry.getResource("P");
  Resource V1 = entry.getResource("V1");
  Resource V2 = entry.getResource("V2");
  if (CV != null) {                      /*case A*/
    Node subject = CV.asNode();
    elg.addTriplePattern(new Triple(Var.alloc(subject.getLocalName()),
         RDF.type.asNode(), S.asNode()));
  } else {                               /*case B*/
    Var v1Var = Var.alloc(V1.asNode().getLocalName());
    Var v2Var = Var.alloc(V2.asNode().getLocalName());
    if (S == null || !S.hasProperty(RDF.type,
        OWLQ.NegatedConstraint)){     /*cases B.b and B.a+*/
      elg.addTriplePattern(new Triple(v1Var, P.asNode(), v2Var));
    } else {                             /*case B.a-*/
      Var additionalPropertyValue = Var.alloc(v1Var.getName() +
          P.getLocalName() + "NOT" + v2Var.getName());
      ElementGroup optGroup = new ElementGroup();
      optGroup.addTriplePattern(new Triple(v1Var, P.asNode(),
          additionalPropertyValue));
      Expr eqV2Expr = new E_Equals(
          new NodeVar(additionalPropertyValue), new NodeVar(v2Var));
      ElementFilter eqV2Filter = new ElementFilter(eqV2Expr);
      optGroup.addElement(eqV2Filter);
      ElementOptional elgOpt = new ElementOptional(optGroup);
      elg.addElement(elgOpt);
      Expr notBoundExpr = new E_LogicalNot(new E_Bound(
          new NodeVar(additionalPropertyValue)));
      ElementFilter notBoundFilter = new ElementFilter(notBoundExpr);
      elg.addElement(notBoundFilter);
    }
  }
}
```

*Figure 4.11: Generating the SPARQL query's selection statements*

engine implements the procedures that were explained in Section 3.3.1 and generates the `OPTIONAL` and `FILTER` elements in Lines 16–26.

The extraction of the result variables is implemented straightforwardly as can be seen in Figure 4.12. The function `extractResultVariables()` obtains an iterator over all the model's nodes, that are the objects of statements with the predicate `owlq:resultVariable` in Line 3. The `while` loop then iterates over those nodes and adds their names to the `LinkedList` of result variables, which is returned in Line 9.

```
 1 private LinkedList<String> extractResultVariables(Model model) {
 2   LinkedList<String> resultVariables = new LinkedList<String>();
 3   NodeIterator resultVariableIter =
         model.listObjectsOfProperty(OWLQ.resultVariable);
 4
 5   while (resultVariableIter.hasNext()) {
 6     Resource resultVariable = (Resource)
           resultVariableIter.nextNode().as(Resource.class);
 7     resultVariables.add(resultVariable.getLocalName());
 8   }
 9   return resultVariables;
10 }
```

*Figure 4.12: Extracting the result variables*

### 4.3.3 Processing Closures of Predicates and Classes

According to the procedures which were depicted in sections 3.3.2 and 3.3.3, the methods `closePredicates()` and `closeClasses()` process a query's closed predicates and classes respectively. Figure 4.13 shows the former of those functions.

The loop in Lines 4–18 iterates over a query's closed predicates. The respective property resource is obtained from the model and is passed to the method `getClassLimitsForPredicateClosure()` in Line 12, along with all of the model's `owl:Restriction` resources. In accordance with considerations on Page 30, that method examines all the restrictions on the given property grouped by their `owl11:onClass` attribute and returns a mapping of class resources to an `int` vector, containing the values of the least and greatest cardinalities, that are relevant for the predicate's closure. Then, for every entry in that mapping, the method `categorizeIndividualsForPredicateClosure()` is used to create the actual restriction classes and classify their prospective members by following the steps in Section 3.3.2.

```java
private void closePredicates(OntModel model) {
  Model query = getModel();
  ResIterator closedPredicates =
      query.listSubjectsWithProperty(RDF.type, OWLQ.ClosedPredicate);
  while (closedPredicates.hasNext()) {
    Resource closedPredicate = closedPredicates.nextResource();
    OntResource propertyRes = model.getOntResource(closedPredicate);
    if (propertyRes == null || !propertyRes.isProperty()) {
      return;
    }
    OntProperty property = propertyRes.asProperty();
    ExtendedIterator queryRestrictions =
        model.listIndividuals(OWL.Restriction);
    HashMap<Resource, int[]> limitsForClasses =
        getClassLimitsForPredicateClosure(property,
        queryRestrictions);
    for (Resource onClass : limitsForClasses.keySet()) {
      int startCardinality = limitsForClasses.get(onClass)[0];
      int endCardinality = limitsForClasses.get(onClass)[1];
      categorizeIndividualsForPredicateClosure(model, property,
          onClass, startCardinality, endCardinality);
      }
    }
  }
```

*Figure 4.13: Computing the closure of predicates*

