D2.5.2 Report on Query Language Design and Standardisation

Jeff Z. Pan (UoM)
Enrico Franconi, Sergio Tessaris (FUB), Birte Glimm (UoM),
Wolf Siberski (L3S), Vassilis Tzouvaras, Giorgos Stamou, (ITI)
Ian Horrocks, Lei Li (UoM) and Holger Wache (VU)

Abstract.
EU-IST Network of Excellence (NoE) IST-2004-507482 KWEB
Deliverable D2.5.2 (WP2.5)

In the progress of realising the Semantic Web, developing and supporting Semantic Web query languages are among the most useful and important research problems. In [PFT+04], we have provided a unified framework for OWL-based rule and query languages. In this report, we focus on the problems of query answering for Semantic Web query languages (such as RDF, OWL DL and OWL-E) in the OWL-QL specification.

Keyword list: description logics, ontology language, query language, RDF, OWL DL, OWL-E

Document Identifier	KWEB/2004/D2.5.2/v1.0
Project	KWEB EU-IST-2004-507482
Version	v1.0
Date	Dec 16, 2004
State	Final
Distribution	public

Knowledge Web Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2004-507482.

University of Innsbruck (UIBK) - Coordinator
Institute of Computer Science
Technikerstrasse 13
A-6020 Innsbruck
Austria
Contact person: Dieter Fensel
E-mail address: dieter.fensel@uibk.ac.at

École Polytechnique Fédérale de Lausanne (EPFL)
Computer Science Department
Swiss Federal Institute of Technology
IN (Ecublens), CH-1015 Lausanne
Switzerland
Contact person: Boi Faltings
E-mail address: boi.faltings@epfl.ch

France Telecom (FT)
4 Rue du Clos Courtel
35512 Cesson Sévigné
France. PO Box 91226
Contact person : Alain Leger
E-mail address: alain.leger@rd.francetelecom.com

Freie Universität Berlin (FU Berlin)
Takustrasse 9
14195 Berlin
Germany
Contact person: Robert Tolksdorf
E-mail address: tolk@inf.fu-berlin.de

Free University of Bozen-Bolzano (FUB)
Piazza Domenicani 3
39100 Bolzano
Italy
Contact person: Enrico Franconi
E-mail address: franconi@inf.unibz.it

Institut National de Recherche en Informatique et en Automatique (INRIA)
ZIRST - 655 avenue de l'Europe -
Montbonnot Saint Martin
38334 Saint-Ismier
France
Contact person: Jérôme Euzenat
E-mail address: Jerome.Euzenat@inrialpes.fr

Centre for Research and Technology Hellas / Informatics and Telematics Institute (ITI-CERTH)
1st km Thermi - Panorama road
57001 Thermi-Thessaloniki
Greece. Po Box 361
Contact person: Michael G. Strintzis
E-mail address: strintzi@iti.gr

Learning Lab Lower Saxony (L3S)
Expo Plaza 1
30539 Hannover
Germany
Contact person: Wolfgang Nejdl
E-mail address: nejdl@learninglab.de

National University of Ireland Galway (NUIG)
National University of Ireland
Science and Technology Building
University Road
Galway
Ireland
Contact person: Christoph Bussler
E-mail address: chris.bussler@deri.ie

The Open University (OU)
Knowledge Media Institute
The Open University
Milton Keynes, MK7 6AA
United Kingdom
Contact person: Enrico Motta
E-mail address: e.motta@open.ac.uk

Universidad Politécnica de Madrid (UPM)
Campus de Montegancedo sn
28660 Boadilla del Monte
Spain
Contact person: Asunción Gómez Pérez
E-mail address: asun@fi.upm.es

University of Karlsruhe (UKARL)
Institut für Angewandte Informatik und Formale
Beschreibungsverfahren - AIFB
Universität Karlsruhe
D-76128 Karlsruhe
Germany
Contact person: Rudi Studer
E-mail address: studer@aifb.uni-karlsruhe.de

University of Liverpool (UniLiv)
Chadwick Building, Peach Street
L697ZF Liverpool
United Kingdom
Contact person: Michael Wooldridge
E-mail address: M.J.Wooldridge@csc.liv.ac.uk

University of Manchester (UoM)
Room 2.32. Kilburn Building, Department of Computer
Science, University of Manchester, Oxford Road
Manchester, M13 9PL
United Kingdom
Contact person: Carole Goble
E-mail address: carole@cs.man.ac.uk

University of Sheffield (USFD)
Regent Court, 211 Portobello street
S14DP Sheffield
United Kingdom
Contact person: Hamish Cunningham
E-mail address: hamish@dcs.shef.ac.uk

University of Trento (UniTn)
Via Sommarive 14
38050 Trento
Italy
Contact person: Fausto Giunchiglia
E-mail address: fausto@dit.unitn.it

Vrije Universiteit Amsterdam (VUA)
De Boelelaan 1081a
1081HV. Amsterdam
The Netherlands
Contact person: Frank van Harmelen
E-mail address: Frank.van.Harmelen@cs.vu.nl

Vrije Universiteit Brussel (VUB)
Pleinlaan 2, Building G10
1050 Brussels
Belgium
Contact person: Robert Meersman
E-mail address: robert.meersman@vub.ac.be

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to writing parts of this document:

Centre for Research and Technology Hellas /Informatics and Telematics Institute
Free University of Bozen-Bolzano
Institut National de Recherche en Informatique et en Automatique
Learning Lab Lower Saxony
University of Manchester
University of Trento
Vrije Universiteit Amsterdam

Changes

Version	Date	Author(s)	Changes
0.1	10.09.04	Jeff Z. Pan	creation
0.2	06.11.04	Birte Glimm and Jeff Z. Pan	added Chapter 3
0.3	13.11.04	Jeff Z. Pan and Wolf Siberski	added Chapter 4
0.4	16.11.04	Vassilis Tzouvaras and Giorgos Stamou	added Chapter 5
0.5	21.11.04	Holger Wache	added Chapter 7
0.6	21.11.04	Ian Horrocks and Lei Li	added Chapter 6
0.65	09.12.04	Jeff Pan and Birte Glimm	revised Chapter 4
0.8	13.12.04	Jeff Pan	added Chapter 1 and 8

Executive Summary

In the progress of realising the Semantic Web, developing and supporting Semantic Web query languages are among the most useful and important research problems. In [PFT$^+$04], we have provided a unified framework for OWL-based rule and query languages. In this report, we focus on the problems of query answering for Semantic Web query languages (such as RDF, OWL DL and OWL-E) in the OWL-QL specification.

Contents

Chapter 1

Introduction

In the progress of realising the Semantic Web, developing and supporting Semantic Web query languages are among the most useful and important research problems. In [PFT$^+$04], we have provided a unified framework for OWL-based rule and query languages. In this report, we will

- fill the gap between the theoretical foundations of the unifying framework introduced in [PFT$^+$04] and the W3C work on the RDF semantics and the SPARQL query language;
- investigate query answering within the OWL-QL formalism, in particular for queries over OWL DL and OWL-E ontologies;
- study various optimisation techniques of combining DL reasoners with database, in order to handle large data sets; and
- report our implementations of an OWL-QL server and a hybrid DL/Database system called Instance Store.

The rest of the report is organised as follows. Chapter 2 provides a connection between the theoretical foundations of the unifying framework introduced in [PFT$^+$04] and the W3C work on the RDF semantics and the SPARQL query language.

Chapters 3 to 5 investigate query answering within the OWL-QL formalism. Chapter 3 presents the OWL-QL formalism, the basic rolling-up techniques to reduce OWL-QL query answering to OWL DL knowledge satisfiability and an implementation of an OWL-QL server. Chapter 4 presents the formal semantics for datatype expression enabled queries and shows how to reduce query answering in OWL-E-QL to knowledge base satisfiability in OWL-E. Chapter 5 discusses a fuzzy extension of OWL-QL.

Chapter 6 and 7 study various optimisation techniques of combining DL reasoners with databases. Chapter 6 provides an in-depth description of the algorithms and implementation of a hybrid DL/Database system called Instance Store. Chapter 7 presents some early ideas on how to optimise instance realisation.

Chapter 8 concludes this report.

Chapter 2

Querying the Semantic Web with Ontologies

2.1 Introduction

The main aim of this chapter to recast the RDF model theory in a more classical logic framework, and to use this characterisation to shed new light on the ontology languages layering in the semantic web. In particular, we will show how the models of RDF can be related to the models of DL based ontology languages, without requiring any change on the existing syntactic or semantic definitions in the RDF and OWL-DL realms.

We first introduce the notion of minimal models for RDF graphs, and we use this notion to characterise RDF entailment. RDF minimal models can also be seen as classical first order structures, that we call DL interpretations. These structures provide the semantic bridge between RDF and description logics based languages. The intuition beyond DL interpretations is that it singles out the concepts and the individuals from an RDF minimal model – possibly in a polymorphic way when the same node is given both the meaning as a class and as an individual. The natural DL interpretation is the one in which concepts and individuals are disjoint. The class of RDF graphs which allow only for natural DL models are called DL compatible.

Once we have characterised RDF graphs in terms of their minimal models, it is possible to understand the notion of logical implication between RDF graphs and DL formulas. In particular, in this chapter we analyse the problem of querying RDF graphs with DL ontologies. We define the certain answer and the possible answer of a query to an RDF graph given an ontology; this is based on the notion of natural DL interpretation of the RDF graph. Finally, we prove an important reduction result. That is, given an RDF graph $\mathcal{S}$ and a query Q, the answer set of Q to $\mathcal{S}$ (as defined by W3C) is the same as the certain answer of Q to $\mathcal{S}$ given the empty KB. This shows a complete interoperability between RDF and DLs. For example, in absence of ontologies, it would be possible to use OWL-

QL to answer queries to RDF graphs, or to use SPARQL to answer queries to ABoxes.

2.2 RDF Model Theory revisited

We first define the notion of minimal model for an RDF graph.

Definition 1 *(Minimal Model)*
A ground instantiation *of an RDF graph $\mathcal{S}$ is obtained by replacing each bnode in $\mathcal{S}$, if any, with some URI. A* restricted ground instantiation *of an RDF graph $\mathcal{S}$ is obtained by replacing each bnode in $\mathcal{S}$, if any, with some element of the set of the URIs appearing in $\mathcal{S}$ together with a set of fresh URIs – that is, not appearing elsewhere in $\mathcal{S}$– in correspondence to each bnode symbol in $\mathcal{S}$.*

An RDF minimal model $\mathcal{I}_{RDF}$ *of an RDF graph $\mathcal{S}$ is a restricted ground instantiation of the graph.*

Note that a minimal model is always finite if the RDF graph is finite, that a ground RDF graph has a unique minimal model, and that a minimal model is a ground RDF graph.

As the following lemma shows, the minimal models of an RDF graph contain *explicitly* all the information entailed by the graph itself.

Lemma 2 *(RDF entailment and minimal models)*

1. *An RDF graph $\mathcal{S}$ entails an RDF graph $\mathcal{E}$ (as defined in [Hay04a]), written $\mathcal{S} \models \mathcal{E}$, if and only if each minimal model of $\mathcal{S}$ contains some ground instantiation of $\mathcal{E}$.*
2. *RDF entailment is NP-complete in the dimension of the RDF graphs.*
3. *RDF entailment is polynomial in the dimension of the graphs if $\mathcal{E}$ is acyclic or ground.*

The proof is based on a reduction to the problem of conjunctive query containment, and by using the interpolation lemma in [Hay04a].

A *DL interpretation* of an RDF graph shows how models of RDF can be seen as interpretations in classical logic.

Definition 3 *(DL Interpretation of an RDF minimal model)*
A DL interpretation of an RDF minimal model *is a description logics (DL) interpretation $\mathcal{I}(\mathcal{I}_{RDF}) = \langle \Delta, \Lambda, \mathbb{C}, \mathbb{R}, \mathbb{F}, \mathbb{O}, \cdot^{\mathcal{I}} \rangle$, where Δ is an abstract domain, Λ is the union of all XML schema datatype value spaces, $\mathbb{C}$ is a set of atomic concepts, $\mathbb{R}$ is a set of atomic roles, $\mathbb{F}$ is a set of datatype features, $\mathbb{O}$ is a set of individuals, and $\cdot^{\mathcal{I}}$ is an interpretation function giving the extension to concepts, roles, and features, such that:*

$$\mathbb{C} \subseteq \textit{URI-references} \setminus \textit{RDF-vocabulary}$$

$$\mathbb{R} \subseteq \textit{URI-references} \setminus \textit{RDF-vocabulary}$$

$$\mathbb{F} \subseteq \textit{URI-references} \setminus \textit{RDF-vocabulary}$$

$$\mathbb{O} \subseteq \textit{URI-references} \setminus \textit{RDF-vocabulary}$$

$$\Delta \supseteq_{\textit{non-empty}} \mathbb{O}$$

$$\cdot^{\mathcal{I}} = \begin{cases} \textit{for each } C \in \mathbb{C} C^{\mathcal{I}} = \{o \in \mathbb{O} \mid \langle o, \texttt{rdf:type}, C\rangle \in \mathcal{I}_{RDF}\} \\ \textit{for each } R \in \mathbb{R} R^{\mathcal{I}} = \{\langle o_1, o_2\rangle \in \mathbb{O} \times \mathbb{O} \mid \langle o_1, R, o_2\rangle \in \mathcal{I}_{RDF}\} \\ \textit{for each } F \in \mathbb{F} F^{\mathcal{I}} = \{\langle o, l2v(l)\rangle \in \mathbb{O} \times \Lambda \mid \langle o, F, l\rangle \in \mathcal{I}_{RDF}\} \\ \textit{for each } o \in \mathbb{O} o^{\mathcal{I}} = o \end{cases}$$

Note that there may be several DL interpretations of a single RDF minimal model, depending on which URI references are associated to concept names, to role names, to datatype features, and to individuals.

An URI reference may be associated to more than one DL syntactic type: polymorphic meanings of URIs are allowed. However note that, just like in the case of contextual predicate calculus (as defined in [CKW93]), there is no interaction between the distinct occurrences of the same URI as a concept name, or as a role name, or as a feature name, or as an individual. This absence of interaction is required for classical first order DLs such as OWL-lite or OWL-DL. For example, given the triple $\langle \texttt{ex:o}, \texttt{rdf:type}, \texttt{ex:o}\rangle$ within an RDF minimal model, it is possible to have a DL interpretation associated to it where both $\mathbb{C}$ and $\mathbb{O}$ include the URI `ex:o`, and such that the individual `ex:o` is in the extension of the concept `ex:o`.

The above definition of DL interpretation of an RDF minimal model is sloppy as far as the role of datatypes is concerned. In fact, in a DL interpretation distinct datatypes should be introduced explicitly. This can be easily induced by the structure of the lexical form of the XML-schema typed literals themselves.

A DL interpretation of an RDF minimal model is *datatype-free* if the RDF literals are also interpreted as individuals in $\mathbb{O}$, and no Λ nor datatype features are given.

Definition 4 *(DL compatible RDF graph)*
Given an RDF minimal model $\mathcal{I}_{RDF}$, the sets $\hat{\mathbb{C}}$, $\hat{\mathbb{R}}$, $\hat{\mathbb{F}}$, $\hat{\mathbb{O}}$ are defined as the minimum sets such that:

for each $\langle o, \texttt{rdf:type}, C\rangle \in \mathcal{I}_{RDF}$, then $o \in \hat{\mathbb{O}}, C \in \hat{\mathbb{C}}$;

for each $\langle o_1, R, o_2\rangle \in \mathcal{I}_{RDF}$, then $o_1, o_2 \in \hat{\mathbb{O}}, R \in \hat{\mathbb{R}}$;

for each $\langle o, F, l\rangle \in \mathcal{I}_{RDF}$ and l is a literal, then $o \in \hat{\mathbb{O}}, F \in \hat{\mathbb{F}}$.

A natural DL interpretation *of an RDF graph $\mathcal{S}$ is the DL interpretation of an RDF minimal model of $\mathcal{S}$ where $\mathbb{C} = \hat{\mathbb{C}}, \mathbb{R} = \hat{\mathbb{R}}, \mathbb{F} = \hat{\mathbb{F}}, \mathbb{O} = \hat{\mathbb{O}}$.*

An RDF graph $\mathcal{S}$ is a DL compatible RDF graph *if for some of its minimal models $\hat{\mathbb{C}}$, $\hat{\mathbb{R}}$, $\hat{\mathbb{F}}$, $\hat{\mathbb{O}}$ are mutually disjoint.*

Note that checking whether an RDF graph is DL compatible and building a natural DL interpretation takes polynomial time with respect to the dimension of the graph. Ground DL compatible RDF graphs have a unique natural DL interpretation.

2.3 Querying with Ontologies

In the previous section we have characterised RDF graphs in terms of their minimal models. It is now possible to understand the notion of logical implication between RDF graphs and DL formulas. We have thus achieved full semantic interoperability between the RDF-like languages and the DL-like languages in the semantic web. In particular, in this section we analyse the problem of querying RDF graphs with DL ontologies.

Definition 5 *(Querying RDF graphs with DL ontologies)*
Given an RDF graph $\mathcal{S}$, consider the DL knowledge bases $\Sigma_{\mathcal{S},i}$, each one with the same TBox expressing some given ontology KB and with the ABox assertions as in the natural DL interpretation associated to the ith minimal model of $\mathcal{S}$. Given a first order query Q over the alphabet of $\mathcal{S}$ without the RDF vocabulary, consider the set $\mathcal{A}_Q^{\mathcal{S}}$, which includes for each i the answer set of Q to $\Sigma_{\mathcal{S},i}$ (in agreement with the semantics as specified in the Knowledge Web deliverable D2.5.1). The certain answer *of Q to $\mathcal{S}$ given the KB is the intersection of all the answer sets in $\mathcal{A}_Q^{\mathcal{S}}$; a* possible answer *of Q to $\mathcal{S}$ given the KB is any of the answer sets in $\mathcal{A}_Q^{\mathcal{S}}$.*

A special case of the theorem above is when we restrict the query to ground DL compatible RDF graphs. This corresponds to querying the unique DL interpretation (trivially) associated to the ground DL compatible RDF graph.

Theorem 6 *(Querying RDF graphs with empty ontologies)*
Given an RDF graph $\mathcal{S}$ and a first order query Q over the alphabet of $\mathcal{S}$ without the RDF vocabulary, the answer set of Q to $\mathcal{S}$ (in agreement with the RDF entailment semantics, as in Lemma 2) is the same as the certain answer of Q to $\mathcal{S}$ given the empty KB.

The problem of query answering with the empty KB is polynomial with respect to the dimension of $\mathcal{S}$.

The proof is based on a reduction to the problem of conjunctive query containment. Note that in this case it is enough to encode as an ABox only the natural interpretation associated to the so called *canonical model*, i.e., the minimal model whose bnodes have been associated to distinct fresh URIs.

Note that a special case of first order query – without the RDF vocabulary – is the case of positive queries, which corresponds to an open formula in the form of a disjunction of conjunctions of, possibly existentially quantified, non-RDF atoms. Positive queries can be expressed in RDF as a disjunction of RDF graphs, with the proviso that the only allowed RDF property is `rdf:type`, and that bnodes do not appear as objects of `rdf:type` triples.

Chapter 3

OWL-QL

Here we address the OWL Query Language (OWL-QL), how to reduce query answering of OWL-QL into Knowledge Base Satisfiability and an implementation of an OWL-QL server.

3.1 Introduction

The OWL-QL specification, proposed by the Joint US/EU ad hoc Agent Markup Language Committee,[1] is a language and protocol for query-answering dialogues using knowledge represented in the Ontology Web Language (OWL). It is a direct successor of the DAML Query Language (DQL) [Fik03], also released by the Joint US/EU ad hoc Agent Markup Language Committee. Both language specifications go beyond the aims of other current web query languages like XML Query [Boa03], an XML [Bra04] query language, or RQL [KAC$^+$02], an RDF [Bec04] query language, in that they support the use of inference and reasoning services for query answering.

The OWL-QL specification suggests a reasoner independent and more general way for agents (clients) to query OWL knowledge bases on the Semantic Web. The specification is given on a structural level with no exact definition of the external syntax. By this it is easily adoptable for other knowledge representation formats (such as RDFS and first order logics), but on the semantic level OWL-QL is properly defined, due to the formal definition of the relationships among a query, a query answer and the knowledge base(s) provided by the specification (see [FHH03], page 10–11, Appendix Formal Relationship between a Query and a Query Answer).

[1]See `http://www.daml.org/committee/` for the members of the Joint Committee.

3.1.1 Queries

OWL-QL queries are conjunctive queries w.r.t. some knowledge bases (or simply *KBs*). A query necessarily includes a *query pattern* that is a collection of OWL statements (axioms) where some URI references [Ber98] or literals are replaced by variables. In a query, the client can specify 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. In this report, must-bind variables, may-bind variables and don't-bind variables are prefixed with "`?`", "$\sim$" and "`!`", respectively.

A client uses an answer KB pattern to specify which knowledge base(s) the server should use to answer the query. An *answer KB pattern* can be either a KB, a list of KB URI references or a variable (of the above three kinds); in the last case, the server is allowed to decide which KB(s) to use. The use of may-bind and don't-bind variables is one of the features that clearly distinguish OWL-QL from standard database query languages (such as SQL [ANS92]) and other web query languages (such as RQL [KAC$^+$02] and XML Query [Bra04]).

Here is an example of a query pattern and an answer KB pattern.

```
queryPattern:  {(hasFather Bill ?f)}
answerKBPattern:  {http://owlqlExample/fathers.owl}
```

Figure 3.1: A query example

Assume that the KB referred to in the answer KB pattern includes the following OWL statements

```
SubClassOf(Person
          restriction(hasFather someValuesFrom(Person)))
Individual(Bill type(Person)),
```

which assure that every person has a father that is also a person and that Bill is a person. It could then be inferred that Bill has a father, but we can't name him, so the OWL-QL server can't provide a binding and returns an empty answer collection. This is of course different if f is specified as a may-bind ($\sim$`f`) or don't-bind (`!f`) variable, in both cases an OWL-QL server should return one answer, but without a binding for $\sim$`f` resp. `!f`.

Assume now that the KB includes the additional statement that Mary has Joe as her father and a query with a must-bind variable for the child (`?c`). The type of the variable `f` for the father would change the answer set as follows:

```
queryPattern:  {(hasFather ?c ?f)}
```

If `f` is a must-bind variable (`?f`), a complete answer set contains only persons whose father is known, in this example (hasFather Mary Joe) where Mary is a binding for `?c` and Joe is a binding for `?f`.

`queryPattern:` $\{$`(hasFather ?c !f)`$\}$
If `f` is a don't-bind variable (`!f`), a complete answer set contains all known persons since it is specified that all persons have a father, but without a binding for `!f`. In this example (hasFather Mary !f), (hasFather Joe !f) and (hasFather Bill !f) should be in the answer set.

`queryPattern:` $\{$`(hasFather ?c ~f)`$\}$
If `f` is a may-bind variable ($\sim$`f`), the complete and non-redundant answer set contains all known persons since it is specified that all persons have a father, but a binding for $\sim$`f` is only provided in case the father is known. In this example (hasFather Mary Joe), (hasFather Joe $\sim$f) and (hasFather Bill $\sim$f) should be in the answer set.

An optional query parameter allows the definition of a pattern that the server should use to return the answers. This *answer pattern* necessarily includes the format of all variables used in the query pattern. If no answer pattern is specified, a two item list whose first item is the querys must-bind variables list and whose second item is the querys may-bind variables list is used as the answer pattern. This is different to the DQL specification, where, for the case that no answer pattern was specified, the query pattern is used as the answer pattern.

Another option for a query is to include a *query premise* (a set of assumptions) to facilitate “if-then” queries, which can't be expressed otherwise since OWL does not support an “implies” logical connective. E.g., to ask a question like “If Bill is a person, then does Bill have a father?” the query premise part includes an OWL KB or a KB reference stating that Joe is a person and the query part is the same as in Figure 3.1. The server will treat OWL statements in the query premise as a regular part of the answer KB and all answers must be entailed by this KB.

3.1.2 Query-Answering Dialogues

To initiate a query-answering dialogue the clients sends a query to an OWL-QL server. The server then returns an *answer bundle*, which includes a (possibly empty) answer set together with either a *termination token* to end the dialogue or a *process handle* to allow the continuation of the query-answering dialogue. A termination token is either *end* to indicate that the server can't for any reasons provide more answers 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. The specification also allows the definition of further termination token, e.g., to provide information about the rejection reasons.

Since an answer bundle can be 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 the number of answers in an answer bundle. If the client specified an answer bundle size bound in the query, the server does not send more answers then allowed by the answer bundle size bound.

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 must 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 resources. Figure 3.2 illustrates the query-answering-dialogue.

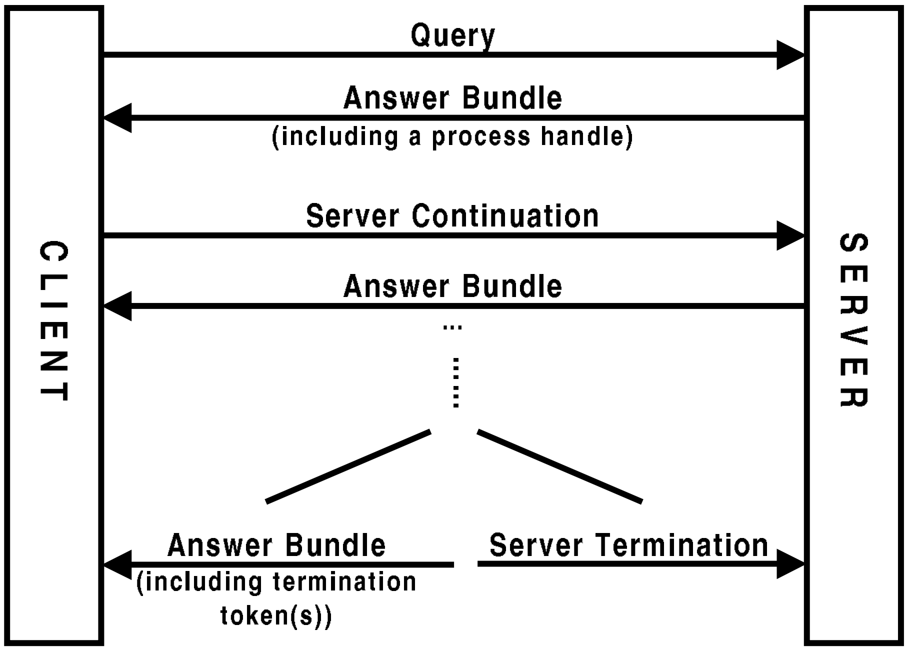


Figure 3.2: The query-answering dialogue

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*, so no 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. An answer is more general (subsumes another) if it only provides fewer bindings for may-bind variables or has less specific bindings for variables that occur only as values of minCardinality or maxCardinality restrictions, e.g., if the KB is true for a binding of 4 for a maxCardinality variable, then it will also be true for a binding of 5, 6, ... 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 that previously delivered answers. Finally servers that guarantee to terminate with termination token *none* are called *complete*.

3.2 Reducing Query Answering to ABox Reasoning

In this section, we show that query answering of *acyclic conjunctive queries* (a formal definition of which is presented in Section 3.2.2) can be reduced to ABox reasoning. Before presenting the details of the reduction, we would like to mention two points here.

- Since there exist no efficient decision procedure for the $\mathcal{SHOIQ}(\mathbf{D}^+)$ DL, the underpinning of OWL DL, we consider the $\mathcal{SHIQ}$ DL in this section.
- Please note that may-bind variables are a combination of distinguished (must-bind) and non-distinguished (don't-bind) variables and are therefore not treated in further detail here. Therefore, in the following reduction we will not consider may-bind variables.

3.2.1 Conjunctive Queries

A conjunctive query q is of the form $q\langle\vec{x}\rangle \leftarrow conj(\vec{x};\vec{y};\vec{z})$. The vector $\vec{x}$ consists of so called distinguished or must-bind variables that will be bound to individual names of the knowledge base used to answer the query. The vector $\vec{y}$ consists of non-distinguished or don't-bind variables, which are existentially quantified variables. The vector $\vec{z}$ consists of individual names, and $conj(\vec{x};\vec{y};\vec{z})$ is a conjunction of atoms. An atom is of the form $v_1 : \mathtt{C}$ or $\langle v_2, v_3\rangle : \mathtt{r}$ where $\mathtt{C}$ is a concept name, $\mathtt{r}$ is a role name and v_1, v_2, v_3 are individual names from $\vec{z}$ or variables from $\vec{x}$ or $\vec{y}$. The left hand side of the query, i.e., $q\langle\vec{x}\rangle \leftarrow$, might be omitted, since it is clear from the prefixes which variables are distinguished ones. Recall that must-bind variable names in a query are prefixed with ?, don't-bind variables are prefixed with !, individual names are not prefixed. Concept names are written in upper case letters, while role and individual names are written in lower case.

3.2.2 Query Graphs

A conjunctive query q can be represented as a directed labelled graph $G(q) := \langle V, E\rangle$, where V is a set of vertices, and E is a set of edges. The set V consists of the union of the elements in $\vec{x}$, $\vec{y}$, and $\vec{z}$. The set E consists of all pairs $\langle v_1, v_2\rangle$, such that $v_1, v_2 \in V$ and $\langle v_1, v_2\rangle : \mathtt{r}$ is an atom in q. A node $v \in V$ is labelled with a concept $C_1 \sqcap \ldots \sqcap C_n$ such that, for each C_i, $v : C_i$ is an atom in q. Each edge $e \in E$ is labelled with a set of role names $\{\mathtt{r} — \langle v_1, v_2\rangle : \mathtt{r}$ is an atom in $q\}$.

