Birte Glimm

A Query Language for Web Ontologies

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informatik
am Fachbereich Elektrotechnik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer.nat Christoph Klauck
Zweitgutachter : Prof. Ian Horrocks, The University of Manchester

Abgegeben am 25. Juni 2004

Birte Glimm

Thema der Bachelorarbeit

A Query Language for Web Ontologies (Eine Anfragesprache für Web Ontologien)

Stichworte

Semantic Web Anfragen, DAML+OIL, DQL, Ontologien, Semantic Web Dienste

Kurzzusammenfassung

Im August 2002 veröffentlichte das Joint United States/European Union ad hoc Agent Markup Language Committee die DAML Query Language (DQL) Abstract Specification. Im April 2003 folgte eine Revision der Spezifikation. Diese Arbeit analysiert die aktuelle Spezifikation und stellt eine (partielle) Implementierung im Rahmen einer Machbarkeitsstudie zur Verfügung. Der implementierte Prototyp nutzt Beschreibungslogik-Syteme, um Anfragen an DAML+OIL Wissensbasen unter Zuhilfenahme von automatischen Schlussfolgerungen zu berechnen. Die Anfragen sind beschränkt auf *conjunctive queries*, die eine baumähnliche Struktur aufweisen. Der Prototyp berechnet erfolgreich Anfragen an Wissensbasen im Semantic Web und belegt, dass eine Umformung von Anfragen in Beschreibungslogik Anfragen genutzt werden kann, um einen DQL Service zur Verfügung zu stellen. Für einen Einsatz in realen Anwendungen ist allerdings eine weitere Optimierung der Anfrageverarbeitung notwendig.

Birte Glimm

Title of the paper

A Query Language for Web Ontologies

Keywords

Semantic Web querying, DAML+OIL, DQL, ontologies, Semantic Web services

Abstract

In August 2002 the Joint United States/European Union ad hoc Agent Markup Language Committee announced the first release of the DAML Query Language (DQL) Abstract Specification. A revision of the specification followed in April 2003. This work analyses the current DQL specification and provides a (partial) implementation as a feasibility study. The prototype uses Description Logic reasoners for automated reasoning services over the DAML+OIL knowledge bases to compute the query answers. The queries are restricted to conjunctive queries that have a tree-like structure. The prototype successfully computes queries against knowledge bases in the Semantic Web. It shows that a query transformation into Description Logics queries is a feasible setting to provide a DQL service. However, further performance improvements are necessary before the prototype is usable in real world applications.

Acknowledgements

First of all I would like to thank my friend Frank and my parents for supporting me during my studies. Without them, none of this would have been possible.

I am deeply grateful for the support of my supervisors Prof. Ian Horrocks and Prof. Dr. Christoph Klauck; especially Ian Horrocks for proofreading and correcting my English mistakes.

Further more, I would like to thank the following people who have helped me during my studies and this final year project: all members of the Information Management Group, especially Daniele Turi for explaining many of the external software components that the DQL server uses and Phillip Lord for LaTeX support; Conny Hedeler, Antoon Goderis and Sven Stegelmeier for proofreading; Christian Morgenstern and several of my fellow students from Hamburg (the list would be too long) and Ubbo Visser and Sebastian Hübner from the Center for Computing Technologies (TZI) in Bremen.

Finally I would like to thank the Stiftung der Deutschen Wirtschaft (Foundation of German Business) for supporting me with a scholarship.

Contents

1 Introduction **1**
- 1.1 Semantic Web Concepts . . . 1
  - 1.1.1 Ontologies . . . 1
  - 1.1.2 Web Ontology Languages . . . 2
  - 1.1.3 Reasoners and Inference Engines . . . 2
- 1.2 The Conceptual Formulation . . . 3

2 Querying a DAML+OIL Knowledge Base **5**
- 2.1 Introduction . . . 5
  - 2.1.1 DAML+OIL . . . 5
  - 2.1.2 Description Logics . . . 6
- 2.2 Querying . . . 7
  - 2.2.1 Extended Retrieval Support . . . 8
  - 2.2.2 Conjunctive Queries . . . 9
  - 2.2.3 Graphs as Query Representation . . . 10
- 2.3 Query Transformation . . . 11
  - 2.3.1 Boolean Queries with one Leaf . . . 11
  - 2.3.2 Boolean Queries with Multiple Leaves . . . 12
  - 2.3.3 Rolling-Up in the Role Direction . . . 13
  - 2.3.4 Rolling-Up with Individual Names . . . 13
  - 2.3.5 Rolling-Up for non Boolean Queries . . . 14

3 The DQL Abstract Specification **16**
- 3.1 Query and Answer Parts . . . 16
- 3.2 A Query-Answering Dialogue . . . 18
- 3.3 Query Classes . . . 19
- 3.4 OWL-QL . . . 19

4 Realisation of a DQL Server Prototype **20**
- 4.1 The Architecture . . . 20
- 4.2 Used Tools, Products and Languages . . . 22
- 4.3 The Components . . . 23
  - 4.3.1 The Web Service Interface . . . 23
  - 4.3.2 The DQL Server Component . . . 24
  - 4.3.3 The Query Parser . . . 24

4.3.4 Knowledge Base Loading . . . 25
4.3.5 Interaction with the Reasoner . . . 26
4.3.6 The Query Graph Component . . . 27
4.3.7 Query Types . . . 29
4.3.8 Query Answers . . . 30
4.3.9 The Answer Set Cache . . . 32
4.4 A Query Processing Sequence . . . 32
4.5 Error Handling . . . 33
4.6 Testing . . . 34
4.7 The DQL Client Interface . . . 34

5 Conclusion **37**
5.1 Improvements for Future Versions . . . 37
5.1.1 Extended Query Support . . . 37
5.1.2 Multi-Thread Safe Reasoner Connections . . . 38
5.1.3 Proper Use of the Termination Token . . . 38
5.1.4 Interaction with the Reasoner . . . 38
5.1.5 Improved Candidate Checks . . . 39
5.2 Identified Improvements for the DQL Specification . . . 39
5.2.1 Security . . . 40
5.2.2 External Query Language Definition . . . 40
5.2.3 Forced Different or Equal Bindings . . . 41
5.2.4 Knowledge Base Loading . . . 41
5.2.5 Answer Bundle Size Bound . . . 41
5.3 Comparison with Other Systems . . . 42
5.3.1 The Stanford OWL-QL Server . . . 42
5.3.2 Racer Query Language . . . 43

References **44**

A Appendix **A1**
A.1 Notation . . . A1
A.2 Abbreviations . . . A1
A.3 The Enclosed CD . . . A2
A.3.1 Application Files . . . A2
A.3.2 Dependent Applications . . . A3
A.3.3 The Report and the References . . . A3
A.3.4 The Project Source Files . . . A3
A.3.5 Documentation . . . A3
A.4 Model Theoretic Semantics of DAML+OIL . . . A4

Chapter 1

Introduction

The foundations of the work presented here were first laid by Tim Berners-Lee [5], who introduced in 1998 his vision for the future architecture of the world wide web. His vision is about a Semantic Web, where resources are not only usable for humans, who are able to interpret the implicit semantics of a web page, but also for machines. Essential to make this idea work, is the explicit annotation of data in a structured way using a well defined terminology. Software processes or so called agents can then interpret the meaning or semantics of a web resource and use this information to complete their automated tasks. A variety of technologies will be required to fulfil this vision.

The implemented query server presented here is just a small part of this, but it will allow agents to query ontologies that are used to store knowledge in the Semantic Web.

1.1 Semantic Web Concepts

Before the conceptual formulation for this report is defined more precisely, the following section introduces the most important underlying concepts of the Semantic Web.

1.1.1 Ontologies

The first concept to mention in a Semantic Web context are ontologies. The term ontology was first introduced by the ancient Greek philosopher Aristoteles (384–322 B.C.) in his “The metaphysical study of the nature of being and existence”. Nowadays the term has been adopted by the Artificial Intelligence community. Willem N. Borst [7] gave a popular definition for an ontology as it is understood by computer scientists: “An ontology is a formal specification of a shared conceptualization”. *Conceptualisation* refers to an abstract model of phenomena in the world that identifies that phenomenon’s relevant concepts. *Formal* means that the ontology has a well defined semantics. *Shared* reflects the notion that an ontology captures consensual knowledge — that is, it is not restricted

to some individual but is accepted by a group. A similar explanation of these terms was given by Fensel et al. [12].

As defined above, an ontology defines classes, also called concepts, that describe the common properties of a collection of individuals, similar to classes and objects in object oriented design and programming. The classes are ordered in a hierarchy using the is-a relation. It is also possible to define roles, which are interpreted as binary relations between objects. The semantics of the terms concept and class are equivalent, but class is mainly used in ontologies and concept in Description Logics. Ontologies are often developed for a particular domain to provide a controlled vocabulary of terms with an explicitly defined and machine processable semantics. An example for a large ontology is GALEN.[1] GALEN provides a formal model of clinical terminology and the GALEN system offers various services to support the management of clinical information.

1.1.2 Web Ontology Languages

During the last five years the foundations were laid to properly define ontologies and today the World Wide Web Consortium (W3C)[2] hosts the common standards for web ontology languages. The basic language to build an ontology is the Resource Description Framework (RDF) [24], which is built on top of XML [8], together with its schema language RDFS [9]. A further extension on top of RDF and RDFS is the Ontology Inference Layer (OIL) [18], an ontology language developed by a group of (largely) European researchers. OIL later was merged with the US approach called DAML-ONT [25], which stands for DARPA Agent Markup Language Ontology language, to give the DAML+OIL standard [21]. The latest W3C recommendation regarding ontology languages is the Web Ontology Language OWL [2], which is largely based on DAML+OIL.

The query language regarded here is used to query knowledge bases in DAML+OIL, but the prototypical implementation already allows to query OWL knowledge bases as well, since OWL will sooner or later replace its predecessor DAML+OIL.

1.1.3 Reasoners and Inference Engines

Beneath the ontologies that represent information, there is another important task to support the Semantic Web vision: reasoners and inference engines will allow agents to make logical inferences over ontologies (also termed to reason) and they make the difference between just “machine readable” and “machine understandable”.

This is possible since ontology languages are closely related to Description Logics (DL), which are decidable[3] subsets of first order logic (FOL). Description Logics are derived

[1] `http://www.opengalen.org`

[2] `http://www.w3.org`

[3] Decidable means that it is in principle possible to specify an algorithm that terminates in all cases.

from the well known frame-based systems, semantic networks and KL-ONE-like knowledge representation systems [31].

The basic building blocks used to define an ontology (classes, properties and individuals) can directly be mapped into Description Logics (concepts, roles and individuals). Concepts are interpreted as sets of individuals and roles are interpreted as binary relations between individuals. Every ontology that is defined in an appropriate ontology language can be translated into DL formulas and a DL reasoner is then able to draw conclusions based on the knowledge given in the ontology. To give an idea of a possible conclusion, consider an ontology that defines a concept *human* and states that *humans are mortal*. If the ontology includes the assertion that *Sokrates is a human*, it can be inferred that *Sokrates is mortal*, even if this is never stated explicitly.

The main reasoning tasks, performed by a DL reasoner, are subsumption, classification, instance checking and satisfiability. Subsumption represents the is-a relation and a subsumption check tests if one concept is more general than (subsumes) another. Classification is the computation of the concept hierarchy based on subsumption, and instance checking means to test if an individual belongs to a given concept. A satisfiability test determines if a concept is contradictory and could never have an instance, e.g., a concept defined as *human and not human* can never have an instance.

For all these tasks the reasoners offer a query interface, but users or agents have many more kinds of queries than e.g., asking for sub-concepts of a given concept, hence some reasoners were equipped with additional query support, but until now querying is one of the weakest supported features of current DL reasoners.

1.2 The Conceptual Formulation

To overcome the limited query support in the Semantic Web the DAML Joint Committee[4] announced in August 2002 the DAML Query Language (DQL) Abstract Specification and replaced it in April 2003 with a new release [13]. The DQL specification describes a protocol and features for a query language in the Semantic Web, and its development was based on user requirements for a query language in the Semantic Web.

Until now there is only one implementation for this standard provided by the Knowledge Systems Laboratory of the Stanford University,[5] which we were unfortunately unable to test since the specified DQL server is unavailable, but they also offer an OWL-QL implementation that supports DAML+OIL and can therefore also be regarded as a DQL server.[6] The approach taken in Stanford was to use a first order logic theorem prover to answer the queries, but as two examples in chapter 5 show, the implementation is in some cases incomplete and in others also incorrect.

[4] http://www.daml.org

[5] http://ksl.stanford.edu/projects/dql

[6] http://ksl.stanford.edu/projects/owl-ql

This fact led to the work presented in this report, which aims to provide a correct implementation that relies on Description Logic reasoners instead of first order logic theorem provers to answer the queries. Incompleteness can not completely be eliminated since for the ontology language DAML+OIL no sound and complete inference algorithm is known so far. A complete and correct solution can so far only be achieved by reducing the knowledge representation language itself.

The DQL specification, described in detail in chapter 3, provides the general framework for this report and can be regarded as a user specification for the prototypical implementation developed as part of this project. A constraint defined for this work is to limit the query support to acyclic conjunctive queries, since until now no algorithm is known to translate arbitrary queries into DL reasoner queries.

To implement a DQL server that relies on DL reasoners, it is necessary to translate the incoming queries into statements that a DL reasoner can process, and to translate the received results from the reasoner back into a form that corresponds to the submitted query. This translation process constitutes the main part of this work and the next chapter gives a detailed description of how queries are translated into one or many DL reasoner queries.

Chapter 2

Querying a DAML+OIL Knowledge Base

2.1 Introduction

Before explaining the query translation process, the general environment is introduced. This includes DAML+OIL knowledge bases, which provide the knowledge used to answer the queries, and Description Logics, which are the underlying logic formalism.

2.1.1 DAML+OIL

As mentioned in the introduction, Description Logics build the formal foundation of the Semantic Web and enable the automated reasoning services. However knowledge bases on the Semantic Web are not directly written in Description Logics, but in languages like DAML+OIL or OWL. These ontology languages were designed in a way that makes it possible to translate them into DL and use the full power of already existing reasoning services. Horrocks [17] illustrated how DAML+OIL and Description Logics fit together. Example 2.1 shows a very simple DAML+OIL knowledge base, to give an impression of what DAML+OIL looks like. A complete and annotated example is available on the DAML website.[1]

The XML syntax of DAML+OIL is quite verbose, although already abbreviations for longer syntactical statements were used in example 2.1. One of the W3C recommendations gives a good overview of the syntax and allowed abbreviations for RDF [23] and since RDF is the underlying layer of DAML+OIL these abbreviations are valid for DAML+OIL too. To avoid such long statements for the remainder of this report, the much shorter Description Logic syntax is introduced in the next section.