```
private void closeClasses(OntModel model) {
  OntModel queryModel = ModelFactory.createOntologyModel(
      PelletReasonerFactory.THE_SPEC);
  queryModel.add(getModel());
  ExtendedIterator closedClasses =
      queryModel.listIndividuals(OWLQ.ClosedClass);
  while (closedClasses.hasNext()) {
    Resource closedClass = (Individual) closedClasses.next();
    RDFList classMembers =
        model.createList(model.listIndividuals(closedClass));
    model.add(closedClass, OWL.oneOf, classMembers);
  }
}
```

*Figure 4.14: Computing the closure of classes*

The method that is responsible for closing classes is shown in Figure 4.14. Every class that is defined as being an `owlq:ClosedClass` is extended by an `owl:oneOf` statement whose object is a list of all the individuals of that class.

## 4.4 Reasoning about Queries

While the last sections described the preparatory work and the mere execution of queries, this section describes, how the OWLQ engine was extended by reasoning capabilities and how a query can be inspected with regard to its satisfiability and containment in another query.

### 4.4.1 Satisfiability

A query's satisfiability can be checked via the `checkQuerySatisfiability()` method from the `OWLQEngine`, which is shown in Figure 4.15.

The method takes an `int` parameter, that denotes the query's index and returns a `SatisfiabilityInformation` object, which is defined in an inner class of the `OWLQEngine`. This structure comprises a list of unsatisfiable classes and constraints, the reason for the query's unsatisfiability and the respective setters and getters. In Lines 4–7, the query's unsatisfiable classes are extracted by the method **extractUnsatisfiableClasses()** from the LoadedQueryModel class, which returns the URIs of those classes, that are equal to **owl:Nothing** and which are then added to the respective **SatisfiabilityInformation** object, that is to be returned. The same is done for the query's constraints in Lines 9–12 according to the process explained in Section 3.6. The call to the method **extractUnsatisfiableConstraints()** in Line 9 creates the restriction classes that result from the constraints and also compare them to **owl:Nothing**. The unsatisfiable constraints, which are returned by the method are again added to the **SatisfiabilityInformation** object. The constraint combination is checked in line 14 by the method **getReasonForUnsatisfiability()**, which will be described in a moment, and Lines 15–16 assign the **String** returned by that function to the **SatisfiabilityInformation** object and return it afterwards.

```
 1  public static SatisfiabilityInformation checkQuerySatisfiability(
        int index) {
 2    SatisfiabilityInformation satInfo = new SatisfiabilityInformation();
 3
 4    LinkedList<String> unsatisfiableClasses =
          loadedQueryModels.get(index).extractUnsatisfiableClasses();
 5    for (String classURI : unsatisfiableClasses) {
 6      satInfo.addUnsatisfiableClass(classURI);
 7    }
 8
 9    LinkedList<String> unsatisfiableConstraints =
          loadedQueryModels.get(index).extractUnsatisfiableConstraints();
10    for (String constraintURI : unsatisfiableConstraints) {
11      satInfo.addUnsatisfiableConstraint(constraintURI);
12    }
13
14    String reasonForUnsatisfiability =
          loadedQueryModels.get(index).getReasonForUnsatisfiability();
15    satInfo.setReasonForUnsatisfiability(reasonForUnsatisfiability);
16    return satInfo;
17  }
```

*Figure 4.15: Checking a query's satisfiability*

While the extraction of a query's unsatisfiable classes and constraints, which are obtained in Lines 4 and 7 respectively, can be useful to find mistakes in the query formulation, they do not necessarily affect the satisfiability of the query as a whole. A query might very well define an unsatisfiable class, but the query can still be satisfiable, if none of its variables range over that class. So, as explained in Section 3.5, the combination of classes and constraints has to be checked to be able to tell whether a query is satisfiable or not. This analysis is made by the `LoadedModel` class' method `getReasonForUnsatisfiability()`. The method uses again the method `extractInformationForSparqlExtraction()` that executes the SPARQL query from Page 20. The `ResultSet` which is returned by that function can be used, according to the steps explained in Section 3.5, to create a model containing an individual for each variable and an assertion for each constraint. This model's `validate()` method is then used to examine its consistency. If the model is not valid which means that a database instance that satisfies the query cannot exist, the `getReasonForUnsatisfiability()` method returns the error message that can be obtained from the model. If the model is valid, the method returns `null`. So if the return value of `getReasonForUnsatisfiability()` is `null`, the query is satisfiable, otherwise, it is not.

### 4.4.2 Query Containment

In Section 3.6, the theoretical aspects of query containment were dealt with, now we go for the implementation. The `OWLQEngine` class offers the method `checkQueryContainments()` (Figure 4.16) to check whether one of its registered queries is contained in another one.

The method has an `int` parameter, which denotes the index of the query ($q$) that is to be examined. It uses the compiled version of $q$ and extracts information about its result variables and constraints in Lines 3–4 by executing the respective methods of the `LoadedQueryModel` class. The methods `getConstraintInformation()`and `getVariableInformation()` will be explained later on.

Now, the function iterates over all of the engine's registered queries but $q$ (Lines 6–17). Within the loop, the function `checkTwoQueriesForContainment()` is used to generate a `ContainmentInformation` object for each of the other queries. Like the `SatisfiabilityInformation` class, this structure is defined in an inner class of the engine and keeps information about the containment of the query $q$ in another, for example whether it is contained at all, which variables and constraints are matched in the other query and which are not. There will be more information about how this class is utilized in the following explanations about the `checkTwoQueriesForContainment()` method and in Chapter 5, which outlines the reference implementation of a servlet that uses the OWLQ engine. After having generated a `ContainmentInformation` object for each of the other queries, finally the method `checkQueryContainments()` returns them in an array.

```java
public static ContainmentInformation[] checkQueryContainments(
    int index) {
  LinkedList<ContainmentInformation> result =
      new LinkedList<ContainmentInformation>();
  ResultSet variablesInfo =
      loadedQueryModels.get(index).getVariableInformation();
  ResultSet constraintInfo =
      loadedQueryModels.get(index).getConstraintInformation();

  for (int i = 0; i < loadedQueryModels.size(); i++) {
    if (i == index) {
      continue;
    }
    Model proofModel = ModelFactory.createDefaultModel();
    proofModel.add(loadedQueryModels.get(index).getOwlPart(true));
    proofModel.add(loadedQueryModels.get(i).getOwlPart(true));