The function $\mathcal{L}(v), v \in V$ returns the label for v. If $\mathcal{L}(v)$ is empty, the top concept ($\top$) is returned. The function $\mathcal{L}(e), e \in E$ returns a set of edge labels for e. The function $\mathcal{L}^{-}(e), e \in E$ returns a set of inverted edge labels, such that $\mathcal{L}^{-}(e) = \{r | r^{-} \in \mathcal{L}(e)\}$. The function $flip(G, \langle v_1, v_2 \rangle)$, $\langle v_1, v_2 \rangle \in E$ creates a new graph $G' := \langle V', E' \rangle$, with $V' := V$, $E' := (E \setminus \{\langle v_1, v_2 \rangle\}) \cup \{\langle v_2, v_1 \rangle\}$, and $\mathcal{L}(\langle v_2, v_1 \rangle) = \mathcal{L}^{-}(\langle v_1, v_2 \rangle)$. The function $pred(v_1), v_1 \in V$ returns a set of vertices $\{v_1 | v_1, v_2 \in V \wedge \langle v_2, v_1 \rangle \in E\}$.

Two vertices $v_1, v_2 \in V$ are adjacent, if $\mathcal{L}(\langle v_1, v_2 \rangle) \neq \emptyset$ or $\mathcal{L}(\langle v_2, v_1 \rangle) \neq \emptyset$. The vertex $v_1 \in V$ is reachable from $v_2 \in V$, if v_1 is adjacent to v_2 or if there is a another vertex $v_3 \in V$ such that v_3 is adjacent to v_1, and v_2 is reachable from v_3. The graph $G(q)$ is cyclic, if there is a $v \in V$, such that $\mathcal{L}(\langle v, v \rangle) \neq \emptyset$ or if there is a $v' \in V$, such that v is adjacent to v' and if one element is removed from $\mathcal{L}(\langle v, v' \rangle)$, v' is still reachable from v. q is an acyclic conjunctive query if $G(q)$ is not cyclic.

3.2.3 The Rolling-up Technique

If a query contains only distinguished variables, one could replace all variables with individual names from the knowledge base and use a sequence of instantiation queries to determine if the statement is true in the knowledge base. To compute a complete query answer set with this approach, it is necessary to test all possible combinations of individual names. This is very costly, and furthermore, this approach is not applicable to queries with non-distinguished variables.

In 2001 Tessaris [Tes01] proposed a rolling-up technique that can be used to eliminate non-distinguished variables from a query. The technique is applicable to acyclic conjunctive queries and the OWL-QL server implemented in Manchester uses this technique to compute the query answers.

The basic idea behind the rolling-up technique is to convert *individual-valued* property atoms into concept atoms. The rationale behind this rolling up can easily be understood by the use of nominals. The *individual-valued* property atom $\langle \mathtt{a}, \mathtt{b} \rangle : r$ can be transformed into the equivalent concept atom $\mathtt{a} : \exists r.\{\mathtt{b}\}$. If we replace $\mathtt{b}$ with a non-distinguished variable $!y$, the corresponding role atom $\langle \mathtt{a}, !y \rangle : r$ can be transformed into the equivalent concept atom $\mathtt{a} : \exists r.\top$ because $!y$ does not have to be bound to a named individual. Furthermore, other concept atoms about the individual $\mathtt{b}$ (being rolled up) can be adsorbed into the rolled up concept atom. For instance, the conjunction

$$\langle \mathtt{a}, \mathtt{b} \rangle : r \ \wedge \ \mathtt{b} : D$$

can be transformed into $\mathtt{a} : \exists r.(\{\mathtt{b}\} \sqcap D)$. Similarly, the conjunction

$$\langle \mathtt{a}, !y \rangle : r \wedge !y : D$$

can be transformed into $\mathtt{a} : \exists r.D$ because D is equivalent to $\top \sqcap D$.

Queries with One Distinguished Variable

Using the rolling-up technique introduced above, we can reduce query answering of queries with only one distinguished variables to retrieval (the problem of determining the set of individuals that instantiate a given concept). The process is best illustrated using the query graph $G(q)$ of a query q (Figure 3.3). For the readers convenience the distinguished variables are represented by a filled node (•), whereas non-distinguished variables and individuals are represented by an unfilled node (∘).

$\langle$?w$\rangle \leftarrow$?w:PERSON $\wedge$ $\langle$?w, !x$\rangle$:owns $\wedge$ $\langle$?w, !y$\rangle$:loves $\wedge$ $\langle$!z, !y$\rangle$:haschild

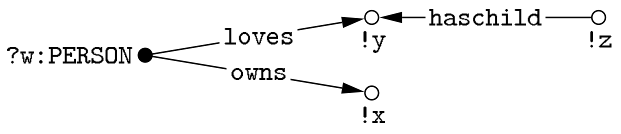


Figure 3.3: A query and its query graph.

First of all, the query graph is transformed into a tree with the distinguished variable as root. The function $flip(G, e), e \in E$ is applied to change edge directions if necessary to transform the graph into a proper tree. The left hand part of Figure 3.4 shows the resulting tree. Then the rolling-up starts from the leaves of the tree. A leaf, e.g., !z, is selected and the vertex and its incoming edge are replaced by conjoining the concept $\exists\,\mathcal{L}(pred(!\mathrm{z}), !\mathrm{z}).\mathcal{L}(!\mathrm{z})$ to the label of $pred(!\mathrm{z})$. The right hand part of Figure 3.4 shows the result of the first rolling-up step. The $\top$ conjunct could be omitted without changing the semantics. This step is applied to each leaf until only the distinguished variable at the root is remaining. The label of the root node can now be used to retrieve the individual names that are valid bindings for the distinguished variable. For this example these are instances of the concept PERSON $\sqcap$ $\exists$owns.$\top$ $\sqcap$ $\exists$loves.($\top \sqcap \exists$haschild^{-}.$\top$).

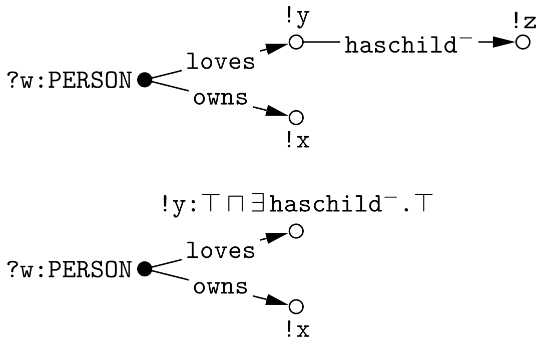


Figure 3.4: Two states of a query graph in the rolling-up process.

Queries with Individual Names

In a DL that supports the oneOf constructor, which allows the definition of a concept by enumerating its instances, the rolling-up can use the individual name directly in the concept expression. Nodes for an individual name can then be treated like a non-distinguished variable with the concept {individual name} as label. For example, the query $\langle ?\texttt{x}\rangle \leftarrow \langle ?\texttt{x, mary}\rangle\texttt{:loves}$ is rolled-up into a retrieval query for instances of the concept $\exists\,\texttt{loves.}\{\texttt{mary}\}$. Unfortunately most reasoners do not support the oneOf constructor, but it is still possible to deal with such queries using a so called representative concept [Tes01]. The representative concept is a so far unused concept name, which is used instead of the individual name, the ABox being extended with an assertion stating that the individual is an instance of its representative concept. E.g., the query could be answered by retrieving the concept instances of $\exists\,\texttt{loves.P}_{mary}$, after the assertion $\texttt{mary:P}_{mary}$ is added to the KB.

Boolean Queries

If the vector $\vec{x}$ is empty, i.e., the query contains no distinguished variables, the query answer is true, iff the knowledge base entails the query with the non-distinguished variables treated as existentially quantified. The boolean query $q \leftarrow \langle\texttt{acar, }!\texttt{x}\rangle\texttt{:ownedby} \wedge\ !\texttt{x:PERSON}$ against the knowledge base in Example 1 should be answered with true, since the existence of such a person is entailed by the KB.

Example 1
$KB = \{\mathcal{T}, \mathcal{A}\}$
$\mathcal{T} = \{CAR \sqsubseteq \exists\, ownedby.PERSON\}$
$\mathcal{A} = \{acar{:}CAR\}$

We can arbitrarily select a non-distinguished variable and treat it as if it were a distinguished one and apply the rolling up techniques presented in previous sections. For instance, the above query can be rolled up to $!\texttt{x:PERSON} \sqcap \exists\,\texttt{ownedby}^{-}\texttt{.}\{\texttt{acar}\}$. If !x would have been a distinguished variable, the query could have been answered with a retrieval query, but here only the assertion must hold that such a thing exists. It must not necessarily be named in the knowledge base.

To answer the query with true, we must prove that the negated rolled-up concept causes an inconsistency in the knowledge base. This is equal to adding a statement that the rolled-up concept implies bottom. In this example the knowledge base becomes indeed inconsistent if we add a statement that there is no instance of the concept PERSON that owns acar, i.e., adding the axiom $(\texttt{PERSON} \sqcap \exists\,\texttt{ownedby}^{-}\texttt{.}\{\texttt{acar}\}) \sqsubseteq \bot$. Therefore the query answer is true, otherwise the query answer would have been false.

Queries with Multiple Distinguished Variables

If a query contains multiple distinguished variables, the query can not be rolled-up into a single DL retrieval query. To avoid a test of all possible combinations of individual names, as necessary for the simple approach described in Section 3.2.3, the rolling-up technique is nevertheless helpful. To start the query answering process, one of the distinguished variables is selected as the root node, and all other variables are treated as non-distinguished. The query graph is transformed into a tree and the rolling-up process is applied as described above for the case with only one distinguished variable. The retrieved individual names are candidates for the binding of the variable. This step is repeated for all distinguished variables.

Not every combination of the retrieved candidates is possible, and to determine the valid combinations further boolean tests are necessary. To avoid as many boolean tests as possible further optimisations can be used at this point.

3.2.4 Optimisation Techniques

One promising approach is to use an iterative process that eliminates unsuitable combinations as soon as possible. Consider, e.g., the query and its query graph in Figure 3.5, where ?x has four candidates (i.e., $x_1 \ldots x_4$), ?y has two candidates (y_1, y_2), and ?z has ten candidates ($z_1, \ldots, z_{10}$) after the rolling-up.

$$\langle \texttt{?x}, \texttt{?y}, \texttt{?z} \rangle \leftarrow \langle \texttt{?x}, \texttt{?y} \rangle : \texttt{r} \wedge \langle \texttt{?y}, \texttt{?z} \rangle : \texttt{s}$$

$$\begin{tikzcd} \bullet \arrow[r, "\texttt{r}"] & \bullet \arrow[r, "\texttt{s}"] & \bullet \\ \texttt{?x}\text{: } (x_1, x_2, x_3, x_4) & \texttt{?y}\text{: } (y_1, y_2) & \texttt{?z}\text{: } (z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9, z_{10}) \end{tikzcd}$$

Figure 3.5: An example query with its query graph and candidates.

If we had not used the rolling-up to retrieve the candidates, the number of necessary boolean tests would have been factorial in the number of named individuals in the KB. With the rolling-up and boolean tests for all possible candidate combinations, the number of tests is still the product of the number of candidates, i.e., 80 tests in this example. An optimised strategy could start at the variable with the most candidates (i.e., ?z) and retrieve the concept instances of $\exists\, s^{-}.P_{y_1}$, where P_{y_1} is the representative concept for y_1. In this way, one can determine which of the candidates for ?z are related to y_1. This is repeated for y_2. By testing for valid pairs first, one can skip many unnecessary test, e.g., if y_1 and z_1 are not related, no tests for candidates of ?x are necessary. The process is repeated for the variable with the next highest number of candidates (i.e., ?x). Compared to the 80 boolean tests necessary before, this approach needs four retrieval queries to determine the valid candidate combinations. However, how many retrieval queries are necessary, depends on the number of candidates for the distinguished variables, but it is clearly cheaper than a test of all candidates and much cheaper than a test of all individual names in the KB.

Another optimisation could use structural knowledge about the roles in the KB to exclude impossible candidate combinations even before the above tests are used. The system developed in Manchester does not yet fully implement these optimisations.

3.3 An Implementation of an OWL-QL Server

3.3.1 Used Tools, Products and Languages

The implementation was realised 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.

The Description Logic reasoner Racer[9] is used in this implementation.

[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

3.3.2 System Architecture

OWL-QL 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) [CGM+04] 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 3.6 shows the architecture of the implemented OWL-QL server, together with the implemented client application. The OWL-QL server part is the main component of this work and is responsible for the rolling-up process as explained in Section 3.2.3. 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 OWL-QL server and forwards the received parameters to the relevant methods of the OWL-QL server component.

The reasoner could be any reasoner that supports the DIG [Bec03a] interface. This implementation has been tested with Racer,[10] 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.

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

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

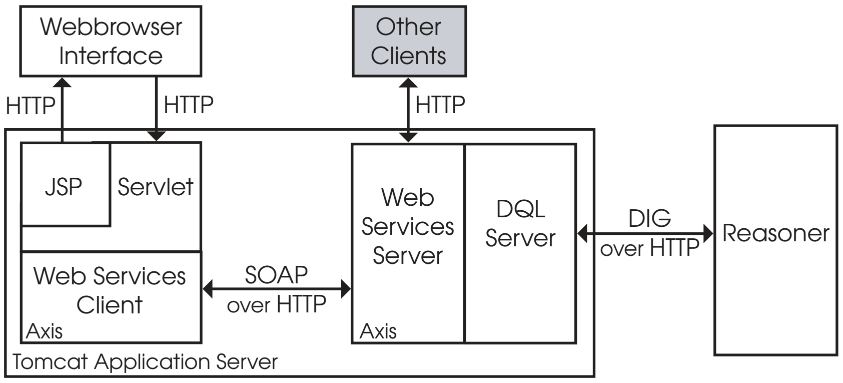


Figure 3.6: The chosen software architecture.

not necessary and can be changed easily.

3.3.3 Components

This section provides details of various component of the architecture.

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 3.7 shows a UML class diagram of the interface class that was used to create the web service and Figure 3.8 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.

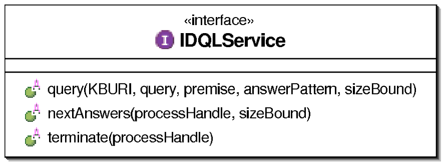

«interface»
IDQLService
query(KBURI, query, premise, answerPattern, sizeBound)
nextAnswers(processHandle, sizeBound)
terminate(processHandle)

Figure 3.7: The web service interface.

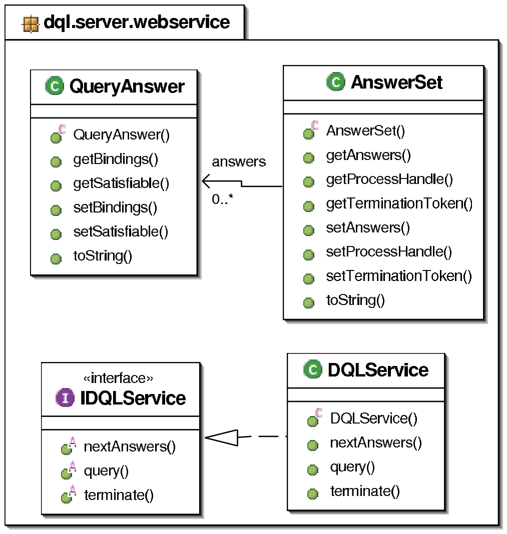


Figure 3.8: The web service package.

The OWL-QL 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 3.9 illustrates the components that are incorporated in the realisation of the service. The components labelled with library are not developed as part of this project.

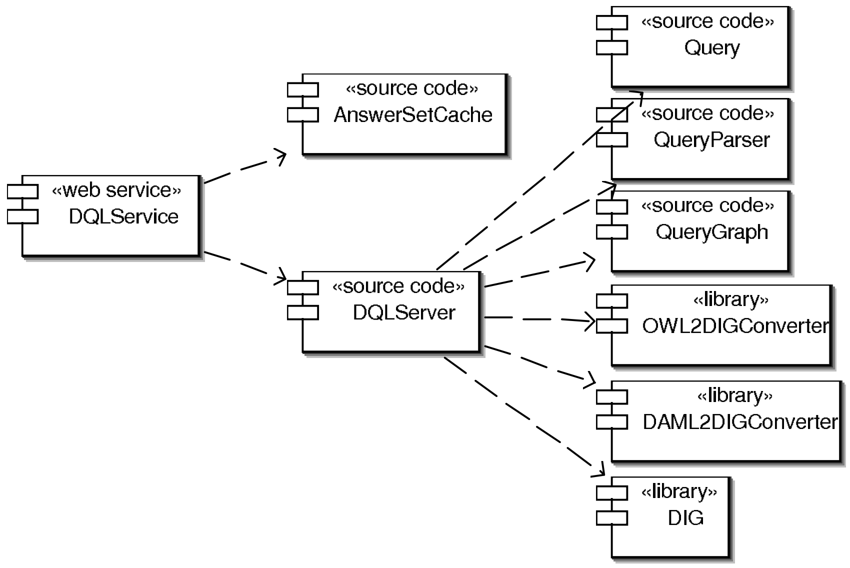


Figure 3.9: An UML deployment diagram of the OWL-QL service.

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 allow a DAML+OIL or OWL query syntax and use a parser such as the one provided with the Jena framework[11] to read the queries.

The different types of variables are indicated by a prefix, as introduced in in the OWL-QL specification: ! 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 3.1.1. To parse the query a small parser was implemented with JavaCC. JavaCC needs a .jj file as input containing an EBNF grammar [Wir77, fSI96] together with actions and token definitions as regular expressions. Table 3.1 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 3.2.2 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 3.1 shows the EBNF grammar used to parse the queries.

[11]`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 3.1: The EBNF grammar for the query parser.

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 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[12] 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 OWL-QL server.

Interaction with the Reasoner

The connection to a reasoner is established over the DIG Interface [Bec03a], which is a standardised XML interface for Description Logics systems developed by the DL Implementation Group (DIG).[13]

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.[14]

The OWL-QL 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`.

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

If this property file is not accessible the OWL-QL 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 was successful one has to call only one method with the reasoner response as a parameter.

Currently all interactions with the reasoner are performed in a kind of batch mode, i.e., 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 OWL-QL 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.

Recently the Jena framework has been extended to support the connection of OWL or DAML+OIL 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[15] is available along with a technical report about the experiences with the DIG standard during the extension of Jena [Dic04], so this is only an alternative for a future version of the OWL-QL 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.

The Query Graph Component

All classes that belong to the graph representation of a query are bundled in the package `dql.server.querygraph`. Figure 3.10 shows an UML class diagram of these classes.

The class `Graph` implements the rolling-up technique as described in Section 3.2.3. 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

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

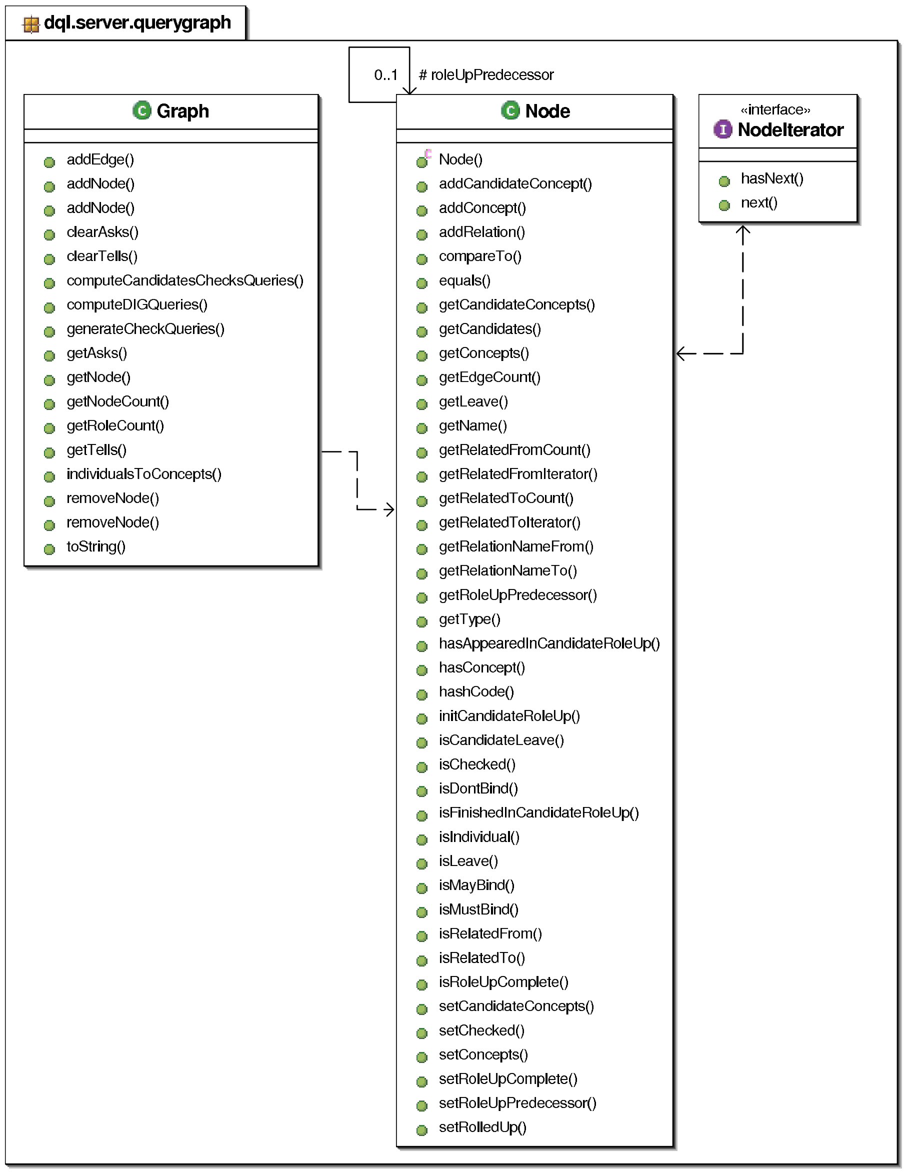


Figure 3.10: The UML class diagram of the query graph classes.

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 3.2.3 for an explanation), then 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 3.2.3.

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.

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,[16] 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 3.11 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 3.12.

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`

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

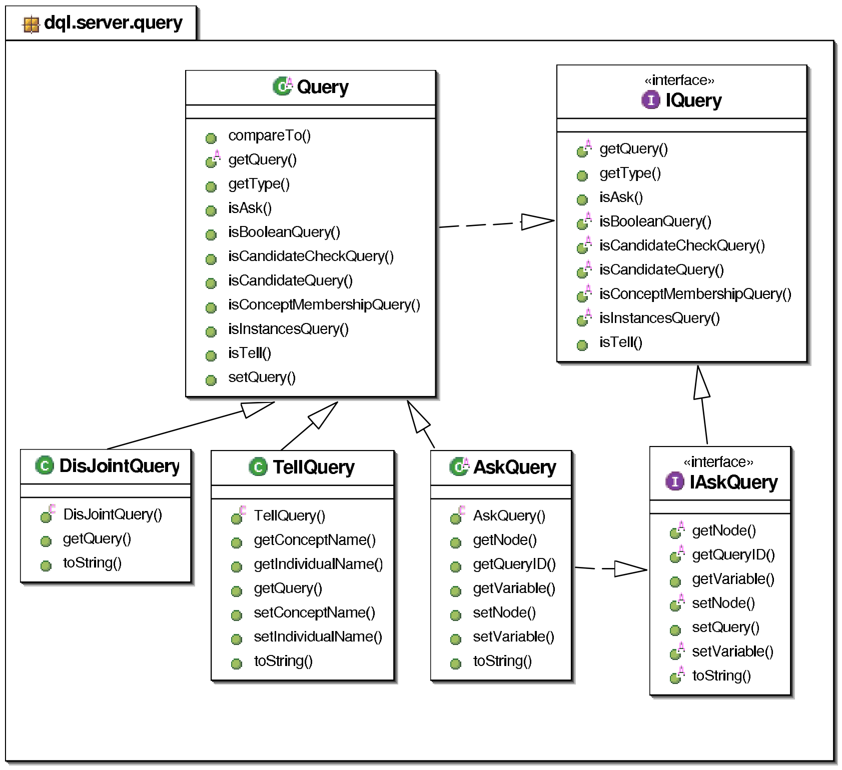


Figure 3.11: The UML class diagram of the query classes.

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 deserialization. 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

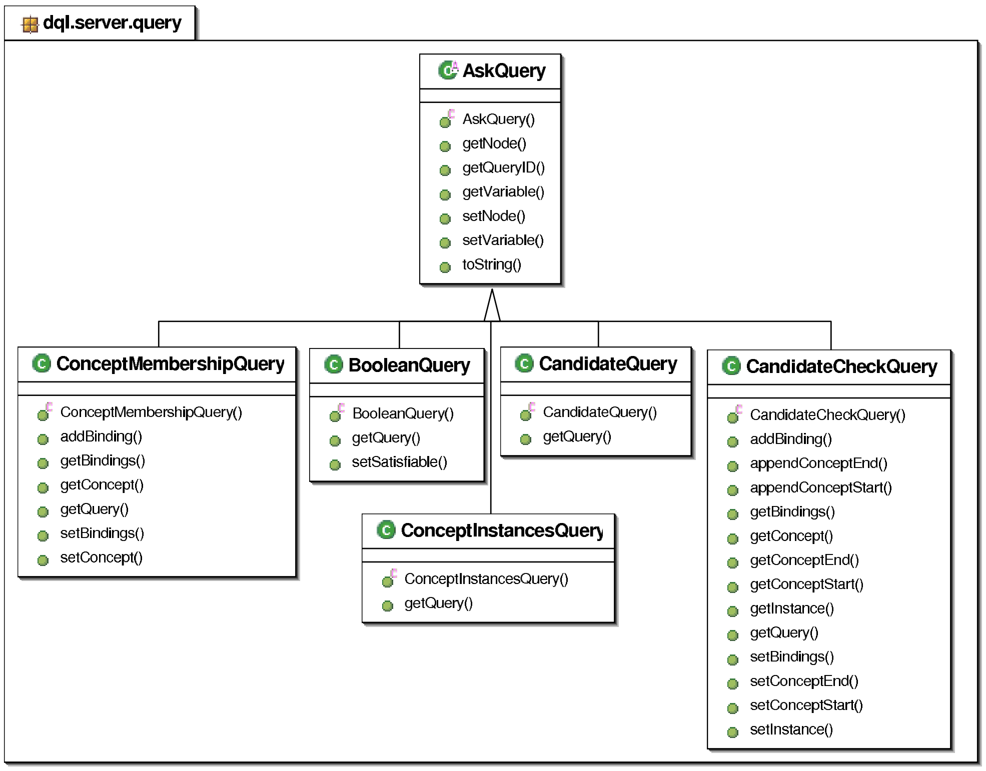


Figure 3.12: The UML class diagram of the AskQuery subclasses.

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 classes `ServerAnswerSet` and `AnswerSetCache` both reside in the package `dql.server` (see Figure 3.13), 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 3.3.3 on page 20.

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

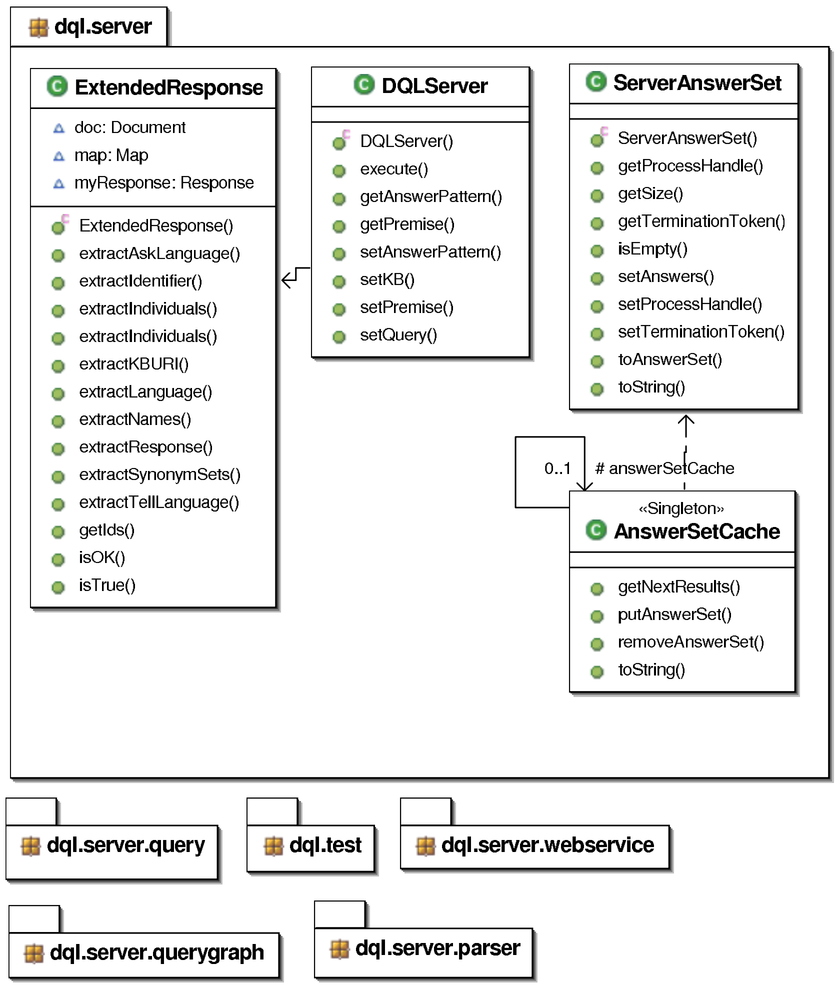


Figure 3.13: The UML class diagram of the package `dql.server`.

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 OWL-QL 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.

A Query Processing Sequence

Figure 3.14 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 server itself is a component with a clear boundary to the offered web service, i.e., the web service can be seen as a client of the component.