[1] `http://www.daml.org/2000/12/daml+oil-walkthru.html`

Example 2.1

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2000/12/daml+oil#"
  xmlns     ="http://myPlace/example#">

  <daml:Ontology rdf:about="">
    <daml:versionInfo>0.1</daml:versionInfo>
    <daml:imports
      rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
  </daml:Ontology>

  <daml:Class rdf:ID="MORTAL"/>
  <daml:Class rdf:ID="HUMAN">
    <rdfs:subClassOf rdf:resource="#MORTAL"/>
  </daml:Class>
  <daml:Class rdf:ID="PHILOSOPHER">
    <rdfs:subClassOf rdf:resource="#HUMAN"/>
  </daml:Class>
  <daml:Class rdf:ID="COUNTRY"/>
  <daml:ObjectProperty rdf:ID="hasresident">
    <rdfs:domain rdf:resource="#COUNTRY"/>
    <rdfs:range rdf:resource="#HUMAN"/>
  </daml:ObjectProperty>

  <PHILOSOPHER rdf:ID="sokrates"/>
  <COUNTRY rdf:ID="greece">
    <hasresident rdf:resource="#sokrates"/>
  </COUNTRY>

</rdf:RDF>
```

2.1.2 Description Logics

The use of Description Logics to express DAML+OIL statements is possible since from a formal point of view all DAML+OIL is equivalent to the Description Logic $\mathcal{SHIQ}$ enriched with the oneOf constructor, i.e., defining a concept by enumerating its instances, and by XML datatypes, i.e., integers or strings. (This was shown by Horrocks [19]). The basic statements are written as `a:C` to denote that the individual `a` is an instance of the concept `C` and `(a,b):r` to denote that the individuals `a` and `b` are related by the role `r`. `C` $\sqsubseteq$ `D` is used for implications, which can also be read as a subconcept relationship. The semantics is that all instances of the concept `C` are always also instances of the concept `D`, so a `C` membership also implies a `D` membership. The same knowledge base as in example 2.1 can then be written as follows:

Example 2.2
HUMAN $\sqsubseteq$ MORTAL
PHILOSOPHER $\sqsubseteq$ HUMAN
greece:COUNTRY
sokrates:PHILOSOPHER
(greece, sokrates):hasresident

In addition to these statements the existential quantification $\exists$ is worth to be introduced here as well, since existential quantifiers are heavily used in the query translation process. A statement like HUMAN $\sqsubseteq \exists$ hasfather.HUMAN means that each instance of the concept HUMAN is related by the hasfather role to another instance of the concept HUMAN, so HUMAN implies the existence of a relationship with another human. The related human need not be known by name, but a query asking for humans that have a father should return all humans. A query asking for pairs of child and father (premised that hasfather has the semantics of relating a child to its father) would however only return humans whose father is known and return these as a pair.

The constructor $\sqcap$ is used to conjunct terms, i.e., the example 2.3 states that there is also a relation to a human over the hasmother role.

Example 2.3 HUMAN $\sqsubseteq \exists$ hasfather.HUMAN $\sqcap \exists$ hasmother.HUMAN

These are not all the constructors of DL or DAML+OIL, but sufficient to explain the general query translation process. A complete overview of the DL constructors and axioms together with their DAML+OIL equivalent is given by van Harmelen [28] and appendix A.4 lists all DAML+OIL statements together with their model theoretic semantics.

2.2 Querying

The previous examples have shown how information about a domain can be stored in a knowledge base. If a knowledge base is stored as part of the Internet, it is available for other software agents or humans, but to access the knowledge comfortably, a good query support is essential.

Most of the currently available OWL reasoners support a query interface and they offer some basic query support to access structural information or information about individuals stored in the knowledge base. Most current reasoners support the following types of queries:

- methods to retrieve all concept, role or individual names
- boolean queries for concept satisfiability, i.e., is the concept inconsistent
- boolean queries for subsumption check, i.e., is one concept more general than another

- boolean queries for disjointness of two concepts
- retrieval: queries for individuals that are instances of a given concept
- realisation: retrieves the most specific concepts that an individual is an instance of
- instantiation: boolean queries for an individual (pair of individuals) being an instance of a given concept (role)

These query facilities allow implicit knowledge to be made available. For example the boolean query for `sokrates:MORTAL` against the knowledge base of example 2.2 will return true. The reason for this is that Sokrates is specified as a philosopher, philosopher is a subconcept of human, and human is a subcncept of mortal; and due to the underlying set theoretic semantics Sokrates is also an instance of the concept mortal.

2.2.1 Extended Retrieval Support

The queries described above are already useful, but users also demand more advanced features such as the use of variables in a query. A query such as `?x:MORTAL`, where `?x` is a variable, can be transformed into a retrieval query, but slightly more complicated queries like example 2.4 that asks for Greek philosophers are not solvable with the normally offered query support.

Example 2.4 `?x:PHILOSOPHER ⊓ (greece, ?x):hasresident`

In the following, names prefixed with ? are used to represent variables. An answer for such a query consists of bindings for the used variables, and if the variables are replaced with their corresponding bindings, the resulting statement must be true in the knowledge base used to answer the query.

Besides these variables a second type of variables is used here for which no binding is expected in the answer. Instead, it is only required that the existence of such an individual is inferred by the used knowledge base. To differentiate these variables from the ones for which a binding is required a ! prefix is used. Both kinds of variables occur in the DQL specification. Common terms for these two kind of variables are *distinguished* or *must-bind* variables for the former and *undistinguished* or *don't-bind* variables for the latter.

The meaning of don't-bind variables can be illustrated by the example in section 2.1.2, where it was stated that every human is related to another human via the `hasfather` role, so the existence of a related human via the `hasfather` role can be inferred for every human. The answer for the query in example 2.5 would return all individuals that are humans as a binding for `?x` and no binding for `!y`, since this is not required. But it is true in the knowledge base that every human has such a related individual.

This is contrary to a database setting, where no such inferences are possible. In a database an unspecified value is represented by `null` and `null` can mean that the father is not known or that the person has no father. In this setting an unspecified father definitely

means that the father is not known, but he exists. This is called the Open World Assumption, whereas databases use the Closed World Assumption and classify everything that is not explicitly specified in the database as false.

Example 2.5 `?x:HUMAN ⊓ (?x, !y):hasfather`

The next example gives a first idea of the query translation process. The query in example 2.6 is a transformation of the query in example 2.5, but the transformation does not change the semantics of the query. The bindings for `?x` must be instances of the concept `HUMAN` and they must be related via the role `hasfather` to another (maybe unnamed) individual. The symbol $\top$ is an abbreviation for $C \sqcup \neg C$, where `C` is an arbitrary concept name. This is a tautology and therefore every individual is an instance of the concept $\top$. Since no specific concept was provided for `!y`, the concept $\top$ is used here.

Example 2.6 `?x:HUMAN ⊓ ∃hasfather.⊤`

A simple approach to find an appropriate binding for must-bind variables of a query could be to replace the variables with individual names from the knowledge base and use a boolean query to check whether the statement (with the replaced variables) is entailed by the knowledge base. If that is the case, the individual names are a valid binding for the used variables. This check has to be done for all individual names in the knowledge base and for more than one variable one must test all possible combinations of variable replacements. This would obviously determine the query answer, but with extremely high costs.

2.2.2 Conjunctive Queries

For some queries Horrocks and Tessaris [20] and Tessaris [26] proposed a solution that is more efficient than the simple testing strategy described above. Their technique is applicable for *conjunctive queries* and transforms queries such as the one given in example 2.5 into equivalent ones as the one in example 2.6. The query in example 2.6 can for example be answered by querying for concept instances of the concept `HUMAN⊓∃ hasfather.⊤`, which is already supported by most of the current DL reasoners.

Tessaris [26] or Wang, Maher, and Topor [29] provide a formal definition of conjunctive queries. For short, a conjunctive query consists of a conjunction of concept and/or role terms that may contain variables. An answer for a conjunctive query replaces some of the variables with individual names from the knowledge base used to answer the query. These individual names are called the bindings for their variables. If a variable is not replaced, the existence of a possible binding must be inferred by the knowledge base. This is illustrated by the example in section 2.1.2, where it was stated that every human is related to another human via the hasfather role, i.e., the existence of a related human via the hasfather role can be inferred for every human. Valid answers then consist solely of terms that are true in the used knowledge base.

Before the query translation process is explained in detail, the next section shows how a graph can be used to represent a query. On the one hand, a graphical representation facilitates the explanation of the query transformation process and on the other hand, the implemented algorithm also uses a graph to transform a query into valid Description Logic reasoner queries.

2.2.3 Graphs as Query Representation

This section shows, how a directed graph can represent a query. In a query graph each variable is represented as a node. For the readers convenience must-bind variables are represented by a filled node (•), whereas don't-bind variables and individuals are represented by an unfilled node (○). Nodes for an individual are labelled with the individual's name. A role assertion corresponds to a directed edge, labelled with the role name. Concept assertions are also labels for the node and appended after a colon, e.g., `?p:PERSON` states that the binding for `?p` must be an instance of the concept `PERSON` and `bill:PERSON` states that Bill is a person. Figure 2.1 shows the graph representation of the query in example 2.7.

Example 2.7

```
?p:PERSON ⊓ (?p, !t):owns ⊓ (!t, red):hasColour
```

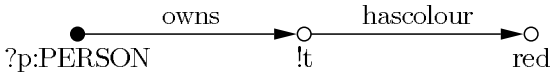

Figure 2.1: Graph representation of the query in example 2.7.

The graph representation in figure 2.1 is a directed acyclic graph and even the underlying undirected graph is acyclic, but there are also queries that produce a cycle. Since the query transformation technique introduced in the next section is not directly applicable to cyclic graphs, the prototype developed as part of this project is limited to tree-like queries that do not have a cycle in their (underlying undirected) graph representation.