    ContainmentInformation contInfo =
        checkTwoQueriesForContainment(variablesInfo, constraintInfo,
        i, proofModel);
    proofModel.close();
    result.add(contInfo);
  }
  ContainmentInformation[] resultAsArray =
      new ContainmentInformation[result.size()];
  return result.toArray(resultAsArray);
}
```

*Figure 4.16: Checking whether a query is contained in another one*

As the analysis of query containment is based on comparing two queries' variables and constraints, those have to be extracted from the comparees. For that purpose, the `LoadedQueryModel` offers the functions `getVariableInformation()` and `getConstraintInformation()` respectively. Both of them return the results of executing a query against the respective OWLQ query. Those queries are shown in Figure 4.17.

The first query extracts information about a query's result variables and the classes on which they depend. Those variables can either directly range over a class, or represent a literal value and originate from a variable definition or constraint on a property of another variable, like in Figure 3.2 on Page 18. In this case, the query returns that class, that the other variable ranges over.

Information about the query's constraints is obtained via the second query. For each constraint, it returns the associated query resource, the constraint itself, the constraint's property and its variables along with either the scopes they range over, or the scopes of the variables they are defined by. All this information is then passed to the function `checkTwoQueriesForContainment()`, which is shown in Figure 4.18.

This method allocates a `ContainmentInformation` object and extracts the variables and constraints from the other query. This information is compared to the other query's variables and constraints in Lines 6 and 18 respectively. The methods `compareResultVariables()` and `compareConstraints()` both return a `HashMap`. In the case of `compareResultVariables()`, the returned map contains a mapping of $q$'s variables to the matching variables of the other query. The `compareConstraints()` method returns a mapping of the *other query's* constraints to $q$'s constraints. As the next step, the method loops over those mappings and adds the information to the `ContainmentInformation` object (see Lines 11, 14, 23 and 26). If a map contains an entry that is mapped to `null`, that means, that no match for the specific variable or constraint was found, this is kept track of by two `boolean` variables (Lines 12 and 24), which are evaluated in Line 29 for the final verdict concerning the containment. Finally the `ContainmentInformation` object is returned.

The result variables of the queries are compared with respect to containment by the method `compareResultVariables()`. It iterates over the result variables of $q$, which are passed to the function as an argument, and tries for each one to find a matching variable in the other query by looping over those. For each variable, it adds an entry in the map, its value being either the name of the

```
PREFIX  owlq: <http://www.semwebtech.org/languages/2006/owlq#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT DISTINCT  ?Variable ?Scope ?Property
WHERE
  { ?Query  owlq:resultVariable  ?Variable.
    ?Scope  a  owlq:Class.
    { ?Scope  owlq:scopesVariable  ?Variable.}
    UNION
    { ?Scope owlq:scopesVariable ?Var2
      { ?Constraint a owlq:Constraint;
                     owlq:onVariable       ?Var2;
                     owlq:onProperty       ?Property;
                     owlq:equalsVariable   ?Variable.
        ?Property  a owl:DatatypeProperty.
        OPTIONAL
        { ?Constraint a ?ConstraintType.
          FILTER ( ?ConstraintType = owlq:NegatedConstraint )
        }
        FILTER ( ! bound(?ConstraintType) )
      }
      UNION
      { ?Var2 owlq:hasVariableDefinition
        [ owlq:onProperty   ?Property;
          owlq:toVariable   ?Variable ]
      }
    }
  }
```

```
PREFIX  owlq: <http://www.semwebtech.org/languages/2006/owlq#>
SELECT  ?Query ?Constraint ?Var1 ?Var1Scope ?Property ?Var2 ?Var2Scope
WHERE
  {  { ?Var1Scope  owlq:scopesVariable  ?Var1.}
       UNION
      { ?Var1Scope  owlq:scopesVariable
         [ owlq:hasVariableDefinition
           [ owlq:toVariable ?Var1 ]
         ]. }
    ?Var2Scope  a  owlq:Class.
      { ?Var2Scope  owlq:scopesVariable  ?Var2.}
      UNION
      { ?Var2Scope  owlq:scopesVariable
          [ owlq:hasVariableDefinition
            [ owlq:toVariable ?Var2 ]
          ]. }
    ?Query       owlq:hasConstraint   ?Constraint.
    ?Constraint  owlq:onVariable      ?Var1;
                 owlq:onProperty      ?Property;
                 owlq:equalsVariable  ?Var2.
  }
```

*Figure 4.17: Obtaining a query's variables and constraints*

```java
private static ContainmentInformation
   checkTwoQueriesForContainment(ResultSet variables,
   ResultSet constraints, int index, Model proofModel) {
  ContainmentInformation contInfo = new ContainmentInformation(index);
  ResultSet otherVariablesInfo =
      loadedQueryModels.get(index).getVariableInformation();
  ResultSet otherConstraintInfo =
      loadedQueryModels.get(index).getConstraintInformation();

  HashMap<String, String> resultVariablesComparison =
      compareResultVariables(proofModel,
      ResultSetFactory.copyResults(variables),otherVariablesInfo);
  boolean variableWithNoCounterpartFound = false;
  for (String q1Variable : resultVariablesComparison.keySet()) {
    String relatedq2Variable =
        resultVariablesComparison.get(q1Variable);
    if (relatedq2Variable == null) {
      contInfo.addVariableWithoutCounterpart(q1Variable);
      variableWithNoCounterpartFound = true;
    } else {
      contInfo.addVariableMapping(q1Variable, relatedq2Variable);
    }
  }

  HashMap<String, String> constraintComparison =
      compareConstraints(proofModel,
      ResultSetFactory.copyResults(constraints), otherConstraintInfo);
  boolean constraintWithNoCounterpartFound = false;
  for (String q2Constraint : constraintComparison.keySet()) {
    String relatedq1Constraint =
        constraintComparison.get(q2Constraint);
    if (relatedq1Constraint == null) {
      contInfo.addConstraintWithoutCounterpart(q2Constraint);
      constraintWithNoCounterpartFound = true;
    } else {
      contInfo.addConstraintMapping(q2Constraint,
          relatedq1Constraint);
    }
  }
    if (!(constraintWithNoCounterpartFound ||
        variableWithNoCounterpartFound)) {
    contInfo.setContained(true);
  }
  return contInfo;
}
```

*Figure 4.18: Checking whether one query is contained in another*

other query's matching variable or `null`, if no match was found. So each of $q$'s variables is compared with each of the other query's variables. For checking, whether one variable matches another, the method `checkVariableContainment()` is used, which is shown in Figure 4.19.