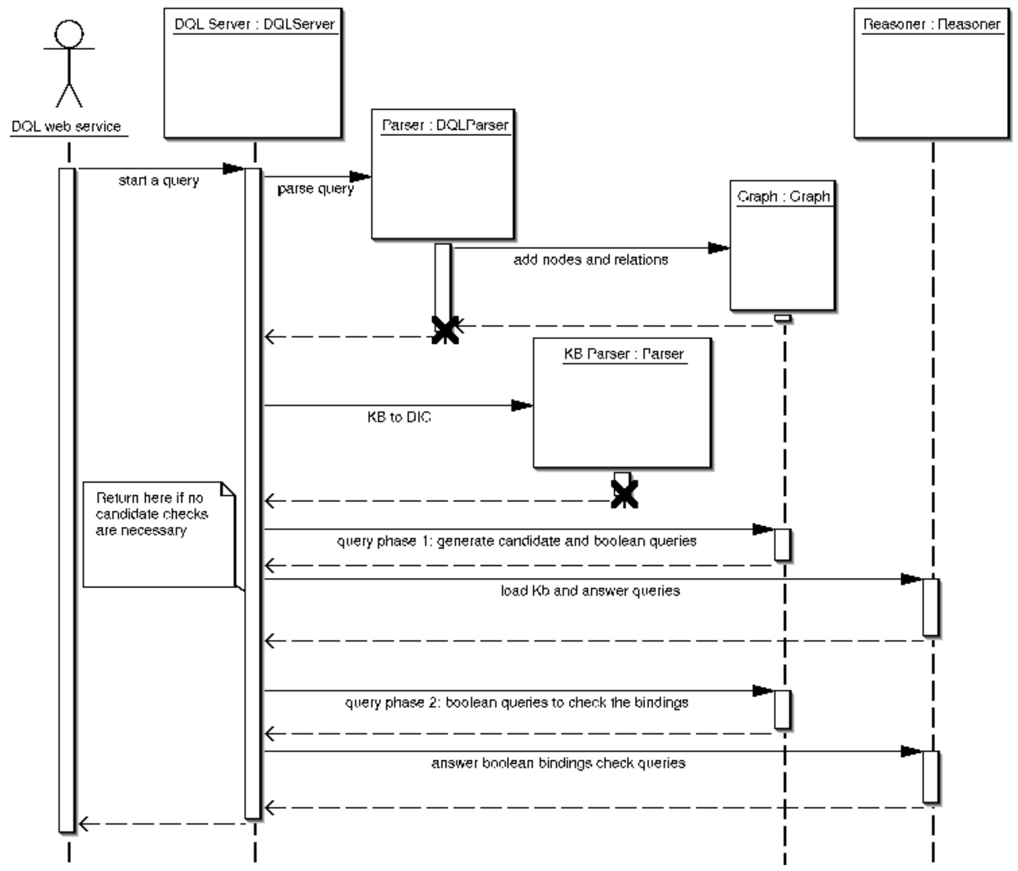

DQL web service
DQL Server : DQLServer
Parser : DQLParser
Graph : Graph
Reasoner : Reasoner
KB Parser : Parser
start a query
parse query
add nodes and relations
KB to DIG
Return here if no candidate checks are necessary
query phase 1: generate candidate and boolean queries
load Kb and answer queries
query phase 2: boolean queries to check the bindings
answer boolean bindings check queries

Figure 3.14: The UML sequence diagram for query answering.

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 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.

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.

Testing

JUnit[17] 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 [Bec02], 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[18] 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 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

[17]`http://www.junit.org`
[18]`http://www.eclipse.org`

also part of the defined Ant deployment process for the OWL-QL server components. The tests help to assure that specified requirements for the software, e.g., defined by the OWL-QL 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.

The 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 19 shows the general layout of the client. It is mainly composed of one servlet[19] 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 3.3.1. After the servlet has received a result from the DQL web service the request is forwarded to a JavaServer Pages (JSP)[20] 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.

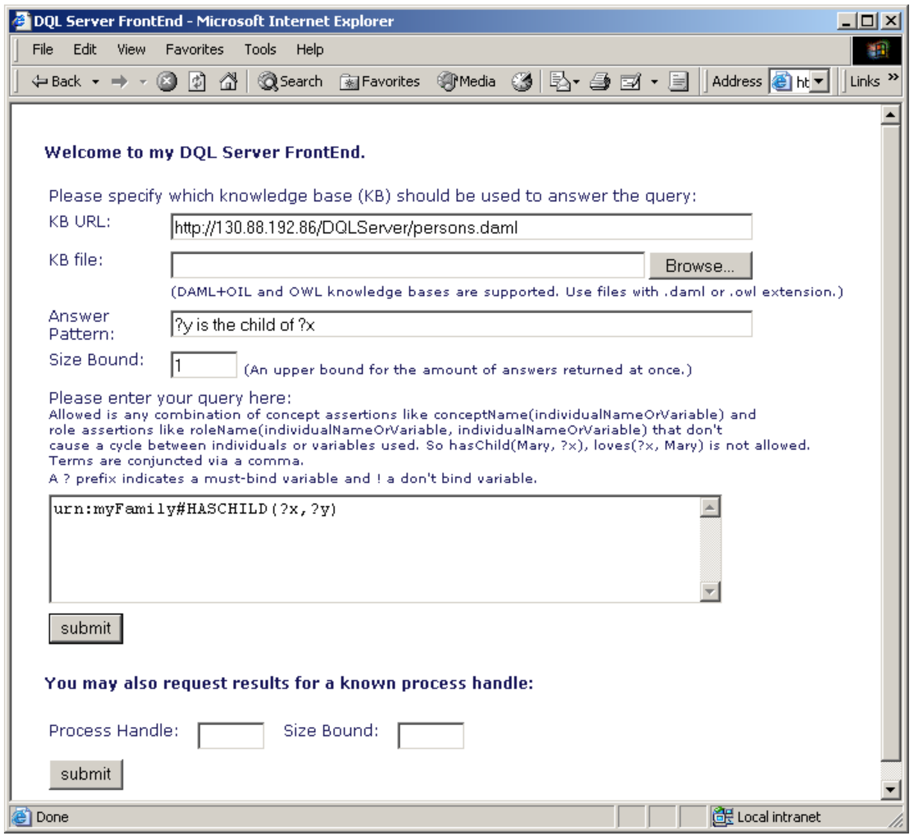
The figures on the following pages illustrate the client interface. Figure 3.15 shows the front-end for the user. It allows to specify a local knowledge base file or the URL of a knowledge base, the answer bundle size bound, the query and an answer pattern. It is necessary to use the fully qualified 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 3.16 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 3.17.

3.4 Related Work

This section introduces other available systems to query OWL knowledge bases and highlights the differences to the system realised in Manchester.

[19] `http://java.sun.com/products/servlet/whitepaper.html`
[20] `http://java.sun.com/products/jsp/whitepaper.html`


DQL Server FrontEnd - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Address Links
Done Local intranet

Welcome to my DQL Server FrontEnd.

Please specify which knowledge base (KB) should be used to answer the query:

KB URL: http://130.88.192.86/DQLServer/persons.daml

KB file: Browse...

(DAML+OIL and OWL knowledge bases are supported. Use files with .daml or .owl extension.)

Answer Pattern: ?y is the child of ?x

Size Bound: 1 (An upper bound for the amount of answers returned at once.)

Please enter your query here:
Allowed is any combination of concept assertions like conceptName(individualNameOrVariable) and role assertions like roleName(individualNameOrVariable, individualNameOrVariable) that don't cause a cycle between individuals or variables used. So hasChild(Mary, ?x), loves(?x, Mary) is not allowed.
Terms are conjuncted via a comma.
A ? prefix indicates a must-bind variable and ! a don't bind variable.

```
urn:myFamily#HASCHILD(?x,?y)
```

submit

You may also request results for a known process handle:

Process Handle: Size Bound:

submit

Figure 3.15: The DQL client start page.

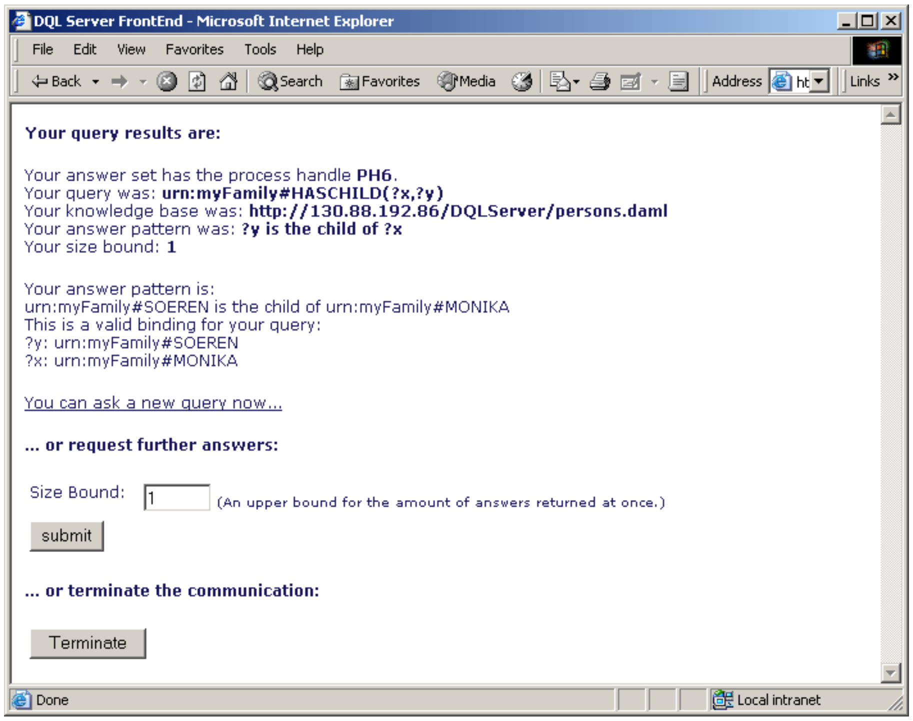
3.4.1 The Stanford OWL-QL Server

The Knowledge Systems Laboratory (KSL) of the Stanford University provides an OWL-QL implementation that supports DAML+OIL and OWL knowledge bases. The system uses the first order logic theorem prover JTP[21] [FJF03] to answer the queries. The OWL-QL server is implemented as a wrapper around the theorem prover. A query consists of DAML+OIL or OWL statements (in RDF triple notation) with URI references replaced by variables. Compared to acyclic conjunctive queries, the supported query language is therefore richer. Unfortunately the system does not answer all allowed queries. For some queries the server simply terminates the communication with a client.

As an example, consider again the KB specified in Example 2. The query $\langle ?\mathrm{x}\rangle \leftarrow ?\mathrm{x}\texttt{:CAR} \wedge \langle ?\mathrm{x}, !\mathrm{y}\rangle\texttt{:ownedby} \wedge !\mathrm{y}\texttt{:PERSON}$ is correctly answered with the binding `acar` for $?\mathrm{x}$. However, the slightly modified query $\langle ?\mathrm{x}\rangle \leftarrow ?\mathrm{x}\texttt{:CAR} \wedge \langle ?\mathrm{x}, !\mathrm{y}\rangle\texttt{:ownedby} \wedge !\mathrm{y}\texttt{:CAR}$, asking for a car that is owned by a car, is also answered with the binding `acar` for $?\mathrm{x}$.

Example 2

[21]`http://www.ksl.stanford.edu/software/JTP`


DQL Server FrontEnd - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Address Links
Done
Local intranet

Your query results are:

Your answer set has the process handle **PH6**.
Your query was: **urn:myFamily#HASCHILD(?x,?y)**
Your knowledge base was: **http://130.88.192.86/DQLServer/persons.daml**
Your answer pattern was: **?y is the child of ?x**
Your size bound: **1**

Your answer pattern is:
urn:myFamily#SOEREN is the child of urn:myFamily#MONIKA
This is a valid binding for your query:
?y: urn:myFamily#SOEREN
?x: urn:myFamily#MONIKA

You can ask a new query now...

... or request further answers:

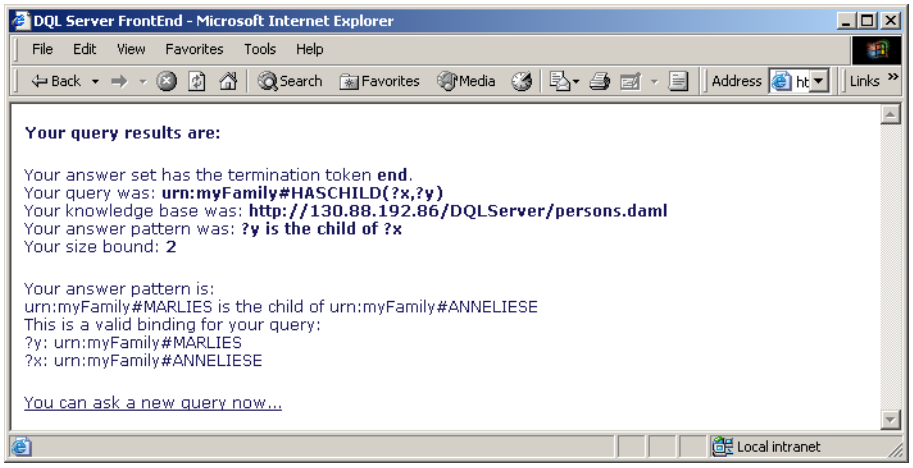
Size Bound: 1 (An upper bound for the amount of answers returned at once.)

submit

... or terminate the communication:

Terminate

Figure 3.16: A DQL client answer page with further answers available.


DQL Server FrontEnd - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Address Links
Local intranet

Your query results are:

Your answer set has the termination token **end**.
Your query was: **urn:myFamily#HASCHILD(?x,?y)**
Your knowledge base was: **http://130.88.192.86/DQLServer/persons.daml**
Your answer pattern was: **?y is the child of ?x**
Your size bound: **2**

Your answer pattern is:
urn:myFamily#MARLIES is the child of urn:myFamily#ANNELIESE
This is a valid binding for your query:
?y: urn:myFamily#MARLIES
?x: urn:myFamily#ANNELIESE

You can ask a new query now...

Figure 3.17: A DQL client answer page with termination token.

$$
\begin{aligned}
KB &= \{\mathcal{T}, \mathcal{A}\}\\
\mathcal{T} &= \{\mathtt{CAR} \sqsubseteq \exists \mathtt{ownedby.PERSON}\}\\
\mathcal{A} &= \{\mathtt{acar{:}CAR}\}
\end{aligned}
$$

The implementation was also tested with a second, more complicated query, see Example 3, against the KB in Figure 3.18. The query asks for individuals that have an `r` successor that is a `C` and has itself an r successor. The difficulty is that in this case there is no nameable instance of the concept `C`, but it can be inferred that either `c1` or `c2` is a `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.

For the slightly modified query in Example 4, in which the individual name b1 is used instead of the must-bind variable ?z, the KSL implementation provides `a1`, `c1`, and `c2` as a binding for ?x. The last two answers are, however, incorrect.

Example 3
$\langle ?x, ?z\rangle \leftarrow \langle ?x,\ !y\rangle : r \wedge \langle !y, ?z\rangle : r \wedge !y : C$

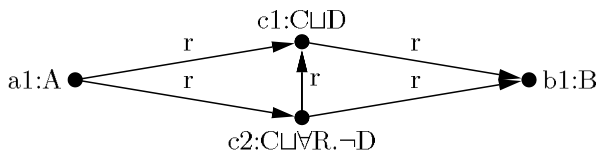


Figure 3.18: The knowledge base used for the queries in Example 3 and 4.

Example 4
$\langle ?x\rangle \leftarrow \langle ?x,\ !y\rangle : r \wedge \langle !y, b1\rangle : r \wedge !y : C$

It seems that the system has difficulties with non-distinguished variables, and queries often cause unexpected results. The reasons for this behaviour could be due to the communication with the used theorem prover or in the theorem prover itself. If the implementation is improved in this respect, however, it would provide a powerful and complete implementation of the OWL-QL specification. For practical use, the system would benefit from better error handling and error explanation and a detailed documentation would be desirable.

3.4.2 The new Racer Query Language

The recently introduced new Racer Query Language (nRQL) [HMW04] is not geared to the DQL specification, therefore it misses all the protocol specific elements, such as termination tokens or the delivery of answers in a bundle with a specifiable size bound. In addition nRQL does not support non-distinguished variables. Although nRQL is far away from the OWL-QL specification, it is nevertheless a step towards better query support, and it is therefore introduced here very briefly. The query language itself is very rich, as it supports the retrieval of variable bindings in arbitrary concept and role expressions. In contrast to the other systems introduced here, all variables are distinguished, even if they are not included in the answer. For an example, the reader may again consider the KB in Example 2 (page 33). The nRLQ query `(retrieve (?x) (and (?x CAR)`

`(?y PERSON) (?x ?y ownedby)))` returns all cars that are owned by a person. Although only cars are in the answer, a named individual must exist in the KB that is specified as owner of the car. As a result the query answer for this example is empty.

Another feature, which was added to nRQL, is negated query atoms, implemented using a negation as failure semantics. This is contrary to the Open World semantics normally used in DL systems (and also by RACER). nRQL uses the same operator (`not`) for negated query atoms and for concept negation, which could probably lead to confusion and the users have to be careful with the formulation of such a query. The nRQL query `(retrieve (?x) (not (?x PERSON)))`, using the negation as failure semantics, therefore returns `acar`. Due to the Open World semantics for concept negation, the modified query `(retrieve (?x) (?x (not PERSON)))` returns an empty answer set, since RACER cannot prove that `acar` is not an instance of the concept `person`.

nRQL offers more features than the ones described here and for details the reader is referred to the RACER documentation.[22]

3.5 Discussion

3.5.1 The OWL-QL Specification

In general, OWL-QL provides a flexible framework in conducting a query-answering dialogue using knowledge represented in OWL. It allows the definition of additional parameters, delegation of queries to another server or the continuation of a query dialogue by other clients that know a valid process handle. If the client specifies an answer bundle size bound, the specification allows an OWL-QL server to compute all answers at once or to compute the answers incrementally, as long as the answer set returned to the client contains not more answers than specified by the answer bundle size bound. The specification also allows the definition of further termination token, e.g. to provide information about the rejection reasons.

The current version of OWL-QL, however, has the following limitations.

External Syntax The specification does not provide any exact syntax definition or a specification of how to communicate the supported conformance level to a client and also other mechanism like time-outs for a query are not specified. This is due to the focus on providing an abstract specification on a structural level and to allow the various syntactical preferences of the different web communities to fit the standard to their needs. An OWL-QL server therefore has to provide this information in a documentation or in an XML Schema [Bir01] [Tho01].

[22]The documentation, which includes a section about nRQL, is available from the RACER download page: `http://www.cs.concordia.ca/~haarslev/racer/download.html`

Semantics As the external syntax has not (yet) been specified, the formal semantics of OWL-QL is presented in a quite general way, and is only included as an appendix of the specification. In particular, the fact that the relationship between the OWL model-theoretic semantics and the OWL-QL semantics has not been specified is not very satisfactory.

Boolean Queries The specification does not specify how to answer boolean queries, i.e., queries with only don't-bind variables or queries with an empty variable list. How to implement a system that can answer queries with only don't-bind variables is described in Section 3.2.3. In the absence of variables query answering is identical to instance checking. In both cases the answer set is empty, instead the answer to such a query is either yes/true or no/false.

Query classes The OWL-QL specification does not introduce the query classes that DQL provides. Since it is difficult for some reasoners to implement all of these requirements, DQL 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 like `?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. Until now it is up to the implementer of an OWL-QL server to provide a documentation of supported query classes and how, if at all, this is communicated to a client. In a real agent-to-agent protocol, however, a client should be able to determine the supported query classes and this is one of the issues a future specification should address.

In short, for an implementer of an OWL-QL server, OWL-QL acts as a guide without a concrete external syntax, a formal relationship with the OWL model-theoretic semantics and proper means to communicate the supported query classes or the conformance level. Until now every implementation has to fill (some of) these gaps and to provide a detailed documentation of how these gaps have been filled.

3.5.2 OWL-QL Systems

Efforts are currently being made, to develop better query support for knowledge representation systems. The establishment of OWL as a W3C recommendation may also promote the proposed OWL-QL specification[23] and so encourage improvements for the currently available systems or the development of new query answering systems.

So far, all introduced systems have some drawbacks. The Stanford implementation covers all features defined by the OWL-QL specification, but delivers in some cases incorrect answers and rejects some queries, without providing an answer. The Manchester

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

implementation does not support all DQL features and is restricted to acyclic conjunctive queries. Both systems are available as Java applications and the Stanford implementation is also available as a servlet, while the Manchester implementation is also available as a web service. Both provide a web client interface and are able to deal with OWL and DAML+OIL knowledge bases.

nRQL provides richer query support, but is not meant as an OWL-QL implementation and is therefore missing many DQL features. In addition, the restriction that a binding is required for all variables, even for those not expected to appear in the answer set, would make it difficult to formulate queries such as the one in Section 3.4 against the KB in Figure 3.18. Apart from this, nRQL is easy to use, and the documentation provides a good introduction to the new features of nRQL.

For all described systems there are still improvements possible. One main topic for query answering systems is scalability. The query answering times for knowledge bases with large amounts of individuals are still far away from the results achieved by databases. For the implementation developed in Manchester, the boolean queries that are necessary to check valid combinations of variable bindings, can cause major delays in case of many candidates. The system would clearly benefit of a further optimisation of this phase in the query answering process, some of which were discussed in Section 3.2.4.

Chapter 4

Querying with OWL-E-QL

This chapter describes how to query with OWL-E-QL, which is an extension of OWL-QL by using OWL-E as the ontology language and by enabling the use of datatype expression in queries. As details of OWL-QL have already been addressed in Chapter 3, we only have to cover: (i) what is OWL-E, (ii) what is the semantics of datatype expression enabled queries, and (iii) how to provide reasoning services for query answering in OWL-E-QL. As a side issue, we also include a short survey on the datatype predicates used in existing Web-related query languages.

4.1 Formal Semantics

4.1.1 Datatypes and Datatype Predicates

Most existing ontology-related formalisms focus on either datatypes (such as RDF(S) and OWL datatyping) or predicates (such as the concrete domain and the type system approach). Pan ([Pan04]) presents a datatype group approach, which provides a unified formalism for datatypes and datatype predicates.

In a datatype group, datatypes and datatype predicates serve different purposes. A datatype d is characterised by its lexical space $L(d)$, value space $V(d)$ and lexical-to-value mapping $L2V(d)$. It can be used to represent its member values through typed literals. A *typed literals* is of the form "v"^^u, where v is a Unicode string, called the *lexical form* of the typed literal, and u is a URI reference of a datatype. A *datatype predicate* (or simply *predicate*) p is characterised by an arity $a(p)$, or a minimum arity $a_{min}(p)$ if p can have multiple arities, and a predicate extension (or simply *extension*) $E(p)$. For instance, $>^{int}_{[20]}$ is a (unary) predicate with $a(>^{int}_{[20]}) = 1$ and $E(>^{int}_{[20]}) = \{i \in V(integer) \mid i > L2V(integer)(\text{"20"})\}$. This example shows that predicates are defined based on datatypes (e.g., $integer$) and their values (e.g., the integer $L2V(integer)$ ("20"), i.e., 20). Predicates are mainly used to represent constraints over values of datatypes which

Abstract Syntax	DL Syntax	Semantics
rdfs:Literal	$\top_{\mathrm{D}}$	$\Delta_{\mathbf{D}}$
owlx:DatatypeBottom	$\bot_{\mathrm{D}}$	$\emptyset$
u a predicate URIref	u	$u^{\mathbf{D}}$
not(u)	$\overline{u}$	if $u \in \mathbf{D}_{\mathcal{G}},\ \Delta_{\mathbf{D}} \setminus u^{\mathbf{D}}$
		if $u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}},\ (\mathsf{dom}(u))^{\mathbf{D}} \setminus u^{\mathbf{D}}$
		if $u \notin \Phi_{\mathcal{G}},\ \bigcup_{n \geq 1} (\Delta_{\mathbf{D}})^n \setminus u^{\mathbf{D}}$
oneOf(“s_1”^^$d_1 \ldots$ “s_n”^^d_n)	$\{$“s_1”^^$d_1, \ldots,$ “s_n”^^$d_n\}$	$\{($“s_1”^^$d_1)^{\mathbf{D}}\} \cup \cdots \cup \{($“$s_n$”^^$d_n)^{\mathbf{D}}\}$
domain$(v_1, \ldots, v_n)$	$[v_1, \ldots, v_n]$	$v_1^{\mathbf{D}} \times \cdots \times v_n^{\mathbf{D}}$
and(P, Q)	$P \wedge Q$	$P^{\mathbf{D}} \cap Q^{\mathbf{D}}$
or(P, Q)	$P \vee Q$	$P^{\mathbf{D}} \cup Q^{\mathbf{D}}$

Table 4.1: OWL-E datatype expressions

they are defined over.

On the other hand, datatypes and datatype predicates are closely related to each other. Datatypes can be regarded as *special* predicates with arity 1 and predicate extensions equal to their value spaces; e.g., the datatype $integer$ can be seen as a predicate with arity $a(integer) = 1$ and predicate extension $E(integer) = V(integer)$. They are *special* because they have lexical spaces and lexical-to-value mappings that ordinary predicates do not have.

The reader is referred to [Pan04] for more details about the datatype group approach.

4.1.2 OWL-E: Extending OWL with Datatype Expressions

Although OWL is rather expressive, it has a very serious limitation; i.e., it does not support customised datatypes and datatype predicates. It has been pointed out that many potential users will not adopt OWL unless this limitation is overcome [Rec04]. To overcome these limitations, [PH04] proposes OWL-E, equivalent to the $\mathcal{SHOIQ}(\mathcal{G})$ DL, which is a decidable extension of both OWL DL and DAML+OIL, which provides customised datatypes and predicates; in fact, [Pan04] shows that all the basic reasoning services of OWL-E are decidable.

OWL-E provides datatype expressions based on the datatype group approach [Pan04], which can be used to represent customised datatypes and datatype predicates. Table 4.1 shows the kind of datatype expression OWL-E supports, where u is a datatype predicate URIref, “s_i”^^d_i are typed literals, $v_1, \ldots, v_n$ are (possibly negated) unary supported predicate URIrefs, P, Q are datatype expressions and $\Phi_{\mathcal{G}}$ is the set of supported predicate URIrefs in a datatype group $\mathcal{G}$. OWL-E provides some new classes descriptions, which are listed in Table 4.2, where $T, T_1, \ldots, T_n$ are datatype properties (where $T_i \not\sqsubseteq^* T_j, T_j \not\sqsubseteq^* T_i$ for all $1 \leq i < j \leq n$),[1] R is an object property, C is a class, E is a datatype expression or a datatype expression URIref, and $\sharp$ denotes cardinality. Note that the first four are

[1] $\sqsubseteq^*$ is the transitive reflexive closure of $\sqsubseteq$.

Abstract Syntax	DL Syntax	Semantics
restriction($\{T\}$ someTuplesSatisfy(E))	$\exists T_1,\ldots,T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \exists t_1,\ldots,t_n.\langle x,t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \le i \le m) \wedge \langle t_1,\ldots,t_n\rangle \in E^{\mathbf{D}}\}$
restriction($\{T\}$ allTuplesSatisfy(E))	$\forall T_1,\ldots,T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \forall t_1,\ldots,t_n.\langle x,t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \le i \le m) \rightarrow \langle t_1,\ldots,t_n\rangle \in E^{\mathbf{D}}\}$
restriction($\{T\}$ minCardinality(m) someTuplesSatisfy(E))	$\geqslant m T_1,\ldots,T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1,\ldots,t_n\rangle \mid \langle x,t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \le i \le m) \wedge \langle t_1,\ldots,t_n\rangle \in E^{\mathbf{D}}\} \ge m\}$
restriction($\{T\}$ maxCardinality(m) someTuplesSatisfy(E))	$\leqslant m T_1,\ldots,T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1,\ldots,t_n\rangle \mid \langle x,t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \le i \le m) \wedge \langle t_1,\ldots,t_n\rangle \in E^{\mathbf{D}}\} \le m\}$
restriction(R minCardinality(m) someValuesFrom(C))	$\geqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \ge m\}$
restriction(R maxCardinality(m) someValuesFrom(C))	$\leqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \le m\}$

Table 4.2: OWL-E introduced class descriptions

datatype group-based class descriptions, and the last two are qualified number restrictions.

4.1.3 Queries and Query Graphs

In this chapter, we consider acyclic conjunctive queries that allow datatype expressions. Formally, a *query* q is of the form

$$\vec{x}\colon q \leftarrow conj(\vec{x};\vec{y};\vec{z})$$

where $conj(\vec{x};\vec{y};\vec{z})$ is a conjunction of *atoms*, $\vec{x}$ is a set of distinguished (or must bind) variables that will be bound to individual names of the knowledge base used to answer the query, $\vec{y}$ is a set of non-distinguished (don't-bind variables) that are existentially quantified variables, and $\vec{z}$ consists of individual names or typed literals. Each atom has one of the forms $v_1 : C$,[2] $\langle v_2, v_3\rangle : r$, $\langle v_4, v_5\rangle : s$, $\langle t_1,\ldots,t_n\rangle :$ E, where C is a concept description, r is an *individual-valued* property, s is a *data-valued* property, E is a datatype expression, $v_1,\ldots,v_4$ are individual names from $\vec{z}$ or *individual-valued* variables from $\vec{x}$ or $\vec{y}$, and v_5 and $t_1,\ldots,t_n$ are typed literals from $\vec{z}$ or *data-valued* variables from $\vec{x}$ or $\vec{y}$. If $conj(\vec{x};\vec{y};\vec{z})$ is empty, the query returns $true$.