Consider for example query 2.8, which is represented by the graph in figure 2.2. The directed graph is acyclic, but the underlying undirected graph has a cycle, so the query is not permitted. It is also possible to construct queries that have a cycle in their directed graph representation and these are also not permitted.

Example 2.8

```
?p:PERSON ⊓ (?p, !t):owns ⊓ (?p, !c):favouritecolour ⊓
(!t, !c):hascolour
```

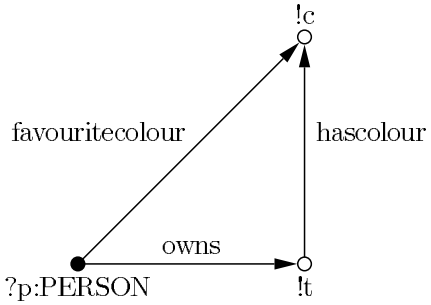

Figure 2.2: Graph representation of the query in example 2.8.

2.3 Query Transformation

Tessaris and Horrocks [20] recently proposed a rolling-up technique to eliminate variables from a query. A simple example was already given by the transformation of the query in example 2.5 into the equivalent query in example 2.6.

2.3.1 Boolean Queries with one Leaf

The rolling-up is first explained by means of example 2.9 and its graph representation in figure 2.3, which contains only don't-bind variables and since no binding for a variable is required, the query can be treated as a *boolean query* with either *yes* as query answer, in case the knowledge base entails the query, or *no* otherwise.

Example 2.9

`!w:PERSON ⊓ (!w, !x):haschild ⊓ !x:PERSON ⊓ (!x, !y):owns ⊓ (!y, !z):hascolour ⊓ !z:COLOUR`

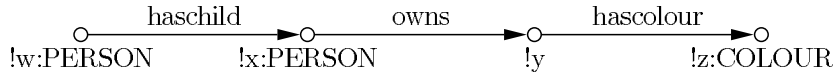

Figure 2.3: A boolean query containing only don't-bind variables.

The rolling-up starts at the leaves of a graph, here at the node for the variable `!z`. This node and its incoming edge state that the node's predecessor has a `hascolour` successor that is an instance of the concept `COLOUR`. The same thing can be expressed by the concept $\exists$ `hascolour.COLOUR`. If this concept is conjuncted with the concepts of the predecessor it replaces the leaf node and its incoming edge. The graph in figure 2.4 therefore has the same semantics as the one in figure 2.3.

Figure 2.4: After the first rolling-up step.

The rolling-up can now be applied again, this time for `!y`, which is now a leaf. Figure 2.5 shows the query graph after the next rolling-up step. The variable `!x` is still a person and is related via the `owns` role to something that is related via the `hascolour` role to a colour, so the semantics are still the same.

Figure 2.5: After the second rolling-up step.

The last edge can be removed with an additional rolling-up step and the result is a graph with only one node.

Figure 2.6: After the third rolling-up step.

In this example the last variable is also a don't-bind variable, therefore the query can be answered with true if the knowledge base entails the existence of an instance of the remaining concept. If `?y` had been a must-bind variable, then the conjunctive query could be answered by a standard retrieval query, i.e., by retrieving all the instances of the concept resulting from the rolling-up procedure.

2.3.2 Boolean Queries with Multiple Leaves

If a query graph has more than one leaf, the conditions generated during the rolling-up process are all appended to the concept description of the predecessor. Example 2.10 shows such a query and the left part of figure 2.7 illustrates the corresponding query concept. The query asks for persons that have a child and own a car. The rolling-up leads to the graph in the right part of figure 2.7 where both conditions are conjoined in the preceding node.

Example 2.10
`?x:PERSON ⊓ (?x, !y):owns ⊓ !y:CAR ⊓ (?x, !z):haschild`

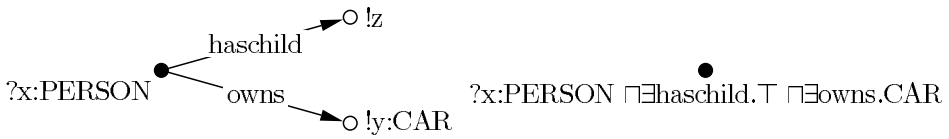

Figure 2.7: The query graph of example 2.10 and its rolled-up equivalent.

2.3.3 Rolling-Up in the Role Direction

Until now, the rolling-up process started at a leaf node, following the incoming edge back to the node's predecessor. If a query causes a graph containing only leaves with outgoing edges, the query is no longer in tree form, but since the underlying undirected graph contains no cycles, the query is still manageable.

Example 2.11 and its appertaining graph in figure 2.8 illustrates such a query. The owns relation can be treated as usual, but then both end nodes contain only outgoing edges. To continue, one can use an inverse role and reduce one of the nodes and its outgoing edge, e.g., the node for `!f` with its `haschild` relation by adding the assertion $\exists$ `haschild`$^{-}$`.FEMALE` to the person node in the middle. Now the rolling-up can continue as usual, since the remaining graph is a proper tree.

Example 2.11
`!f:FEMALE` $\sqcap$ `!p:PERSON` $\sqcap$ `!m:MALE` $\sqcap$ `!c:CAR` $\sqcap$ `(!f, !p):haschild` $\sqcap$ `(!m, !p):haschild` $\sqcap$ `(!m, !c):owns`

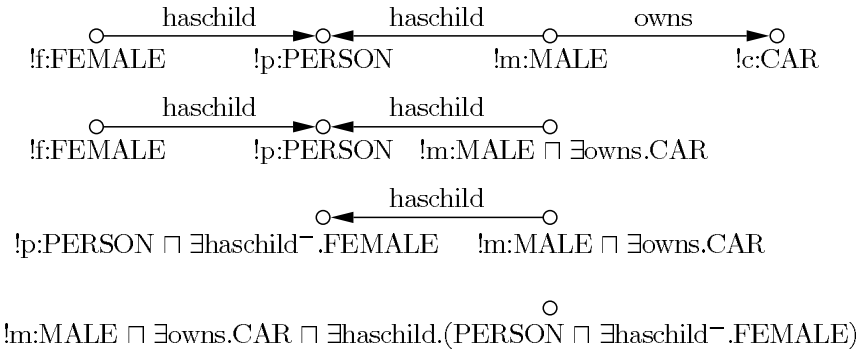

Figure 2.8: The query graph of example 2.11 and the rolling-up steps.

2.3.4 Rolling-Up with Individual Names

Until now a query contained only variables, but a user can also mix in individual names. This section shows how to deal with that in the rolling-up process. Consider the query in example 2.12, which asks for persons that own a red thing and `red` is an individual name. In a DL that supports the one-of constructor, which allows the definition of a concept by enumerating its individuals, the rolling-up step can use the one-of constructor to replace the node for the individual `red` and its `hascolour` edge by adding the assertion $\exists$ `hascolour.`$\{$`red`$\}$ to its predecessor node. The one-of constructor is denoted as a set of individuals: $\{$`individual1, individual2, ...`$\}$. Unfortunately most reasoners do not support this operator, but an indirect way can help to deal with such queries anyway. As described by Tessaris [26], a so called representative concept, with a so far unused concept name, can be used instead of the individual. Of course the ABox[2] has to be

[2]The ABox is the part of the knowledge base that contains assertional knowledge about individuals.

extended with an assertion stating that the individual is an instance of its representative concept. A representative concept is denoted here as $\mathtt{P}_a$, where a is the individual name. So in this example the assertion $\mathtt{red:P}_{red}$ is added to the knowledge base and then the query is transformed into the concept description $\mathtt{PERSON} \sqcap \exists\, \mathtt{hascolour.P}_{red}$.

Example 2.12
`?x:PERSON ⊓ (?x, !y):owns ⊓ (?y, red):hascolour`

2.3.5 Rolling-Up for non Boolean Queries

The last section already mentioned how to deal with queries containing at most one must-bind variable. The rolling-up process simply ends at this variable, and a query asking for concept instances will return the bindings for the variable. Queries with more than one must-bind variable need a different approach, since the rolling-up for don't-bind variables eliminates the variables and replaces them with sufficient conditions attached to their predecessor nodes. As a result the reasoner does not return any bindings for them. The simplest possible approach to solve a query with multiple must-bind variables is to submit a boolean query for every possible combination of individuals substituted for the must-bind variables. Unfortunately this approach is very costly. To avoid the test of every possible combination, the rolling-up process can be used to compute possible candidates first. Boolean queries are then only necessary for the computed candidates.

Example 2.13 shows such a query and figure 2.9 shows a graph representation of the knowledge base that is used to answer the query. The knowledge base consists only of an ABox. The relations between the individuals that are expected to be in the answer set are already highlighted.

Example 2.13
`?x:PERSON ⊓ ?y:CAR ⊓ ?z:COLOUR ⊓ (?x, ?y):owns ⊓ (?y, ?z):hascolour`

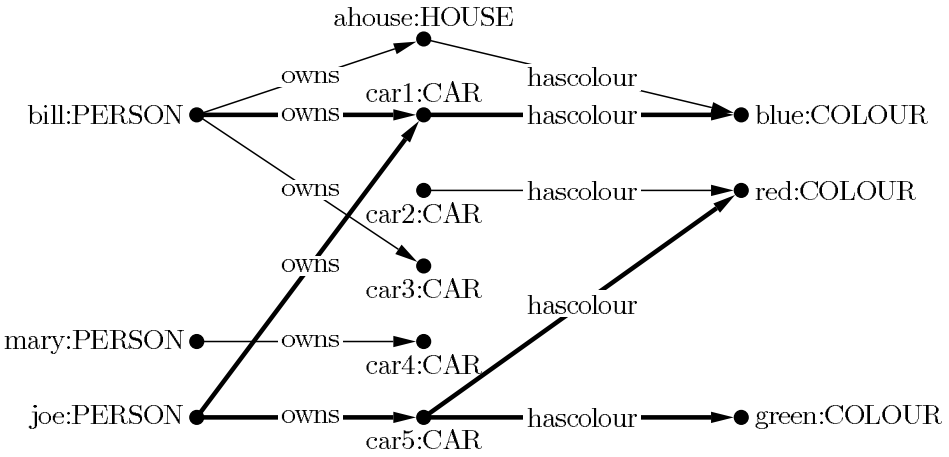

Figure 2.9: The knowledge base used for the query in example 2.13.

For the rolling-up, every must-bind variable is treated separately. In the first step, ?x is treated as the only must-bind variable and all other variables are treated as don't-bind variables.

The rolling-up for ?x then yields a retrieval query for the concept expression:

$$\texttt{PERSON} \sqcap \exists \texttt{owns}.(\texttt{CAR} \sqcap \exists \texttt{hascolour}.\texttt{COLOUR})$$

Instances of this concept, and therefore candidates for the binding of the variable ?x, are bill and joe. The same is now done for ?y, i.e., ?y is treated as the only must-bind variable and the rolling-up yields to the concept:

$$\texttt{CAR} \sqcap \exists \texttt{owns}^{-}.\texttt{PERSON} \sqcap \exists \texttt{hascolour}.\texttt{COLOUR}$$

Instances of this concept are car1 and car5. For ?z the rolling-up results in:

$$\texttt{COLOUR} \sqcap \exists \texttt{hascolour}^{-}.(\texttt{CAR} \sqcap \exists \texttt{owns}^{-}.\texttt{PERSON})$$

Instances of this concept, and therefore candidates for the binding of ?z are blue, red and green.

Boolean queries can now be used to find out which bindings for ?x, ?y and ?z belong together. Compared to the testing of all possible combinations the preceding rolling-up process and candidate retrieval reduces the number of boolean queries significantly.

A server implementing these rolling-up technique can answer conjunctive queries with don't and must-bind variables. This allows the implementation of a query answering server that complies with the proposed DQL Abstract Specification introduced in the next chapter.

Chapter 3

The DQL Abstract Specification

In August 2002 the Joint United States/European Union ad hoc Agent Markup Language Committee[1] released the first version of the DAML Query Language Abstract Specification, which was replaced in April 2003 by the current release [13]. The specification goes beyond the aims of other current web query languages such as XML Query [6], an XML query language, or RQL [22], an RDF query language, in that it supports the use of inference and reasoning services for query answering.

The specification is given on a structural level with no exact definition of the external syntax. This was done with the intention to leave the specification easily adoptable for other knowledge representation formats, such as the Web Ontology Language OWL, which is a W3C standard recommendation since February 2004. The specification focuses on defining the semantics of queries and the interaction between a querying agent and a query answering server.

3.1 Query and Answer Parts

To initiate a query-answering dialogue, a client sends a query to a DQL server. The query necessarily includes a *query pattern* that is a collection of knowledge base statements where some URI references [4] or literals are replaced by variables. Therefore, if the knowledge base contains the assertion that Bill is a child of Mary, Bill should be in the answers of a query asking for Mary's children. Table 3.1 illustrates this. On the left hand side is a part of a DAML+OIL knowledge base, with individuals and concepts in a knowledge base provided as URI references. The right hand side shows a query that is equivalent to the conjunctive query term `(mary,?x):haschild` $\sqcap$ `mary:PERSON`.

The client also specifies for which variables the server has to provide a binding (*must-bind variables*), for which the server may provide a binding (*may-bind variables*) and for which variables no binding (*don't-bind variables*) should be returned. May-bind variables are a combination of must-bind and don't-bind variables, and the rolling-up

[1] See `http://www.daml.org/committee`