```
 1 private static boolean checkVariableContainment(Resource v1,
       Resource c1, Property r1, Resource v2, Resource c2, Property r2) {
 2   if (r1 == null ^ r2 == null) {
 3     return false;
 4   } else {
 5     if (r1 != null && !r1.equals(r2)) {
 6       return false;
 7     }
 8   }
 9   if (!isSubclass(c1, c2)) {
10     return false;
11   }
12   return true;
13 }
```

*Figure 4.19: Checking two variables for containment*

In this method, two variables are checked for containment. In spite of this term being a bit fuzzy, because one variable $v_1$ can not be 'contained' in another variable $v_2$, we use it as a synonym for 'each of the possible assignments to $v_1$ is contained in the set of possible assignments to $v_2$'. However, the method gets two variables along with either the classes they range over or the classes they are defined by and the property on which they are defined (consider again the first query in Figure 4.17). The method returns `true` if and only if `v1` is contained in `v2`. For example one variable (`v1`) could range over a class `:City` (`c1`), and the other one (`v2`) could result from a variable definition on the property `mon:name` (`r2`) of a class `:River` (`c2`). In this case, of course variable containment would not be given, because of `r1` not being equal to `r2`, that they are neither both `null` nor do they represent the same property. The containment is confirmed, if and only if the properties are the same (including both of them being `null`) and the scope of `v1` is a subclass of the scope that `c2` ranges over (Line 9).

After having built the map of variable containment relationships, the engine proceeds with comparing the queries' constraints. The process resembles the variables' comparison, only that it is done by comparing the other query's constraints to $q$'s instead of the other way around (see Section 3.6). The engine loops over the other constraints in an outer loop and for each one, tries to find a counterpart in $q$ in the inner loop. As it only makes sense to check two constraints for containment, if they are either both negated or both not negated, only if this is the case, the pair is passed to the method `checkConstraintContainment()`, which is shown in Figure 4.20.

This method is responsible for comparing two constraints for containment. Its parameters are the resources, that describe the constraints, which is their properties and the scopes their variables range over, along with the `proofModel` from from the method `checkTwoQueriesForContainment()` (Figure 4.18) and a `boolean` parameter that denotes whether the constraints are negated or not. That method tries to prove the containment by contradiction, according to the containment formulae from pages 49 and 50, respectively, by proving that their respective complements lead to an inconsistency.

At first, two resources `x1` and `x2` are created and `x1` is made a member of `c1`. The next steps depend on whether the comparees are negated or not. For negated constraints, `x2` is made a member of `d1` and the relationship between `x1` and `x2` on the property `r1` is stated in Line 12. This concludes the leftmost half of the containment equation. For the other part, now the the restriction class `r2x2` is created as those resources, that have a predicate `r2`, with the object `x2` (Line 13). By creating that restriction class's complement in Line 14, `notr2x2` now represents the class of those resources, which do *not* have a predicate `r2`, with the object `x2`. The statement `s`, which is created in Line 15, denotes that `x1` belongs to that class, which would mean that $x_1\ r_2\ x_2$ does not hold, which resembles the rightmost part of the containment equation. Finally, Line 15 checks, whether `d2` is a subclass of `d1` and assigns the result to the variable `conditionForD1andD2`.

For non-negated constraints, the necessary steps are taken in Lines 18–23, adapted to the according containment formula.

The result is computed in Line 25. If the constraint, which is represented by `c1`, `d1` and `r1` is contained in the constraint denoted by `c2`, `d2` and `r2`, first of all, `c1` must be a subclass of `c2`, which in terms of the equations would cause $x_1 \notin C_2$ not to hold. If the subclass relationship is given, the next step is the subclass

```java
private static boolean checkConstraintContainment(Model inputModel,
    Resource c1, Property r1, Resource d1, Resource c2, Property r2,
    Resource d2, boolean isNegated) {
  boolean contained;
  OntModel model = getAllModelMetadata();
  model.add(inputModel);