Here is an example query q_1

$$\begin{aligned} ?\mathtt{x}\colon q_1 \leftarrow\ & \langle !\mathtt{y}, ?\mathtt{x}\rangle\colon hasParent \wedge\ !\mathtt{y}\colon \mathsf{Male} \wedge\ \langle !\mathtt{y}, !\mathtt{z}\rangle\colon birthYear \\ & \wedge\ \langle !\mathtt{y}, !\mathtt{w}\rangle\colon marriedYear \wedge\ !\mathtt{z}\colon (=^{int}_{[1960]} \vee =^{int}_{[1962]}) \wedge\ \langle !\mathtt{w}, !\mathtt{z}\rangle\colon >, \end{aligned}$$

where $?\mathtt{x}$ is a distinguished *individual-valued* variable, !y is a non-distinguished *individual-valued* variable, !z, !w are non-distinguished *data-valued* variables, $hasParent$ is an *individual-valued* property, Male is a concept name, $birthYear$ and $marriedYear$ are *data-valued* variables, $\ge_{1940}, \le_{1990}$ are unary datatype predicates and $>$ is a binary datatype predicate.

[2]We avoid the more common notation of $C(v_1)$ etc. because it is confusing when C is a complex concept description.

A conjunctive query q can be represented by a directed labelled graph $G(q)$, called query graph, in which there is a normal vertex $\mathtt{x}$ for each individual, typed literal or variable $\mathtt{x}$ in the query, and a normal edge r from a normal node $\mathtt{x}$ to a normal vertex $\mathtt{y}$ for each property atom $\langle \mathtt{x}, \mathtt{y} \rangle : \mathtt{r}$ in the query. Obviously, there are two kinds of normal vertices, viz. *individual-valued* vertices and *data-valued* vertices. For the readers convenience the distinguished variables are represented by a filled vertex ($\bullet$), whereas non-distinguished variables and individuals are represented by an unfilled vertex ($\circ$). Besides normal vertices, a query graph can contain special vertices, called *datatype vertices*, which represent datatype expressions. For each datatype expression atom $\langle \mathtt{t_1}, \dots, \mathtt{t_n} \rangle : \mathrm{E}$ in a query, there exist datatype edges (represented by dotted lines) in $G(q)$ which relate *data-valued* vertices $\mathtt{t_1}, \dots, \mathtt{t_n}$ to the datatype vertex E, labelled with the positions of $\mathtt{t_1}, \dots, \mathtt{t_n}$ in $\langle \mathtt{t_1}, \dots, \mathtt{t_n} \rangle : \mathrm{E}$. For instance, query (4.1.3) corresponds to the query graph presented in Figure 4.1.

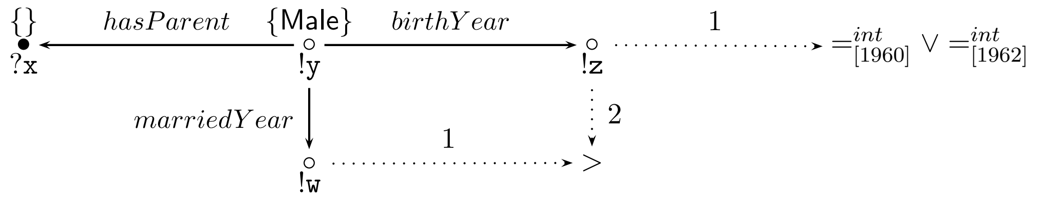


Figure 4.1: A Query Graph

A query graph $G(q)$ is a tuple $\langle \mathbf{V_n}, \mathbf{E_n}, \mathbf{V_d}, \mathbf{E_d} \rangle$, where $\mathbf{V_n}$ is the set of all the normal vertices, $\mathbf{V_d}$ is the set of all the datatype vertices, $\mathbf{E_n}$ is the set of all the normal edges and $\mathbf{E_d}$ is the set of all the datatype edges. Each *individual-valued* vertex $v \in \mathbf{V_n}$ is labelled with $\mathcal{L}(v)$, which is a set of concept descriptions. *Datatype-valued* vertices do not have labels. Each normal edge $e \in \mathbf{E_n}$ is labelled with $\mathcal{L}(e) = r$ such that $\langle start(e), end(e) \rangle : r$ is a property atom in q, where $start$ and end are functions that return the starting and ending vertices of an edge, respectively. Each datatype vertex $p \in \mathbf{V_d}$ is labelled with $\mathcal{L}(p) = E$ where E is a datatype expression. To simplify the presentation, we use $\mathcal{L}(p)$ to represent a datatype vertex p in query graphs. Each datatype edge $g \in \mathbf{E_d}$ is labelled with $\mathcal{L}(g)$, which is an integer and represents the position of $start(g)$ in the corresponding datatype expression atom of the query q.

Two vertices $v_1, v_2 \in \mathbf{V_n} \cup \mathbf{V_d}$ are adjacent, if $\mathcal{L}(\langle v_1, v_2 \rangle) \neq \emptyset$ or $\mathcal{L}(\langle v_2, v_1 \rangle) \neq \emptyset$. Let v_1, v_2, v_3 be vertices, a *path* connects two vertices and it is defined recursively as follows.

- if $\mathcal{L}(\langle v_1, v_2 \rangle) \neq \emptyset$, the set $\{\langle v_1, v_2 \rangle{:}\mathcal{L}(\langle v_1, v_2 \rangle)\}$ is a path connecting v_1 to v_2;
- if the set ϕ is a path connecting v_1 to v_2, the set ϕ' is a path connecting v_2 to v_3 and $\phi \cap \phi' = \emptyset$, then $\phi \cup \phi'$ is a path connecting v_1 to v_3;
- if ϕ is a path from v_1 to v_2, then it is a path from v_2 to v_1 as well.

A *normal path* is a path that all vertices on it are normal vertices. A *(normal) cycle* is a (normal) path connecting a variable vertex to itself. A query graph $G(q)$ is *(normally) cyclic* if a sub-graph of it is a (normal) cycle. A query q is (normally) acyclic if $G(q)$ is not (normally) cyclic.

A datatype vertex p is local (w.r.t. a *individual-valued* vertex v) if all the related *data-valued* vertices (by some datatype edges) of p are adjacent to v; in this case, we call v the master *individual-valued* vertex of the datatype vertex p. A query q is said to be only with local datatype expressions if each datatype vertex p is local w.r.t. some *individual-valued* vertex v in $G(q)$. In this chapter, we consider normally acyclic conjunctive queries only with local datatype expressions.

4.2 Datatypes and Datatype Predicates in Web-related Query Languages

The goal of this section is to give an overview of how to include constraint expressions in query language for the Semantic Web context, and which kind of expressions should be supported. This overview is built from two sources. One source are existing SW query languages. As the SW shouldn't be an island, we have also drawn information from common query language of other areas, namely RDBMSs (SQL) and XML (XQuery).

RDF itself and all extensions (as RDFS, OWL) and query languages don't specify their own data model for atomic data (RDF literals), but reuse the work done in the XML area, especially XML Schema [BE01].

We use the terminology from XQuery and XPath Data Model [FMM$^+$04] to describe literal values:

- An atomic type is a primitive simple type or a type derived by restriction from another atomic type.
- the set of primitive types is listed in the specification (see Figure 4.2).

Of the XML Data Model, only atomic types and values can be used in the Semantic Web context, list and union values aren't allowed for RDF literals. This means that some of the XQuery operators and functions are not applicable in the this context. On the other hand, functions for RDF-related data types (RDF nodes, RDF collections) have to be provided.

[HBEV04] have described important features for RDF query languages. The following of these are related to data types and built-in functions:

- direct support for collection-related functions
- support for XML Schema datatypes

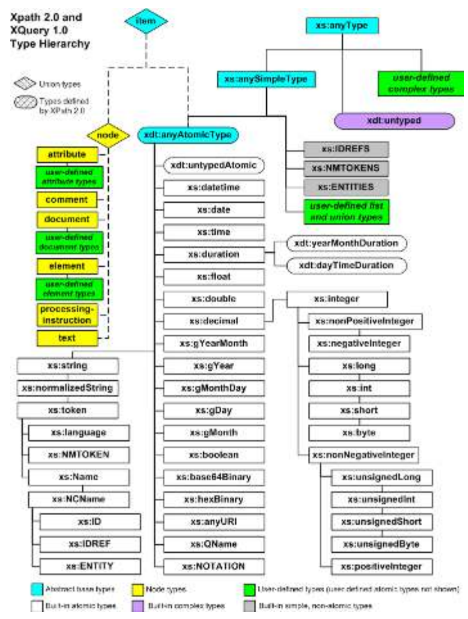


Figure 4.2: XQuery Data Model Type Hierarchy (from [FMM+04]

- support for URI-related functions (e.g. namespace filtering)
- multi-language support

We will refer to these in the later sections

4.2.1 Handling of variable constraints in existing query languages

Essentially there are two approaches to handling variable constraints:

- **Constraint expressions** built-in functions return arbitrary atomic types, the resulting constraint expression(s) must be of type boolean.
  SQL, XQuery and all SQL-like RDF languages (e.g. RDQL, RQL, SeRQL, SPARQL) use (part of) a where clause to add such constraint expressions to a query.[3]
- **Constraint predicates** there are only built-in predicates, which are satisfied if the arguments are in the relation specified by the corresponding constraint clause operator (e.g. $(sum\ ?x\ ?y\ ?z)$ is satisfied if $?x\ = ?y + ?z$. This approach is used by rule-based RDF languages, e.g. SWRL, TRIPLE, QEL and DQL [4]

[3] XQuery operators are just syntactic sugar to facilitate inline operators (e.g. in '2+2') additionally to prefix expressions.

[4] DQL uses a knowledge base where some nodes are variables to specify a query. No built-in predicates are part of the specification, only equality is supported (implicitly).

These approaches are different in style and syntax, but equally powerful. A language having n constraint predicates can be converted to a language with one constraint predicate ('satisfied') which is satisfied if the boolean argument equals 'true' and n constraint clause operators/functions. For example, the conjunctive constraint clause
$(greaterThan\ ?x\ ?y) \wedge (sum\ ?x\ ?a\ ?b) \wedge (sum\ ?y\ ?c\ ?d)$
could be translated to
$(satisfied\ (?a+?b > ?c+?d))$
As the latter type of expressions is used in SQL and XQuery, as well as in all non-rule-based RDF query languages, it seems reasonable to integrate such a syntactic approach into rule-based languages as well (possibly as alternative syntax).

4.2.2 Built-in Functions/Predicates in current RDF query languages

In *RDQL*, a query consists of an RDF graph template specifying the structure of matching subgraphs and additional constraints of the form <variable operator constant>. Equality operators (=, !=), comparison operators (<, >) and a pattern matching operator for strings (˜=) are available. There is no formal specification of these operators. Boolean operators to construct more complex expressions are also provided.

SeRQL provides numeric comparison operators, string pattern matching and functions for RDF node type checks (isResource, isLiteral). These can be combined using boolean operators.

SPARQL SPARQL is based on RDQL, but it is planned to rely on XQuery operators and functions instead of the ones provided in RDQL. Details are not yet provided ([PS04], section 12).

SWRL built-in predicates are mostly based on corresponding XQuery functions and operators. For primitive datatypes a selection of the most important XQuery expressions are supported. Additionally, predicates regarding collections and URIs are provided.

4.2.3 RDF(S) Related Predicates

Support for Collections RDF as well as OWL have a notion of collections. While current query languages allow to query these inderectly by referring to the graph structure for representation of the collection, there is no direct support.

A query language should have the following functions related to collections:

- $(member\ ?c\ ?x)$ satisfied if $?c$ is a collection and contains $?x$.
- $(union\ ?r\ ?c\ ?d)$, $(intersection\ ?r\ ?c\ ?d)$, $(subtraction\ ?r\ ?c\ ?d)$ the common set operator and bind $?r$ to the respective resulting set.

For Sequences, the following operators are useful:

- $(indexOf\ ?r\ ?c\ ?x)$ binds $?r$ to position of $?x$ in $?c$.
- $(concat\ ?r\ ?c\ ?d)$ binds r to the concatenation of $?c$ and $?d$.

These operators should be able to work on linked lists (as used in OWL-DL) as well. It is an interesting question how XQuery sequence support and RDF collections support could be aligned.

Support for Resource Types A query may also require that an RDF node is of a specific type. For example, this is necessary to return the transitive closure of all anonymous resources connected to a non-anonymous resource. Following types exist:

- Literal.
- Resource
  - Anonymous resource
  - Non-Anonymous resource

A predicate $(nodeType\ ?x\ ?t)$ is satisfied if $?t$ is one of these four type specifiers, and $?x$ is a resource of the requested type.

URI predicates While in general URIs are supposed to be opaque, in RDF it is often useful to split them into their namespace and local name parts. Thus, the following functins should be provided:

- $(namespace\ ?r\ ?u)$ binds $?r$ to the namespace part of uri $?u$.
- $(localname\ ?r\ ?u)$ binds $?r$ to the local name part of uri $?u$.

4.2.4 Functions and operators for XML atomic types

XQuery already provides an extensive set of functions and operators on common atomic types as string, numerics and date. The most promising approach seems to draw on these efforts and take over at least the semantics of these functions as defined in [MME04]. We refer to this document and the SWRL specification [HPSB$^+$04] regarding a suitable subset of XQuery operators for the RDF context.

4.3 An Extended Rolling-up Algorithm

In general, query answering with datatypes is harder than that without datatypes. If a datatype-free query contains only distinguished variables, one could replace all variables with individual names from the knowledge base and check if the grounded query is logically implied by the knowledge base. This is impossible for non-datatype-free queries because there are infinite numbers of typed literals.

In this section, we extend the rolling up technique presented in [Tes01] to support query answering with normally acyclic conjunctive queries with local datatype expressions. The basic idea behind the rolling-up technique is to convert *data-valued* property atoms and datatype expression atoms into concept atoms. Informally speaking, there are three cases.

- **No datatype expression atoms:** The rationale behind this rolling up can easily be understood by the use of the oneOf constructor for datatypes. The *data-valued* property atom $\langle \mathsf{a}, \text{“18”\^\^xsd:integer} \rangle : \mathit{age}$ can be transformed into the equivalent concept atom $\mathsf{a}\colon \exists \mathit{age}.\{\text{“18”\^\^xsd:integer}\}$, where $\{\text{“18”\^\^xsd:integer}\}$ is the datatype containing only one value, i.e., the integer 18. Now let us consider the *data-valued* property atom $\langle \mathsf{a}, !\mathsf{y} \rangle\colon \mathit{age}$ where we have a non-distinguished variable instead of a typed literal. Similarly, it can be transformed into the equivalent concept atom $\mathsf{a}\colon \exists \mathit{age}.\top_{\mathrm{D}}$, where $\top_{\mathrm{D}}$ is the datatype predicate that represents the whole datatype domain.

- **Datatype expressions with arity 1:** A unary datatype expression atom with the rolled up *data-valued* variable can be absorbed into the corresponding concept atom. For instance, $\langle \mathsf{a}, !\mathsf{y} \rangle : \mathit{age} \ \wedge \ !\mathsf{y} :<^{\mathit{int}}_{[20]}$ can be transformed into the equivalent concept atom $\mathsf{a}\colon \exists \mathit{age}. <^{\mathit{int}}_{[20]}$.

- **Datatype expressions with arbitrary arities:** Similarly, a datatype expression atom with arbitrary arity can be absorbed into the corresponding master concept atom. For instance, $\langle \mathsf{a}, !\mathsf{y} \rangle\colon \mathit{income} \ \wedge \ \langle \mathsf{a}, !\mathsf{z} \rangle\colon \mathit{expense} \ \wedge \ \langle !\mathsf{y}, !\mathsf{z} \rangle\colon >$ can be transformed into the equivalent concept atom $\mathsf{a}\colon \exists \mathit{income}, \mathit{expense}. >$. In this example, the datatype predicate $>$ is local w.r.t. the individual a.

In what follows, we present the rolling-up algorithm in more details. Given a query graph[5] $G = \langle \mathbf{V_n}, \mathbf{E_n}, \mathbf{V_d}, \mathbf{E_d} \rangle$ and an *individual-valued* vertex $v \in \mathbf{V_n}$, the query graph can be transformed into a normal tree (with root v_0), in which the direction of each normal edge points from the root v_0 to the leaves. The directions of normal edges can be satisfied by the application of the $\mathit{flip}(G, \langle x, y \rangle)$ function when necessary. The $\mathit{flip}(G, \langle x, y \rangle)$ function returns a new graph $G' = \langle \mathbf{V_n}', \mathbf{E_n}', \mathbf{V_d}', \mathbf{E_d}' \rangle$ with $\mathbf{V_n}' = \mathbf{V_n}, \mathbf{E_n}' = (\mathbf{E_n} \setminus$

[5]We will specify the kind of requirement for such a query graph later in the chapter.

$\langle x, y\rangle) \cup \langle y, x\rangle, \mathbf{V_d}' = \mathbf{V_d}, \mathbf{E_d}' = \mathbf{E_d}$, and $\mathcal{L}(\langle y, x\rangle) = Inv(\mathcal{L}(\langle x, y\rangle))$.[6]

The process can be illustrated using the query graph G_1 of the query q_1:

$$\begin{aligned} ?\mathtt{x}\colon q_1 \leftarrow \quad & \langle !\mathtt{y}, ?\mathtt{x}\rangle\colon hasParent \wedge\ !\mathtt{y}\colon \mathsf{Male} \wedge\ \langle !\mathtt{y}, !\mathtt{z}\rangle\colon birthYear \\ & \wedge\ \langle !\mathtt{y}, !\mathtt{w}\rangle\colon marriedYear \wedge\ !\mathtt{z}\colon (=_{1960} \vee =_{1962}) \wedge\ \langle !\mathtt{w}, !\mathtt{z}\rangle\colon>, \end{aligned}$$

$$\begin{tikzcd}
\underset{?\mathtt{x}}{\overset{\{\}}{\bullet}} & \underset{!\mathtt{y}}{\overset{\{\mathsf{Male}\}}{\circ}} \arrow[l, "hasParent"'] \arrow[r, "birthYear"] \arrow[d, "marriedYear"'] & \underset{!\mathtt{z}}{\circ} \arrow[r, dotted, "1"] \arrow[d, dotted, "2"] & =_{1960} \vee =_{1962} \\
 & \underset{!\mathtt{w}}{\circ} \arrow[r, dotted, "1"] & >
\end{tikzcd}$$

$flip(G_1, \langle !\mathtt{y}, ?\mathtt{x}\rangle)$ returns the following query graph G_{12}, which contain a normal tree.

$$\begin{tikzcd}
\underset{?\mathtt{x}}{\overset{\{\}}{\bullet}} \arrow[r, "hasParent^-"] & \underset{!\mathtt{y}}{\overset{\{\mathsf{Male}\}}{\circ}} \arrow[r, "birthYear"] \arrow[d, "marriedYear"'] & \underset{!\mathtt{z}}{\circ} \arrow[r, dotted, "1"] \arrow[d, dotted, "2"] & =_{1960} \vee =_{1962} \\
 & \underset{!\mathtt{w}}{\circ} \arrow[r, dotted, "1"] & >
\end{tikzcd}$$

As the resulting normal tree contains no datatype vertices, we should reduce type literal atoms and datatype expression vertices.

The reduction of typed literal vertices and *data-valued* vertices can be satisfied by the application of the function $removeTL(G)$. Let t be a typed literal vertex representing the typed literal "s"^^u, d the datatype vertex t is adjacent to,[7] $\mathcal{L}(d) = E$, i the label (integer) of the datatype edge connecting t and d. $removeTL(G)$ rewrites the label of d as $E\mid_{i=\text{“}s\text{”\^{}\^{}}u}$, which is a parameterised datatype expression, and removes t and the datatype edge connecting t and d. For instance, given the query graph G_2 of the query q_2

$$?\mathtt{x}\colon q_2 \leftarrow ?\mathtt{x}\colon \mathsf{Person} \wedge\ \langle ?\mathtt{x}, !\mathtt{y}\rangle\colon age \wedge\ \langle !\mathtt{y}, \text{“18”\^{}\^{}xsd:integer}\rangle\colon> .$$

$$\begin{tikzcd}
\underset{?\mathtt{x}}{\overset{\{\mathsf{Person}\}}{\bullet}} \arrow[r, "age"] & \underset{!\mathtt{y}}{\circ} \arrow[r, dotted, "1"] & > \\
 & \text{“18”\^{}\^{}xsd:integer} \arrow[ur, dotted, "2"'] &
\end{tikzcd}$$

$removeTL(G_2)$ returns the following query graph G_{22}.

$$\begin{tikzcd}
\underset{?\mathtt{x}}{\overset{\{\mathsf{Person}\}}{\bullet}} \arrow[r, "age"] & \underset{!\mathtt{y}}{\circ} \arrow[r, dotted, "1"] & >\mid_{2=\text{“18”\^{}\^{}xsd:integer}}
\end{tikzcd}$$

Note that the $>$ predicate has the arity of $a(>) = 2$ and extension $E(>) = \{\langle i, j\rangle \mid i > j \text{ and } i \in V(integer) \text{ and } j \in V(integer)\}$, while the parameterised predicate

[6]The function $Inv(r)$ returns the inverse of a property r; e.g., $Inv(love) = love^-$ and $Inv(love^-) = love$.

[7]Here we assume that each typed literal vertex is adjacent to only *one* datatype vertex.

$>|_{2=\text{“18”}\hat{}\hat{}\text{xsd:integer}}$ has arity $a(>|_{2=\text{“18”}\hat{}\hat{}\text{xsd:integer}}) = 1$ and $E(>|_{2=\text{“18”}\hat{}\hat{}\text{xsd:integer}}) = \{i \mid i > L2V(integer)\,(\text{“18”})$ and $i \in V(integer)\}$.

The reduce of datatype expression vertices and *data-valued* vertices can be satisfied by the application of the function $removeDV(G)$. Let d be a datatype expression vertex, the arity of $\mathcal{L}(d)$ is n, v the master *individual-valued* vertex of d, $v_1, \ldots, v_n$ the *data-valued* vertices between v and d. $removeDV(G)$ adds the concept description $\exists s_1, \ldots, s_n.E$ into $\mathcal{L}(v)$, where $s_i = \mathcal{L}(\langle v, v_i\rangle), E = \mathcal{L}(d)$, and removes the datatype vertex d and all datatype edges connecting d and $v_1, \ldots, v_n$. This step is applied on all datatype vertices in the query graph. $removeDV(G)$ then removes all the *data-valued* vertices and the normal edges connecting them and corresponding *individual-valued* vertices. The resulting query graph is a normal tree. For instance, $removeDV(G_{12})$ returns the following normal tree G_{13}.

$$\begin{tikzcd}
\overset{\{\}}{\underset{?\mathtt{x}}{\bullet}} \arrow[r, "hasParent^-"] & \overset{\{\ \mathsf{Male}, \exists birthYear.(=_{1960} \vee =_{1962}),\ \exists marriedYear, birthYear. > \}}{\underset{!\mathtt{y}}{\circ}}
\end{tikzcd}$$

Finally, the rolling-up from the leaves of the normal tree to the root v_0 can be satisfied by the application of the function $removeLeaf(G)$. Let l be a leaf, v the adjacent *individual-valued* vertex of l. $removeLeaf(G)$ adds the concept description $\exists r.C$ into $\mathcal{L}(v)$, where $r = \mathcal{L}(\langle v, l\rangle), C = C_1 \sqcap \ldots \sqcap C_n$ where $C_1, \ldots, C_n \in \mathcal{L}(l)$ (if $\mathcal{L}(l) = \emptyset$, $C = \top$; if l represents an individual a, $C = \{\mathsf{a}\}$), and removes l and the normal edge connecting v and l. This step is applied to each leaf until only the distinguished variable at the root is remaining - here the order of the reduction of leaves is not important. For instance, $removeLeaf(G_{13})$ returns the following normal tree G_{14}.

$$\overset{\{\exists hasParent^-.(\mathsf{Male} \sqcap \exists birthYear.(=_{1960} \vee =_{1962}) \sqcap \exists marriedYear, birthYear. >)\}}{\underset{?\mathtt{x}}{\bullet}}$$

Therefore, with the help of the rolling-up algorithm, query answering of query q_1 is reduced to the problem of retrieving all the instances of the concept description

$$\exists hasParent^-.(\mathsf{Male} \sqcap \exists birthYear.(=_{1960} \vee =_{1962}) \sqcap \exists marriedYear, birthYear. >).$$

In the rest of this chapter, when we say we roll up a vertex x of a query graph, we mean we roll up the query graph into a normal tree with only one vertex x.

4.4 Reducing Query Answering in OWL-E-QL to Knowledge Base Satisfiability in OWL-E

4.4.1 Boolean Queries

If there are no distinguished variables in a normally acyclic query, there are two possibilities here:

1. There exist some non-distinguished variables in a query q. We can randomly pick up a non-distinguished variable $!\mathtt{x}$ from q and and roll up $!\mathtt{x}$. Hence, we can transform the q into
$$q \leftarrow !\mathtt{x}\colon C$$
where C is the conjunction of all the concept descriptions in the label of the *individual-valued* vertex representing $!\mathtt{x}$ in the resulting query graph. Therefore, query answering of q is reduced to concept satisfiability of C.

2. There exist no non-distinguished variables in a query q. We can randomly pick up an individual $\mathtt{a}$ from q and choose it as the root of the normal tree and apply the rolling-up algorithm. Therefore, we can transform the q into
$$q \leftarrow \mathtt{a}\colon C$$
where C is the conjunction of all the concept descriptions in the label of the *individual-valued* vertex representing $\mathtt{a}$ in the resulting query graph. Therefore, query answering of q is reduced to instance checking $\mathtt{a}\colon C$.

4.4.2 Acyclic Queries without Datatype Expression Atoms

In this section, we consider acyclic queries with only class atoms, *individual-valued* property atoms and *data-valued* property atoms. We assume that the knowledge base is consistent.

All Variables are Distinguished

Given a query q, the algorithm of answering queries in which all variables are distinguished consists of the following steps:

1. Roll-up each *individual-valued* variable $?\mathtt{v_i}$ in q and retrieve a set of individuals O_i as candidates of $?\mathtt{v_i}$. Let us take the following query q_3 as an example:

$\langle ?\mathtt{v}_1, ?\mathtt{v}_2, ?\mathtt{u}_1, ?\mathtt{u}_2\rangle\colon q_3 \leftarrow \langle ?\mathtt{v}_1, ?\mathtt{v}_2\rangle\colon \mathit{brother} \wedge\ \langle ?\mathtt{v}_1, ?\mathtt{u}_1\rangle\colon \mathit{age} \wedge\ \langle ?\mathtt{v}_2, ?\mathtt{u}_2\rangle\colon \mathit{age}.$

We can roll up $?\mathtt{v}_1$ and $?\mathtt{v}_2$ and retrieve their candidates O_1 and O_2.

2. Get the valid candidate combinations. We construct a super-query q' of q by removing all the datatype property atoms from q. Each candidate combination is tested by the corresponding boolean query, which is constructed by replacing each distinguished variable in q' with its corresponding candidate in the combination. Note that if there are only datatype property atoms in a query, then all the combination are valid. In the above example, the datatype property atoms-free super query q'_3 of q_3 is

   $$\langle ?\mathtt{v}_1, ?\mathtt{v}_2 \rangle : q'_3 \leftarrow \langle ?\mathtt{v}_1, ?\mathtt{v}_2 \rangle : brother.$$

   If $O_1 = \{a_1, a_2\}$, $O_2 = \{b_1, b_2, b_3\}$, we can test the candidate combination $\langle ?\mathtt{v_1} \rightarrow a_1, ?\mathtt{v}_2 \rightarrow \mathtt{b}_1 \rangle$ by replacing $?\mathtt{v_1}$ with $\mathtt{a_1}$ and $?\mathtt{v_1}$ with $\mathtt{b_1}$ in the query q'_3 and turn it into a boolean query as follows:

   $$q''_3 \leftarrow \langle \mathtt{a}_1, \mathtt{b}_1 \rangle : brother.$$

   If the above query returns $true$, then the candidate combination is valid; otherwise, it is not.

   If there exists no valid combination, the result of a query is an empty set; otherwise, we proceed with step 3. We call objects in valid combinations *c-valid* (c for combination) candidates of corresponding variables.

3. Get the values for all the *data-valued* variables. This can be done in two steps.

   (a) Get the explicitly stated individuals and values pairs $EIDPairs$ for each *data-valued* variable $?\mathtt{u_i}$. Let $\langle ?\mathtt{v_i}, ?\mathtt{u_i} \rangle : s$ be a datatype property atom, $Objects(?\mathtt{v_i})$ be the set of c-valid candidates of $?\mathtt{v_i}$. For each c-valid candidate $c \in Objects(?\mathtt{v_i})$ of $?\mathtt{v_i}$, if there exists any sub-property s' of s such that $s'(c,t)$ is in the ABox, we store the mapping $\langle ?\mathtt{v_i} \mapsto c, ?\mathtt{u_i} \mapsto t \rangle$ into $EIDPairs$.

   (b) Get the implicitly stated individuals and values pairs $IIDPairs$ for each *data-valued* variable $?\mathtt{u_i}$ that does not appear in $EIDPairs$. Let $\langle ?\mathtt{v_i}, ?\mathtt{u_i} \rangle : s$ be a datatype property atom, $Objects(?\mathtt{v_i})$ be the set of c-valid candidates of $?\mathtt{v_i}$. For each c-valid candidate $c \in Objects(?\mathtt{v_i})$ of $?\mathtt{v_i}$, we check the most specific class D of c and see if D implies any fixed value for any sub-property s' of s. There can be several cases here:

      i. there exist a sub-class $\exists s'. =_t$ of D, or
      ii. there exist sub-classes $\exists s'.d$ and $\forall s'. =_t$ of D,
      iii. the variants of the above two cases that involves the use of inverse roles. For instance, D implies some fixed value of s if $\exists r.(\forall r^-.(\exists s'. =_t))$ is a sub-class of D.

      Note that in some datatypes $= t$ can have some variants too. For instance, in the integer datatype, $=^{int}_{[24]}$ is equivalent to $>^{int}_{[23]} \wedge <^{int}_{[25]}$.