```
<PERSON rdf:ID="mary">
  <haschild rdf:resource="#bill">
</PERSON>
```

```
<PERSON rdf:ID="mary">
  <haschild rdf:resource=?x>
</PERSON>
```

Table 3.1: A query pattern example.

technique described in the previous chapter is sufficient to deal with may-bind variables too. Similar to the introduced prefixes `?` and `!` for must-bind and don't-bind variables respectively, the prefix $\sim$ is used to indicate a may-bind variable.

The knowledge base in example 3.1 can be used to illustrate these different kinds of variable. A query such as `(?x, ?y):hasfather`, where `?x` and `?y` are both must-bind variables, would have one answer with `bill` as a binding for `?x` and `joe` as a binding for `?y`. The query `(?x, ~y):hasfather`, where `~y` is now a may-bind variable would have two answers. One with the binding `bill` for `?x` and the binding `joe` for `~y` and one with the binding `joe` for `?x` and no binding for `~y`. The second answer is caused by the assertion that every person is related to another person via the `hasfather` role, so it is true that Joe also has a father, but the name of the father is unknown. If `y` would have been a don't bind variable, the answer set would also contain all known persons as a binding for `?x`, but no binding for `y` would have been returned, whether the father is known or not.

Example 3.1

```
PERSON ⊑ ∃hasfather.PERSON
bill:PERSON
joe:PERSON
(bill, joe):hasfather
```

The client may also specify an *answer knowledge base pattern* which specifies the knowledge base(s) the server should use to answer the query or use a variable to let the server decide which knowledge base to use. The server is also free to delegate the query to another server. The implemented prototype does not support the specification of multiple knowledge bases and it does not use delegation, since there are no other servers available at the moment. Delegation only makes sense, if the server has means to find out which knowledge base would be useful to answer the query or if the server has some well known knowledge bases from which clients expect to receive the answer for their query and neither is currently the case.

An optional query parameter allows the definition of a pattern that the server should use to return the answers. If it is omitted, the server uses the query pattern instead. An *answer pattern* necessarily includes all variables used in the query pattern. An example answer pattern for the query `(?x, ?y):hasfather` is the natural language sentence "?y is the father of ?x". The DQL server uses this pattern in every answer and replaces the variable names with the appropriate bindings.

The server bundles answers together in an answer set and since such an *answer bundle* can become very large and the computation can take a long time, the specification also

allows to specify an *answer bundle size bound* that is an upper bound for number of answers in an answer bundle.

Another option for a query is to include a *query premise* to facilitate "if-then" queries, which can't be expressed otherwise since DAML+OIL does not support an implies logical connective. To ask a question such as "If Bill is a person, then does Bill have a father?", the query premise part includes a DAML+OIL knowledge base or a knowledge base reference stating that Joe is a person and the query part asks for the father of Bill. A server must treat DAML+OIL statements in the query premise as a regular part of the answer knowledge base and all answers must be entailed by this knowledge base. A premise is not supported by the prototype, but since it does not affect the query algorithm itself, a future version of this DQL server could add statements in the premise to the knowledge base before the query algorithm starts.

3.2 A Query-Answering Dialogue

To answer a query, the server returns an answer set, which may be empty, together with a termination token. A *termination token* is either *end* to indicate that the server cannot provide more answers for any reason or *none* to assert that no more answers are possible. If a server is unable to deal with a query, e.g., due to syntactical errors, a *rejected* termination token is sent in the answer.

If the client specifies an answer bundle size bound in the query, the server does not send more answers then allowed by the answer bundle size bound. Together with the answer the server sends either a termination token to end the dialogue or an arbitrarily chosen *process handle* to allow the continuation of the query-answering-dialogue. However, even if the server sends a process handle with the answer it does not guarantee that there are more answers.

To continue a dialogue the client sends a *server continuation* request including the process handle and an answer bundle size bound for the next answer bundle. A server continuation may not necessarily be sent from the same client. The client can also pass the process handle to another client that then continues the query answering dialogue. If the server can't deliver any more answers for a server continuation request, it sends a termination token together with the probably empty answer set.

If the client does not want to continue the dialogue, the client can send a *server termination* request including the process handle. The server can use a received server termination request to possibly free up resources. Figure 3.1 illustrates the sequence of a query-answering-dialogue.

The specification provides some attributes for a server to promote the delivered quality of service or the so called *conformance level*. A server can guarantee to be *non-repeating*, i.e., no two answers with the same binding are delivered. The strictest level is called a *terse* server and only the most specific answers are delivered to the client. For example an answer is more general (subsumes another) if it only provides fewer bindings

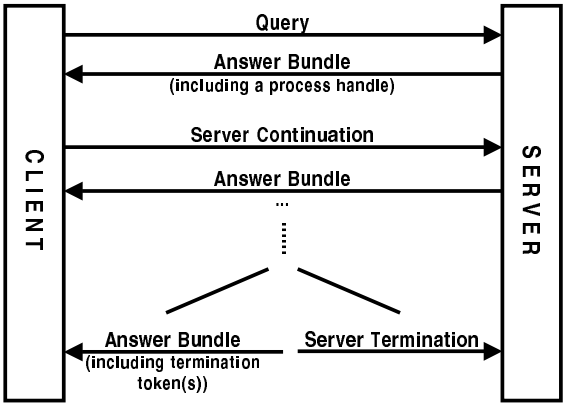

Figure 3.1: The query-answering dialogue.

for may-bind variables. Since this demand is very high for a server that produces the answers incrementally, a less restrictive conformance level is *serially terse*, where all delivered answers are more specific than previously delivered answers. Finally servers that guarantee to terminate with termination token *none* are called *complete*.

3.3 Query Classes

Since it is difficult to implement all of these requirements, the DQL specification explicitly allows a partial implementation. A DQL server can restrict itself to special *query classes*, e.g., a server may only support queries that conform to a pattern such as `?x rdf:type C`, where C is an DAML+OIL class expression, or `?x daml:subClassOf ?y` and reject all other queries. The server is then said to apply to these query classes.

3.4 OWL-QL

For OWL, the successor of DAML+OIL, there is also a first proposal of a query language called OWL-QL.[2] The proposal is very similar to the released DQL specification, but it is not yet official. Since OWL gains more and more popularity and is also accepted as a W3C standard, the implemented prototype already supports OWL knowledge bases to answer the queries and as soon as a formal OWL-QL specification is available the server could be extended to fully support an OWL-QL specification.

One thing that is missing in the proposed OWL-QL specification are the query-classes mentioned in the previous section. This would make it much more difficult to provide an implementation of an OWL-QL specification than it is for the DQL specification and the final OWL-QL specification is hopefully extended in this regard.

[2] `http://ksl.stanford.edu/projects/owl-ql`

Chapter 4

Realisation of a DQL Server Prototype

Now that the rolling-up technique to answer conjunctive queries and the DQL specification itself have been introduced, this chapter explains how the prototypical implementation of a DQL server has been realised and explains why distinct design decisions have been taken.

The software development process for this project follows an evolutionary prototyping approach. Not all features for a deployable product are implemented yet, but the prototype is meant to be extended and provides a basis for more advanced versions. From a software engineering point of view it is a vertical prototype model, because it implements all layers from the client over the query rolling-up process to the reasoning component, but it is not complete with regard to every functionality described in the DQL specification. It covers the most difficult part of a DQL server implementation and leaves simpler parts open for future versions. In addition to meeting the requirements of the specification there are also a lot of nice to have features or methods to improve the performance that were not implemented, but chapter 5 lists some ideas for future versions of the system. The experiences with this prototype also lead to suggestions for a future version of a DQL or OWL-QL specification, which are also covered in chapter 5.

4.1 The Architecture

DQL was designed as an agent-to-agent communication protocol and the knowledge bases used to answer a query may be distributed over various sources in the Semantic Web. Due to this requirement a web service architecture was chosen for the project realisation. Web services allow communication with different clients, i.e., a .NET application can interact with the service or a client written in Java or anything else that supports HTTP as a communication protocol. In addition, web services are self describing and their interfaces can be explored by parsing their web services description language (WSDL) [10] file.

Web services were favoured here over other middleware such as CORBA or Java RMI.

They are well standardised now and are able to use multiple high level protocols, such as HTTP or SMTP, to communicate with a remote service and do not depend on a specific programming language. Java RMI is in comparison only usable between Java applications, which is a clear limitation for an agent-to-agent communication protocol. CORBA does not expose this restriction, but compared to web services it is not so easy to use. Furthermore, much more efforts are currently made to extend web service standards and frameworks or services such as registries to promote an available service. The rich set of additional tools and services, like transaction services, concurrency control or authentication available for CORBA will surely also be available for web services in the future and currently theses services are not needed for the realisation of a DQL server.

Part of this project is also an example web client that allows a user to send queries to the server and then displays the answers as an HTML document.

Figure 4.1 shows the architecture of the implemented DQL server, together with the implemented client application. The DQL Server part is the main component of this work and is responsible for the rolling-up process as explained in chapter 2. The web service offers three methods: one to initiate a query dialogue, one to request more answers for a process handle of a formerly asked query and one to terminate a query-answering dialogue. This component then interacts with the main DQL Server and forwards the received parameters to the relevant methods of the DQL Server component.

The reasoner could be any reasoner that supports the DIG [1] interface. This implementation has been tested with Racer,[1] since Racer implements all ABox reasoning methods defined in the DIG interface.

The grey box symbolises other client applications such as a rich Java Swing GUI, a .NET application, another web service that uses the DQL server as part of its service or any other application that can use a web service.

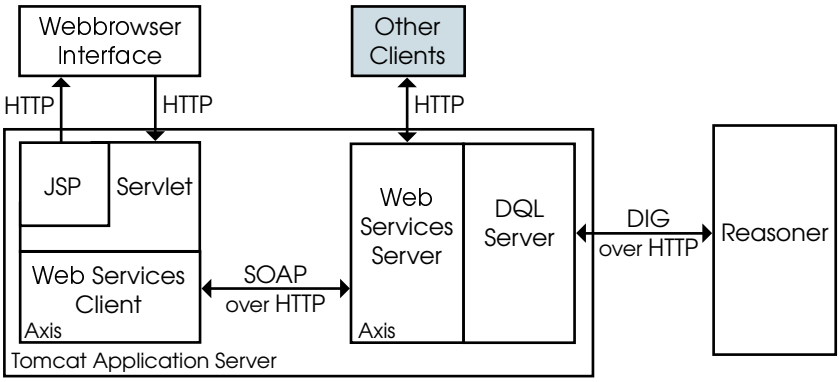

Figure 4.1: The chosen software architecture.

The web service client and the server of the provided implementation are both located on the same physical machine and therefore hosted by the same Tomcat instance. This is not necessary and can be changed easily.

[1] http://www.sts.tu-harburg.de/~r.f.moeller/racer

4.2 Used Tools, Products and Languages

The implementation was done in Java. The reason for this is that all other components that are used in this project, e.g., the DAML+OIL to DIG converter or the DIG interface classes, are also written in Java, and a rich number of frameworks for web services are also available in Java. To realise such a project in the given amount of time also makes it necessary to fall back on as much experience with tools and languages as possible, otherwise too much time would be spent on familiarisation with new tools. Java was, therefore, the best candidate for the implementation language, and the set up of other tools was more or less easy.

As an application server Jakarta Tomcat[2] with the Axis[3] web service framework was chosen. Axis is Apache's most recent web service framework, and compared to its successor Apache SOAP it supports the Web Service Description Language (WSDL). Application developers can generate the Java classes for a web service client from a .wsdl file.

JUnit[4] served as a testing framework for the project and an Ant[5] script deploys both the client and the server application to the Tomcat web server and can also run the JUnit tests to assert that the deployed files work as expected. For CVS versioning the savannah project server of the Hamburg University of Applied Sciences was used. Apache's log4J[6] served as a logging framework. It is easy to use and provides several predefined categories, such as info, warning and error. A configuration file defines the verbosity and the output medium on an application or on a per class level. During the development various outputs were logged, but due to performance losses this is reduced to only error logging in the final version of the prototype.

To parse the queries, a small parser was generated using JavaCC (Java Compiler Compiler),[7] which is similar to the well known Lex/ Yacc programs or their successors Flex/ Bison.[8] The differences to Lex/ Yacc are that JavaCC produces Java code instead of C. Furthermore it is a LL(k) parser generator, i.e., it parses top-down, while Yacc is a LALR(1) parser generator that parses bottom-up. Top-down parsing is completely sufficient for parsing the queries, and the use of a Java parser allows smooth interaction with the other components.

As Description Logic reasoner Racer[9] was used.

[2] http://jakarta.apache.org/tomcat
[3] http://ws.apache.org/axis
[4] http://www.junit.org
[5] http://ant.apache.org
[6] http://logging.apache.org/log4j/docs
[7] https://javacc.dev.java.net
[8] http://dinosaur.compilertools.net
[9] http://www.sts.tu-harburg.de/~r.f.moeller/racer

4.3 The Components

The following sections describe the components involved in and developed for the DQL web service following the direction from the web service interface to the core query-answering component. All Java classes are equipped with detailed JavaDoc comments and to find out how a special method works, the reader is advised to look at the provided API documentation.

4.3.1 The Web Service Interface

To start a query-answering-dialogue a client calls the `query()` method of the DQL web service with the necessary parameters to answer the query (the query, the URL of a knowledge base and optionally an answer bundle size bound and an answer pattern). A method parameter for the premise is already implemented, but the values are currently ignored, since the allowed time for the project made it necessary to focus on the main parts and the premise can easily be added later without major changes to the query-answering algorithm. The premise should be transferred to the reasoner before the queries are sent, since statements in the premise have to be treated as if they were a normal part of the knowledge base.

The web service interface also offers the method `nextResults()`, which allows the request of further answers for a given process handle. The method `terminate()` ends a query-answering-dialogue for a given process handle. Currently all answers are produced for the first query call and if more answers are available than allowed by the answer bundle size bound, the rest of the answers is stored on the server together with the process handle.

Figure 4.2 shows a UML class diagram of the interface class that was used to create the web service and figure 4.3 shows the classes that are relevant for the web service. All these classes are in the package `dql.server.webservice`. `DQLService` is an implementation of the `IDQLService` interface and the classes `AnswerSet` and `QueryAnswer` are types that are used to deliver query answers to a client. The `DQLService` class is not the real implementation; the class follows the facade design pattern and delegates the parameters to the corresponding components and delivers query answers to the client.

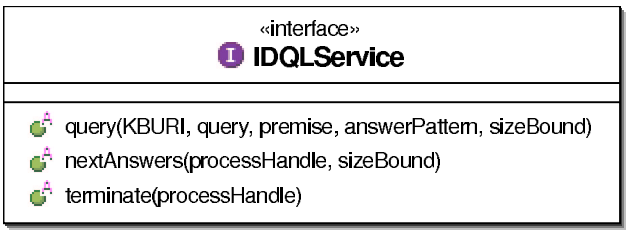

Figure 4.2: The web service interface.

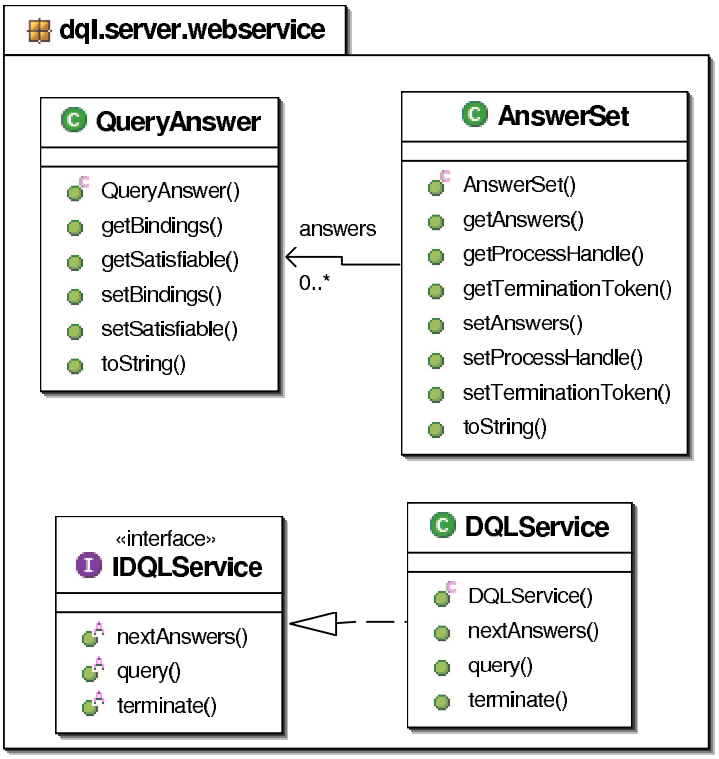

Figure 4.3: The web service package.

4.3.2 The DQL Server Component

The main component is the class `DQLServer`. It passes the query to a query parser component, the knowledge base to a converter (a component that converts DAML+OIL or OWL to DIG statements) and forwards the converted knowledge base to the reasoner. It also initiates the rolling-up process on the produced query graph and finally returns the computed answers back to the `DQLService` class. The `DQLServer` class is not responsible for storing answers in a cache, since this is not part of the query answering process. Instead the `DQLService` facade class uses the class `AnswerSetCache` that is responsible for storing and returning cached answers.

All parts that belong to the main component are stored in the package `dql.server`. The UML deployment diagram in figure 4.4 illustrates the components that are incorporated in the realisation of the DQL service. The components labelled with library are not developed as part of this project.

4.3.3 The Query Parser

The queries are currently not written in DAML+OIL or OWL, since only a subset of these languages is supported (conjunctive queries) and the syntax of a query would be very long in DAML+OIL or OWL. An extended version of the server could of course

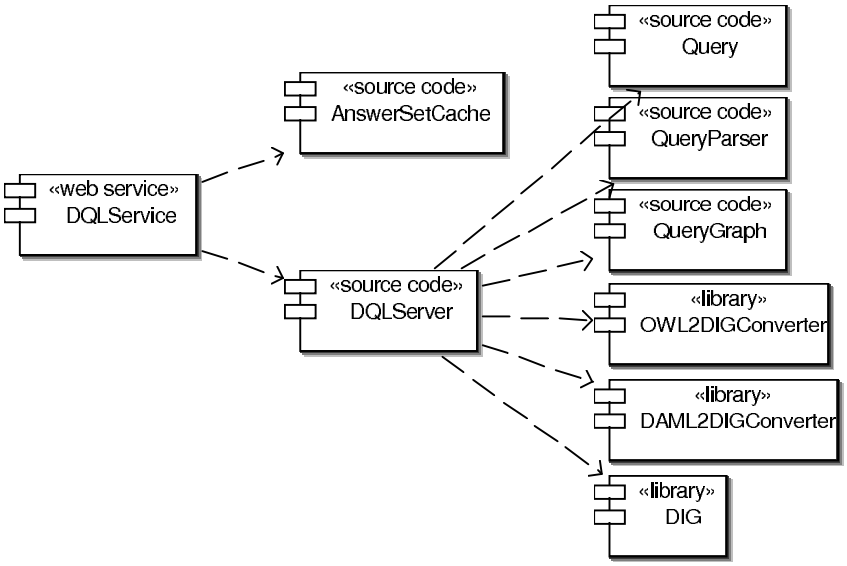

Figure 4.4: An UML deployment diagram of the DQL service.

allow a DAML+OIL or OWL query syntax and use a parser such as the one provided with the Jena framework[10] to read the queries.

The different types of variables are indicated by a prefix, as introduced in chapter 2: ! is the prefix to indicate a don't bind variable and ? is the prefix for must-bind variables. May-bind variables are currently not supported as already mentioned in section 3.1. To parse the query a small parser was implemented with JavaCC. JavaCC needs a .jj file as input containing an EBNF grammar [14, 30] together with actions and token definitions as regular expressions. Table 4.1 shows the used EBNF grammar. The non-terminals are `query`, `term`, `crName`, `objectName` and `roleFiller` and the terminals are characters, like '(', or defined regular expression, denoted as `<MB>`, `<DB>` and `<ID>` for a must-bind variable, a don't-bind variable or an individual, concept or role name respectively. The regular expression `<STDCHAR>` is used as shortcut. The parser also builds the query graph as described in section 2.2.3 while parsing a query. To realise this, a graph object is instantiated before the parsing starts, and the actions for the non-terminals contain corresponding Java method calls to add a node, a role or a concept assertion to a node.

The grammar file for JavaCC and all files that are generated by JavaCC are in the Java package `dql.server.parser`.

Table 4.1 shows the EBNF grammar used to parse the queries.

4.3.4 Knowledge Base Loading

The knowledge bases are passed to the class `DQLServer` as URIs, so they could reference a file on the local file system or they could point to a knowledge base available over the

[10] `http://jena.sourceforge.net`