  OntResource x1 = createIndividual(c1);
  OntResource x2 = model.createcreateOntResource(null);
  boolean conditionForD1andD2;
  Statement s;
  if (isNegated) {
    x2.addRDFType(d1);
    x1.addProperty(r1, x2);
    HasValueRestriction r2x2 = model.createHasValueRestriction(null,
        r2, x2);
    ComplementClass notr2x2 = model.createComplementClass(null, r2x2);
    s = model.createStatement(x1, RDF.type, notr2x2);
    conditionForD1andD2 = isSubclass(model.getBaseModel(), d1, d2);
  } else {
    x2.addRDFType(d2);
    x1.addProperty(r2, x2);
    HasValueRestriction r1x2 = model.createHasValueRestriction(null,
        r1, x2);
    ComplementClass notr1x2 = model.createComplementClass(null, r1x2);
    s = model.createStatement(x1, RDF.type, notRr1x2);
    conditionForD1andD2 = isSubclass(model.getBaseModel(), d2, d1);
  }
  contained = isSubclass(model.getBaseModel(), c1, c2) &&
      conditionForD1andD2 && !statementSatisfiable(model, s);
  model.close();
  return contained;
}
```

*Figure 4.20: Checking two constraints for containment*

relationship of d1 and d2, whose direction depends on whether the constraints are negated. Applied to the containment equations this would cause $x_2 \notin D1$ and $x_2 \notin D_2$ to be false respectively. For the constraint to be contained, the only thing left is its property, that is r2 must be a sub-property of r1 or vice versa. This is exactly the opposite of the statement s, so if s is satisfiable, that leads to the last operand being false. Should s not be satisfiable, all the necessary conditions for the constraint's containment are given and the method returns true in Line 27.

## 4.5 Performance

Most of the information that the OWLQ engine provides is almost instantly available. The computation times given here apply to a single query with an average complexity.

The following information is computed independently of the knowledge base, so the execution is performed in near-constant time and takes less than a second.

- compilation
- transformation to OWL and SPARQL

The computation time of the tasks listed below depends on the scale of the knowledge base.

- examination of satisfiability or containment in another query
  (within a second, uses only metadata)
- closure of classes
  (within a few seconds)
- closure of predicates
  (several minutes for cardinalities up to about 5, unfeasible for two-digit cardinalities)

The final execution of a query involves stating the extracted SPARQL portion against the knowledge base and and having the results computed by the Pellet reasoner. As the answering of SPARQL queries is not a feature that is specific for OWLQ, its computation is not affected by the engine.

# 5 Reference Implementation Example of an OWLQ Servlet

So far, the main features of the OWLQ engine have been presented and explained, now this chapter shows how to use the engine inside an example application. This is accomplished by means of a Java servlet for the Apache Tomcat Servlet/JSP container[1]. As it is meant to be a demonstration object, the example servlet is kept simple. It uses static HTML pages, but generates them from static strings and dynamic information about the models and queries used.

The servlet extends the abstract `javax.servlet.http.HttpServlet` class by implementing its two main functions `doGet()` and `doPost()` which are responsible to process HTTP GET and POST requests, respectively. Besides the servlet-specific classes, the servlet has to import the OWLQ engine itself and its two inner classes `ContainmentInformation` and `SatisfiabilityInformation`.

Providing access to the OWLQ engine, the servlet serves two purposes: modifying the engine's state in terms of adding and removing models and queries and presenting data that is output from the engine, such as results, containment or satisfiability information. Everything that is meant to modify the engine's data is done in POST requests and thus handled by the `doPost()` method, whereas information is obtained via GET requests which are processed by the function `doGet()`.

## 5.1 The doGet() Method

This method is responsible of processing GET requests to the servlet, which is done by first returning the HTML header, then, depending on which page was requested, delegating the construction of the HTML body to the servlet's

[1]The OWLQ servlet was written for version 6 of Tomcat, which is available through `http://tomcat.apache.org/download-60.cgi`.

respective function and finally outputting the HTML footer. The servlet is able to display four different pages, namely the *overview* page (`index.html`), one to *edit* models and queries, one to *view* information about queries, such as results or containment and an *error* page. The methods that generate those pages will be outlined in this section, however the listings is this chapter show slightly simplified versions of the real source code, which omit most of the HTML tags outputting.

### 5.1.1 Presenting an Overview

The servlet's `index.html` page is used to show an overview of the loaded models and queries and providing access to the OWLQ engine's functionality such as adding and removing models, examining a query's results, satisfiability or containment. The first part of this overview is generated by the code snipped shown in Figure 5.1.

```
String[] loadedQueriesNames = OWLQEngine.getLoadedQueryNames();
if (loadedQueriesNames.length > 0) {
  int i = 0;
  for (String queryName : loadedQueriesNames) {
    out.write("queryName + "<input type=\"submit\" value=\"Edit\"" +
      " name=\"editq" + i + "\"/>" + "<input type=\"submit\"" +
      " value=\"Remove\" name=\"removeq" + i + "\"/>");
    if (OWLQEngine.hasGeneratedClasses(i)) {
      out.write("<input type=\"submit\"" +
        " value=\"View generated classes\"" +
        " name=\"viewg + i + "\"/>");
    }
    out.write("<input type=\"submit\" value=\"View results\"" +
      " name=\"viewr" + i + "\"/>");
    i++;
  }
}
out.write("<input type=\"submit\" name=\"addq\"" +
  " value=\"Add query\"/><br/>");
```

*Figure 5.1: Generating an overview over the loaded queries*

At first the servlet accesses the OWLQ engine in Line 1 to ask about the loaded queries and prints each of their names in a table along with buttons to edit or remove the specific query, or view its results. An additional button is shown for queries with constraints, to view the OWL classes, that were generated out of these constraints. The knowledge whether there are such classes is obtained from the engine in Line 8. After this section, the servlet uses the engine's method `getLoadedModelNames()` to display a table containing the models, again with buttons to edit or remove each of them.

The next part is the displaying of information on each query's satisfiability. That portion of the `index.html` page is generated by the lines listed in Figure 5.2. For each of the queries, a corresponding `SatisfiabilityInformation` object is

```
 1  out.write("<h3>Query " + loadedQueriesNames[i] + "</h3>");
 2  SatisfiabilityInformation satInfo =
 3    OWLQEngine.checkQuerySatisfiability(i);
 4  if (!satInfo.isSatisfiable()) {
 5    out.write("This query is not satisfiable: "
 6        + satInfo.getReasonForUnsatisfiability());
 7  }
 8  if (satInfo.hasUnsatisfiableClasses()) {
 9    out.write("Unsatisfiable classes: ");
10    for (String className : satInfo.getUnsatisfiableClasses()) {
11      out.write(className)
12    }
13  }
14  if (satInfo.hasUnsatisfiableConstraints()) {
15    out.write("Unsatisfiable constraints: ");
16    for (String constraintName : satInfo.getUnsatisfiableConstraints()){
17      out.write(constraintName);
18    }
19  }
```

*Figure 5.2: Displaying information on queries' satisfiability*

generated in Line 3, which is used for further analysis. First, it is checked if the query is satisfiable at all in Line 4. If it is not, the reason is output. Then, if the query contains any unsatisfiable classes or constraints, they are listed in Lines 8–19.

The final part of the overview page, namely the containment information portion, is generated by the excerpt in Figure 5.3. In line 2, the servlet obtains an