Some Data-valued Variables are Non-distinguished

Now we consider the case when all individual-valued variables are distinguished, but some of the data-valued variables are non-distinguished. In this query answering can be achieved by rolling up the non-distinguished data-valued variables first. The function $removeDV(G)$ can be used to eliminate all the non-distinguished data-valued variables. After this has been done, the procedure described above can be used to answer the query.

As an example, consider a slightly modified query from the example used in the previous section.

$$\langle ?\mathtt{v}_1, ?\mathtt{v}_2, ?\mathtt{u}_1\rangle : q_4 \leftarrow \langle ?\mathtt{v}_1, ?\mathtt{v}_2\rangle : brother \wedge \ \langle ?\mathtt{v}_1, ?\mathtt{u}_1\rangle : age \wedge \ \langle ?\mathtt{v}_2, !\mathtt{u}_2\rangle : age.$$

Here the data-valued variable $!\mathtt{u}_2$ is non-distinguished and can be eliminated by applying the function $removeDV(G(q4))$ once. As a result the concept expression of $\exists age.\top_\mathrm{D}$ is conjoined with the label of the vertex $?\mathtt{v}_2$. Now procedure described in the previous section is applicable.

Some Individual-valued Variables are Non-distinguished

Here, a query may contain individual-valued variables that are existentially quantified, but the knowledge base used to answer the query must not necessarily include a named individual as a binding for the variable. Consider for example the knowledge base in Example 5 and the query

$$\langle ?\mathtt{x}, ?\mathtt{u}\rangle : q_5 \leftarrow \langle ?\mathtt{x}, !\mathtt{y}\rangle : brother \wedge \ \langle !\mathtt{y}, ?\mathtt{u}\rangle : age.$$

Example 5
KB = $\{\mathcal{T}, \mathcal{A}\}$
$\mathcal{T}$ = $\{$*Male* $\sqsubseteq \neg$ *Female*
$\top \sqsubseteq \forall$ *brother.Male*
$\top \sqsubseteq \forall$ *sister.Female*
brother $\sqsubseteq$ *sibling*
sister $\sqsubseteq$ *sibling*$\}$
$\mathcal{A}$ = $\{$*john:(=1 brother* $\sqcap$ *=1 sister* $\sqcap$ *=2 sibling)*
$\langle john, francis\rangle : sibling$
$\langle john, andrea\rangle : sibling$
$\langle francis\rangle$*:(= age 20)*
$\langle andrea\rangle$*:(= age 20)*$\}$

From the knowledge base we know that the individual named john has exactly one sister and exactly one brother. In addition we know that andrea and francis are the names of john's siblings, but we do not know who is the brother and who is the sister. Nevertheless, we know that both are of the age 20 and therefore, the query has an answer in which john is a valid binding for the individual-valued variable ?x and 20 is a valid binding for the data-valued variable ?u. However, if the specified age would have been different, the query would have no answer, since there is no binding for ?u.

In this case there is no straight forward algorithm to retrieve the query answers. The rolling up can be used to retrieve candidates. For this query the verification of valid combinations is unnecessary, since there is only one distinguished individual-valued property. Queries with multiple distinguished variables will still need the verification of valid combinations. The third step, however, can only be applied, if the data-valued variables are connected to distinguished variables. In this case, step three of the case where all variables were distinguished can be used to derive the valid bindings for the data-valued variables. If the data-valued variables are connected to undistinguished individual-valued variables, some answers may be found by treating the individual-valued variables as distinguished ones, without delivering the found bindings in the query answer. However, the answer for the given example would not be returned. To retrieve also these answers, further reasoning is necessary that we will investigate in future work.

4.4.3 Normally Acyclic Queries

All Variables are Distinguished

In addition to the already described case where all variables are distinguished, here we cover scenarios where the query includes datatype expressions. The beginning of the query answering process is the same as for the case where all variables are distinguished, but there are no datatype expressions in the query. Firstly the candidates for the individual-valued variables are retrieved and the valid candidate combinations are determined. In the third step the values for the data-valued variables are retrieved. In addition to these steps, a forth step is necessary to verify the valid combinations for the data-valued variables.

In the example query graph given in Figure 4.3, we could imagine to retrieve more than one value $?u_2$. In this case the retrieved value for $?u_1$ has to be tested with all retrieved values for $?u_2$ to see which are valid for the given datatype expression. In general all combinations of candidates for the data-valued variables have to be tested, as it is necessary for the individual-valued variables.

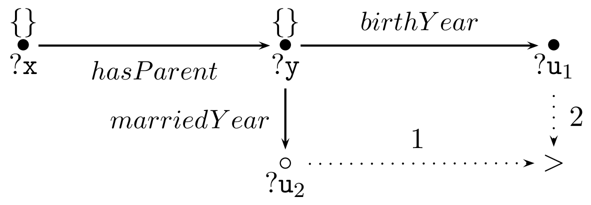


Figure 4.3: A Query Graph

Some Data-valued Variables are Non-distinguished

In this scenario we can retrieve and validate the candidates for the individual-valued variables as before. A different procedure is necessary, if not all data-valued variables are distinguished. If all are non-distinguished, we can roll up all data-valued variables as described in Section 4.3. In case some data-valued variables are distinguished and others are not for a datatype expression, there are more steps necessary:

1. Validate that a solution is possible in the given knowledge base. To determine this, one can treat all involved data-valued variables as non-distinguished and do the rolling up as described.

2. Retrieve candidates for the distinguished data-valued variable. If there are no candidates the query has no answer.

3. The retrieved candidates have to be tested, to determine which are valid in combination with the non-distinguished variables and candidates for other data-valued variables. In this step the distinguished variables are replaced with a candidate. Therefore, the resulting query is free of distinguished data-valued variables and the can be handled as described in Section 4.3.

Some Individual-valued Variables are Non-distinguished

The process of determining valid bindings for a mixture of distinguished and non-distinguished individual-valued variables has already been described in Section 3.2.3 of Chapter 3. After determining the valid candidate combinations for the individual-valued variables, one can imagine two situations.

- All data-valued variables are connected to a distinguished variable. In this case, the bindings for the data-valued variables have to be retrieved and tested for each candidate. The process is the same as described in the case of only distinguished variables, since for the retrieval of valid bindings for the data-valued variables only the candidates of the distinguished master individual-valued vertex are taken into account.

- There are data-valued variables connected to non-distinguished variables. In this case it is difficult to determine all valid query answers. The reasons for this have already been described for the case where some individual-valued variables are non-distinguished but the query did not contain datatype expression.

4.5 Summary

In this chapter, we discuss the query answering in OWL-E-QL. We show how the existing rolling up techniques can be extended to support datatype expression-enabled queries. We also provide a short survey on the datatype predicates used in Web-related query languages.

Chapter 5

A Fuzzy Extension

5.1 Introduction

The representation of uncertainty and imprecision has received a considerable attention in database and query services. The currents efforts are focused to extend the existed knowledge formalisms to deal with the imperfect nature of real word information (which is likely the rule and not the exception). The use of DLs in the context of the semantic web points out the necessity of extending DLs with capabilities, which allow the treatment of the uncertain and imperfect knowledge. In fact classical DLs are insufficient for describing real retrieval situations, as the retrieval is usually not only a yes or no question: (i) the representation of the knowledge which the system have access to is inherently imperfect; and (ii) the relevance of the content to a query can thus be established only up to a limited degree. Because of this, we need a logic in which, rather than taking crisp decisions whether a KB entails a query or not, we are able to enrich and *rank* the retrieved objects according to how strongly the systems believes in their relevance to a query.

The choice of fuzzy set theory to extend DLs plays a twofold role: (i)it directly models semantic-based retrieval, and (ii) it offers an ideal framework for more sophisticated query processes. From a syntactical point of view fuzzy DLs provides fuzzy *assertions*, that is, expressions of type $\langle a, n \rangle$, where a is a crisp assertion and $n \in [0, 1]$. We use the term fuzzy simple assertion, fuzzy axiom, and a fuzzy Knowledge Base (KB) with the obvious meaning. Then, $\langle \exists hasHeight.Height(i), .7)$ is a fuzzy simple assertion with intended meaning "the membership degree of constant i to concept $\exists hasHeight.Height$ is .7". From a semantics point of view, fuzzy logic captures the notion of imprecise concept, i.e. a concept for which a clear and precise definition is not possible. Fuzzy concepts play a key role in information retrieval. For instance, in the previous example the semantics are that the person(i) is medium tall.

In D2.5.1 is presented the framework for extending DLs with fuzzy logic. It is presented a way to extend OWL with the notion of fuzzy assertions. The extension in the current syntax of OWL that we propose is to add an assertion degree representing the degree that an OWL individual belongs to an OWL class or two OWL individuals in an OWL relation. In addition to the fuzzy assertion extension, we propose a way to extend the SWRL syntax with degrees of importance. The degree of importance is assigned in the atoms of a SWRL rule representing the degree of importance of the atoms for the activation of the rule. In this way, the atoms in the head of a rule can be activated with an assertion degree depending the assertion degrees of the involved variables and the degrees of importance of the atoms in the body of the rule. There are two main differences between the assertion degree and the degree of importance: i)they have different semantics ii)they have different way of calculation. Basically, these degrees are used to manage two different kinds of uncertainty as explained in the following section.

In this chapter is presented the way to extend the query languages based in DLs and more specifically the OWL-Q Language, with fuzzy logic. We provide the syntactic as well as the semantic extensions necessary for constructing fuzzy queries in OWL-QL. OWL-QL is indented to be a candidate standard language and protocol for query-answering dialogues among Semantic Web computational agents during which answering agents may derive answers to questions posed by query agents.

The structure of this chapter is as follows: the first section presents a survey on past and current work involved with fuzzy queries and extensions of query languages with fuzzy logic. In the second section we analyse the two kinds of uncertainty that exist in real life applications. Also, we sumurise the work done in the D2.5.1 for the extension of OWL and SWRL with fuzzy operators. Finally, we present the notion of fuzzy entailment for implementing fuzzy queries. In section 3, we describe the extensions in OWL-QL necessary to realise fuzzy queries. Also we present the way to construct fuzzy assumptions. In the last section, we present a use case for better understanding the need of fuzzy logic in query languages.

5.2 Queries and Uncertainty - State of the Art

As hardware becomes more powerful and as software becomes more sophisticated, it is increasingly possible to make use of multimedia data such as images and video. If we wish to access multimedia data through a database system a number of new issues arise. For example a multimedia database might deal with pictures that have a complicated coloring pattern and also contains a number of shapes. These differences between multimedia databases and traditional databases bread the need of extending the applicability of traditional databases; hence some new techniques have been proposed to deal with uncertain or incomplete information [Zad65]. Fuzzy sets and fuzzy logics have been introduced into database systems for this purpose [MK85].

Since then fuzzy databases were widely used and a lot of research was made in this area. A fuzzy database library has been build by Omron Corporation [Cor92] and the standard relational SQL has been extended to Fuzzy (relational) SQL [QWC^{+}93]. Yang and others [QCJ+95] stated that despite the fact that nested queries allowed users to express their queries in a convenient way their evaluations were very inefficient if they were implemented in a naive way as nested loops. They have extended and modified those unnesting techniques for fuzzy databases and they also provided some new unnesting techniques for fuzzy databases.

In many fuzzy databases [BF82, PH88, JZ86, DD90, SA90, MCG+93] in which the meaning of a linguistic fuzzy set such as ”young”, is represented by a fuzzy set and thus its membership function. So one membership function is used to interpret a fuzzy term under all circumstances. Zhang et. al. [WCBN95] stated that similarly to real word a fuzzy term must have several meanings among which one must be chosen dynamically according to a given context, proposing that fuzzy databases systems must support multiple and dynamic interpretation of fuzzy terms. They achieved that by a scaling process that was used to transform a pre-defined meaning of a fuzzy term into an appropriate meaning in the given context. Sufficient conditions were given for a nested fuzzy query with relative quantifiers to be unnested for an efficient evaluation. They also proposed an attribute dependent interpretation in order to model the applications in which the meaning of the fuzzy term in an attribute must be interpreted with respect to values in other related attributes. For this purpose two necessary and sufficient conditions for a tuple to have a unique attribute-dependent interpretation were provided. They described an interpretation system that allows queries to be processed based on the attribute-dependent interpretation of the data and also two techniques grouping and shifting to improve the implementation.

Papadias et.al. [DND99]worked on the configuration similarity in the context of Digital Libraries, Spatial Databases and Geographical Information systems. The queries in these systems retrieved all databases configurations that matched an input description. Their approach introduced a framework for configuration similarity that takes into account all major types of spatial constraints. They also defined appropriate fuzzy similarity measures for each type of constraint to provide flexibility and allow the system to capture real-life needs. Ending they also applied pre-processing techniques to explicate constraints in the query.

Ending Morris and Jankowski [AP00] combined fuzzy sets and databases in multiple criteria spatial decision making. Spatial decision making is a fundamental function of contemporary Geographic Information Systems (GIS). One of the most fertile GIS development areas is integrating multiple criteria decision models into GIS querying mechanisms. The classic approach for this integration was to use Boolean techniques of MCDM with crisp representations of spatial objects (features) to produce static maps as query answers. They visually represented query results more precisely by implementing fuzzy sets membership as a method for representing the performance of decision alternatives on evaluation criteria, fuzzy methods for both criteria weighting and capturing geographic preferences and fuzzy object oriented spatial databases for feature storage.

5.3 Representing Queries Using Fuzzy Logic

As previously described, fuzzy logic and queries have been combined in many ways to manage uncertain and imprecise knowledge. Before we describe how to represent and implement queries using fuzzy logic, we will define the two kinds of uncertainty that exist in real-life applications. We will analyse the two kinds of uncertainty by demonstrating an example of the use case presented in the last section. Let us consider that an advertisement company requires a female model who is tall and thin. Since queries need artificial precision, this query is formed as:

Query("List all the female models, which are
over 175 cm and under 60 kilos")

The query pattern is as follows:

Query Pattern $\{$(hasSex ?p ?a)(type ?a female)$\wedge$(hasHeight ?p ?c)
(type ?c Height $\geq$ 175)$\wedge$(hasWeight ?p ?d)(type ?d Weight $\leq$ 60)$\}$
Must-Bind Variables List: (?p)
Answer pattern $\{$(hasSex ?p "Female" $\wedge$ hasHeight ?p "over 175 cm"
$\wedge$ hasWeight ?p "under 60 kilos")$\}$
Answer1: ("Mary is a female model who is over 175 cm and is under 60 kilos")
Answer2: ("Susan is a female model who is over 175 cm and is under 60 kilos")

The above situation happens having a crisp query in a crisp KB. In a 700 models database the answers that make the query true (entails the KB) are "Mary and Susan". However, after a closer look in the database, we can find out that there are more than 50 models that could satisfy *to some degree* this query if we didn't have the crisp thresholds. In a such conjunctive query, if one of the atoms of the query does not entails the KB we get an empty answer. If, for example, the model "Adriana", which satisfies the thin sentence, but is under 1cm only in the height sentence, is rejected. The second type of uncertainty is introduced when the query sentences are implemented with an equal degree of importance. It could happen, for example, that the advertisement company is more interested, for the model, to be tall than to be thin. This means, apart from limited query answers, that we cannot rank the answers of the query according to the user needs. If, for example, "Mary" is 185cm tall and 65 kilos and "Susan" is 185cm tall and 55 kilos and the degrees of importance of the atoms is 1 and 0.5 for the weight and the height respectively, then "Susan" should be ranked before "Mary". To conclude, in the above example, is clearly presented, the two kinds of uncertainty that exist real-life applications. Of course, this is not always the case. There are as many queries that do not incorporate uncertainty. The advantage of using fuzzy logic for managing these two kinds of uncertainty, is that the crisp case is implemented as a subcase of the fuzzy case, which means that we fuzzy logic does not replace the existed logic but extends it. In D2.5.1 we have proposed a way to manage imprecise and uncertain knowledge by extending DL based Semantic Web Languages (OWL) with fuzzy logic. In the next paragraph we sumurise this work.

The fuzzy DL is based on the definition of the *fuzzy interpretation*. A fuzzy interpretation $\mathcal{I}$ consists of a non empty set $\Delta^{\mathcal{I}}$ and the mapping functions:

$$C^{\mathcal{I}} : \Delta^{\mathcal{I}} \longrightarrow [0,1]$$

$$R^{\mathcal{I}} : \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \longrightarrow [0,1]$$

assigning fuzzy sets to concepts and roles, respectively. For example if $\alpha \in \Delta^{\mathcal{I}}$ then $A^{\mathcal{I}}(a)$ gives the degree that the object a belongs to the fuzzy concept A, i.e $A^{\mathcal{I}}(a) = 0.8$.

Table 5.1 summarises the syntax and the semantics of some constructors and terminological and assertional axioms. The first column provides the name of the constructor, the second its syntax and the third its semantics.

Table 5.1: Some Concept Constructors, Assertional and Terminological Axioms

Name	Syntax	Semantics $(a \in \Delta^{\mathcal{I}})$
Top	$\top$	$\top^{\mathcal{I}}(a) = 1$
Bottom	$\bot$	$\bot^{\mathcal{I}}(a) = 0$
Fuzzy Intersection	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}}(a) = t(C^{\mathcal{I}}(a), D^{\mathcal{I}}(a))$
Fuzzy Union	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}}(a) = u(C^{\mathcal{I}}(a), D^{\mathcal{I}}(a))$
Fuzzy negation	$\neg C$	$(\neg C)^{\mathcal{I}}(a) = c(C^{\mathcal{I}}(a))$
Fuzzy Value Restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}}(a) = \inf_{b \in \Delta^{\mathcal{I}}} w_t(R^{\mathcal{I}}(a,b), C^{\mathcal{I}}(b))$
Fuzzy existential quantifier	$\exists R.C$	$(\exists R.C)^{\mathcal{I}}(a) = \sup_{b \in \Delta^{\mathcal{I}}} t(R^{\mathcal{I}}(a,b), C^{\mathcal{I}}(b))$
Concept Inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}(\forall a \in \Delta^{\mathcal{I}} \mid C^{\mathcal{I}}(a) \leq D^{\mathcal{I}}(a))$
Role Inclusion	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$ $(\forall (a,b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(a,b) \leq S^{\mathcal{I}}(a,b))$
Concept Equality	$C \equiv D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$ $(\forall a \in \Delta^{\mathcal{I}} \mid C^{\mathcal{I}}(a) = D^{\mathcal{I}}(a))$
Role Equality	$R \equiv S$	$R^{\mathcal{I}} = S^{\mathcal{I}}(\forall (a,b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ $\mid R^{\mathcal{I}}(a,b) = S^{\mathcal{I}}(a,b))$
Concept Assertion	$C(a)$	$(C(a))^{\mathcal{I}}(a) = C^{\mathcal{I}}(a) > 0$
Role Assertion	$R(a,b)$	$(R(a,b))^{\mathcal{I}}(a,b) = R^{\mathcal{I}}(a,b) > 0$

The concepts and the roles in classical OWL are interpreted as crisp sets, i.e an individual either belongs to the set or not. However, many real-life concepts are vague in the sense that they do not have precisely defined membership criteria. In fuzzy OWL an individual belongs to a degree of confidence to the set (membership). This means that, for example, the individual ”Peter” might belong to the degree of confidence of“0.8” to the concept set“TallPerson”.

In fuzzy SWRL, a weight representing the degree of importance of the atoms of the body of the rule, is added. A rule now means that if the antecedent atoms A_1, A_2, A_n

are activated to the assertion degrees $a_1, a_2, a_n \in [0, 1]$, and have degrees of importance $b_1, b_2, b_n \in [0, 1]$ then the consequent hold to an assertion degree $c \in [0, 1]$ that can be computed from a_1, a_2, a_n and b_1, b_2, b_n with the aid of fuzzy operators. For example:

If(hasSmallHeight ?p ?w, “0.4”)∧(hasLargeWeight ?p ?r, “0.8”)→(fatPerson ?p)

If the assertion degrees of p, w to the relation $hasSmallHeight$ is $a_1 = 0.5(seefigure5.1)$ and the assertion degrees of p, r to the relation $hasLargeWeight$ is $a_2 = 0.9$ then the p must have assertion degree to the concept $fatPerson, c = 0.8$. The difference between the assertion degrees and the degrees of importance is that the assertion degrees show the membership values a_1, a_2, a_n of the variables included in the atoms A_1, A_2, A_n to the concepts or relations they belong to, and the degrees of importance show how important is each antecedent atom in order to detect the head atom.

The fuzzy extensions in DLs proposed in D2.5.1 and sumurised in the previous paragraphs, present a way to manage the two kinds of uncertainty. These extensions were based on the notion fuzzy assertion, and and not he notion of the degree of importance in the atoms of a rule-axiom that shows the wight of each atom for the activation of the head of the rule. A rule is distinguished from a query from the fact that a query uses more variable bindings and has only head atoms. A query may have zero or more answers, each of which provides bindings of URI references or literals to some of the variables in the query pattern such that the conjunction of the answer sentences, produced by applying the bindings to the query pattern and considering the remaining variables in the query pattern to be existentially quantified, is entailed by a KB called the *answer* KB. A fuzzy query is similar to the crisp query apart from the fact that the query answers may have a degrees of importance, and the conjunction of the query answers are fuzzy entailed by a fuzzy KB (a KB with fuzzy assertions). A crisp KB entails a query answer ψ

$$KB \models \psi,$$

iff every model of the KB also *satisfies* (is a model of) ψ. A fuzzy KB fuzzy entails a query answer ψ

$$KB_f \models_f \psi,$$

iff every model of the fuzzy KB satisfies to some degree $e \in (0, 1]$ ψ. Fuzzy entailment occurs when

$$C \sqcap \neg C \neq \bot,$$

which is the case for fuzzy concepts, or when

$$C \equiv D,$$

to some degree. In the D2.5.1 was defined only the notion of fuzzy assertion and not the notions of fuzzy equality and fuzzy entailment, since it was difficult to understand where these extensions are useful for.

5.4 Fuzzy OWL-QL

As previously described, in classical OWL-QL each binding in a query answer is a URI references or a literal that either explicitly occurs as a term in the answer KB or is a term in OWL. That is, OWL-QL is designed for answering queries of the form "What URIs references and literals from the answer KB and OWL denote objects that make the query pattern true?". A variable that has a binding in a query answer is identified in the query answer. OWL-QL supports existentially quantified answers by enabling the client to designate some of the query variables for which answers will be accepted with or without variables. That is, each variable that occurs in a OWL-QL query is considered to be a *must-bind, a may-bind variable or a don't bind variable*. Answers are required to provide bindings for all the must-bind variables, may provide bindings for any of the may-bind variables, and are not to provide bindings for any of the don't-bind variables.

In fuzzy OWL-QL each binding in a query answer is, as in the classical OWL-QL, a URI reference or a literal. The difference of fuzzy OWL-QL from the classical one, is that is used a fuzzy KB (fuzzy Abox) to retrieve the answers, and therefore we can use fuzzy concepts and fuzzy relations in the queries, such as "tallPerson, fatPerson, hasMediumHeight" together with assertion degrees representing the membership value of the object to the corresponding concepts and relations. In addition to the fuzzy assertion, the user may assign degrees of importance to the query sentences denoting the influence that a specific sentence must have in the query answer. For example, in the query *"List all the models that hasLargeHeight and hasMidleAge"* the user might be more interested in the height sentence than in the age sentence. In this case, the user can assign degree of importance 1 to the height sentence and 0.5 to the age sentence. In this way the query engine is enabled to produce ranked answers according to the user needs. Finally, as described in the previous section, a KB must entail all the query sentences, since they are conjunctive. In the fuzzy case, the decision whether a KB entails a query sentence is not crisp (yes or no). A fuzzy KB fuzzy entails a query sentence to a degree $e \in (0, 1]$.

As in classical OWL-QL, in the a fuzzy OWL-QL query-answering dialogue is initiated by a client sending a query to an OWL-QL server. A fuzzy OWL-QL query is an object necessarily containing a query pattern that specifies a collection of fuzzy OWL sentences in which some URI references are considered to be variables. The example presented in the previous section has the form in fuzzy OWL-QL:

Query("List all the female models, which are
tall and thin")

The query pattern is as follows:

Query Pattern {(hasSex ?p ?a)(type ?a female)∧(hasLargeHeight ?p ?c <0.8>)
(type ?c LargeHeight)∧(hasMediumWeight ?p ?d <0.5>)(type ?d MediumWeight)}
Must-Bind Variables List: (?p)
Answer pattern {(hasSex ?p "Female" ∧ hasLargeHeight ?p largeHeight
∧ hasMediumWeight ?p mediumWeight)}
Answer1: ("Mary is a female model who is 185cm tall (largeHeight=0.8)
and is 65 kilos (mediumWeight=0.4)
Answer2: ("Susan is a female model who is 175cm (LargeHeight=0.6)
and is 50 kilos (mediumWeight=0.9)
Answer3: ("Helen is a female model who is 170cm (LargeHeight=0.5)
and is 50 kilos (mediumWeight=0.9)

In this example we used the fuzzy relations "hasLargeHeight, hasMediumWeight" and the fuzzy concepts "LargeHeight, MediumWeight" to manage the uncertainty introduced by the concepts "Thin, Tall". In this way, a person who is 183cm tall hasLargeHeight=0.65 and hasMediumHeight=0.3, as depicted in figure 5.1. Also we have assigned degrees of importance, 0.8 for the height sentence and 0.5 for the weight sentence. That is, that the user is more interested for the model to be tall that thin. The answers are, in the fuzzy case, ranked. The ranking value $R \in [0, 1]$ is the calculated as:

$$R = Inf \omega_t[\mathbf{K, A}],$$

where *A* correspond to the fuzzy relation that has the assertion degrees of a query sentences, and *K* correspond to the fuzzy relation that has the degrees of importance and ω_t is a fuzzy implication (see D2.5.1). In the above example, the rank R for Answer1 can be computed as:

$$K = [1.0 \quad 0.8 \quad 0.5],\ A_{mary} = \begin{bmatrix} 1.0 \\ 0.8 \\ 0.4 \end{bmatrix}$$

$$R_{mary} = 0.8,$$

where ω_t is the implication of the Product t-norm. Accordingly are computed $R_{susan} = 0.6$ and $R_{helen} = 0.5$. In this way, we do not restrict the query answer with crisp thresholds and thus i)we get more answers and ii)the answers are ranked.

Classical OWL-QL facilitates the representation of "If Then" queries by enabling a query to optionally include a *query premise* that is an OWL KB or a KB reference. When a premise is included in a query, it is considered to be included in the answer KB. Fuzzy OWL-QL can have a *fuzzy query premise*, which means that we can assign degrees of importance to the query premise, and influence the ranking of the answers

Query('If C1 is LongHair and C1 is the typeOfHair of W1,
Then what is height of W1")
Premise{(type C1 LongHair)(hasTypeOfHair C1 W1 <0.8>)}
Query Pattern: {(hasLargeHeight W1 ?x <0.6>)}
Must-Bind Variables List: (?x)
...

5.5 Use case

In order to understand the need of the proposed fuzzy logic extensions, we will demonstrate a use case. Let us consider a casting company that has a large multimedia database consisting of visual and textual information about person-models. This company has a user interface for inserting the textual and visual characteristics of the models as instances of a predefined ontology. It also provides a query engine to search for models with special characteristics depending the context and the subject of the advertisement. The visual characteristics of a model consist of the images of the models together with some low-level information. Low-level information consist of the visual descriptors of an image(MPEG-7 visual descriptors), which are used for visual queries. Visual queries are included in the sense that a user can provide an image of a model and query for models with similar low-level characteristics (colour, shape, etc.). In the textual case a user can query the database providing high-level information about the models (such as the name, the height, the type of the hair etc.). The textual characteristics are inserted by a domain-expert manually in the database (KB), However, the visual characteristics are inserted automatically using a visual descriptor extraction algorithm, which automatically analyses the inserted image and stores as instances the values of the detected visual descriptors in a visual descriptor ontology. The same algorithm analyse the visual query image. The extracted visual descriptors are then form a query pattern, which is true if it is entailed by the KB, as in the textual case.

In the following paragraphs we provide a sample of the Tbox, the Abox, a couple of rules and a diagram showing how the assertion degrees are calculated, of the textual information of the models.