```
query       → term ("," term)*
term        → crName "(" objectName roleFiller ")"
crName      → <ID>
objectName  → <MB> | <DB> | <ID>
roleFiller  → ("," objectName)?
<MB  : ["?","#","a"-"z","A"-"Z","0"-"9","_"]
       (":","#","a"-"z","A"-"Z","0"-"9","_")* >
<DB  : ["!","#","a"-"z","A"-"Z","0"-"9","_"]
       (":","#","a"-"z","A"-"Z","0"-"9","_")* >
<ID  : ["#","a"-"z","A"-"Z","0"-"9","_"]
       (":","/",".","?","-","#","a"-"z","A"-"Z","0"-"9","_")* >
```

Table 4.1: The EBNF grammar for the query parser.

Hyper Text Transfer Protocol (HTTP) or the File Transfer Protocol (FTP). The URIs must end with .daml for a DAML+OIL knowledge base or with .owl for an OWL knowledge base. The OWL standard[11] specifies three sublanguages, which are called OWL Lite, OWL DL and OWL Full. Current Description Logic reasoners are not able to use all features of OWL Full, which is the most expressive sublanguage of OWL. Knowledge bases that contain such unsupported features are rejected by the DQL server.

Depending on the type of the ontology (DAML+OIL or OWL) they are passed to the appropriate DIG converter. Both converters are libraries from the University of Manchester and transform DAML+OIL or OWL into DIG statements. These statements are then passed to the reasoner that is currently connected to the DQL Server.

4.3.5 Interaction with the Reasoner

The connection to a reasoner is established over the DIG Interface [1], which is a standardised XML interface for Description Logics systems developed by the DL Implementation Group (DIG).[12]

A part of the DIG project is the Java API to communicate with DIG compliant reasoners, like Racer or FaCT++. All parts of the DIG project are available from the Sourceforge home page.[13]

The DQL Server tries to read the URL for the reasoner from a properties file that is named `dqlserver.properties` and is located in the package `dql.server`. If this property file is not accessible the DQL server tries to connect to `http://localhost:8080/` to see if a local reasoner is available there. If none of this works, all query() method calls will cause an exception.

The class `ExtendedResponse` in the package `dql.server` implements methods that facilitate the analysis of the reasoner's response, e.g., to see if the knowledge base loading

[11] `http://www.w3.org/TR/2004/REC-owl-features-20040210`
[12] `http://dl.kr.org/dig`
[13] `http://dig.sourceforge.net`

was successful one has to call only one method with the reasoner response as a parameter.

Currently all interactions with the reasoner are done in a kind of batch mode, so all requests (tell and ask) are collected for the first phase of the algorithm and if necessary also for the second phase to check the candidates for must-bind variables and then sent to the reasoner at once. This limits the network transportation overhead to a minimum, since the reasoner may not necessarily run on the same physical machine as the DQL server.

The DIG interface was chosen since it offers an implementation independent way for the communication with a reasoner. The standard becomes more and more accepted and has currently been updated to version 1.1. This additional indirection, compared to a direct connection to a reasoner over its proprietary interface, may cause longer query answering times, but it was preferred since it allows an easy switch between all reasoners that support the interface.

While writing this report the Jena framework has been extended to support the connection of DAML+OIL or OWL knowledge bases to a DL reasoner over the DIG standard, so this framework could be an alternative to the converters used here. The DQLServer class could build a Jena model for the knowledge bases and use it to interact with the reasoner. Currently the implementation is not yet included in an official Jena release and very little documentation[14] is available along with a technical report about the experiences with the DIG standard during the extension of Jena [11], so this is only an alternative for a future version of the DQL server. It would also be necessary to test if a switch to Jena would increase the performance, otherwise there is no need to change the components.

4.3.6 The Query Graph Component

All classes that belong to the graph representation of a query are bundled in the package `dql.server.querygraph`. Figure 4.5 shows an UML class diagram of these classes.

The class `Graph` implements the rolling-up technique as described in section 2.3. The graph contains a list of its nodes and a node is represented by the Java class `Node`. The nodes manage their relations to other nodes with an adjacent list. An adjacent list is more applicable than a centrally managed matrix for the relations since the graph is build incrementally while parsing the query. For each role assertion a directed edge is added from the outgoing node to its successor and vice versa, but the inverse direction is kept separately, since it is only needed to traverse the graph and is not part of the query. The class `NodeIterator` allows a convenient iteration over all related nodes. Although the query is represented as a directed graph the term leaf is used here. This is explained by the fact that the underlying undirected graph is per definition in tree form and a node is called leaf here, if it is a leaf in the underlying undirected graph.

The method `startRollingUp()` initialises the rolling-up process. First all individuals are replaced by their representative concepts (see section 2.3.4 for an explanation), then

[14] `http://jena.sourceforge.net/how-to/dig-reasoner.html`

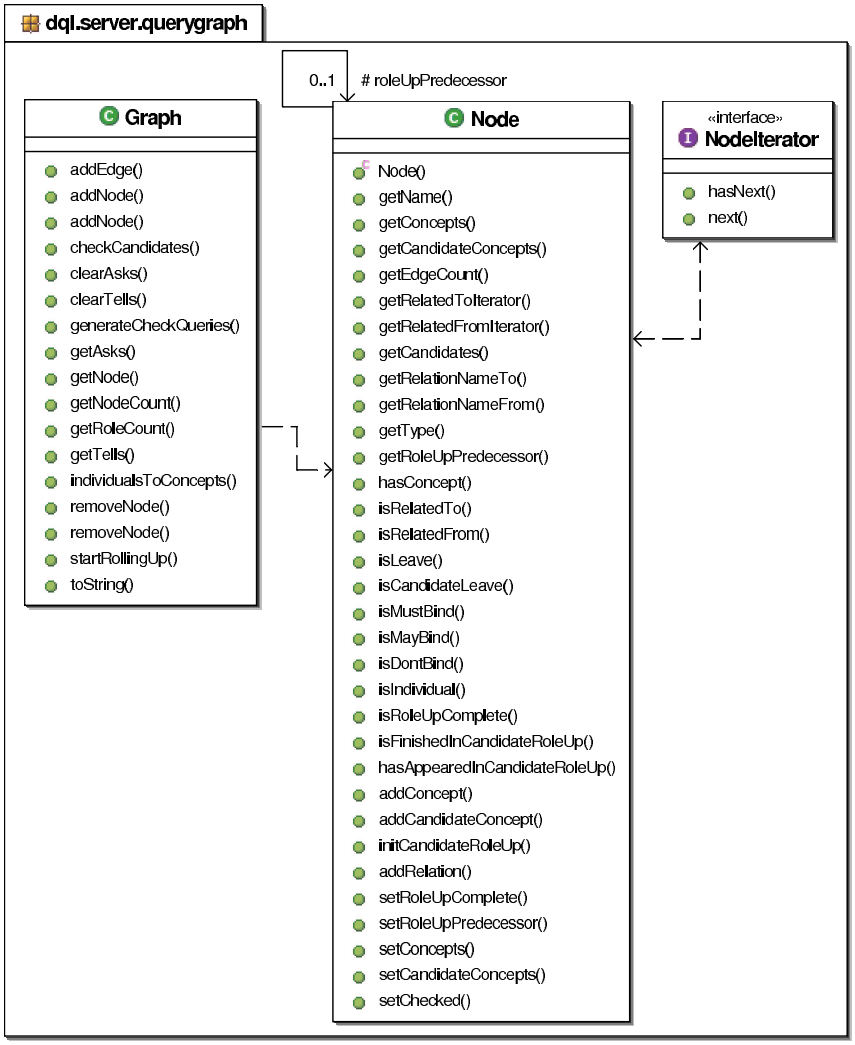

Figure 4.5: The UML class diagram of the query graph classes.

all individual or don't-bind leaves are rolled-up until only one node is left or this process must stop since only must-bind variables are leaves. If only one node is left, the query can be transformed to a boolean query or to a concept instance query. Otherwise the rolling-up technique is used to compute candidates for the bindings of the must-bind variables as described in section 2.3.5.

After this first rolling-up phase the generated queries are sent to the reasoner. If the query contains at most one must-bind variable the reasoner already returns the final query answer, otherwise the reasoner returns candidates for the bindings of the must-bind variables.

If at least one of the must-bind variables has no candidates for its binding, the query has an empty answer set and the query-answering algorithm terminates. Otherwise boolean queries for each possible candidate combination are sent to the reasoner to test which combinations are valid answers.

4.3.7 Query Types

In this implementation all interactions with the reasoner are regarded as queries. There are mainly two types of them: ask queries that want to know something from the reasoner, e.g., which individuals are instances of a concept, and tell queries that pass information to the reasoner, e.g., that an individual is an instance of a concept. The terms tell and ask are also used in the DIG specification. Since there are different types of queries for tell as well as for ask queries, the package `dql.server.query` contains different query type classes arranged in an inheritance hierarchy, together with two interfaces that allow users of the classes to interact with all (ask) queries in the same way. Tell queries are only used for the representative concepts of individuals and to state that all representative concepts are disjoint,[15] i.e., the tell queries are derived directly from the abstract query superclass, while ask queries are arranged in a deeper inheritance hierarchy under the abstract class `AskQuery`. Figure 4.6 shows the type hierarchy without the subclasses of the abstract class `AskQuery` for a better overview. The class `AskQuery` with its subclasses is illustrated in figure 4.7.

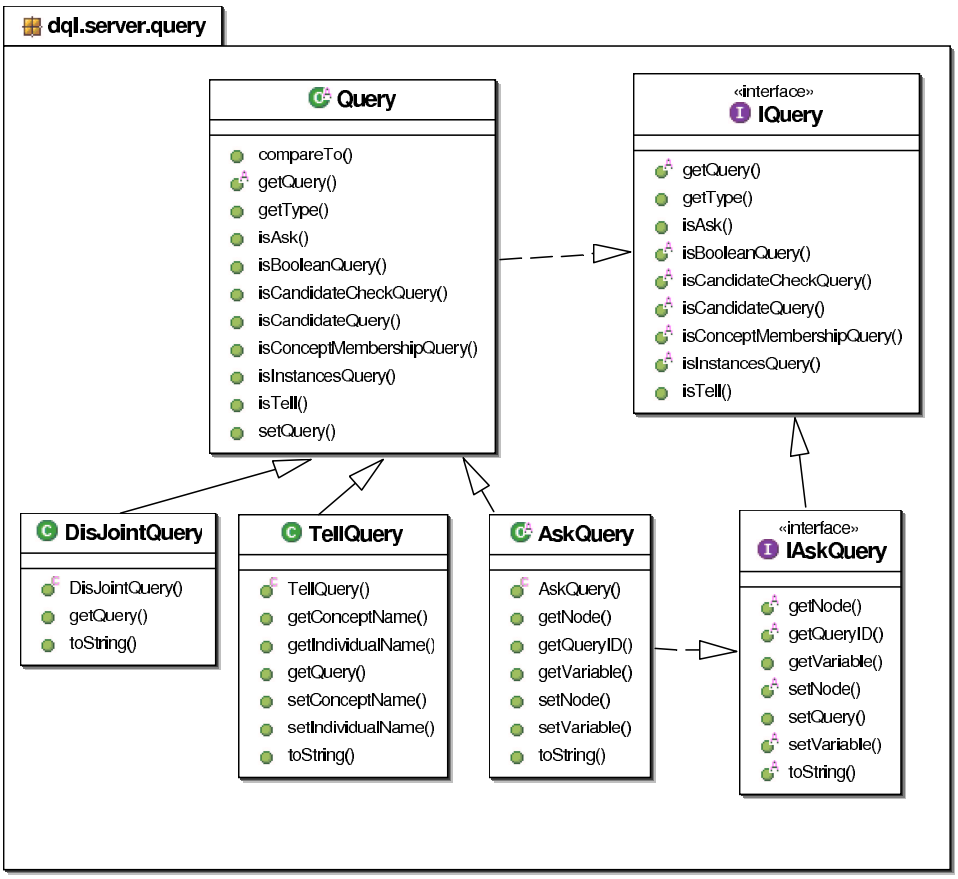

Figure 4.6: The UML class diagram of the query classes.

[15] Current Description Logic reasoners impose the Unique Name Assumption (UNA) for individuals, and the disjointness axiom keeps this for the representative concepts.

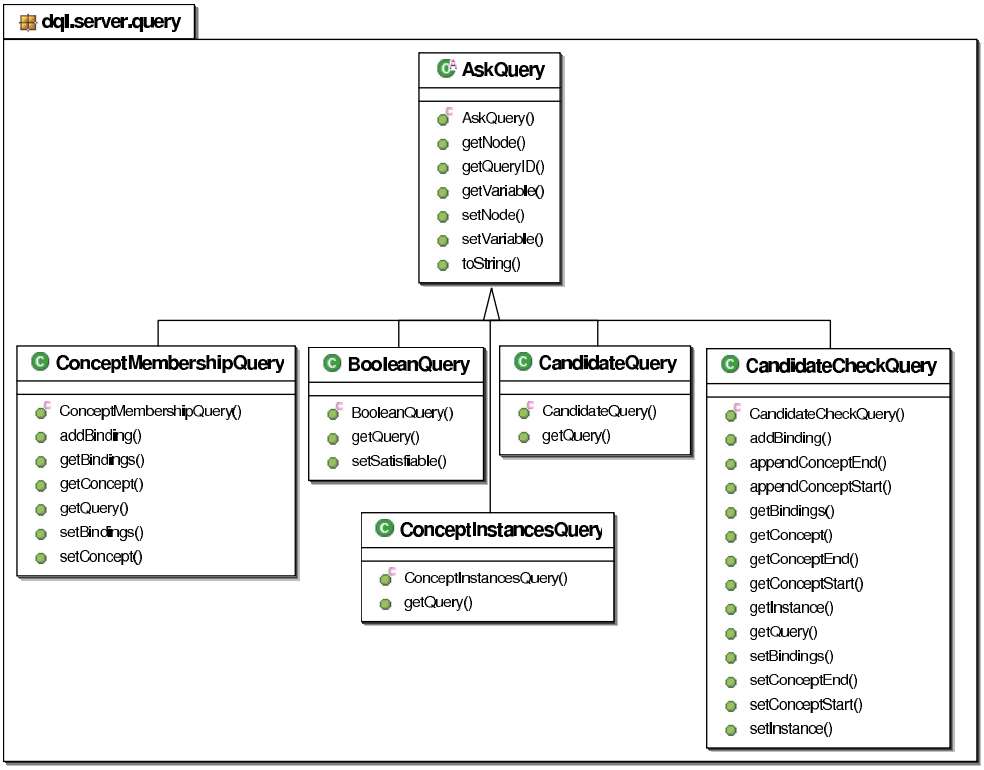

Figure 4.7: The UML class diagram of the AskQuery subclasses.

4.3.8 Query Answers

Query answers are returned in a set represented by the Java class `AnswerSet`. An answer set contains at least one answer and at most as many answers as allowed by the answer bundle size bound variable or all computed answers if the sizeBound variable is zero or negative. Normally the Java class Integer with the value null would be more applicable, but for a web service the class Integer and the primitive type `int` are both mapped to the XML schema type `xsd:int` for transportation over the SOAP protocol and both types are then unmarshalled to an primitive Java type int. Therefore, the `DQLServer` class works with Integer as preferred and the `DQLService` class, which is the web interface facade, works with `int` and does the mapping to Integer.

In addition to the answers for a query an answer set also includes the termination token or the process handle, whichever is appropriate.

On the server side the answers are stored in the class `ServerAnswerSet`. This class can be stored in the answer set cache and provides a method to receive an answer set of a specified size for delivery to the client. In this way it is easy to prepare the next answer set for the specified size of a `nextResults()` request. In addition, the use of a simpler answer set class as the return value of the web service avoided the implementation of special serializers and deserializers for the class. If the class complies with the Java Bean Standard, which specifies that a class has to have an empty default constructor and `getVariable()` plus `setVariable()` methods for each used instance variable and

nothing else, the default Java Bean serializer class can be used for serialization and de-serialization. This also saves time for the client implementers of the web service, since they also need not implement a serializer.

A query can have two kinds of answer. If the query contained no must-bind variables the returned answer set consists of only one answer with true as its value if all parts of the query are entailed by the used knowledge base and false otherwise. The returned answer contains no bindings in this case. If the query contained at least one must-bind variable the answer set may contain more answers. Each answer contains one binding for each must-bind variable. These bindings are stored in a map. If all must-bind variables in a query are replaced by their binding, and all remaining don't-bind variables are treated as existentially quantified, the query must be entailed by the knowledge base used to answer the query.

The class `ServerAnswerSet` and the class `AnswerSetCache` both reside in the package `dql.server` (see figure 4.8), while the classes `AnswerSet` and `QueryAnswer` together with their interfaces are located in the `dql.server.webservice` package, since they are delivered to the client of the web service. A UML class diagram for this package was already given in section 4.3.1 on page 24.

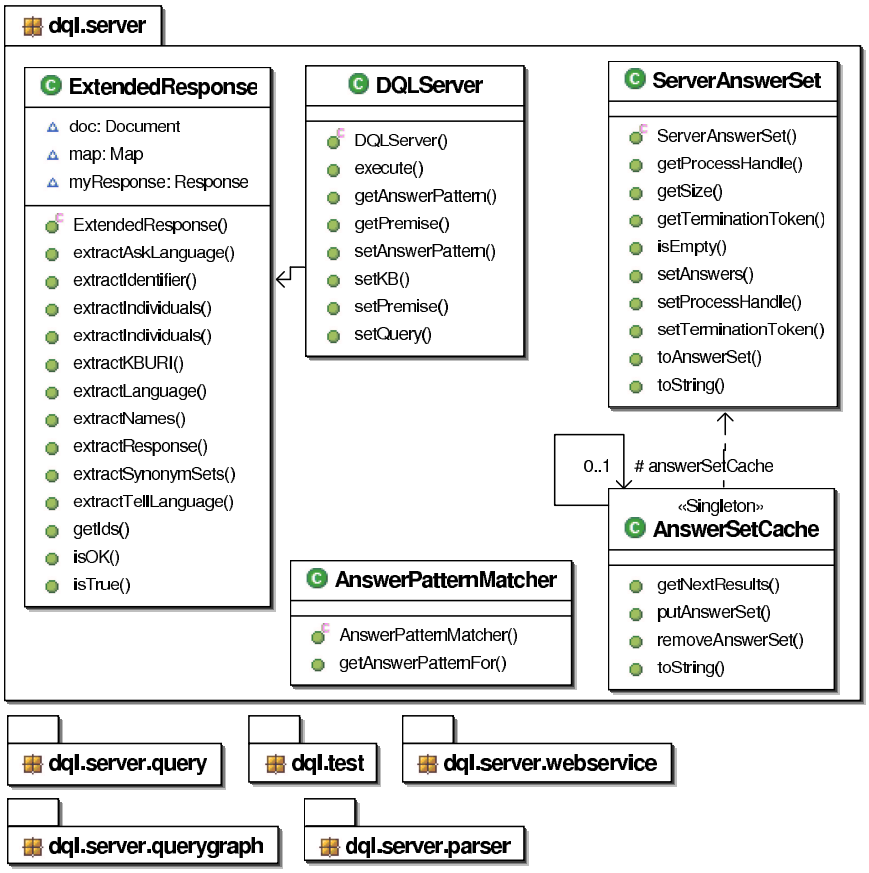

Figure 4.8: The UML class diagram of the package `dql.server`.

4.3.9 The Answer Set Cache

If a query has more answers than the server is allowed to return, the remaining answers are stored in an answer set cache. The corresponding Java class is `AnswerSetCache` in the package `dql.server`. The class is implemented as a singleton, to ensure that only one instance is available in the system. This is necessary for two reasons:

1. Web services can't guarantee (without extra efforts) that two requests from the same client are mapped to the same object on the server, i.e., if the `query()` method is executed by one object this object need not be the one that also handles a `nextResults()` request for the client. This makes it impossible to store the answers in an instance variable. This behaviour is known as web sessions. In a session the state of the application is saved on a per client basis. Web services can be forced to support sessions, but a normal configuration does not support this.
2. The DQL specification allows any client that has a valid process handle to request more answers for this handle, even if the original `query()` request was sent by another client. For this reason a normal web session would also not be suitable.

With a singleton only one instance of a class is available and this instance stores the answer sets and returns them on demand. When an answer set becomes empty it is removed from the cache and if a client requests an answer set that is not in the cache an empty answer set with an end termination token is returned.

4.4 A Query Processing Sequence

Figure 4.9 is an UML sequence diagram illustrating the collaboration of the components during a query answering process. The actor `DQL web service` is also a software component, namely the web service answering the query request, but the DQL server itself is a component with a clear boundary to the offered web service, i.e., the DQL web service can be seen as a client of the component.