```
 1  ContainmentInformation[] containmentInfo =
 2    OWLQEngine.checkQueryContainments(i);
 3  for (ContainmentInformation contInfo : containmentInfo) {
 4    out.write(loadedQueriesNames[contInfo.getIndex()] + ": ");
 5    if (contInfo.isContained()) {
 6      out.write("contained");
 7    } else {
 8      out.write("not contained");
 9    }
10  }
11  out.write("<input type=\"submit\" " +
12    "value=\"Show details\" name=\"viewc" + i + "\"/>");
```

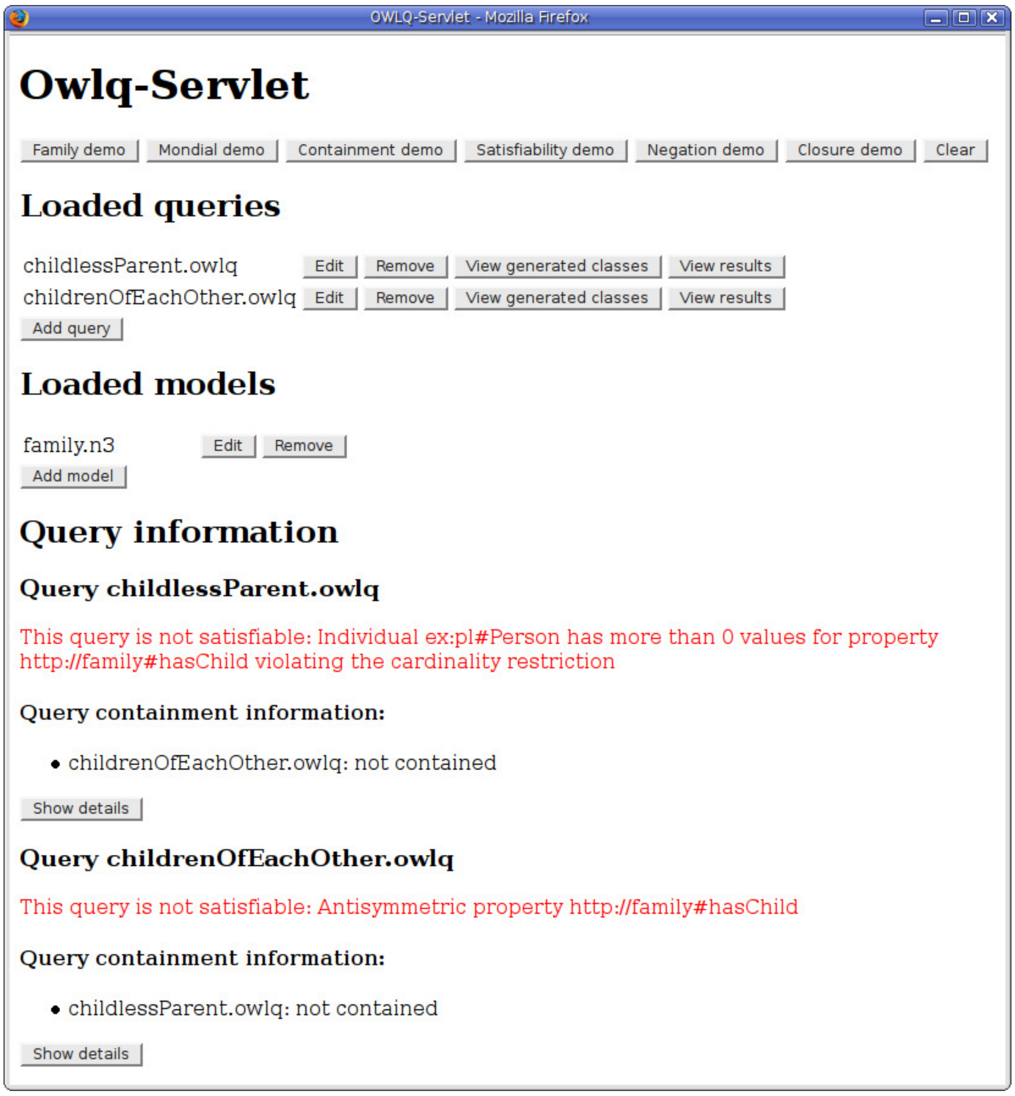
*Figure 5.3: Presenting containment information*

array of `ContainmentInformation` objects from the engine, over which it iterates in Lines 3–10. For each of the array's entries, the according other query's name is output using the query's index (Line 4, along with whether the current query is contained in it (Lines 5–9). Finally, a button is added, that leads to a page with more detailed containment information. An example screenshot of this page with two loaded queries and one model is shown in Figure 5.4.

### 5.1.2 Editing Models and Queries

To be able to add and edit models and queries via the servlet, an edit page is displayed whose body is created by the servlet's `serveEditPage()` method, which is shown in Figure 5.5.

This page is used to both add and edit both models and queries, therefore, Lines 4–16 check whether the subject to edit is a model or a query and whether a model or query with the given index is already present in the OWLQ engine. The parameter `index` is passed by the servlet's `doGet()` method which calls `serveEditPage()` and depends on which button was clicked on the overview page. If a model's or query's *edit* button was clicked, `index` is the according query's index. If it was the *add* button, `index` will be the number of loaded



Owlq-Servlet

Loaded queries

childlessParent.owlq

childrenOfEachOther.owlq

Loaded models

family.n3

Query information

Query childlessParent.owlq

This query is not satisfiable: Individual ex:pl#Person has more than 0 values for property http://family#hasChild violating the cardinality restriction

Query containment information:

- childrenOfEachOther.owlq: not contained

Query childrenOfEachOther.owlq

This query is not satisfiable: Antisymmetric property http://family#hasChild

Query containment information:

- childlessParent.owlq: not contained

*Figure 5.4: A screenshot of the servlet's overview page*