Tbox

$Woman \equiv Person \sqcap Female$
$Man \equiv Person \sqcap Male$
$CastingPerson \equiv Person \sqcap \forall HasPersonalInformation.PersonalInformation$
$\sqcap \forall HasMeasurements.Measurements \sqcap \forall HasTypeOfHair.Hair$
$PersonalInformation \equiv \forall hasName.Name \sqcap \forall hasLastName.Name \sqcap$
$\forall hasAge.Age$
$\sqcap \forall hasDOB.DOB \sqcap \forall hasAddress.Adress \sqcap \forall hasMobilenumber.Number$
$Measurements \equiv \forall hasHeight \sqcap \forall hasWeight.Weight$
$\sqcap \forall hasShoeSize.Size$
$Hair \equiv \exists hasHairQuality.HairQuality \sqcap$
$\exists hasHairLength.HairLength \sqcap \exists hasHairColour.HairColour$

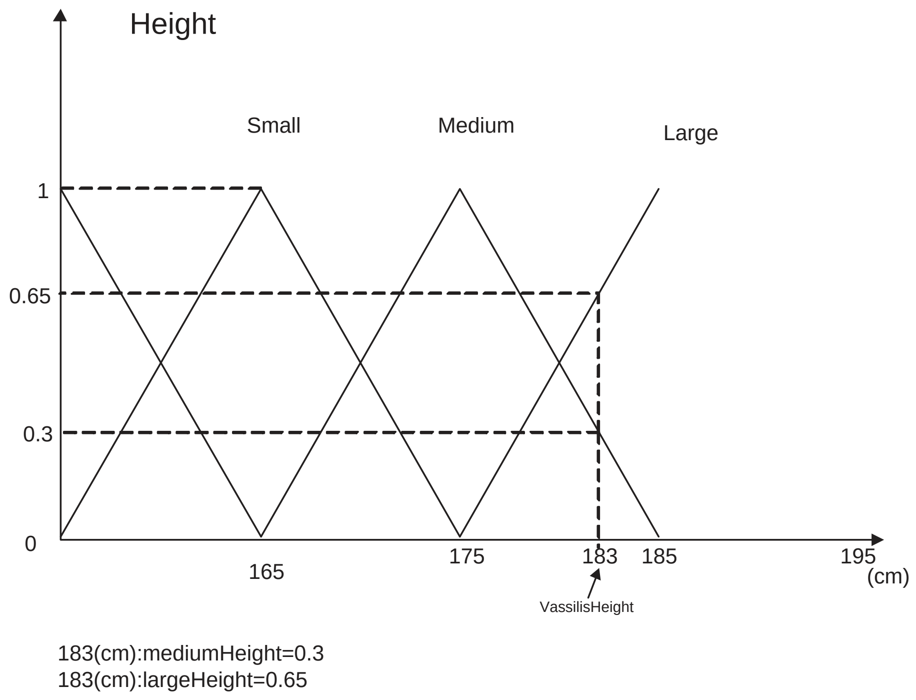


Figure 5.1: The fuzzy partition of Height

Entry: no1		
Personal Information		
Name: Vassilis	*LastName:* Tzouvaras	*Age:* 29
Address: Hatzi 7	*Mobile:* 6937295722	*D.O.B.:* 07.08.75
Measurements		
Height: 183cm	*Weight:*90	*ShoeSize:* 44
Hair		
Quality: good	*Length:* short	*Style:* frizy

Abox:

$\{\langle no1 : CastingPerson = 1\rangle, \langle (no1, Vassilis) : hasName = 1\rangle, \langle (no1, Tzouvaras) : hasLastName = 1\rangle, \langle (no1, 29) : hasAge = 1\rangle, \langle (no1, Hatzi7) : hasAddress = 1\rangle, \langle (no1, 6937295722) : hasMobilenumber = 1\rangle, \langle (no1, 183cm) : hasMediumHeight = .3\rangle, \langle (no1, 183) : haslargeHeight = .65\rangle, \langle (no1, 34) : mediumWaste = 0.7\rangle, \langle (no1, 34) : hasLargeWaste = 0.3\rangle, \langle (no1, 44) : hasMediumShoeSize = .9\rangle, \langle (no1, 44 : hasLargeShoeSize = 0.1\rangle, \langle (no1, Long) : hasLongHair = 0.3\rangle, \langle (no1, good) : hasQualityHair = 0.8\rangle, \langle (no1, frizy) : hasTypeOfHair = 1\}$

Rule 1: $IF \quad hasMediumWeight \quad AND \quad hasLargeHeight(a, b) \quad THEN \quad ThinPerson(a)$
Rule 2: $If \quad HasSmallHeight(a, c) \quad AND \quad HasLargeWeight(a, b) \quad THEN \quad FatPerson(a)$

Chapter 6

The Instance Store

6.1 Introduction

One of the main features of the W3C's OWL ontology language [DCv$^+$02] is that there is a direct correspondence between (two of the three "species" of) OWL and *Description Logics* (*DL*s) [HPS03]. This means that DL reasoners can be used to reason about OWL ontologies, and in particular to answer both class based queries (e.g., asking if the class "Semantic Web researcher" is a subclass of the class "Computer Scientist") and instance retrieval queries (e.g., a query that asks for all the individuals in the ontology that are instances of the class "person who works at a university whose research interests include Semantic Web and Description Logics").

Unfortunately, while existing techniques for *TBox* reasoning (i.e., reasoning about the concepts in an ontology) seem able to cope with real world ontologies [Hor98, HM01a], it is not clear if existing techniques for *ABox* reasoning (i.e., reasoning about the individuals in an ontology) will be able to cope with realistic sets of instance data. This difficulty arises not so much from the computational complexity of ABox reasoning, but from the fact that the number of individuals (e.g., annotations) might be extremely large.

The so called *Instance Store* is a system that addresses this problem by using a hybrid DL/Database architecture to answer queries against ontologies containing large numbers of individuals. The idea behind the Instance Store is to provide efficient (but still sound and complete) query answering by maximising the use of the Database and minimising calls to the DL reasoner.

A prototype of the Instance Store has been implemented by researchers in the Information Management Group at the University of Manchester. Currently the prototype can only deal with a *role-free* ontology, i.e., an ontology that does not contain any axioms asserting role relationships (properties) between pairs of individuals, but work is underway to extend the Instance Store to deal with arbitrary ontologies. In this chapter we will describe the functioning of the existing Instance Store, illustrate its performance with some experimental results, and outline how the Instance Store design will be extended to deal with arbitrary ontologies.

The remainder of the chapter is structured as follows: in Section 6.2 we motivate the design of the Instance Store; in Section 6.3 we give some details of Description Logics that will be needed in the later sections; in Section 6.4 we describe the architecture and implementation of the role-free instance store; in Section 6.5 we present the results of an empirical evaluation that we have carried out using the role-free instance store; in Section 6.6 we describe how the Instance Store approach will be extended to deal with arbitrary ontologies; and in Section 6.8 we conclude with a discussion.

6.2 Background and Motivation

Although the restrictions of the existing Instance Store may seem a rather severe, the functionality provided turns out to be precisely what is required by many applications, and in particular by applications where ontology based terms are used to describe/annotate and retrieve large numbers of objects. Examples include the use of ontology based vocabulary to describe documents in "publish and subscribe" applications [UCD$^+$03], to annotate data in bioinformatics applications [GO] and to annotate web resources such as web pages [DEG$^+$03] or web service descriptions [LH03] in Semantic Web applications. Indeed, we have successfully applied the Instance Store to perform web service discovery [CDT04], to search over the gene ontology [GO] and its associated instances (see below), and in an application to guide gene annotation [BTMS04].

Using a database in order to support (a restricted form of) ABox reasoning is certainly not new (see Section 6.7 for a discussion of related work), but to the best of our knowledge the Instance Store is the first such system that is general purpose (i.e., can deal with any ontology without customising the database schema), provides sound and complete reasoning, and places no a-priori restriction on the size of the ontology.

In order to evaluate the Instance Store design, and in particular its ability to provide scalable performance for instance retrieval queries, we have performed a number of experiments using the Instance Store to search over a large (50,000 concept) gene ontology and its associated very large number (up to 650,000) of individuals – instances of concept descriptions formed using terms from the ontology. In the absence of other specialised reasoners we have compared the performance of the Instance Store with that of RACER [HM01b] (the only publicly available DL system that supports full ABox reasoning for an expressive DL) and of FaCT [Hor98] (using TBox reasoning to simulate reasoning with a role-free ABox).

6.3 Description Logics

Description Logics [BCM$^+$03] are a family of knowledge representation formalisms evolved from early *frame systems* [Min75] and *semantic networks* [Qui68]. DLs use an object oriented modelling paradigm, describing the world in terms of individuals, concepts (classes) and roles (relationships); they are distinguished from their ancestors by having a precise *semantics* which enables the description and justification of automated deduction processes.

The *semantics* of a DL is given in terms of interpretations. An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation) and an interpretation function $\cdot^{\mathcal{I}}$ which maps every individual to an element of $\Delta^{\mathcal{I}}$, every concept to a subset of $\Delta^{\mathcal{I}}$, and every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Concepts may be either atomic (i.e., a concept name) or concept expressions formed using the operators provided by the DL. The interpretations of concept expressions must obey appropriate semantic conditions, e.g., the interpretation of the conjunction $C \sqcap D$ of two concepts C and D must be equal to the intersection of the interpretations of the individual concepts, i.e., $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$. (See, e.g., [BCM$^+$03] for full details.)

A DL knowledge base (KB) is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox. A TBox is a set of axioms of the form $C \sqsubseteq D$, where C and D are concepts; an ABox is a set of axioms of the form $x : C$ or $\langle x, y \rangle : R$, where x, y are individuals, C is a concept and R is a role. An interpretation $\mathcal{I}$ satisfies a TBox axiom $C \sqsubseteq D$ when $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and it satisfies ABox axioms $x : C$ and $\langle x, y \rangle : R$ when $x^{\mathcal{I}} \in C^{\mathcal{I}}$ and $\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$ respectively. An interpretation $\mathcal{I}$ satisfies a TBox $\mathcal{T}$ (ABox $\mathcal{A}$) when it satisfies all of the axioms in $\mathcal{T}$ ($\mathcal{A}$); such an interpretation is called a model of $\mathcal{T}$ ($\mathcal{A}$). An interpretation is a model of a KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ when it is a model of both $\mathcal{T}$ and $\mathcal{A}$.

Given a KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, basic reasoning tasks include:

Satisfiability: a concept C is satisfiable w.r.t. $\mathcal{T}$ ($\mathcal{K}$) iff there exists some model $\mathcal{I}$ of $\mathcal{T}$ ($\mathcal{K}$) s.t. $C^{\mathcal{I}} \neq \emptyset$.

Subsumption: a concept C is subsumed by a concept D w.r.t. $\mathcal{T}$ ($\mathcal{K}$) iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{T}$ ($\mathcal{K}$); we will write this as $\mathcal{T} \models C \sqsubseteq D$ ($\mathcal{K} \models C \sqsubseteq D$).

Instantiation: an individual x is an instance of a concept C w.r.t. $\mathcal{K}$ iff $x^{\mathcal{I}} \in C^{\mathcal{I}}$ in every model of $\mathcal{K}$; we will write this as $\mathcal{K} \models x : C$.

Other reasoning tasks such as *Classification* (computing the subsumption partial ordering, or *hierarchy*, of the atomic concepts in $\mathcal{T}$) and *Retrieval* (computing the individuals in $\mathcal{A}$ that instantiate a given concept) can be reduced to subsumption and instantiation respectively. *Realisation*, the task of computing the most specific (w.r.t. subsumption) atomic concepts in $\mathcal{T}$ that are instantiated by a given individual, can be reduced to a combination of retrieval and classification, i.e., for an individual x and an atomic concept C in $\mathcal{T}$, C

Description	Syntax	Semantics
atomic concept name	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
arbitrary negation	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
existential restriction	$\exists R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
universal restriction	$\forall R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$

Table 6.1: Syntax and Semantics of $\mathcal{SHF}$ concept expressions

realises x iff x is an instance of C and there is no atomic concept $D \neq C$ in $\mathcal{T}$ such that x is an instance of D and C subsumes D. Finally, two concepts C and D are *equivalent*, written $C \equiv D$, iff $C \sqsubseteq D$ and $D \sqsubseteq C$.

6.3.1 The Description Logic $\mathcal{SHF}$

We will be particularly interested in the $\mathcal{SHF}$ Description Logic as this is the logic implemented in the Instance Store. $\mathcal{SHF}$ is an extension of the basic DL $\mathcal{AL}$ [SSS91] to include *negation* of arbitrary concepts, *transitive roles*, *role hierarchy* and *functional roles*. Given a set of concept names (CN) and a set of role names (RN), *concept expressions* in $\mathcal{SHF}$ are formed according to the following syntax rules:

$$C, D \rightarrow \top \mid \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \forall R.C \mid \exists R.C$$

where A is a concept name, C and D are concept expressions, and R is a role name.

In addition we assume that the set $F \subseteq RN$ of *functional* roles and the set $R_+ \subseteq RN$ of *transitive* roles are disjoint, i.e., $F \cap R_+ = \emptyset$. Moreover, we impose the limitation that there is no role P, Q such that $P \in R_+$, $Q \in F$ and $P \sqsubseteq Q$. The semantics of $\mathcal{SHF}$ concepts is shown in Table 6.1

In the most general case, $\mathcal{SHF}$ TBox axioms have the form:

$$C \sqsubseteq D,\ R \sqsubseteq S \mid C \equiv D,\ R \equiv S$$

where C,D are concept expressions and R,S are role names. Axioms of the first kind are called *inclusions*, while axioms of the second kind are called *equalities*; an equality can be seen as an abreviation for a symetrical pair of inclusion axioms, i.e., $C \equiv D$ is an abreviation for $C \sqsubseteq D$ and $D \sqsubseteq C$.

Since role inclusion axioms and equality axioms contain role names only, a taxonomy of role names can be built based on the inclusion and equality relations among the set of role axioms, and a relation $\preceq$ can be defined as a *partial order* on the transitive closure of $\{R \sqsubseteq S \mid R, S \in RN\} \cup \{R \equiv S \mid R, S \in RN\} \subseteq \mathcal{T}$ to represent the role taxonomy.

6.3.2 The Instance Store Notation

We now introduce some new notation used, for convenience, in this paper. For a TBox $\mathcal{T}$, an ABox $\mathcal{A}$, and a concept C:

- $C\downarrow_{\mathcal{T}}$ for the set of atomic concepts in $\mathcal{T}$ subsumed by C; these are the equivalents and descendants of C in $\mathcal{T}$.
- $\lceil C \rceil_{\mathcal{T}}$ for the set of most specific atomic concepts in $\mathcal{T}$ subsuming C; if C is itself an atomic concept in $\mathcal{T}$ then clearly $\lceil C \rceil_{\mathcal{T}} = \{C\}$.

6.4 The Role-Free Instance Store

An ABox $\mathcal{A}$ is role-free if it contains only axioms of the form $x : C$. We can assume, without loss of generality, that there is exactly one such axiom for each individual as $x : C \sqcup \neg C$ holds in all interpretations, and two axioms $x : C$ and $x : D$ are equivalent to a single axiom $x : (C \sqcap D)$. It is well known that, for a role-free ABox, instantiation can be reduced to TBox subsumption [Hol96, Tes97]; i.e., if $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, and $\mathcal{A}$ is role-free, then $\mathcal{K} \models x : D$ iff $x : C \in \mathcal{A}$ and $\mathcal{T} \models C \sqsubseteq D$. Similarly, if $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ and $\mathcal{A}$ is a role-free ABox, then the instances of a concept D could be retrieved simply by testing for each individual x in $\mathcal{A}$ if $\mathcal{K} \models x : D$. However, this would clearly be very inefficient if $\mathcal{A}$ contained a large number of individuals.

An alternative approach is to add a new axiom $C_x \sqsubseteq D$ to $\mathcal{T}$ for each axiom $x : D$ in $\mathcal{A}$, where C_x is a new atomic concept; we will call such concepts *pseudo-individuals*. Classifying the resulting TBox is equivalent to performing a complete realisation of the ABox: the most specific atomic concepts that an individual x is an instance of are the most specific atomic concepts that subsume C_x and that are not themselves pseudo-individuals. Moreover, the instances of a concept D can be retrieved by computing the set of pseudo-individuals that are subsumed by D.

The problem with this latter approach is that the number of pseudo-individuals added to the TBox is equal to the number of individuals in the ABox, and if this number is very large, then TBox reasoning may become inefficient or even break down completely (e.g., due to resource limits).

The basic idea behind the Instance Store is to overcome this problem by using a DL reasoner to classify the TBox and a database to store the ABox, with the database also being used to store a complete realisation of the ABox, i.e., for each individual x, the concepts that x realises (the most specific atomic concepts that x instantiates). The realisation of each individual is computed using the DL (TBox) reasoner when an axiom of the form $x : C$ is added to the Instance Store ABox.

A retrieval query Q to the Instance Store (i.e., computing the set of individuals that instantiate a concept Q) can be answered using a combination of database queries and TBox reasoning. Given an Instance Store containing a KB $\langle \mathcal{T}, \mathcal{A} \rangle$ and a query concept Q, retrieval involves the computation of the following sets of individuals for which we introduce a special notation:

- I_1 denotes the set of individuals in $\mathcal{A}$ that realise *some* concept in $Q\downarrow_{\mathcal{T}}$;
- I_2 denotes the set of individuals in $\mathcal{A}$ that realise *every* concept in $\lceil Q \rceil_{\mathcal{T}}$.

The Instance Store algorithm to retrieve the instances of Q can be then described as follows:

1. use the DL reasoner to compute $Q\downarrow_{\mathcal{T}}$;
2. use the database to find the set of individuals I_1;
3. use the reasoner to check whether Q is equivalent to any atomic concept in $\mathcal{T}$; if that is the case then simply return I_1 and *terminate*;
4. otherwise, use the reasoner to compute $\lceil Q \rceil_{\mathcal{T}}$;
5. use the database to compute I_2;
6. use the reasoner and the database to compute I_3, the set of individuals $x \in I_2$ such that $x : C$ is an axiom in $\mathcal{A}$ and C is subsumed by Q;
7. return $I_1 \cup I_3$ and *terminate*.

Proposition 1 *The above procedure is sound and complete for retrieval, i.e., given a concept Q, it returns all and only individuals in $\mathcal{A}$ that are instances of Q.*

The above is easily proved using the fact that we assume, without loss of generality, that for each individual there is only one axiom associated to it.

6.4.1 An Optimised Instance Store

In practice, several refinements to the above procedure are used to improve the performance of the Instance Store. In the first place, as it is potentially costly, we should try to minimise the DL reasoning required in order to compute realisations (when instance axioms are added to the ABox) and to check if individuals in I_1 are instances of the query concept (when answering a query).

One way to (possibly) reduce the need for DL reasoning is to avoid repeating computations for "equivalent" individuals, e.g., individuals x_1, x_2 where $x_1 : C_1$ and $x_2 : C_2$ are ABox axioms, and C_1 is equivalent to C_2. Since checking for semantic equivalence between two concepts would require DL reasoning (which we are trying to avoid), the optimised Instance Store only checks for syntactic equality using a database lookup. (The chances of detecting equivalence via syntactic checks could be increased by transforming concepts into a syntactic normal form, as is done by optimised DL reasoners [Hor03], but this additional refinement has not yet been implemented in the Instance Store.) Individuals are grouped into equivalence sets, where each individual in the set is asserted to be an instance of a syntactically identical concept, and only one representative of the set is added to the Instance Store ABox as an instance of the relevant concept. When answering queries, each individual in the answer is replaced by its equivalence set.

Similarly, we can avoid repeated computations of sub and super-concepts for the same concept (e.g., when repeating a query) by caching the results of such computations in the database.

DL reasoning can also be avoided when the query concept Q is not equivalent to any atomic concept in $\mathcal{T}$, but when Q *is* equivalent to the intersection of the concepts in $\lceil Q \rceil_{\mathcal{T}}$, i.e., where

$$Q \equiv \bigsqcap_{C \in \lceil Q \rceil_{\mathcal{T}}} C.$$

In this case it is not necessary to compute I_3, as the answer to the query is clearly I_2, i.e., the set of individuals in $\mathcal{A}$ that realise *every* concept in $\lceil Q \rceil_{\mathcal{T}}$.

Finally, the number and complexity of database queries also has a significant impact on the performance of the Instance Store. In particular, the computation of I_1 can be costly as $Q \downarrow_{\mathcal{T}}$ may be very large. One way to reduce this complexity is to store not only the most specific concepts instantiated by each individual, but to store *every* concept instantiated by each individual. As most concept hierarchies are relatively shallow, this does not increase the storage requirement too much, and it greatly simplifies the computation of I_1: it is only necessary to compute the (normally) much smaller set of most general concepts subsumed by Q and to query the database for individuals that instantiate some member of such set. On the other hand, the computation of I_2 is slightly more complicated as I_1 must be subtracted from the set of individuals that instantiate every concept in $\lceil Q \rceil_{\mathcal{T}}$. Empirically, however, the savings when computing I_1 seems to far outweigh the extra cost of computing I_2.

6.4.2 Implementation

We have implemented the Instance Store using a component based architecture that is able to exploit existing DL reasoners and databases. The core component is a Java application [isw] talking to a DL reasoner via the DIG interface [Bec03b] and to a relational database via JDBC. We have tested it with FaCT [Hor98] and RACER reasoners and MySQL, Hypersonic, and Oracle databases.

```
initialise(Reasoner reasoner,
            Database db, TBox t)
addAssertion(Individual i, Concept C)
retract(Individual i)
retrieve(Concept Q): Set⟨Individual⟩
```

Figure 6.1: Instance Store basic functionality

The basic functionality of the Instance Store is illustrated by Figure 6.1. The four basic operations are `initialise`, which loads a TBox into the DL reasoner, classifies the TBox and establishes a connection to the database; `addAssertion`, which adds an axiom $i : D$ to the Instance Store; `retract`, which removes any axiom of the form $i : C$ (for some concept C) from the Instance Store; and `retrieve`, which returns the set of individuals that instantiate a query concept Q. As the Instance Store ABox can only contain one axiom for each individual, asserting $i : D$ when $i : C$ is already in the ABox is equivalent to first removing i and then asserting $i : (C \sqcap D)$.

In the current implementation, we make the simplifying assumption that the TBox itself does not change. Extending the implementation to deal with monotonic extensions of the TBox would be relatively straightforward, but deleting information from the TBox might require (in the worst case) all realisations to be recomputed.

6.5 Empirical Evaluation

To illustrate the scalability and performance of the Instance Store we describe the tests we have performed using the gene ontology and its associated instance data. We also illustrate how this compares with existing non-specialised ABox reasoning techniques by describing the same tests performed using RACER and FaCT (the latter using the pseudo-individual approach discussed in Section 6.4).

The gene ontology (*GO*) itself, an ontology describing terms used in gene products and developed by the Gene Ontology Consortium [The00], is little more than three taxonomies of gene terms, with a single role being used to add "part-of" relationships. However, the ontology is large (47,012 atomic concepts) and the instance data, obtained by mining the GO database [Go 03] of gene products, consists of 653,762 individual axioms involving 48,581 distinct complex DL expressions using three more roles.

The retrieval performance tests use two sets of queries. The first set was formulated with the help of domain experts and consists of five realistic queries that might be posed by a biologist. The second set consists of six artificial queries designed to test the effect on query answering performance of factors such as the number of individuals in the answer, whether the query concept is equivalent to an atomic concept (if so, then the answer can be returned without computing I_3), and the number of candidate individuals in I_2 for which DL reasoning is required in order to determine if they form part of the answer. The characteristics of the various queries with respect to these factors is shown in Tables 6.2 and 6.3.

Table 6.2: Query characteristics (realistic queries)

Query	Equivalent to Atomic Concept	No. of Instances in Answer	No. of "candidates" in I_2
Q1	Yes	2,641	n/a
Q2	No	0	284
Q3	No	3	284
Q4	Yes	7,728	n/a
Q5	Yes	25	n/a

Table 6.3: Query characteristics (artificial queries)

Query	Equivalent to Atomic Concept	No. of Instances in Answer	No. of "candidates" in I_2
Q6	No	13,449	551
Q7	No	11,820	116
Q8	No	12	603
Q9	No	19	19
Q10	Yes	4,543	n/a
Q11	Yes	1	n/a

The tests were performed using two machines *M1* (Linux, 850MHz Intel Pentium III, 256Mb RAM) and *M2* (Windows 2000, 2.5GHz Intel Pentium IV processor, 512Mb RAM). For the Instance Store we run version 1.2 on $M1$ with a MySQL-4.0.12 database on *M1* and connecting to a FaCT-2.34.13 reasoner running remotely on *M2*. For the tests on RACER we run RACER-1.7.7 and for the pseudo-individual tests we used FaCT-2.34.13, both on *M2*.

6.5.1 Loading and Querying Tests

In these tests, we compared the performance of the Instance Store with that of RACER using the GO TBox and differently sized and randomly selected subsets of the GO ABox. The Instance Store was first initialised with the GO TBox (it took FaCT approximately 1,020 CPU seconds to classify the TBox), then, for each ABox, we measured the time (in CPU seconds) taken to load the ABox into the Instance Store and the time taken to answer each of the queries.

In the case of RACER, we carried out the same tests in two different ways. In both cases we first initialised RACER with the GO TBox (it took RACER approximately 1,620

CPU seconds to classify the TBox), then loaded the ABox. In the first form of the test, we then used the *realize-abox* function to force RACER to compute a complete realisation of the ABox before answering any queries; this is roughly equivalent to the Instance Store, which effectively computes a complete realisation while loading the ABox. We timed how long RACER took to to realise the ABox and, if the realisation was successfully completed, how long it took to answer each of the queries. In the second form of the test, we simply timed how long it took RACER to answer each of the queries without first forcing it to realise the ABox.

Table 6.4: The Instance Store and RACER load and realise times (CPU seconds)

Number of	Distinct	Load & Realise (s)	
Individuals	Descriptions	The Instance Store	RACER
200	155	189	180
500	330	405	3,420
1,000	591	804	22,320
2,000	1,017	1,395	fault
5,000	2,024	2,906	fault
10,000	3,299	5,988	fault
20,000	5,364	11,057	fault
50,000	9,760	21,579	fault
100,000	15,147	33,456	fault
200,000	23,387	56,613	fault
400,000	35,800	96,503	fault
653,762	48,581	140,623	fault

The times taken by the Instance Store and by RACER to load and realise the various ABoxes are shown in Table 6.4. The time take by the Instance Store to load the ABoxes increases more slowly than their size: for ABox size 200, the Instance Store takes about 1s to add each individual axiom; by the time the ABox size has reached 400,000 this has fallen to approximately 0.25s per axiom. In view of the equivalent individuals optimisation employed by the Instance Store, however, it may be more relevant to consider the time taken per distinct description: this increases from about 1s per description for the size 200 ABox (which contains 155 distinct descriptions) to approximately 3s per description for the size 653,762 ABox (which contains 48,581 distinct descriptions).

The time taken by RACER to realise the smallest ABox is roughly the same as that taken by the Instance Store. As the ABox size grows, however, the time taken by RACER increases rapidly, and at ABox size 1,000 it is already taking approximately 22s per axiom. For larger ABoxes, RACER broke down due to a resource allocation error in the underlying Lisp system.

While the times taken by the Instance Store to load (and, in effect, to realise) the larger ABoxes are quite significant, it is able to deal with the 653,762 axiom ABox, whereas

RACER failed to realise a 2,000 axiom ABox. Moreover, the load/realise operation only needs to be performed once—an added advantage of the Instance Store is that the database provides for persistence of the realised ABox. Depending on the nature of the application, it may also be more normal for instance data to be added to the Instance Store over time rather than all at once as in our experiment.

Tables 6.5 and 6.6 give the results for the Instance Store when answering each of the five realistic queries and six artificial queries described in Tables 6.2 and 6.3. In addition to the time taken (in CPU seconds) to answer the queries, the number of candidate individuals in I_2 is also given as this is one of the major factors in determining the "hardness" of the query: for each individual in I_2, the Instance Store must use the DL reasoner to determine if the individual instantiates the query concept. The time taken to answer these queries is also plotted against the size of the ABox in Figure 6.2; note the logarithmic scales on both axes.

Table 6.5: The Instance Store realistic query times (CPU seconds) and cardinality of I_2

Number of	Q1		Q2		Q3		Q4		Q5	
Individuals	IS	$\|I_2\|$	IS	$\|I_2\|$	IS	$\|I_2\|$	IS	$\|I_2\|$	IS	$\|I_2\|$
200	8.6	n/a	1.4	1	1.9	2	4.2	n/a	1.0	n/a
500	8.6	n/a	1.9	2	2.0	2	4.2	n/a	1.1	n/a
1,000	8.8	n/a	2.1	3	2.1	3	4.5	n/a	1.1	n/a
2,000	8.8	n/a	3.7	3	2.1	3	4.7	n/a	1.1	n/a
5,000	8.8	n/a	4.0	5	2.2	5	4.8	n/a	1.2	n/a
10,000	9.2	n/a	4.3	6	3.1	6	4.9	n/a	1.2	n/a
20,000	9.7	n/a	4.8	13	4.5	13	5.5	n/a	1.1	n/a
50,000	10.1	n/a	7.1	20	6.9	20	6.6	n/a	1.2	n/a
100,000	11.4	n/a	9.6	34	9.5	34	8.2	n/a	1.2	n/a
200,000	11.5	n/a	20.2	85	19.2	85	10.9	n/a	1.2	n/a
400,000	15.0	n/a	33.8	151	33.9	151	17.4	n/a	1.2	n/a
653,762	23.0	n/a	55.4	241	55.1	241	35.3	n/a	1.3	n/a

As can be seen, the time taken to answer queries becomes quite large when the number of individuals in I_2 is large. In these cases, the time taken to check if these individuals instantiate the query concept (roughly 0.2s per individual) dominates other factors. The number of "distinct" individuals in the answer also has a significant impact on query answering performance: when there are many such individuals, the database query required in order to compute the complete answer set (i.e., retrieving the union of the equivalence sets of these individuals) can be quite time consuming. In the case of Q9 with the largest ABox, for example, the relevant database query takes 19.5s (out of a total of 25.7s).