Several actions have been taken to improve performance. One optimisation is to execute fast tasks that may cause an end of the query-answering process as early as possible, e.g., parsing a query is normally fast, since queries are much shorter than for example a knowledge base and if there is a syntax error in the query none of the other components need to be involved.

In two cases the process is finished after the first query phase. One case is, if at most one must-bind variable was in the query, then the first reasoner response already includes the query answer. The other is, if the query is not entailed by the knowledge base. This results in an empty candidate set for at least one must-bind variable or a returned false value for a boolean query asking if a specified individual exists in the knowledge base or is an instance of a given concept.

In all other cases a second interaction with the reasoner is necessary to verify all possible

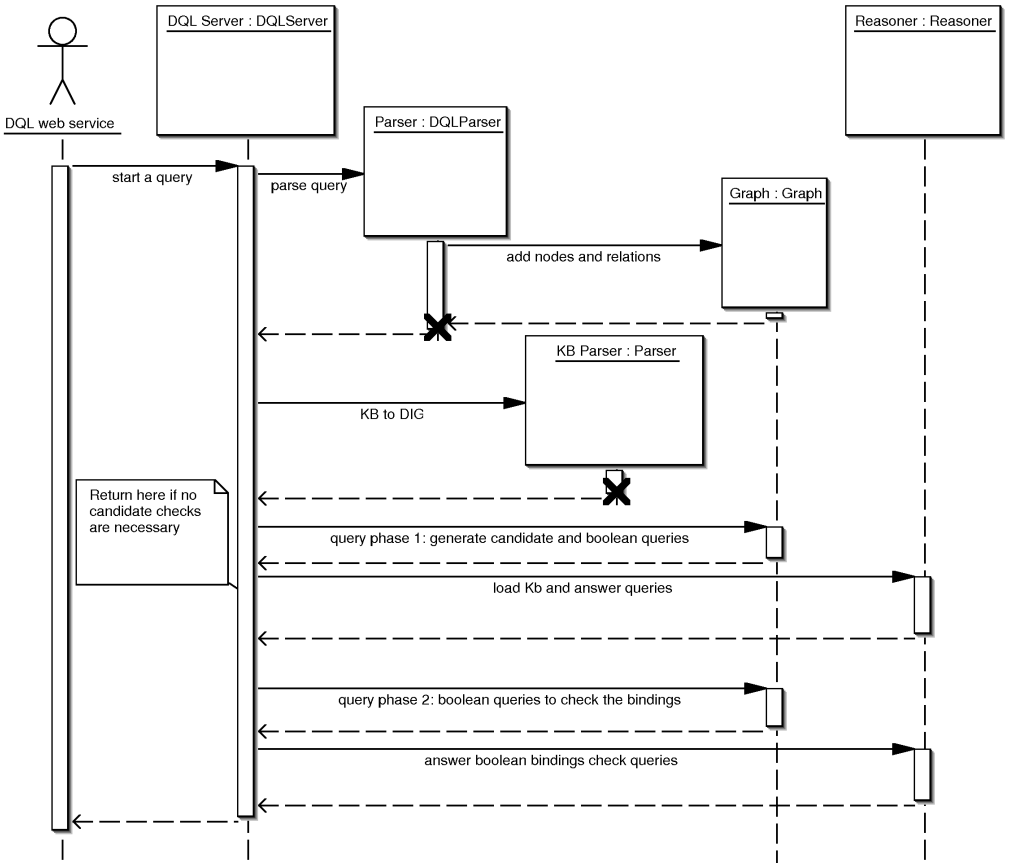

Figure 4.9: The UML sequence diagram for query answering.

combinations of the received binding candidates. This is the most costly part of the implementation besides the loading time for a knowledge base that is determined by the size of the knowledge base itself.

4.5 Error Handling

The specification defines that if for any reasons a server can not deal with a query it has to return the termination token *rejected* in an empty answer set. In addition to this, the provided implementation also defines a `getErrorMessage()` method that contains an explanation of the caused error or failure.

Whenever an error occurs in the DQL server component, e.g., a syntax error in the query or knowledge base or the reasoner may be unavailable for some reason, the error is caught, logged and re-thrown with an appropriate description of the exception. The DQL web service (that is the facade class DQLService) catches all exceptions, creates an empty answer set with rejected termination token and the message of the caught exception, i.e., whenever the service is available the client will receive an answer set for its query and in case of an error this answer set also provides an explanation.

4.6 Testing

JUnit[16] is a regression testing framework to support developers in the software development process. A good introduction into test driven software development is given by Kent Beck [3], one of the authors of JUnit. For each software unit the developer should write a test that executes defined methods and asserts that defined conditions are met before and/or after a method has been executed. A regression test runs the unit tests of all components. This can help to find possibly occurring side effects, after a change in one of the components. If a tests does not result in a defined condition, the test fails and therefore also the whole test suite fails. For example the Eclipse IDE[17] has a build in graphical user interface for JUnit that signals green if all tests were executed as expected and red otherwise and the used deployment tool Ant also supports the execution of JUnit tests as part of a software build process.

For the DQL server, tests were implemented for all larger components, which test different methods against predefined results. The tests can be executed on demand and they are also part of the defined Ant deployment process for the DQL server components. The tests help to assure that specified requirements for the software, e.g., defined by the DQL specification, are met and they save time, since it is not necessary to test every class after a change again by executing the class's main method with different examples.

4.7 The DQL Client Interface

Another part of this implementation is a web service client. This was not specified as part of the project, but is rather useful to demonstrate the system. In addition, it shows one possibility of how the provided web service may be used.

The implementation is not described in much detail, since it is not of the realisation of a DQL server, but the system architecture diagram on page 21 shows the general layout of the client. It is mainly composed of one servlet[18] that collects the parameters that a user enters into an HTML form and passes the parameters to the DQL web service. All classes needed for the interaction with the web service were build by the wsdl2java program that is a part of the Jakarta Axis framework, see also section 4.2. After the servlet has received a result from the DQL web service the request is forwarded to a JavaServer Pages (JSP)[19] page. JSP are much easier to use for HTML output than a servlet, since a servlet can generate output only by using Java's `PrintWriter` classes while JSP can conveniently switch between Java and HTML parts.

The figures on the following pages illustrate the client interface. Figure 4.10 shows the front-end for the user. It allows to specify a local knowledge base file or the URL of a

[16] `http://www.junit.org`
[17] `http://www.eclipse.org`
[18] `http://java.sun.com/products/servlet/whitepaper.html`
[19] `http://java.sun.com/products/jsp/whitepaper.html`

knowledge base, the answer bundle size bound, the query and an answer pattern. It is necessary to use the fully qualified names for concept, role and individual names as in the knowledge base itself. The user can also specify a process handle and request more answers for this. If there are answers stored for the process handle on the server the server will return them.

Figure 4.11 shows the answer page. If the answer included a process handle to indicate that the client can make further calls, the client can choose one of three options: to request more answers (then the size bound for the next answer set must be given), to terminate this request and hereby allow the server to free resources or to start a new call. If the server has no more answers in its cache a termination token is returned and the user has only the option to ask a new query. This is displayed in figure 4.12.

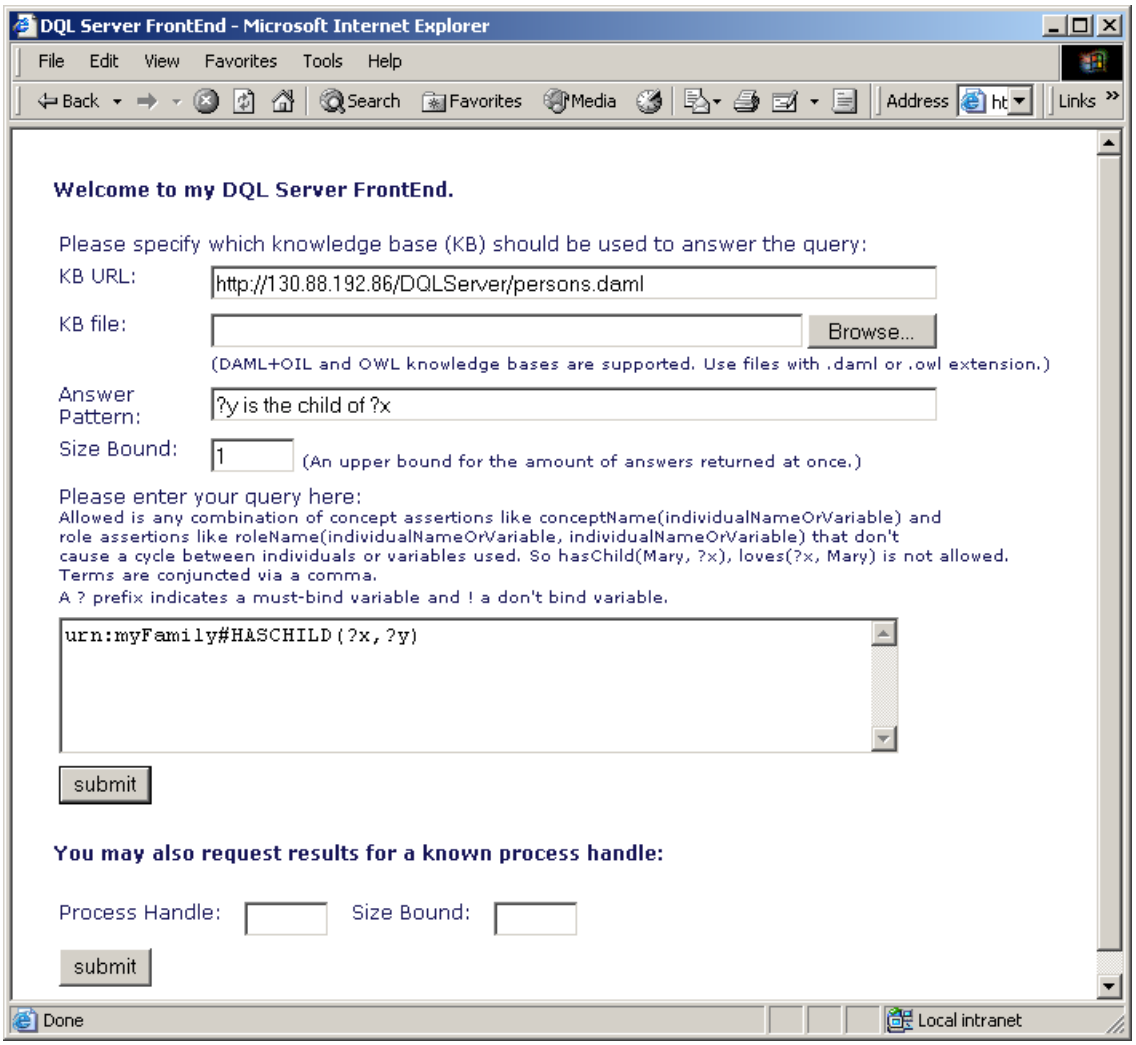

Figure 4.10: The DQL client start page.

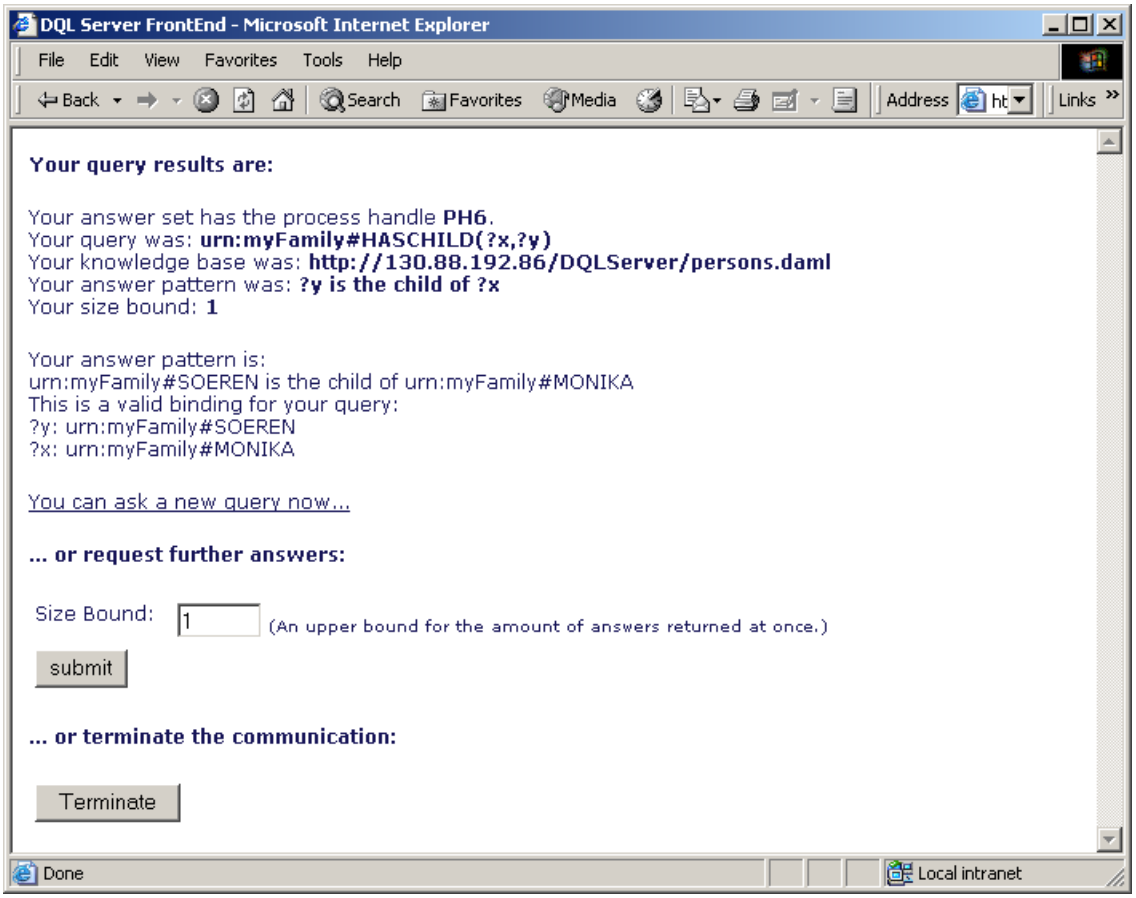

Figure 4.11: A DQL client answer page with further answers available.

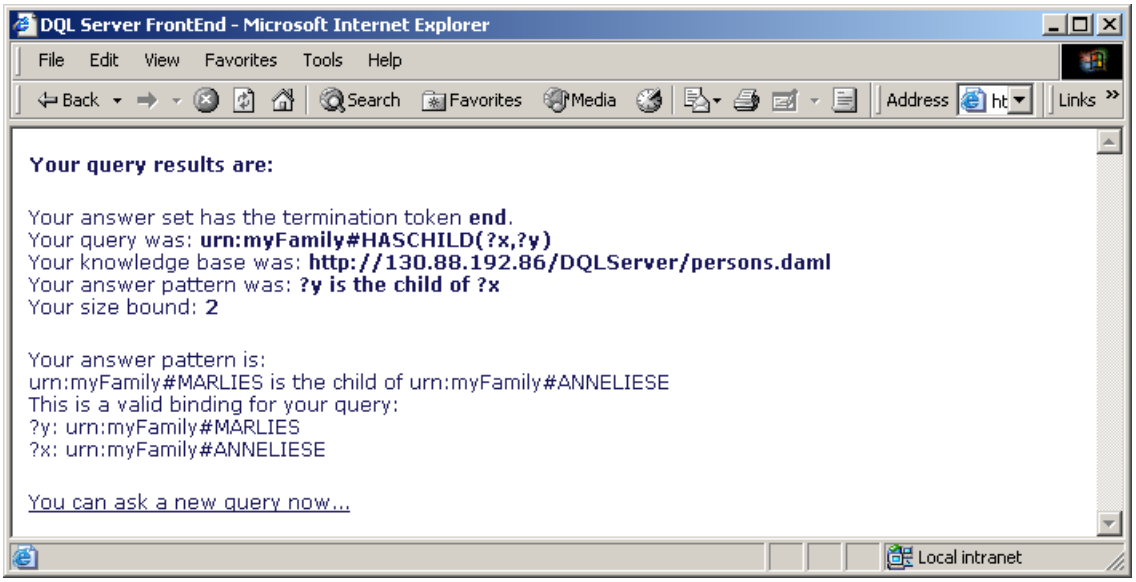

Figure 4.12: A DQL client answer page with termination token.

Chapter 5

Conclusion

The preceding chapters have given an overview of a query answering algorithm and its implementation in a DQL server prototype that uses a Description Logic reasoner in the background. Although limited in functionality the approach worked well for acyclic conjunctive queries. A further extension of the implemented prototype is in some regards easy, e.g., the support of further query types such as a query for sub-concepts of a given concept, but there are still some open questions that are addressed in this chapter. The chapter also lists some improvements for a future version of the prototype and compares it with other query-answering systems.

5.1 Improvements for Future Versions

5.1.1 Extended Query Support

Future versions could implement other query classes such as queries asking for subconcepts. These kind of queries and many other simple queries are already supported by most of the current reasoners, thus an extended implementation has to translate such queries into a form that the reasoner uses, but no application of complicated techniques like the rolling-up technique described here are necessary.

In this prototype the conjuncted queries are not allowed to include cycles, since the used rolling-up technique then fails to find a proper starting-point in the graph representation. Tessaris [26] developed some approaches, but there is still no general solution for the rolling-up technique to deal with cycles, so research in this field is necessary to overcome this limitation.

5.1.2 Multi-Thread Safe Reasoner Connections

For this first prototype the connection to the reasoner is not multi-thread safe, i.e., client requests have to be serialized for the interaction with the reasoner. The DIG protocol provides features for the simultaneous use of multiple knowledge bases, so this could be achieved with some extra effort. One main point here is a change regarding the used DAML+OIL converter, since the converted knowledge bases always include a statement to clear everything else in the reasoner. To identify a specific knowledge base it is also necessary to include a unique identifier in the knowledge base loading statement, which is not supported by the converter. A solution would be to change the generated DIG statements before they are passed to the reasoner or switch to the Jena framework, as proposed in section 4.3.5.

5.1.3 Proper Use of the Termination Token

A useful feature for a future version would be the proper use of the termination token. Currently, the termination token *end* is used to end every query-answering-dialogue, but the use of the termination token *none*, to indicate that the returned answer set is complete, would give the client a better information about the quality of the received results, although the use of the termination token end is not contrary to the specification.

Since completeness is not achievable for DAML+OIL in general, either the server must check which subset of the language has been used or the reasoner itself must communicate whether the answer set is complete or not (but this kind of information is currently not available over the DIG interface). As already mentioned in section 4.3.5, the Jena RDF framework was extended during the time of this report to support the connection to a DL reasoner over the DIG interface [11]. Because of this extension it is worth investigating, if it is much more expensive to build up a model of the knowledge base in Jena and communicate to the reasoner over this model instead of simply converting the knowledge base. This would provide the advantage that one can use the Jena framework to inspect the knowledge base and look for statements that may cause incompleteness and use the end termination token only in these cases.

5.1.4 Interaction with the Reasoner

Currently, the implementation computes all answers at once and then stores answers in a cache in the case of an answer bundle size bound that is smaller than the number of computed answers. This approach minimises the number of interactions with the reasoner. This was done because the reasoner may reside on a physically different machine and no general assumptions can be made about the quality of the connection to the reasoner. This approach also avoids saving the whole state of a request in the server, which would be necessary to continue a started query later on. For a continuation, a reload of the knowledge base is also necessary or the knowledge base must remain in the reasoner's memory until the query is finally terminated. A reloading of the knowledge base will

possibly decrease or even eliminate the time savings, while the holding of many large knowledge bases in memory may slow down the reasoner or cause a permanent memory lack.

For a small number of received candidates for a binding, the current method of batched queries is definitely the best approach, but large candidate sets and the computation of many more answers than requested by a client may cause time delays for the delivery of a first answer set.

Another method to interact with the reasoner is to send a boolean query directly after an answer candidate has been assembled to test if the used bindings are valid and do so until an answer set of the desired size is ready or all candidates have been tested. This strategy does not consider the connection speed to the reasoner and causes more communication overhead, since a new request is sent for each possible answer. It would also be necessary to store the complete state of a query with the resulting shortcomings described above.

Advanced connection handling could combine both of these strategies depending on different factors such as the quality of the connection to the reasoner (e.g., measured by sending pings or a simple request to the reasoner and average the measured times), the number of candidate answers with respect to the specified answer bundle size bound, the available memory or other factors of the current system environment.

5.1.5 Improved Candidate Checks

The additional candidate checks for must-bind variables may be the cause of major delays in query answer retrieval, but unfortunately there are only few optimisations that eliminate this additional check. On improvement is to use structural knowledge (e.g., knowledge about transitive roles) to eliminate some of the candidates.

Two other possible methods take advantage of the knowledge about how many candidates each node has. One method starts at a node with few candidates and checks the candidates for direct successors first. If these are not suitable, more distant nodes with probably many candidates can be skipped. The other method starts at the node with the most candidates and asks which of these candidates fit to a tuple of candidates for the other nodes. Both methods could reduce the number of necessary boolean checks significantly, but which one is the best for a specific situation is future work.