```
private void serveEditPage(PrintWriter out, int index,
    boolean isAQuery) {
  String modelString = new String(), modelName = new String();

  if (isAQuery) {
    if (index >= 0 && index < OWLQEngine.getNumberOfLoadedQueries()) {
      modelString = OWLQEngine.getQuery(index, null);
      modelName = OWLQEngine.getQueryName(index);
      OWLQEngine.removeQuery(index);
    }
  } else {
    if (index >= 0 && index < OWLQEngine.getNumberOfLoadedModels()) {
      modelString = OWLQEngine.getModelString(index, null);
      modelName = OWLQEngine.getModelName(index;
      OWLQEngine.removeModel(index);
    }
  }

  String editPageString = "<form action=\"index.html\"" +
      "method=\"post\" enctype=\"multipart/form-data\">" +
      "Model name: <input name=\"modelName\" type=\"text\"" +
      "value=\" + modelName + "\"/>" +
      "<textarea name=\"modelText\">" +
      modelString + "</textarea>" +
      "Load from file:" +
      "<input name=\"modelFile\" type=\"file\"/>" +
      "<input type=\"submit\" value=\"Submit\"/></form>";

  if (isAQuery) {
    editPageString = editPageString.replaceAll("model", "query");
    editPageString = editPageString.replaceAll("Model", "Query");
  }
  out.write(editPageString);
}
```

*Figure 5.5: Creating the servlet's edit page*

queries or models respectively, which will lead to no existing query or model with the index being found in Lines 4–16.

The page's content is created in Lines 18–31, which create input fields for the model's or query's name and its actual content. In the case of a model or query is to be added, these fields are blank, whereas in the editing case, they are filled with the respective model's or query's actual values in Lines 21 and 23. Finally, the resulting page body is output in Line 32. Figure 5.6 shows a screenshot of the edit page.

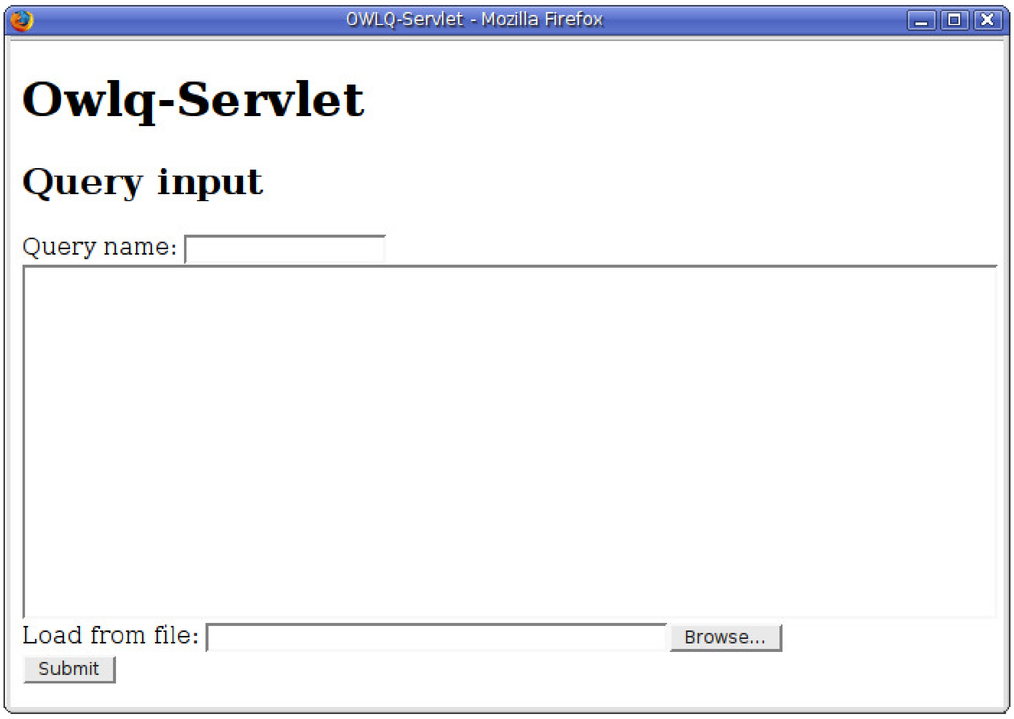


*Figure 5.6: A screenshot of the servlet's edit page*

### 5.1.3 Showing Detailed Information

While the servlet's `index.html` page just displays some basic information about which models and queries are loaded and which queries are contained in other ones, the edit page is used to display a query's generated classes, results or containment details. This page is created by the `serveViewPage()`, which is depicted in Figure 5.7. Again, the `doGet()` method passes two parameters,

```
 1  private void serveViewPage(PrintWriter out, int index, char code)
        throws ServletException {
 2    if (code == 'g') {
 3      out.write("Generated classes for " +
            OWLQEngine.getQueryName(index) + "<pre>" +
            OWLQEngine.getGeneratedClassesString(index,
            OWLQEngine.getQueryLanguage(index)) + "</pre>");
 4    } else if (code == 'r') {
 5      out.write("Query results for " + OWLQEngine.getQueryName(index) +
            "<pre>" + OWLQEngine.getQueryResults(index) + "</pre>");
 6      } else if (code == 'c') {
 7      serveContainmentViewPage(out, index);
 8    } else {
 9      throw new ServletException();
10    }
11  }
```

*Figure 5.7: Creating the servlet's edit page*

namely `index` and `code`. The former denotes the query's index and the latter specifies, which information is to view. In the case of the requested information being the generated classes (`'g'`) or the query results (`'r'`), that information is output in Lines 3 and 5, respectively. The containment information (`'c'`), is displayed by the `serveContainmentViewPage()`, an excerpt of which is given in Figure 5.8.

Like for the overview page, at first, the `ContainmentInformation` array is retrieved from the engine in Line 2. For all of its entries, the details are output by obtaining the information from the respective `ContainmentInformation` object. The code fragment in Figure 5.8 outputs the information about which variables