When the query concept is determined to be semantically equivalent to an atomic concept in the TBox, as is the case with Q1, Q4, Q5, Q10 and Q11, then no further DL

Table 6.6: The Instance Store artificial query times (CPU seconds) and cardinality of I_2

Number of Individuals	Q6		Q7		Q8		Q9		Q10		Q11	
	IS	$\|I_2\|$	IS	$\|I_2\|$	IS	$\|I_2\|$	IS	$\|I_2\|$	IS	$\|I_2\|$	IS	$\|I_2\|$
200	2.4	2	2.1	3	1.6	1	2.0	1	1.7	n/a	1.8	n/a
500	2.6	4	2.1	3	2.0	3	2.1	1	1.7	n/a	1.8	n/a
1,000	3.0	8	2.3	3	2.0	3	2.1	1	2.2	n/a	1.9	n/a
2,000	3.4	9	2.4	4	2.2	4	2.3	1	1.8	n/a	1.7	n/a
5,000	4.5	15	3.0	7	2.9	9	2.5	1	1.9	n/a	1.9	n/a
10,000	7.1	32	4.2	13	6.0	21	2.5	1	1.8	n/a	1.8	n/a
20,000	10.9	58	5.4	19	11.5	38	2.9	1	2.1	n/a	1.7	n/a
50,000	17.4	101	7.3	31	23.8	81	3.3	1	1.9	n/a	1.8	n/a
100,000	27.3	164	8.9	45	31.9	147	5.2	2	1.7	n/a	1.8	n/a
200,000	44.4	273	13.1	64	40.1	268	7.9	7	1.9	n/a	1.8	n/a
400,000	70.9	416	16.4	85	68.1	430	15.8	11	1.9	n/a	1.9	n/a
653,762	111.8	551	22.1	116	104.0	603	25.7	19	1.9	n/a	1.9	n/a

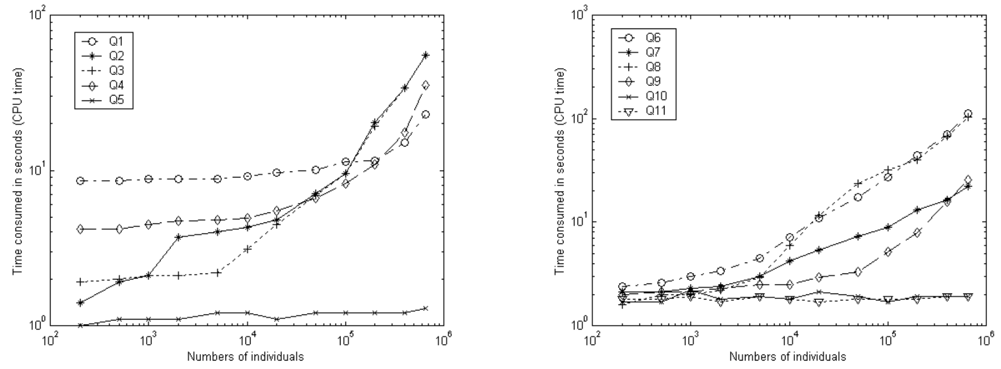


Figure 6.2: The Instance Store realistic (above) and artificial (below) query times -v- ABox size

reasoning is required. In these cases, the time taken to answer the query changes much more slowly with ABox size, and is mainly determined by the answer size. With Q4, for example, the time taken to answer the query rises to over 35s with the largest ABox, when the answer contains 7,728 individuals.

Tables 6.7 and 6.8 give the results for RACER when answering the same sets of five realistic and six artificial queries used to test the Instance Store, both in the case where the ABox has been realised (R) and where it has not (N). Timings are only approximate, as precise measurements were not possible when using the RACER server under Windows.

Table 6.7: RACER realistic query times (CPU seconds), realised (R) and not (N)

Number of	Q1		Q2		Q3		Q4		Q5	
Individuals	R	N	R	N	R	N	R	N	R	N
200	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0
500	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0
1,000	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0
2,000	n/a	≈60	n/a	≈240	n/a	≈60	n/a	≈150	n/a	≈210
5,000	n/a	≈240	n/a	≈420	n/a	≈240	n/a	≈360	n/a	≈300
10,000	n/a	≈1,080	n/a	≈1,080	n/a	≈660	n/a	≈720	n/a	≈930
20,000	n/a	fault	n/a	fault	n/a	fault	n/a	fault	n/a	fault

Table 6.8: RACER artificial query times (CPU seconds), realised (R) and not (N)

Number of	Q6		Q7		Q8		Q9		Q10		Q11	
Individuals	R	N	R	N	R	N	R	N	R	N	R	N
200	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0
500	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0
1,000	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0
2,000	n/a	≈120	n/a	≈120	n/a	≈60	n/a	≈60	n/a	≈210	n/a	≈180
5,000	n/a	≈420	n/a	≈270	n/a	≈420	n/a	≈480	n/a	≈330	n/a	≈390
10,000	n/a	≈1500	n/a	≈1120	n/a	≈1020	n/a	fault	n/a	≈810	n/a	≈780
20,000	n/a	fault	n/a	fault	n/a	fault	n/a	fault	n/a	fault	n/a	fault

In the cases where the ABox had been realised, queries were answered almost instantly, but results are only available for the relatively small ABoxes that RACER was able to realise (up to 1,000 individuals). In the cases where the ABox was not realised, answers were again returned almost instantly for smaller ABoxes, but when the ABox size exceeded 1,000 individuals the answer times increased dramatically, and for ABoxes larger than 10,000 individuals (larger than 5,000 in the case of Q9) RACER again broke down due to a resource allocation error in the underlying Lisp system.

It should be mentioned that the results for the Instance Store include significant communication overheads (both with the database and DL reasoner), which was not the case for RACER since queries were posed directly via the RACER command line interface.

6.5.2 Pseudo-individual Tests

As discussed in Section 6.4, one way to deal with role-free ABoxes is to treat individuals as atomic concepts in the TBox (pseudo-individuals). To test the feasibility of this approach, and to compare it with the Instance Store, we again used the GO TBox and ABox, and the realistic and artificial queries described above. In the pseudo-individual approach, ABox axioms of the form $x : C$ are treated as TBox axioms of the form $C_x \sqsubseteq C$, and retrieving the instances of a query concept Q means retrieving the pseudo-individuals that are subsumed by Q. In order to make the comparison as fair as possible, we did not use all of the individuals in the GO ABox, but only 48,581 individuals corresponding to the distinct concept expressions used to describe individuals in the GO ABox—the equivalence set optimisation described in Section 6.4.1 can obviously be used with the pseudo-individual approach as well. The FaCT system was used in these tests as RACER broke down when trying to classify the GO TBox augmented with the pseudo-individuals, again due to a resource allocation error in the underlying Lisp system.

In order to get some idea as to how the pseudo-individual approach would scale with increasing ABox (and hence TBox) size, we tried computing the concepts subsumed by each query with the GO TBox alone (which contains 47,012 concept names) and with the TBox augmented with the pseudo-individuals derived from the GO ABox (a total of 95,593 concept names). The answers to these DL queries include normal TBox concepts that are subsumed by the query concepts as well as any relevant pseudo-individuals, but the answer could easily be filtered so as to leave only the pseudo-individuals. (An alternative approach would be to add a concept PI to the TBox, representing pseudo-individuals, and conjoin PI to both pseudo-individual axioms and subsumption queries used to retrieve pseudo-individuals.) The results of these tests are given in Table 6.9. It is important to note that they do not include the time required to expand answers to include sets of equivalent individuals—as discussed above, this can be quite time consuming for some queries (e.g., 19.5s in the case of Q9 with the largest ABox).

Table 6.9: Pseudo-individual query time (CPU seconds) and answer size

Query	GO TBox		GO TBox + ABox	
	Time	Answer Size	Time	Answer Size
Q1	8.1	220	233.3	2,861
Q2	1.3	1	1.2	1
Q3	0.2	1	1.4	4
Q4	26.0	881	631.8	8,609
Q5	0.5	2	5.2	27
Q6	4.3	86	176.6	2,450
Q7	1.4	1	10.0	147
Q8	1.3	1	1.5	7
Q9	1.4	1	3.5	22
Q10	4.2	109	114.4	1,407
Q11	0.5	1	2.0	2

As can be seen from the results, the time taken to compute the answers to the queries is heavily dependent on the size of the answers, and, in the case of Q4 with the pseudo-individual augmented TBox, the time was over 600s. This is in contrast to the Instance Store, where the size of answer had comparatively little effect on the time taken to answer queries. For queries with relatively small answers, however, the pseudo-individual approach was highly effective, even for queries that were time consuming to answer using the Instance Store.

6.6 Query Answering with an Extended Instance Store

In this section we introduce an algorithm for instance retrieval in an $\mathcal{SHF}$ knowledge base. The algorithm can be divided into two steps. The first step transforms a general ABox into multiple new ABoxes, the second step is to use these newly constructed ABoxes answer instance retrieval properly.

6.6.1 Preliminaries

As the Instance Store does not respect the Unique Name Assumption (UNA), two separate individual names could be inferred to be identical. In the following, we present the definitions which are used to detect syntactically whether two individual names represent the very same element in a given ABox.

Definition 7 **($Source_{\mathcal{A}}(o, R), groupRole_{\mathcal{A}}(o, RG)$)** *Given an ABox $\mathcal{A}$, an individual name o, and a role name R in $\mathcal{A}$, the relation $Source_{\mathcal{A}}(o, R)$ holds iff there is a role name $R' \preceq R$ such that either $o\colon \exists R'.C \in \mathcal{A}$ or, for some individual name o', $\langle o, o' \rangle\colon R' \in \mathcal{A}$.*

Given a set of role names RG=$\{R_i \mid 1 \leq i \leq n\}$, and an individual name o in a KB $\langle \mathcal{T}, \mathcal{A} \rangle$, the relation $groupRole_{\mathcal{A}}(o, RG)$ holds iff, for any two role names R_ℓ and R_m in RG, the following two conditions are satisfied:

- *$Source_{\mathcal{A}}(o, R_\ell)$ and $Source_{\mathcal{A}}(o, R_m)$; and*
- *there exist a set of role names $\{L_1, \cdots, L_{n-1}\}$ and a set of functional roles $\{F_1, \cdots, F_n\}$ in $\mathcal{T}$, such that $R_\ell \preceq F_1, L_1 \preceq F_1, L_1 \preceq F_2, L_2 \preceq F_2, \cdots, L_i \preceq F_i, L_i \preceq F_{i+1}, \cdots, L_{n-1} \preceq F_n, R_m \preceq F_n$ and $Source_{\mathcal{A}}(o, L_i)$ for $i = 1, \cdots, n-1$.*

■

$Remarks$: The $groupRole_{\mathcal{A}}(o, RG)$ implies all role names in RG are functional ones. It also takes into account the possible interaction between the role hierarchy $\mathcal{H}$ and the functional restrictions $\mathcal{F}$. Basically, a set of role names RG are $groupRole_{\mathcal{A}}(o, RG)$ related if they are either functional or have some functional super role, and there are assertions in the ABox as shown in Definition 7 that force every R_i-successor o_i of the individual name o to be interpreted as the same element. This can be better understood by considering the following role hierarchy situation in which the relation $groupRole_{\mathcal{A}}(o, \{R_\ell, R_m\})$ holds:

For each of the role names $R_\ell, L_1, \cdots, L_{n-1}, R_m$, for all models $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $o^{\mathcal{I}}$ has successors $o_\ell, o_1, \cdots, o_{n-1}, o_m \in \Delta^{\mathcal{I}}$. By Definition 7, every role name has a functional super role, the functional restriction therefore forces every two successors to be the same element: o_ℓ and o_1, o_1 and o_2, $\cdots$, o_{n-1} and o_m, which in turn forces all the successors to be the same element. In particular, if $groupRole_{\mathcal{A}}(o, \{R_\ell, R_m\})$, the R_ℓ-successor o_ℓ of $o^{\mathcal{I}}$ is then forced to be the very same element as the R_m-successor o_m of $o^{\mathcal{I}}$.

Definition 8 ($sameAs_{\mathcal{A}}(o_1, o_2)$) *Given an ABox $\mathcal{A}$, two individual names o_1 and o_2, the relation*
$sameAs_{\mathcal{A}}(o_1, o_2)$ in $\mathcal{A}$ holds if there exists some individual name o with $\langle o, o_1 \rangle : R$, $\langle o, o_2 \rangle : S$, and $groupRole_{\mathcal{A}}(o, \Gamma)$ with $R, S \in \Gamma$. ■

Definition 9 (label) *Given an ABox $\mathcal{A}$, the* label *$\mathcal{L}(x)$ of an individual name x in $\mathcal{A}$ is defined as the conjunction of all concepts in the concept assertions about the individual name x:*

$$\mathcal{L}(x) := \begin{cases} \bigsqcap_{\{C \mid x:C \in \mathcal{A}\}} C & \textit{if the set } \{C \mid x : C \in \mathcal{A}\} \textit{ is not empty} \\ \top & \textit{otherwise} \end{cases}$$

■

CLAIM: [1] Given a TBox $\mathcal{T}$, an ABox $\mathcal{A}$ and an individual name o in $\mathcal{A}$, for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $o^{\mathcal{I}} \in \mathcal{L}(o)^{\mathcal{I}}$ holds. **Proof:** Let o be an individual name in $\mathcal{A}$ with a non-empty set $\{C_1, \cdots, C_n\} = \{C \mid o : C \in \mathcal{A}\}$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. By Definition 9, $\mathcal{L}(o) = C_1 \sqcap \cdots \sqcap C_n$. Since $\mathcal{I}$ is a model of $\mathcal{A}$, $o^{\mathcal{I}} \in C_i^{\mathcal{I}}$ for all $1 \leq i \leq n$. Hence $o^{\mathcal{I}} \in C_1^{\mathcal{I}} \cap \cdots \cap C_n^{\mathcal{I}}$ and, by the semantics, $o^{\mathcal{I}} \in (C_1 \sqcap \cdots \sqcap C_n)^{\mathcal{I}} = \mathcal{L}(o)^{\mathcal{I}}$.

If $\{C \mid o : C \in \mathcal{A}\}$ is empty, by Definition 9, $\mathcal{L}(o) = \top$. Hence $o^{\mathcal{I}} \in \Delta^{\mathcal{I}} = \top^{\mathcal{I}} = \mathcal{L}(o)^{\mathcal{I}}$. ■

Given two individual names o_1 and o_2 in the ABox $\mathcal{A}$, the relation $reachable(o_2, o_1)$ holds iff
$\langle o_1, o_2 \rangle : R$ in $\mathcal{A}$. Let $reachable^+$ be the transitive closure of $reachable$, i.e., $reachable^+(o_2, o_1)$ means a directed "role assertion chain" from o_1 to o_2 can be found in the ABox.

Definition 10 ((a)cyclic ABox) *An ABox $\mathcal{A}$ is* cyclic *iff there exists some individual name o in $\mathcal{A}$ such that $reachable^+(o, o)$. An ABox that is not cyclic is called* acyclic. ■

6.6.2 Precompleting an $\mathcal{SHF}$ ABox

The step of precompleting an $\mathcal{SHF}$ ABox is based upon a DL technique called *precompletion* [Hol96, Tes97]. It extends the original ABox using a set of syntactic rules. When no further rules can be applied, all information implicit in the role assertions has been made explicit through adding more concept assertions and making equalities between individuals explicit. Note that, for the DL $\mathcal{SHF}$, because of the non-determinism of its precompletion rules, many different precompletions can be derived from a single ABox.

In the following we present a set of nondeterministic syntactic rules which extend the original ABox. It will be shown that an interpretation $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ iff it is also a model of a precompletion of $\mathcal{A}$ derived using these rules.

To simplify the description of the algorithm, we assume that all concepts in the labels of individual names are in *negation normal form* (NNF), where negation can appear only in front of atomic concepts. Arbitrary $\mathcal{SHF}$ concepts can be transformed into equivalent ones in negation normal form using De Morgan's laws and rules including $\neg\neg C \mapsto C$, $\neg\exists R.C \mapsto \forall R.\neg C$ and $\neg\forall R.C \mapsto \exists R.\neg C$ [Hor97]. Moreover, we assume that all concept axioms in the TBox are in the form $\top \sqsubseteq C$ where C is an arbitrary concept expression. For DLs with negation, it is easy to show that any concept axioms of the form $C_1 \sqsubseteq C_2$ is equivalent to $\top \sqsubseteq (\neg C_1 \sqcup C_2)$ [Tes97].

Definition 11 ($Rep_{\mathcal{A}}$**)** *Given an ABox $\mathcal{A}$, $Rep_{\mathcal{A}}$ is a set containing pairs of individual names from $\mathcal{A}$.* ■

Definition 12 (precompletion rules) *Given a knowledge base $\langle\mathcal{T},\mathcal{A}\rangle$ and a set $Rep_{\mathcal{A}}$, the* precompletion rules *for $\mathcal{SHF}$ are defined as follows:*

1. *$\rightarrow_{\sqsubseteq}$ rule:*

   if o is in $\mathcal{A}$, $\top \sqsubseteq C \in \mathcal{T}$, and $o\colon C \notin \mathcal{A}$, then add $o\colon C$ to $\mathcal{A}$.

2. *$\rightarrow_{\sqcap}$ rule:*

   if $o\colon C_1 \sqcap C_2 \in \mathcal{A}$, and either $o\colon C_1 \notin \mathcal{A}$ or $o\colon C_2 \notin \mathcal{A}$, then add $o\colon C_1$ and $o\colon C_2$ to $\mathcal{A}$.

3. *$\rightarrow_{\sqcup}$ rule:*

   if $o\colon C_1 \sqcup C_2 \in \mathcal{A}$, $o\colon C_1 \notin \mathcal{A}$, and $o\colon C_2 \notin \mathcal{A}$, then choose $D{=}C_1$ or $D{=}C_2$, and add $o\colon D$ to $\mathcal{A}$.

4. *$\rightarrow_{\exists^1}$ rule:*

   if $o\colon \exists R.C \in \mathcal{A}$, $\langle o,o'\rangle : S \in \mathcal{A}$, $groupRole_{\mathcal{A}}(o,\{R,S,\cdots\})$, and $o'\colon C \notin \mathcal{A}$, then add $o'\colon C$ to $\mathcal{A}$.

5. *$\rightarrow_{\forall^1}$ rule:*

   if $o\colon \forall R.C \in \mathcal{A}$, $\langle o, o' \rangle\colon S \in \mathcal{A}$, there exists a role name $R' \preceq R$ such that $groupRole_{\mathcal{A}}(o, \{R', S, \cdots\})$, and $o'\colon C \notin \mathcal{A}$, then add $o'\colon C$ to $\mathcal{A}$.

6. *$\rightarrow_{\forall}$ rule:*

   if $o\colon \forall R.C \in \mathcal{A}$, $\langle o, o' \rangle\colon S \in \mathcal{A}$, $S \preceq R$, and $o'\colon C \notin \mathcal{A}$, then add $o'\colon C$ to $\mathcal{A}$.

7. *$\rightarrow_{\forall^+}$ rule:*

   if $o\colon \forall T.C \in \mathcal{A}$, $\langle o, o' \rangle\colon S \in \mathcal{A}$, there is a transitive role name R such that $S \preceq R \preceq T$, and $o'\colon \forall R.C \notin \mathcal{A}$, then add $o'\colon \forall R.C$ to $\mathcal{A}$.

8. *$\rightarrow_{sameAs}$ rule:*

   if $sameAs_{\mathcal{A}}(o, o')$, then add (o, o') to $Rep_{\mathcal{A}}$ and replace all occurrences of o in $\mathcal{A}$ with o'.

■

$Remarks$: Since the $\rightarrow_{\forall^1}$ rule only works on functional roles, the $\rightarrow_{\forall}$ rule can not be merged with $\rightarrow_{\forall^1}$ rule. Since a transitive role can not be a sub-role of a functional role, there is no need for a functional role version for $\rightarrow_{\forall^+}$ rule. The $\rightarrow_{sameAs}$ rule does not make $\rightarrow_{\forall^1}$ rule redundant—considering the following counterexample, $o\colon \exists R'$, $o\colon \forall R.C$, $\langle o, o' \rangle\colon S$ and $groupRole_{\mathcal{A}}(o, \{R', S\})$.

Definition 13 ($\mathcal{T}$-precompleted ABox) *Given a knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ and a set $Rep_{\mathcal{A}}$, the ABox $\mathcal{A}$ is called $\mathcal{T}$*-precompleted *iff none of the precompletion rules can be applied.*

■

Starting with the original ABox $\mathcal{A}$ and the empty set $Rep_{\mathcal{A}}$, the precompletion rules will non-deterministically generate *one* $\mathcal{T}$-precompleted ABox. If a searching strategy is applied upon these rules, however, multiple $\mathcal{T}$-precompleted ABoxes can then be found. Note that there may exist exponentially many
$\mathcal{T}$-precompleted ABoxes $\mathcal{A}', \mathcal{A}'', \cdots$ generated due to the non-deterministic $\rightarrow_{\sqcup}$ rules. Each of the $\mathcal{T}$-precompleted ABoxes, however, can be generated using polynomial space in the size of original $\mathcal{A}$.

Definition 14 (leaf node) *Given a $\mathcal{T}$-precompleted acyclic ABox $\mathcal{A}$, we call an individual name o a* leaf node *if, for any individual name x and role name R, $\langle o, x \rangle\colon R \notin \mathcal{A}$.*

■

Definition 15 (extended label) *Given an acyclic ABox $\mathcal{A}$, the* extended label $\mathcal{L}'(x)$ *of an individual name x in $\mathcal{A}$ is inductively defined as follows:*

$$\mathcal{L}'(x) := \begin{cases} \mathcal{L}(x) & \textit{if } x \textit{ is a leaf node} \\ \mathcal{L}(x) \sqcap \bigsqcap_{\langle x,x' \rangle :\, R} \exists R.\mathcal{L}'(x') & \textit{otherwise} \end{cases}$$

■

$Remarks$: The acyclicity condition in the above definition is to guarantee the termination— due to the presence of cycles among role assertions, the extended label generation process will not terminate.

Definition 16 (subconcept) *The* subconcept $sub(D)$ *of an $\mathcal{SHF}$-concept D is the closure of the subexpression of D and is inductively defined as follows:*

1. *if D is of the form $\neg C$, $\forall R.C$ or $\exists R.C$, then $sub(D) = \{D\} \cup sub(C)$;*
2. *if D is of the form $C_1 \sqcap C_2$ or $C_1 \sqcup C_2$, then $sub(D) = \{D\} \cup sub(C_1) \cup sub(C_2)$;*
3. *otherwise $sub(D) = \{D\}$.*

■

Definition 17 (consistent) *An ABox $\mathcal{A}$ is* consistent *with respect to a TBox $\mathcal{T}$, if there is an interpretation that is a model of $\langle \mathcal{T}, \mathcal{A} \rangle$.* ■

Definition 18 ($\mathcal{T}$-derivable) *Given a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$, $\mathcal{A}'$ is called* $\mathcal{T}$-derivable *from $\langle \mathcal{T}, \mathcal{A} \rangle$ if $\mathcal{A}'$ is $\mathcal{T}$-precompleted, consistent, and obtained from $\langle \mathcal{T}, \mathcal{A} \rangle$ and an empty set $Rep_{\mathcal{A}}$ by application of the precompletion rules.* ■

Lemma 19 *Given a consistent $\mathcal{T}$-precompleted ABox $\mathcal{A}$, an individual name o in $\mathcal{A}$, and a role name R in $\mathcal{A}$, the relation $Source_{\mathcal{A}}(o, R)$ holds iff, for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, there exists some element y in $\Delta^{\mathcal{I}}$ such that $(o^{\mathcal{I}}, y) \in R^{\mathcal{I}}$.*

Proof: "$\Rightarrow$" Let o be an individual name, let R be a role name in a $\mathcal{T}$-precompleted ABox $\mathcal{A}$ with
$Source_{\mathcal{A}}(o, R)$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. By Definition 7, there exists a role name $R' \preceq R$, such that either $o\colon \exists R'.C$ or for some individual name o', $\langle o, o' \rangle \colon R' \in \mathcal{A}$. Since $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A} \rangle$, this implies $o^{\mathcal{I}} \in (\exists R'.\top)^{\mathcal{I}}$ or $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in R'^{\mathcal{I}}$. Since $(\exists R'.\top)^{\mathcal{I}} \subseteq (\exists R.\top)^{\mathcal{I}}$ and $R'^{\mathcal{I}} \subseteq R^{\mathcal{I}}$, this implies that $o^{\mathcal{I}} \in (\exists R.\top)^{\mathcal{I}}$ or $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in R^{\mathcal{I}}$. Since $o^{\mathcal{I}} \in (\exists R.\top)^{\mathcal{I}}$ implies that $o^{\mathcal{I}} \in \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a, b) \in R^{\mathcal{I}}\}$, we can see that, in either case, there exists some element y such that $(o^{\mathcal{I}}, y) \in R^{\mathcal{I}}$.

"$\Leftarrow$" We prove this direction by proving its counterpositive, i.e., "Given a consistent $\mathcal{T}$-precompleted ABox $\mathcal{A}$, an individual name o in $\mathcal{A}$, and a role name R in $\mathcal{A}$, if the

relation
$Source_{\mathcal{A}}(o, R)$ does not hold, then there exists a model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$ such that, for any element $y \in \Delta^{\mathcal{I}}$, $(o^{\mathcal{I}}, y) \notin R^{\mathcal{I}}$." Let o be an individual name, let R be a role name in a $\mathcal{T}$-precompleted ABox $\mathcal{A}$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. We now show a witness model $\mathcal{I}'$ of $\langle \mathcal{T}, \mathcal{A} \rangle$ can be constructed based on $\mathcal{I}$. We remove all $(o^{\mathcal{I}}, y)$ tuples from $R^{\mathcal{I}}$ and $R'^{\mathcal{I}}$ in $\mathcal{A}$ with $R' \preceq R$ (y is an arbitrary element in $\Delta^{\mathcal{I}}$). Since the relation $Source_{\mathcal{A}}(o, R)$ does not hold, we know that either $o\colon \exists R'.C$ or for some individual name o', $\langle o, o' \rangle\colon R'$ with $R' \preceq R$ can not be found in $\mathcal{A}$. Therefore, the resulting model $\mathcal{I}'$ is still a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. Since all $(o^{\mathcal{I}}, y)$ tuples are removed during the construction of $\mathcal{I}'$, we have, for any element $y \in \Delta^{\mathcal{I}'}$, $(o^{\mathcal{I}'}, y) \notin R^{\mathcal{I}'}$.

■

Lemma 20 *Given a consistent ABox $\mathcal{A}$, an individual name o in $\mathcal{A}$, and a role name set Γ.*

1. *if the relation $groupRole_{\mathcal{A}}(o, \Gamma)$ holds, then every role name in Γ is functional, and for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $R_i, R_j \in \Gamma$, $(o^{\mathcal{I}}, x) \in R_i^{\mathcal{I}}$, $(o^{\mathcal{I}}, y) \in R_j^{\mathcal{I}}$, then $x = y$.*

2. *if $\mathcal{A}$ is $\mathcal{T}$-precompleted, every role name R_i in Γ is functional, and for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, for any $R_i, R_j \in \Gamma$, $(o^{\mathcal{I}}, x) \in R_i^{\mathcal{I}}$, $(o^{\mathcal{I}}, y) \in R_j^{\mathcal{I}}$, such that $x = y$, then the relation $groupRole_{\mathcal{A}}(o, \Gamma)$ holds.*

Proof: We shall prove the first claim first. Let o be an individual name, let $\Gamma = \{R_1, \cdots, R_n\}$ be a role name set in a consistent ABox $\mathcal{A}$ with $groupRole_{\mathcal{A}}(o, \Gamma)$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. By Definition 7, all the role names R_i in Γ are thus functional because they are either functional or have some functional super role ($F_1, \cdots, F_n$). Hence, for each of these role names R_i, $o^{\mathcal{I}}$ has at most one successor, say x_i.

In the following, we are going to show that all x_i, for $1 \leq i \leq n$, are equal. Let us arbitrarily choose two role names R_ℓ and R_m from Γ. By Definition 7, there exist a set of role names $\{L_1, \cdots, L_{n-1}\}$ and a set of functional roles $\{F'_1, \cdots, F'_n\}$, such that $R_\ell \preceq F'_1, L_1 \preceq F'_1, L_1 \preceq F'_2, L_2 \preceq F'_2, \cdots, L_i \preceq F'_i, L_i \preceq F'_{i+1}, \cdots, L_{n-1} \preceq F'_n, R_m \preceq F'_n$. Moreover, for each L_i, the relation $Source_{\mathcal{A}}(o, L_i)$ holds by definition, which implies that, for each L_i, there exists an element y_i in $\Delta^{\mathcal{I}}$ such that $(o^{\mathcal{I}}, y_i) \in L_i^{\mathcal{I}}$ (Lemma 19). Since relations $Source_{\mathcal{A}}(o, R_\ell)$ and $Source_{\mathcal{A}}(o, R_m)$ also hold by definition, there exist two element ℓ and m in $\Delta^{\mathcal{I}}$ such that $(o^{\mathcal{I}}, \ell) \in R_\ell^{\mathcal{I}}$ and $(o^{\mathcal{I}}, m) \in R_m^{\mathcal{I}}$.

Since $(o^{\mathcal{I}}, \ell) \in R_\ell^{\mathcal{I}}, (o^{\mathcal{I}}, y_1) \in L_1^{\mathcal{I}}, R_\ell^{\mathcal{I}} \subseteq F_1'^{\mathcal{I}}$ and $L_1^{\mathcal{I}} \subseteq F_1'^{\mathcal{I}}$, we know $\{(o^{\mathcal{I}}, \ell), (o^{\mathcal{I}}, y_1)\} \subseteq F_1'^{\mathcal{I}}$. Due to the functionality of $F_1'^{\mathcal{I}}$, we can conclude that $\ell = y_1$. Similarly, we can apply the same deduction to role name pairs L_1 and L_2, L_2 and L_3, $\cdots$, L_{n-1} and R_m, such that $y_1 = y_2$, $y_2 = y_3$, $\cdots$, $y_{n-1} = m$ which obviously induces $\ell = m$.

Analogously, the same arguments can be applied to any pair of role names R_i, R_{i+1}

in Γ. Therefore, for every model $\mathcal{I}$ of $\langle\mathcal{T},\mathcal{A}\rangle$, there exists only one element x in $\Delta^{\mathcal{I}}$, such that, for each role name R_i in Γ, $(o^{\mathcal{I}},x)\in R_i^{\mathcal{I}}$.

We prove the second claim by proving its counterpositive, i.e., "Given a consistent $\mathcal{T}$-precompleted ABox $\mathcal{A}$, an individual name o in $\mathcal{A}$, and a functional role name set Γ, if the relation $groupRole_{\mathcal{A}}(o,\Gamma)$ does not hold, then there exists a model $\mathcal{I}$ of $\langle\mathcal{T},\mathcal{A}\rangle$, and there exists two role names $R_i, R_j\in\Gamma$ with $(o^{\mathcal{I}},x)\in R_i^{\mathcal{I}}$ and $(o^{\mathcal{I}},y)\in R_j^{\mathcal{I}}$, such that $x\neq y$."