5.2 Identified Improvements for the DQL Specification

During the development of the DQL server prototype some shortcomings of the existing specification occurred and some improvements for a next release of the specification are suggested here.

5.2.1 Security

One major point is that clients can request further answers for a query if they know a valid process handle. The client need not be the same client that originally initialised the query dialogue, so clients can guess a process handle and request answers for it. This may not be problematic, but if a client has passed a knowledge base to the server that is not available to the public the server may probably give away (some aspects of) this protected knowledge. Technical solutions are available to prevent such hijacking of query answers, the simplest one being the use of sessions. A more secure approach would be the use of an access control policy language such as XACML [15].

5.2.2 External Query Language Definition

Another point is the undefined external query language for a DQL server. Although this allows an easy adaptation of the standard to other knowledge representation languages, as has been done for OWL with the OWL-QL specification proposal, interoperability between different DQL server implementations is nearly impossible. It is desirable to extend the specification with regards to an external language that covers at least the basic operations.

A possible solution could also be the release of a DQL Concrete Specification that extends the DQL Abstract Specification. For new knowledge representation languages the abstract specification could remain unchanged and just a new Concrete Specification has to be released.

Error Handling

Closely related to this is also the definition of proper error handling. The specification defines three termination tokens and the rejected token signals an error, but provides no further explanations. The specification allows the definition of further termination tokens, but the invention of arbitrary termination tokens by each implementer is not helpful and does not allow agents to find out the reason for an error without knowledge about implementation details.

Conformance Level and Query Classes

It is also not clear how a server can report its conformance level (e.g., does the server only return non-redundant answers) or its supported query classes (e.g., conjunctive queries). Unless there is a means to communicate such a level an agent cannot use this information. Humans may read this in documentation, but an agent-to-agent protocol should make this information available in machine understandable form.

5.2.3 Forced Different or Equal Bindings

While testing the server with different queries the need arose to specify that two variables should have different bindings. An example would be a query asking for children whose mother is married to a person that is not the father of the child. Without a kind of differentFrom statement one can not exclude that the binding for the husband is not the same as the binding for the father. OWL includes a differentFrom statement in its language but DAML+OIL does not, i.e., a language extension with differentFrom and sameAs statements for DQL should be considered. Example 5.1 shows such a query: if Mary is married to Joe and Bill is their child, a valid binding for the query would be `?m:mary`, `?c:bill`, `?h:joe` and `?f:joe` and there are no means to specify that `?f` and `?h` should have a different binding.

Example 5.1
`(?m, ?c):haschild` $\wedge$ `(?m, ?h):marriedto` $\wedge$ `(?c, ?f):hasfather`

5.2.4 Knowledge Base Loading

The DQL specification allows the client to use a variable instead of giving a reference to a specific knowledge base. This can help to speed up the query-answering process, since the server can use some permanently loaded knowledge bases (such as a default knowledge base) or the server could reuse an already loaded knowledge base. In this case it would be desirable to allow clients to query for available knowledge bases and specify one of those, a feature that is not envisaged by the current DQL specification. Whenever a client provides a knowledge base itself an additional parameter could be added to tell the server if this knowledge base could be cached for further queries of this client or if the knowledge base may be used for other clients also. If caching is allowed, a client should be equipped with a method to force a reload, e.g., by providing a last change date for the knowledge base, similar to the method used to force a reload of an HTML page that is cached by a proxy.

5.2.5 Answer Bundle Size Bound

Currently the specification defines that a request for more answers for a process handle needs an answer bundle size bound parameter. This does not consider that a client perhaps wants to receive all remaining answers at once, which is allowed for the first query request. Of course a client could specify a very large answer bundle size bound, but a simple solution would be to omit the size bound parameter as is allowed for the first request.

5.3 Comparison with Other Systems

Apart from this prototype there is one other implementation of the DQL specification made available by the Knowledge Systems Laboratory of the Stanford University.[1] Other implementations are currently not available, but the reasoner Racer has recently been extended to support a very rich query language called Racer Query Language (RQL) [16] and is also compared to the DQL prototype presented here.

5.3.1 The Stanford OWL-QL Server

Unfortunately, the DQL server implementation provided by Stanford University is not running and leaves the user with a Java connection exception, since the server that is referenced in the application[2] is not available. Stanford also provides an implementation for the DQL successor OWL-QL[3] that can use DAML+OIL knowledge bases and uses the same technique (JTP[4] as a first order logic theorem prover to answer the queries) in the background. This implementation is regarded as comparable to the DQL server and was used instead for this comparison.

While testing this server it produced some curious results for more complicated queries. Example 5.2 contains a query that was given to the server for the knowledge base illustrated in example 5.1 and also contains a statement that the concepts `A`, `B`, `C` and `D` are disjoint. This was added since DAML+OIL has no unique name assumption and for this example the server should not assume that two of the individual names point to the same individual. However, the results of the Stanford server are the same without the disjointness axioms.

Example 5.2

`(?x, !y):r` $\wedge$ `(!y, ?z):r` $\wedge$ `(!y):C`

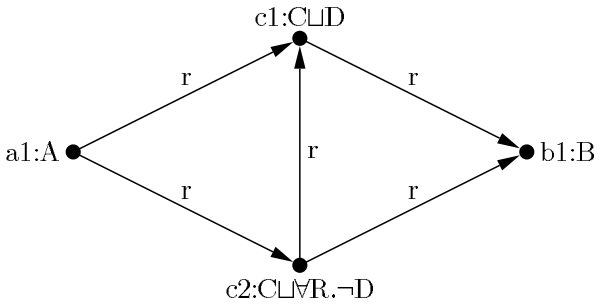

Figure 5.1: The knowledge base used for the query in example 5.2.

[1] `http://ksl.stanford.edu/projects/dql`
[2] `http://onto2.stanford.edu:8080/dql/servlet/DQLServer`
[3] `http://ksl.stanford.edu/projects/owl-ql`
[4] `http://www.ksl.stanford.edu/software/JTP`

The query is difficult, since there really exists no binding for `!y`, but it is clear that either `c1` or `c2` must be a `C`. It is just not decidable which one is the `C`. If `!y` is a don't-bind variable, as in this case, the query has exactly one answer, namely `a1` as a binding for `?x` and `b1` as a binding for `?z`.

The Stanford's OWL-QL server does not find the correct answer tuple but ends the dialogue with termination token end and is compliant with the specification in this case.

However, if the query is changed slightly, and instead of `?z` the individual `b1` is given (see example 5.3), the server produces incorrect answers. The returned answer set (actually it is no longer a set since it contains duplicates so the term answer bag would be more appropriate) is {`a1, a1, c2, c2, c1`} which is cannot be explained with respect to the knowledge base since only `a1` is connected to `b1` over an (not nameable) instance of the concept `C`.

Example 5.3 `(?x, !y):r ∧ (!y, b1):r ∧ (!y):C`

The prototype implemented as part of this project answers both of these queries correctly, the first one in example 5.2 with `a1` as binding for `?x` and `b1` as binding for `?z` and the second one with `a1` as binding for `?x`.

Altogether the Stanford implementation seems to offer richer query facilities, e.g., it allows to query for subconcepts of a given concept, which is not yet included in this prototype, but it terminates for many queries with the termination token end without delivering answers or even produces incorrect answers.

5.3.2 Racer Query Language

The Racer Query Language (RQL) [16] offers extensive query support, but it uses a different approach to answer queries and does not support don't-bind variables. This makes query answering easier, since such ambiguities, as for `c1` and `c2` in the knowledge base of the previous example, cannot occur in a query. Moreover, RQL does not comply with the DQL specification and is therefore not really comparable to this implementation. To optimise the computation of the bindings, RQL uses heuristics from the field of Constraint Programming such as instantiation of the most constrained variable first.

References

[1] Sean Bechhofer, *The DIG Description Logic interface: DIG/1.1*, Tech. report, University of Manchester, Oxford Road, Manchester M13 9PL, February 2003, http://dl-web.man.ac.uk/dig/2003/02/interface.pdf.

[2] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein, *OWL web ontology language reference*, Tech. report, W3C, February 2004, http://www.w3.org/TR/2004/REC-owl-ref-20040210.

[3] Kent Beck, *Test driven development: By example*, Addison-Wesley Pub Co, November 2002.

[4] T. Berners-Lee, R. Fielding, and L. Masinter, *RFC 2396: Uniform Resource Identifiers (URI): Generic syntax*, URL, August 1998.

[5] Tim Berners-Lee, Mark Fischetti, and Michael L. Dertouzos, *Weaving the web: The original design and ultimate destiny of the world wide web by its inventor*, Harper San Francisco, 1999.

[6] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, *XQuery 1.0: An XML query language*, URL, November 2003, http://www.w3.org/TR/2003/WD-xquery-20031112.

[7] Willem N. Borst, *Construction of engineering ontologies for knowledge sharing and reuse*, Ph.D. thesis, Universiteit Twente, Enschede, The Netherlands, September 1997.

[8] Tim Bray, Jean Paoli, Michael Sperberg-McQueen, Eve Maler, and François Yergeau, *Extensible markup language (XML) 1.0 (third edition)*, URL, February 2004, http://www.w3.org/TR/2004/REC-xml-20040204.

[9] Dan Brickley, *RDF vocabulary description language 1.0*, URL, February 2004, http://www.w3.org/TR/2004/REC-rdf-schema-20040210.

[10] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer, and Sanjiva Weerawarana, *Web Services Description Language (WSDL) version 2.0 part 1: Core language*, URL, March 2004, http://www.w3.org/TR/2004/WD-wsdl20-20040326.

[11] Ian Dickinson, *Implementation experience with the DIG 1.1 specification*, Technical Report HPL-2004-85, Hewlett-Packard, Digital Media Systems Laboratory, Bristol, May 2004, http://www.hpl.hp.com/techreports/2004/HPL-2004-85.pdf.

[12] Dieter Fensel, Frank van Harmelen, Ian Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider, *OIL: An ontology infrastructure for the semantic web*, IEEE Intelligent Systems **16** (2001), no. 2, 38–45.

[13] Richard Fikes, Pat Hayes, and Ian Horrocks, *DAML Query Language (DQL) abstract specification*, URL, April 2003, http://www.daml.org/2003/04/dql.

[14] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), *ISO/IEC 14977 : 1996(E)*, 1996, http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[15] Simon Godik and Tim Moses, *eXtensible Access Control Markup Language (XACML) version 1.0*, URL, February 2003, http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf.

[16] Volker Haarslev, Ralf Möller, Ragnhild Van Der Straeten, and Michael Wessel, *Extended query facilities for Racer and an application to software-engineering problems*, To appear in: Proceedings of the International Workshop on Description Logics (DL-2004), Whistler, BC, Canada, June 2004.

[17] Ian Horrocks, *DAML+OIL: a reason-able web ontology language*, Proceedings of EDBT 2002, Lecture Notes in Computer Science, no. 2287, Springer, March 2002, pp. 2–13.

[18] Ian Horrocks, Dieter Fensel, Jeen Broekstra, Stefan Decker, Michael Erdmann, Carol A. Goble, Frank van Harmelen, Michael Klein, Steffen Staab, and Rudi Studer, *The Ontology Interchange Language OIL*, Tech. report, Free University of Amsterdam, 2000, http://www.ontoknowledge.org/oil/TR/oil.long.html.

[19] Ian Horrocks, Ulrike Sattler, and Stephan Tobies, *Practical Reasoning for Expressive Description Logics*, Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99) (H. Ganzinger, D. McAllester, and A. Voronkov, eds.), Lecture Notes in Artificial Intelligence, no. 1705, Springer-Verlag, 1999, pp. 161–180.