```java
private void serveContainmentViewPage(PrintWriter out, int index) {
  ContainmentInformation[] containmentInfo =
      OWLQEngine.checkQueryContainments(index);
  String[] loadedQueriesNames = OWLQEngine.getLoadedQueryNames();
  out.write("Detailed containment information for " +
      loadedQueriesNames[index] + ":");
  for (ContainmentInformation contInfo : containmentInfo) {
    if (!contInfo.isContained()) {
      out.write("not ");
    }
    out.write("contained in " +
        loadedQueriesNames[contInfo.getIndex()] + ":");

    if (contInfo.hasVariablesWithCounterpart()) {
      out.write("Matching variables:");
      for (String element : contInfo.getVariableMappings().keySet()) {
        out.write(element + " is contained in " +
            contInfo.getVariableMappings().get(element));
      }
    }

    if (contInfo.hasVariablesWithNoCounterpart()) {
      out.write("Missing variables:");
      for (String element : contInfo.getVariablesWithNoCounterpart())
          {
        out.write(element + " has no counterpart");
      }
    }

    /* output the same information for the query's constraints */
  }
}
```

*Figure 5.8: Outputting containment information*

of one query match the ones of the other query (Line 14) and which ones do not have a counterpart (Line 21). The code, that outputs the details about the containment of constraints looks quite the same and is therefore omitted in the listing.

## 5.2 The doPost() Method

The `doPost()` method is responsible for adding models and queries to the OWLQ engine. If a `POST` request is sent to the servlet, which corresponds to a model or query being committed from the edit page (Figure 5.6), the function reads the form data from the request and assigns each if the form's field to a `String` variable. The next part of the method processes that data and has the OWLQ engine create models and queries respectively.

```
 1  if (queryFile.equals("")) {
 2    if (!(queryText.equals(""))) {
 3      OWLQEngine.addQuery(queryText, queryName);
 4    }
 5  } else {
 6    if (queryName.equals("")) {
 7      queryName = queryFileName;
 8    }
 9    OWLQEngine.addQuery(queryFile, queryName);
10  }
```

*Figure 5.9: Adding models and queries from the servlet*

Figure 5.9 lists that part of the function, that is responsible for adding a query to the OWLQ engine. If the `queryFile` input field is empty (Line 1), which means that the query is not to be loaded from a file, the text area (`queryText`) is used as input. If it is non-empty, its content is added as a query in Line 3.

If the query is to be loaded from a file (Line 5 et seq.), the file's content (`queryFile`) is passed to the engine's `addQuery()` method in Line 9, along with either the name given in the `queryName` input field or, if the field is empty, the file name.

# 6 Conclusion

The OWLQ query language uses RDF triples to represent queries as Semantic Web data. In addition to providing the features to state conjunctive queries including closures, negated constraints and a form of sub-queries based on specifying result classes in OWL, OWLQ allows for making statements and semantically reasoning about a query.

Besides a way to examine a query's satisfiability, a heuristic approach to comparing two queries regarding their containment has been shown.

The implementation developed within the scope of this thesis is able to execute OWLQ queries and cope with some basic analysis tasks. The information about a query's satisfiability or containment in another query can be computed within near-constant time.

Some possible ideas for starting points for further work on the OWLQ query language and the engine's implementation are pointed out below:

## OWLQ language elements

- equivalents to the SPARQL `UNION` and `OPTIONAL` constructs,
- relational operators for `owlq:Constraint`, e.g. `owlq:greaterThan`.

## OWLQ engine implementation

- answering a query by utilizing the results of some other query it is contained in,
- partial query execution, e.g. listing possible values for a result variable without considering constraints, listing which constraints eliminate which assignments.

# Acknowledgements

I would like to thank Prof. Dr. Wolfgang May for his supervision. I am grateful for his remarkable and anything but self-evident scientific and moral support which enabled me writing this thesis.

Also, I would like to express my gratitude to Prof. Dr. José Júlio Alves Alferes for his co-supervision and Dipl. Biol. Franz Schenk for his technical assistance.

Special thanks go to my family, friends and fellow students and especially to my girlfriend, all of whom bolstered me when things went wrong. I truly appreciate their support, patience and understanding.

# List of Figures

# Acronyms

**RDF** Resource Description Framework

**OWL** Web Ontology Language

**SPARQL** SPARQL Protocol and RDF Query Language

**DIG** DL Implementors Group

**URI** Unified Resource Identifier

**OWA** Open World Assumption

**CWA** Closed World Assumption

# Bibliography

[1] DL Implementation Group (DIG).
`http://dl.kr.org/dig/`.

[2] Jena Framework API.
`http://jena.sourceforge.net/javadoc/index.html`.

[3] Jena – A Semantic Web Framework for Java.
`http://jena.sourceforge.net`.

[4] Pellet: The Open Source OWL DL Reasoner.
`http://pellet.owldl.com`.

[5] Ashok K. Chandra and Philip M. Merlin.
Optimal implementation of conjunctive queries in relational data bases.
In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, New York, NY, USA, 1977. ACM.

[6] Phokion G. Kolaitis and Moshe Y. Vardi.
Conjunctive-Query Containment and Constraint Satisfaction.
In *PODS*, pages 205–213. ACM Press, 1998.

[7] Wolfgang May and Franz Schenk.
OWLQ: A Semantic Semantic Web Query Language.
Institute for Computer Science, University of Göttingen, 2007.

[8] Giorgos Serfiotis, Ioanna Koffina, Vassilis Christophides, and Val Tannen.
Containment and Minimization of RDF/S Query Patterns.
In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 607–623. Springer, 2005.