[20] Ian Horrocks and Sergio Tessaris, *Querying the semantic web: a formal approach*, Proceedings of the 13th International Semantic Web Conference, ISWC (Ian Horrocks and J. Hendler, eds.), Lecture Notes in Computer Science, no. 2342, 2002, pp. 177–191.

[21] Ian Horrocks, Frank van Harmelen, and Peter F. Patel-Schneider, *DAML+OIL*, URL, March 2001, http://www.daml.org/2001/03/daml+oil-index.html.

[22] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl, *RQL: A declarative query language for RDF*, Proceedings of the eleventh international conference on World Wide Web (Honolulu, Hawaii, USA), ACM Press, New York, USA, May 2002, pp. 592–603.

[23] Graham Klyne, Jeremy J. Carroll, and Brian McBride, *Resource Description Framework (RDF) concepts and abstract syntax*, URL, February 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210.

[24] Ora Lassila and Ralph Swick, *Resource Description Framework (RDF) model and syntax specification*, URL, February 1999, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

[25] DARPA Agent Markup Language (DAML) Program, *DAML-ONT initial release*, URL, October 2000, http://www.daml.org/2000/10/daml-ont.html.

[26] Sergio Tessaris, *Questions and answers: reasoning and querying in Description Logic*, Phd thesis, University of Manchester, 2001.

[27] Frank van Harmelen, Peter F. Patel-Schneider, and Ian Horrocks, *A model-theoretic semantics for DAML+OIL*, URL, March 2001, http://www.daml.org/2001/03/model-theoretic-semantics.html.

[28] Frank van Harmelen, Peter F. Patel-Schneider, and Ian Horrocks, *Reference description of the DAML+OIL (march 2001) ontology markup language*, URL, March 2001, http://www.daml.org/2001/03/reference.

[29] Junhu Wang, Michael Maher, and Rodney Topor, *Rewriting general conjunctive queries using views*, Proceedings of the thirteenth Australian conference on Database technologies (Australia Melbourne, Victoria, Australia), ACM International Conference Proceeding Series, vol. 5, Australian Computer Society, Inc., Darlinghurst, Australia, 2002, pp. 197–206.

[30] Niklaus Wirth, *What can we do about the unnecessary diversity of notation for syntactic definitions?*, Communications of the ACM archive **20** (1977), 822–823.

[31] William A. Woods and James G. Schmolze, *The KL-ONE family*, Computer and Mathematics with Applications, special issue: Semantic Networks in Artificial Intelligence **23** (1992), no. 2-5, 133–177.

Appendix A

Appendix

A.1 Notation

Ontology and Description Logics	
class/concept names	uppercase
role/property names	lowercase
individual/instance names	lowercase
representative concept for the individual i	P_i

Variables	
don't-bind variables	! prefix
may-bind variables	~ prefix
must-bind variables	? prefix

Graphs	
individuals	unfilled node
don't-bind variables	unfilled node
must-bind variables	filled node

A.2 Abbreviations

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CVS	Concurrent Versions System
DAML	DARPA Agent Markup Language
DAML+OIL	DARPA Agent Markup Language with Ontology Inference Layer
DARPA	Defense Advanced Research Projects Agency
DIG	DL Implementation Group

DL	Description Logic
DQL	DAML Query Language
EBNF	Extended Backus-Naur Form
FOL	First Order Logic
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
JSP	Java Server Pages
KB	Knowledge Base
LL(k)	A parser parsing **l**eft-right with **l**eftmost derivation with **k** tokens of look-ahead.
LALR	A parser parsing with 1 token of **l**ook-**a**head from **l**eft-to-right with **r**ightmost-derivation.
OIL	Ontology Inference Layer
OWL	Web Ontology Language
OWL-QL	OWL Query Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RMI	Remote Method Invocation
RQL	Racer Query Language
RQL	RDF Query Language
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
UML	Unified Modelling Language
UNA	Unique Name Assumption
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WSDL	Web Service Description Language
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

A.3 The Enclosed CD

This Bachelor report contains an appendix of program listings, hardware descriptions etc. on a CD (disk or supplementary booklet). This Appendix is deposited with Prof. Dr. Klauck.

A.3.1 Application Files

The attached CD includes the developed prototype and the client application as web archive. The server application is named `DQLServer.war` and can be found in the direc-

tory DQLServer and the client application is named DQLClient.war and can be found in the directory DQLClient. Both applications include all necessary libraries. To install the applications they need to be copied into the webapps directory of a Tomcat server.

The applications have been tested with Tomcat 4.1.29, J2SDK 1.4.2_02 and Racer 1.7.19 on a Windows 2000 workstation. Since Racer and Tomcat both work on port 8080 the Tomcat port has been changed to the normal HTTP port 80. If instead the reasoner should run on a different port the dqlserver.properties file in the DQLServer.war in the directory WEB-INF/classes has to be adopted.

If the web service client should run on a different physical machine the client files for the web service have to be generated again with a .wsdl file from the running DQL server or the IP address has to be changed manually in the web service client class.

A.3.2 Dependent Applications

All applications needed to run or to compile the DQL server and the client on a Windows machine are included in the directory software. This includes Tomcat 4.1.29, J2SDK 1.4.2_02, JavaCC 3.2, JUnit 3.8.1 and Racer 1.7.19. The use of the applications is limited to the respective licence agreements.

A.3.3 The Report and the References

This report is included as Adobe Acrobat file named report.pdf. The Acrobat Reader is available at http://www.adobe.com/products/acrobat/readstep2.html.

The references include some web links, mainly W3C standards. The links for the W3C standards point to exactly the version that is referred to in the report and the W3C does not changes these links even if a newer version of a specification appears, but to make sure that all references are available for the reader, the CD includes an offline version of the links in the directory references. All references that include a URL are listed in the file index.html and the link of each reference points to the offline version of the URL.

A.3.4 The Project Source Files

All developed source code is included in the directory src inside the DQLServer and DQLClient directory and an Ant build file is located at the top level folder of each project folder.

A.3.5 Documentation

The JavaDoc API documentation for the DQL server and the client can be found in their respective project directories in the subfolder javadoc. The documentation includes de-

tailed comments for each method.

A.4 Model Theoretic Semantics of DAML+OIL

Table A.2 shows the model theoretic semantics of RDF triples relevant to DAML+OIL. This table is an extract from the DAML+OIL webpage (see [27]), but the syntax was adapted to the one used throughout this report.

C and D represent classes, P and S represent roles, A and B represent individuals, L a literal and R a restriction. Note that in DAML+OIL there is no Unique Name Assumption as used by DL reasoners, so A and B are interpreted as sets of names for an individual. DAML+OIL distinguishes between a non-empty set of DAML+OIL objects, denoted by AD, and a disjoint set of XML Schema data types, like integers, denoted by DD. The domain, denoted by UD (in DL Δ), is the union of AD and DD.

The interpretation function $(\cdot^I)$ applied to a class maps into subsets of either AD or DD, restrictions are mapped into subsets of AD, roles into subsets of $AD \times UD$ (object properties: $AD \times AD$, datatype properties: $AD \times DD$) and individuals resp. RDF literals into subsets of AD resp. DD.

The notation $P(x)$ is the set of objects that form the image of x under P, for P a set of 2-tuples.

Syntactic Structure	Semantic Constraint
(rdf:type, C, rdfs:Class)	$C^I \subseteq UD$
(rdf:type, C, Class)	$C^I \subseteq AD$
(rdf:type, C, Datatype)	$C^I \subseteq DD$
(rdf:type, C, Restriction)	$C^I \subseteq AD$
(rdf:type, P, Property)	$P^I \subseteq AD \times UD$
(rdf:type, P, ObjectProperty)	$P^I \subseteq AD \times AD$
(rdf:type, P, DatatypeProperty)	$P^I \subseteq AD \times DD$
	$\text{Thing}^I = AD$
	$\text{Nothing}^I = \emptyset$
	$(\text{rdfs:Literal})^I = DD$
L	for L a literal, $L^I \subseteq DD$ and if x is in the interpretation of an XML Schema datatype then $x \in L^I$ iff x has L as its lexical representation for some XML Schema datatype
(rdf:type, A, C)	$A^I \subseteq C^I$
(rdf:type, A, D) (rdf:value, A, L) (rdf:type, L, rdfs:Literal)	for D an XML Schema datatype, A^I is the singleton set containing the element of D^I that has lexical representation L, provided that there is one, otherwise $A^I = \emptyset$

Syntactic Structure	Semantic Constraint
(P, A, B)	$(x,y) \in P^I$, for some $x \subseteq A^I$ and $y \subseteq B^I$, provided that $A^I \subseteq AD$
(equivalentTo, C, D) (equivalentTo, R, S) (equivalentTo, A, B)	$C^I = D^I$ $R^I = S^I$ $A^I = B^I$
(rdfs:subClassOf, C, D)	$C^I \subseteq D^I$
(rdfs:subPropertyOf, P, S)	$P^I \subseteq S^I$
(sameClassAs, C, D)	$C^I = D^I$
(samePropertyAs, P, S)	$P^I = S^I$
(sameIndividualAs, A, B)	$A^I = B^I$
(disjointWith, C, D)	$C^I \cap D^I = \emptyset$
(differentIndividualFrom, A, B)	$A^I \cap B^I = \emptyset$
(rdf:type, $\{C_1,\ldots,C_n\}$, Disjoint)	$C_i^I \cap C_j^I = \emptyset$ for $1 \leq i < j \leq n$
(unionOf, C, $\{C_1,\ldots,C_n\}$)	$C^I = (C_1^I \cup \ldots \cup C_n^I) \cap AD$
(disjointUnionOf, C, $\{C_1,\ldots,C_n\}$)	$C^I = (C_1^I \cup \ldots \cup C_n^I) \cap AD$ $C_i^I \cap C_j^I = \emptyset$ for $1 \leq i < j \leq n$
(intersectionOf, C, $\{C_1,\ldots,C_n\}$)	$C^I = C_1^I \cap \ldots \cap C_n^I \cap AD$
(complementOf, C, D)	$C^I \cap D^I = \emptyset$ $C^I \cup D^I = AD$
(oneOf, C, $\{A_1,\ldots,A_n\}$)	$C^I = A_1^I \cup \ldots \cup A_n^I \cap AD$
(rdfs:domain, P, C)	if $(x,y) \in P^I$ then $x \in C^I$
(rdfs:range, P, C)	if $(x,y) \in P^I$ then $y \in C^I$
(inverseOf, P, S)	for $y \in AD$, $(x,y) \in P^I$ iff $(y,x) \in S^I$
(rdf:type, P, TransitiveProperty)	for $y \in AD$, if $(x,y) \in P^I$ and $(y,z) \in P^I$ then $(x,z) \in P^I$
(rdf:type, P, UniqueProperty)	if $(x,y) \in P^I$ and $(x,z) \in P^I$ then $y = z$
(rdf:type, P, UnambiguousProperty)	for $y \in AD$, if $(x,y) \in P^I$ and $(z,y) \in P^I$ then $x = z$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (toClass, R, C)	$x \in R^I$ iff $P^I(\{x\}) \subseteq C^I$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (hasValue, R, V)	$x \in R^I$ iff $\lvert P^I(\{x\}) \cap V^I \rvert > 0$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (hasClass, R, C)	$x \in R^I$ iff $\lvert P^I(\{x\}) \cap C^I \rvert > 0$

Syntactic Structure	Semantic Constraint
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (minCardinality, R, n)	$x \in R^I$ iff $\|P^I(\{x\})\| \geq n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (maxCardinality, R, n)	$x \in R^I$ iff $\|P^I(\{x\})\| \leq n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (cardinality, R, n)	$x \in R^I$ iff $\|P^I(\{x\})\| = n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (minCardinalityQ, R, n) (hasClassQ, R, C)	$x \in R^I$ iff $\|P^I(\{x\}) \cap C^I\| \geq n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (maxCardinalityQ, R, n) (hasClassQ, R, C)	$x \in R^I$ iff $\|P^I(\{x\}) \cap C^I\| \leq n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, ObjectProperty) (cardinalityQ, R, n) (hasClassQ, R, C)	$x \in R^I$ iff $\|P^I(\{x\}) \cap C^I\| = n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (toClass, R, C)	$x \in R^I$ iff $P^I(\{x\}) \subseteq C^I$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (hasValue, R, V)	$x \in R^I$ iff $\|P^I(\{x\}) \cap V^I\| > 0$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (hasClass, R, C)	$x \in R^I$ iff $\|P^I(\{x\}) \cap C^I\| > 0$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (minCardinality, R, n)	$x \in R^I$ iff $\|P^I(\{x\})\| \geq n$

Syntactic Structure	Semantic Constraint
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (maxCardinality, R, n)	$x \in R^I$ iff $\lvert P^I(\{x\})\rvert \leq n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (cardinality, R, n)	$x \in R^I$ iff $\lvert P^I(\{x\})\rvert = n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (minCardinalityQ, R, n) (hasClassQ, R, C)	$x \in R^I$ iff $\lvert P^I(\{x\}) \cap C^I\rvert \geq n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (maxCardinalityQ, R, n) (hasClassQ, R, C)	$x \in R^I$ iff $\lvert P^I(\{x\}) \cap C^I\rvert \leq n$
(rdf:type, R, Restriction) (onProperty, R, P) (rdf:type, P, DatatypeProperty) (cardinalityQ, R, n) (hasClassQ, R, C)	$x \in R^I$ iff $\lvert P^I(\{x\}) \cap C^I\rvert = n$

Table A.2: Model Theoretic Semantics of DAML+OIL

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung *Informatik PO 2001* nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

_____ _____

Ort, Datum Unterschrift