Let $\mathcal{A}$ be a consistent $\mathcal{T}$-precompleted ABox, let o be an individual name in $\mathcal{A}$, let Γ be a functional role name set, and let $\mathcal{I}$ be a model of $\langle\mathcal{T},\mathcal{A}\rangle$. We now show a witness model $\mathcal{I}'$ of $\langle\mathcal{T},\mathcal{A}\rangle$ can be constructed based on $\mathcal{I}$.

By precondition, there exists two role names $R_i, R_j\in\Gamma$ with $(o^{\mathcal{I}},x)\in R_i^{\mathcal{I}}$ and $(o^{\mathcal{I}},y)\in R_j^{\mathcal{I}}$. If the element $x\neq y$, then the model $\mathcal{I}$ is the witness model and we are done. If the element $x=y$, we first remove all $(o^{\mathcal{I}},x)$ tuples from $R_i^{\mathcal{I}}$ and $R'^{\mathcal{I}}$ in $\mathcal{A}$ with $R'\preceq R_i$. For each tuple we removed, we add $(o^{\mathcal{I}},z)$ to $R_i^{\mathcal{I}}$ and $R'^{\mathcal{I}}$ with $z\in\Delta^{\mathcal{I}}$ and $z\neq y$, thus we constructed a new model $\mathcal{I}'$ of $\langle\mathcal{T},\mathcal{A}\rangle$. Since the relation $groupRole_{\mathcal{A}}(o,\Gamma)$ does not hold, by Definition 7, we know that there exist at least a pair of role names $R_i, R_j\in\Gamma$ do not share their successors of $o^{\mathcal{I}}$ as the same element. Without loss of generality we assume R_i, R_j are such a pair of role names, therefore the functionalities of R_i, R_j are not violated and the model $\mathcal{I}'$ is the witness model. ■

Lemma 21 *Given a TBox $\mathcal{T}$, an acyclic ABox $\mathcal{A}$, and an individual name o in $\mathcal{A}$, for every model $\mathcal{I}$ of $\langle\mathcal{T},\mathcal{A}\rangle$, $o^{\mathcal{I}}\in\mathcal{L}'(o)^{\mathcal{I}}$ holds.*

Proof: Let o be an individual name in an acyclic ABox $\mathcal{A}$, and let $\mathcal{I}$ be a model of $\langle\mathcal{T},\mathcal{A}\rangle$. We prove this lemma by structural induction on the extended label definition.

- BASIS: The individual name o is a *leaf node*. By Definition 15, $\mathcal{L}(o)=\mathcal{L}'(o)$, therefore $\mathcal{L}'(o)^{\mathcal{I}}=\mathcal{L}(o)^{\mathcal{I}}$. Since $o^{\mathcal{I}}\in\mathcal{L}(o)^{\mathcal{I}}$ holds by Claim 1, this implies that $o^{\mathcal{I}}\in\mathcal{L}'(o)^{\mathcal{I}}$.

- INDUCTION: Let $\mathcal{L}'(o)$ be an extended label built by the inductive definition as follows:

$$\mathcal{L}'(o) := \mathcal{L}(o)\sqcap \bigsqcap_{\langle o,x_i\rangle:\,R_i\in\mathcal{A}} \exists R_i.\mathcal{L}'(x_i)$$

  By the induction hypothesis, for every individual name x_i with $\langle o,x_i\rangle : R_i\in\mathcal{A}$, we have $x_i^{\mathcal{I}}\in\mathcal{L}'(x_i)^{\mathcal{I}}$. To show that $o^{\mathcal{I}}\in\mathcal{L}'(o)^{\mathcal{I}}$, we have to show that $o^{\mathcal{I}}\in\mathcal{L}(o)^{\mathcal{I}}$ and, for each $\langle o,x_i\rangle : R_i\in\mathcal{A}$, $o^{\mathcal{I}}\in(\exists R_i.\mathcal{L}'(x_i))^{\mathcal{I}}$. The first point follows from Claim 1. For the second claim, let $\langle o,x_i\rangle : R_i\in\mathcal{A}$. Then we have

$$o^{\mathcal{I}}\in\{a\in\Delta^{\mathcal{I}}\mid (a,x_i^{\mathcal{I}})\in R_i^{\mathcal{I}}\}$$

Since $x_i^{\mathcal{I}} \in \mathcal{L}'(x_i)^{\mathcal{I}}$ holds by induction assumption, it implies that

$o^{\mathcal{I}} \in \{a \in \Delta^{\mathcal{I}} \mid (a, x_i^{\mathcal{I}}) \in R_i^{\mathcal{I}}\} \subseteq \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R_i^{\mathcal{I}} \wedge b \in \mathcal{L}'(x_i)^{\mathcal{I}}\} = (\exists R_i.\mathcal{L}'(x_i))^{\mathcal{I}}$

Therefore

$$o^{\mathcal{I}} \in \mathcal{L}(o)^{\mathcal{I}} \cap \bigcap_{\langle o, x_i \rangle :\, R_i \in \mathcal{A}} (\exists R_i.\mathcal{L}'(x_i))^{\mathcal{I}}$$

and thus $o^{\mathcal{I}} \in \mathcal{L}'(o)^{\mathcal{I}}$.

■

Lemma 22 *Given a consistent ABox $\mathcal{A}$, and two individual names o_1, o_2 in $\mathcal{A}$.*

1. *if the relation $sameAs\mathcal{A}(o_1, o_2)$ holds, then for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$.*
2. *if $\mathcal{A}$ is $\mathcal{T}$-precompleted, for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, and $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$, then (o_1, o_2) is in $Rep_{\mathcal{A}}$.*

Proof: We shall prove the first claim first. Let o_1, o_2 be individual names in a consistent ABox $\mathcal{A}$ with $sameAs_{\mathcal{A}}(o_1, o_2)$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. By Definition 8, there exists some individual name o with $\langle o, o_1 \rangle : R$, $\langle o, o_2 \rangle : S$, and $groupRole_{\mathcal{A}}(o, \Gamma)$ with $R, S \in \Gamma$. Hence, by Lemma 20, $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$.

We prove the second claim by proving its counterpositive, i.e., "Given a consistent $\mathcal{T}$-precompleted ABox $\mathcal{A}$, two individual names o_1, o_2 in $\mathcal{A}$, if (o_1, o_2) is not in $Rep_{\mathcal{A}}$, then there exists a model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, such that $o_1^{\mathcal{I}} \neq o_2^{\mathcal{I}}$."

Let o_1, o_2 be individual names in a consistent $\mathcal{T}$-precompleted ABox $\mathcal{A}$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. We now show a witness model $\mathcal{I}'$ of $\langle \mathcal{T}, \mathcal{A} \rangle$ can be constructed based on $\mathcal{I}$. We assume that $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$, otherwise $\mathcal{I}$ is the witness model and we are done. We take an element $x \in \Delta^{\mathcal{I}}$ with $x \neq o_2^{\mathcal{I}}$, and make $o_1^{\mathcal{I}} = x$. Since (o_1, o_2) is not in $Rep_{\mathcal{A}}$, by Definition 12, we know that the relation $sameAs\mathcal{A}(o_1, o_2)$ does not hold. By Definition 8, we know that there does not exist some individual name o with $\langle o, o_1 \rangle : R$, $\langle o, o_2 \rangle : S$, and $groupRole_{\mathcal{A}}(o, \Gamma)$ with $R, S \in \Gamma$. Since the functionalities of functional roles are not violated, the resulting model $\mathcal{I}'$ is still a model of $\langle \mathcal{T}, \mathcal{A} \rangle$ and it is the witness model.

■

6.6.3 Soundness and Completeness for Precompletion

This section presents the proof for soundness and completeness of the precompletion rules we proposed in last section.

Proposition 2 *The process of using the set of precompletion rules to extend the original ABox always terminates, and each $\mathcal{T}$-derivable ABox has a size which is polynomial with respect to the size of the original knowledge base.*

Proof: (Sketched). Termination of this process is an immediate consequence of the following observations. Let $|Ind(\mathcal{A})|$ be the number of individual names in the ABox $\mathcal{A}$ and $|C(\mathcal{T})|$ be the number of concept axioms in the TBox $\mathcal{T}$. The applicability of the $\rightarrow_{\sqsubseteq}$ rule is bounded by the number of $|Ind(\mathcal{A})|$*$|C(\mathcal{T})|$. The $\rightarrow_{\sqcap}$, $\rightarrow_{\sqcup}$, $\rightarrow_{\exists^1}$, $\rightarrow_{\forall^1}$ and $\rightarrow_{\forall}$ rules always introduce concepts into labels which are *subconcept*s of the original ones. Because the number of subconcepts is polynomial w.r.t. the size of a given knowledge base, only finitely many concept assertions can be added. The applicability of the $\rightarrow_{sameAs}$ rule is bounded by the number of *sameAs* related individual name pairs which is always less than $|Ind(\mathcal{A})| * (|Ind(\mathcal{A})| - 1)/2$. Therefore the process of applying the precompletion rules will terminate after finitely many steps.

To obtain an upper bound on the size of each $\mathcal{T}$-derivable ABox, we can use the results from the termination analysis. The number of different concept assertions that can be generated through $\rightarrow_{\sqsubseteq}$, $\rightarrow_{\sqcap}$, $\rightarrow_{\sqcup}$, $\rightarrow_{\exists^1}$, $\rightarrow_{\forall^1}$, $\rightarrow_{\forall}$ and $\rightarrow_{sameAs}$ rules cannot exceed the number of $|Ind(\mathcal{A})|$*$|C(\mathcal{T})|$, and this number is polynomial w.r.t. the size of the original KB. The number of different concept assertions that can be generated using $\rightarrow_{\forall+}$ rule is bound by the number of transitive role names and role assertions in the original KB. Therefore the size of each $\mathcal{T}$-derivable ABox is polynomial with respect to the size of the original KB. ■

Proposition 3 *All the precompletion rules preserve consistency of the ABox, i.e., given a TBox $\mathcal{T}$, an ABox $\mathcal{A}$, and a model $\mathcal{I}$ of $\langle\mathcal{T},\mathcal{A}\rangle$, if a precompletion rule is applicable, then there exists an ABox $\mathcal{A}'$ obtained after this rule application, such that $\mathcal{I}$ is also a model of $\langle\mathcal{T},\mathcal{A}'\rangle$.*

Proof: Suppose that $\mathcal{I}$ is a model of knowledge base $\langle\mathcal{T},\mathcal{A}\rangle$:

1. Let o be in $\mathcal{A}$, let $\top \sqsubseteq C \in \mathcal{T}$ and let $o\colon C \notin \mathcal{A}$. Then applying the $\rightarrow_{\sqsubseteq}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$=$\mathcal{A} \cup \{o\colon C\}$.

   Since $\top \sqsubseteq C \in \mathcal{T}$, we have $\Delta^{\mathcal{I}} \subseteq C^{\mathcal{I}}$, and then $o^{\mathcal{I}} \in C^{\mathcal{I}}$. Therefore $o\colon C$ is satisfied by $\mathcal{I}$, and thus $\mathcal{I}$ is also a model of $\langle\mathcal{T},\mathcal{A}'\rangle$.

2. Let $o\colon C_1 \sqcap C_2 \in \mathcal{A}$ and let either $o\colon C_1 \notin \mathcal{A}$ or $o\colon C_2 \notin \mathcal{A}$. Then applying the $\rightarrow_{\sqcap}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$=$\mathcal{A} \cup \{o\colon C_1, o\colon C_2\}$.

   Since $o\colon C_1 \sqcap C_2 \in \mathcal{A}$, we have $o^{\mathcal{I}} \in (C_1 \sqcap C_2)^{\mathcal{I}}$. By the semantics $(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$, hence $o^{\mathcal{I}} \in C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$. Therefore $o\colon C_1$ and $o\colon C_2$ are satisfied by $\mathcal{I}$, and thus $\mathcal{I}$ is also a model of $\langle\mathcal{T},\mathcal{A}'\rangle$.

3. Let $o\colon C_1 \sqcup C_2 \in \mathcal{A}$ and let $o\colon C_1 \notin \mathcal{A}$ and $o\colon C_2 \notin \mathcal{A}$. Then applying the $\rightarrow_{\sqcup}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$=$\mathcal{A} \cup \{o\colon C_1\}$ or $\mathcal{A}'$=$\mathcal{A} \cup \{o\colon C_2\}$.

   Since $o\colon C_1 \sqcup C_2 \in \mathcal{A}$, we have $o^{\mathcal{I}} \in (C_1 \sqcup C_2)^{\mathcal{I}}$. By the semantics $(C_1 \sqcup C_2)^{\mathcal{I}}$ = $C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$, hence $o^{\mathcal{I}} \in C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$. If $o^{\mathcal{I}} \in C_1^{\mathcal{I}}$ holds, we add $o\colon C_1$ to $\mathcal{A}$; otherwise if $o^{\mathcal{I}} \in C_2^{\mathcal{I}}$ holds, we add $o\colon C_2$ to $\mathcal{A}$. Hence we can apply the $\rightarrow_{\sqcup}$ rule such that $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A}' \rangle$ for $\mathcal{A}'$ the resulting ABox.

4. Let $o\colon \exists R.C \in \mathcal{A}$, let $\langle o, o' \rangle : S \in \mathcal{A}$, let $groupRole_{\mathcal{A}}(o, \{R, S, \cdots\})$, and let $o'\colon C \notin \mathcal{A}$. Then applying the $\rightarrow_{\exists^1}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$=$\mathcal{A} \cup \{o'\colon C\}$.

   Since $o\colon \exists R.C \in \mathcal{A}$, there exists an element $x \in C^{\mathcal{I}}$ such that $(o^{\mathcal{I}}, x) \in R^{\mathcal{I}}$. Since $\langle o, o' \rangle\colon S \in \mathcal{A}$, we have $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in S^{\mathcal{I}}$. Since $groupRole_{\mathcal{A}}(o, \{R, S, \cdots\})$, we have $o'^{\mathcal{I}}$ = x by Lemma 20. Then we can see that $o'^{\mathcal{I}} \in C^{\mathcal{I}}$. Therefore $o'\colon C$ is satisfied by $\mathcal{I}$, and thus $\mathcal{I}$ is also a model of $\langle \mathcal{T}, \mathcal{A}' \rangle$.

5. Let $o\colon \forall R.C \in \mathcal{A}$, let $\langle o, o' \rangle\colon S \in \mathcal{A}$, let $groupRole_{\mathcal{A}}(o, \{R', S, \cdots\})$ for some role name $R' \preceq R$, and let $o'\colon C \notin \mathcal{A}$. Then applying the $\rightarrow_{\forall^1}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$=$\mathcal{A} \cup \{o'\colon C\}$.

   Since $o\colon \forall R.C \in \mathcal{A}$, every element x with $(o^{\mathcal{I}}, x) \in R^{\mathcal{I}}$ must be in $C^{\mathcal{I}}$. Since $groupRole_{\mathcal{A}}(o, \{R', S, \cdots\})$, there exists an element y in $\Delta^{\mathcal{I}}$ such that $(o^{\mathcal{I}}, y) \in R'^{\mathcal{I}}$ (Definition 7 and Lemma 19). Since $R'^{\mathcal{I}} \subseteq R^{\mathcal{I}}$, we have $(o^{\mathcal{I}}, y) \in R^{\mathcal{I}}$ and $y \in C^{\mathcal{I}}$. Since $\langle o, o' \rangle\colon S \in \mathcal{A}$, we have $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in S^{\mathcal{I}}$. Since $groupRole_{\mathcal{A}}(o, \{R', S, \cdots\})$, we have $o'^{\mathcal{I}}$ = y by Lemma 20. Then we can see that $o'^{\mathcal{I}} \in C^{\mathcal{I}}$. Therefore $o'\colon C$ is satisfied by $\mathcal{I}$, and thus $\mathcal{I}$ is also a model of $\langle \mathcal{T}, \mathcal{A}' \rangle$.

6. Let $o\colon \forall R.C \in \mathcal{A}$, let $\langle o, o' \rangle\colon S \in \mathcal{A}$, let $S \preceq R \in \mathcal{T}$, and let $o'\colon C \notin \mathcal{A}$. Then applying the $\rightarrow_{\forall}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$=$\mathcal{A} \cup \{o'\colon C\}$.

   Since $o\colon \forall R.C \in \mathcal{A}$, every element x with $(o^{\mathcal{I}}, x) \in R^{\mathcal{I}}$ must be in $C^{\mathcal{I}}$. Since $\langle o, o' \rangle\colon S \in \mathcal{A}$, we have $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in S^{\mathcal{I}}$. Since $S^{\mathcal{I}} \subseteq R^{\mathcal{I}} \in \mathcal{T}$, we have $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in R^{\mathcal{I}}$ and $o'^{\mathcal{I}} \in C^{\mathcal{I}}$. Therefore $o'\colon C$ is satisfied by $\mathcal{I}$, and thus $\mathcal{I}$ is also a model of $\langle \mathcal{T}, \mathcal{A}' \rangle$.

7. Let $o\colon \forall T.C \in \mathcal{A}$, let $\langle o, o' \rangle\colon S \in \mathcal{A}$ with a transitive role name R such that $S \preceq R \preceq T \in \mathcal{T}$, and let $o'\colon \forall R.C \notin \mathcal{A}$. Then applying the $\rightarrow_{\forall^+}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$=$\mathcal{A} \cup \{o'\colon \forall R.C\}$.

   Since $o\colon \forall T.C \in \mathcal{A}$, every element x with $(o^{\mathcal{I}}, x) \in T^{\mathcal{I}}$ must be in $C^{\mathcal{I}}$. Since $\langle o, o' \rangle\colon S \in \mathcal{A}$, we have $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in S^{\mathcal{I}}$. Since $S \preceq R \preceq T \in \mathcal{T}$, we have $S^{\mathcal{I}} \subseteq R^{\mathcal{I}} \subseteq T^{\mathcal{I}}$ which means $(o^{\mathcal{I}}, o'^{\mathcal{I}}) \in R^{\mathcal{I}}$. If there exists an element y in $\Delta^{\mathcal{I}}$ such that $(o'^{\mathcal{I}}, y) \in R^{\mathcal{I}}$, then $(o^{\mathcal{I}}, y) \in R^{\mathcal{I}}$ due to the transitivity of $R^{\mathcal{I}}$. Since $R \preceq T$, the element y must be in $C^{\mathcal{I}}$. Therefore $o'^{\mathcal{I}} \in (\forall R.C)^{\mathcal{I}}$ and then $o'\colon \forall R.C$ is satisfied by $\mathcal{I}$, and thus $\mathcal{I}$ is also a model of $\langle \mathcal{T}, \mathcal{A}' \rangle$.

8. Let $sameAs_{\mathcal{A}}(o, o')$ and o be in $\mathcal{A}$. Then applying the $\rightarrow_{sameAs}$ rule to $\mathcal{A}$ yields $\mathcal{A}'$ which is obtained through replacing all occurrences of o in $\mathcal{A}$ with o'.

Since $o^{\mathcal{I}} = o'^{\mathcal{I}}$ by Lemma 22, all newly generated concept assertions and role assertions of o' through replacements are satisfied by $\mathcal{I}$, and thus $\mathcal{I}$ is also a model of $\langle \mathcal{T}, \mathcal{A}' \rangle$.

Therefore, for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, if a precompletion rule is applicable, then there exists an ABox $\mathcal{A}'$ obtained after the rule application, such that $\mathcal{I}$ is still a model of $\langle \mathcal{T}, \mathcal{A}' \rangle$. ∎

Proposition 4 *Given a TBox $\mathcal{T}$, an ABox $\mathcal{A}$ and all ABoxes $\mathcal{A}_1^{pc}, \mathcal{A}_2^{pc}, \cdots, \mathcal{A}_n^{pc}$ $\mathcal{T}$-derivable from $\langle \mathcal{T}, \mathcal{A} \rangle$.*

1. *If $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A} \rangle$, then $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A}_i^{pc} \rangle$ for some i;*
2. *If $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A}_i^{pc} \rangle$ for some i, then there exists an extension $\mathcal{I}'$ of $\mathcal{I}$ that is a model of $\langle \mathcal{T}, \mathcal{A} \rangle$.*

Proof: Let $\mathcal{T}$ be a TBox, let $\mathcal{A}$ be an ABox, let $\mathcal{A}_i^{pc}$ be an ABox $\mathcal{T}$-derivable from $\langle \mathcal{T}, \mathcal{A} \rangle$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. The first claim (soundness) can be proved by induction on the process of applying precompletion rules:

- BASIS: For the basis, let $\mathcal{A}_1$ be a consistent ABox extended from the original ABox $\mathcal{A}$ after one step application using some precompletion rule. Since $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A} \rangle$ by assumption, we can apply the rule in a way such that $\mathcal{I}$ is still a model of some $\langle \mathcal{T}, \mathcal{A}_1 \rangle$ by Proposition 3;
- INDUCTION: Now assume n $\geq$ 1, let $\mathcal{A}_{n+1}$ be a consistent ABox extended from a consistent ABox $\mathcal{A}_n$ after one step application using some precompletion rule. By induction, $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A}_n \rangle$. By Proposition 3, we can apply the rule in a way such that $\mathcal{I}$ is still a model of some $\langle \mathcal{T}, \mathcal{A}_{n+1} \rangle$.

By Proposition 2, the process of using the set of precompletion rules to extend the original ABox always terminate. Hence, after finitely many steps of precompletion rule applications, the precompletion rules can be applied in a way such that $\mathcal{I}$ is still a model of some $\langle \mathcal{T}, \mathcal{A}_i^{pc} \rangle$.

We now prove the second claim (completeness). Let $\mathcal{T}$ be a TBox, let $\mathcal{A}$ be an ABox, let $\mathcal{A}_i^{pc}$ be an ABox $\mathcal{T}$-derivable from $\langle \mathcal{T}, \mathcal{A} \rangle$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A}_i^{pc} \rangle$. We can construct a new ABox $\mathcal{A}_i^{pc'}$ as follows: for each individual name pair in $Rep_{\mathcal{A}}$, recover all the replacements of individuals names and add them to $\mathcal{A}_i^{pc'}$. We now construct a new model $\mathcal{I}'$ for $\langle \mathcal{T}, \mathcal{A}_i^{pc'} \rangle$ based on $\mathcal{I}$, for each individual name o_i' replaced by o_i, if $o_i^{\mathcal{I}} = x$ with $x \in \Delta^{\mathcal{I}}$, make $o_i'^{\mathcal{I}} = x$. By Lemma 22, the resulting model $\mathcal{I}'$ is still a model of $\langle \mathcal{T}, \mathcal{A}_i^{pc'} \rangle$. Since $\mathcal{A}$ is a subset of any constructed $\mathcal{A}_i^{pc'}$, if there exists a model $\mathcal{I}'$ of $\langle \mathcal{T}, \mathcal{A}_i^{pc'} \rangle$, then it is also a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. Therefore, if $\mathcal{I}$ is a model of $\langle \mathcal{T}, \mathcal{A}_i^{pc} \rangle$, then there exists an extension $\mathcal{I}'$ of $\mathcal{I}$ that is a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. ∎

Now we can concentrate on the derived ABoxes and show how instance retrieval in $\mathcal{A}$ can be realised using derived ABoxes.

6.6.4 Answering instance retrieval

An ABox is consistent if and only if it has a consistent derived ABox (Proposition 4). When it comes to instance retrieval, computing instances of a given concept using only one obtained consistent derived ABox is not *sound*—an individual is an instance of a given concept in the original ABox if and only if it is instance of the given concept in every consistent derived ABox. Taking this matter into account, the step of answering instance retrieval, is therefore defined as follows:

Definition 23 (acyclic answering procedure) *The* acyclic answering procedure *returns x to query Q w.r.t. a TBox $\mathcal{T}$ and an acyclic ABox $\mathcal{A}$ if for each ABox $\mathcal{A}'$ $\mathcal{T}$-derivable from $\mathcal{A}$, the extended label $\mathcal{L}'(x) \sqsubseteq_{\mathcal{T}} Q$ in $\mathcal{A}'$.* ■

Proposition 5 *Let $\mathcal{T}$ be a TBox, $\mathcal{A}$ be a consistent $\mathcal{T}$-precompleted acyclic ABox, Q be a concept. For every model $\mathcal{I}$ of $\langle\mathcal{T},\mathcal{A}\rangle$, $o^{\mathcal{I}} \in Q^{\mathcal{I}}$ iff $\mathcal{L}'(o) \sqsubseteq_{\mathcal{T}} Q$.*

Proof: "$\Leftarrow$" Let o be an individual name in a consistent $\mathcal{T}$-precompleted acyclic ABox $\mathcal{A}$, let Q be a concept, and let $\mathcal{I}$ be a model of $\langle\mathcal{T},\mathcal{A}\rangle$. Since $o^{\mathcal{I}} \in \mathcal{L}'(o)^{\mathcal{I}}$ holds (Lemma 21), and $\mathcal{L}'(o)^{\mathcal{I}} \subseteq Q^{\mathcal{I}}$ holds, we can see that $o^{\mathcal{I}} \in Q^{\mathcal{I}}$.

"$\Rightarrow$" Let o be an individual name in a consistent $\mathcal{T}$-precompleted acyclic ABox $\mathcal{A}$, let Q be a query concept, and let $\mathcal{I}$ be a model of $\langle\mathcal{T},\mathcal{A}\rangle$. We prove this direction by structural induction on the extended label definition:

- BASIS: The basis case is when the individual name o is a *leaf node*. By Definition 15, $\mathcal{L}(o) = \mathcal{L}'(o)$. Since $o^{\mathcal{I}} \in Q^{\mathcal{I}}$, we have $\mathcal{L}(o) \sqcap \neg Q \sqsubseteq \bot$. Since $\mathcal{L}(o) = \mathcal{L}'(o)$, we have $\mathcal{L}'(o) \sqcap \neg Q \sqsubseteq \bot$. Therefore $\mathcal{L}(o) \sqsubseteq Q$.

- INDUCTION: We prove this step by proving its counterpositive, i.e., "If there exists a model $\mathcal{I}$ of $\langle\mathcal{T},\mathcal{A}\rangle$, such that $\mathcal{L}'(o)^{\mathcal{I}} \not\subseteq Q^{\mathcal{I}}$ holds, then $o^{\mathcal{I}} \notin Q^{\mathcal{I}}$ holds."

  Let $\mathcal{L}'(o)$ be the extended label of o built by the inductive definition as follows:

  $$\mathcal{L}'(o) := \mathcal{L}(o) \sqcap \bigsqcap_{\langle o,x_i\rangle:\, R_i \in \mathcal{A}} \exists R_i.\mathcal{L}'(x_i)$$

  By assumption there exists a model $\mathcal{I}$ of $\langle\mathcal{T},\mathcal{A}\rangle$, such that $\mathcal{L}'(o)^{\mathcal{I}} \not\subseteq Q^{\mathcal{I}}$ holds. Then we may assume there exist a model $\hat{\mathcal{I}}$ and an element a in $\Delta^{\hat{\mathcal{I}}}$, with $a \in \mathcal{L}'(o)^{\hat{\mathcal{I}}}$ and $a \notin Q^{\hat{\mathcal{I}}}$. In the following we show that it is possible to define the model $\hat{\mathcal{I}}$ in such a way that $a = o^{\hat{\mathcal{I}}}$.

Since $a \in \mathcal{L}'(o)^{\hat{\mathcal{I}}}$, therefore $a \in \mathcal{L}(o)^{\hat{\mathcal{I}}}$ and $a \in (\exists R_i.\mathcal{L}'(x_i))^{\hat{\mathcal{I}}}$; that is, $a \in \{x \in \mathcal{L}(o)^{\hat{\mathcal{I}}} \mid (x, b) \in R_i^{\hat{\mathcal{I}}} \wedge b \in \mathcal{L}'(x_i)^{\hat{\mathcal{I}}}\}$. The individual name o must suffice its concept assertions and role assertions for every model including $\hat{\mathcal{I}}$, therefore $o^{\hat{\mathcal{I}}} \in \mathcal{L}(o)^{\hat{\mathcal{I}}}$ and $(o^{\hat{\mathcal{I}}}, x_i^{\hat{\mathcal{I}}}) \in R_i^{\hat{\mathcal{I}}}$; that is, $o^{\hat{\mathcal{I}}} \in \{x \in \mathcal{L}(o)^{\hat{\mathcal{I}}} \mid (x, x_i^{\hat{\mathcal{I}}}) \in R_i^{\hat{\mathcal{I}}}\}$.

Following the result from the $\Leftarrow$ direction, for some concept C, if for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $\mathcal{L}'(x_i)^{\mathcal{I}} \subseteq C^{\mathcal{I}}$ holds, then $x_i^{\mathcal{I}} \in C^{\mathcal{I}}$. Following our induction assumption, if for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $x_i^{\mathcal{I}} \in C^{\mathcal{I}}$ holds, then $\mathcal{L}'(x_i)^{\mathcal{I}} \subseteq C^{\mathcal{I}}$ holds. Because $\hat{\mathcal{I}}$ is also a model of $\langle \mathcal{T}, \mathcal{A} \rangle$, we can conclude that $\mathcal{L}'(x_i)^{\hat{\mathcal{I}}} \subseteq C^{\hat{\mathcal{I}}}$ iff $x_i^{\hat{\mathcal{I}}} \in C^{\hat{\mathcal{I}}}$. Hence the element set $\{x \in \mathcal{L}(o)^{\hat{\mathcal{I}}} \mid (x, b) \in R_i^{\hat{\mathcal{I}}} \wedge b \in \mathcal{L}'(x_i)^{\hat{\mathcal{I}}}\}$ is the same as $\{x \in \mathcal{L}(o)^{\hat{\mathcal{I}}} \mid (x, x_i^{\hat{\mathcal{I}}}) \in R_i^{\hat{\mathcal{I}}}\}$. Therefore, the element a satisfies the condition of being $o^{\hat{\mathcal{I}}}$. Since we know that $a \notin Q^{\hat{\mathcal{I}}}$, $o^{\hat{\mathcal{I}}} \notin Q^{\hat{\mathcal{I}}}$.

■

Lemma 24 *Given a TBox $\mathcal{T}$, an acyclic ABox $\mathcal{A}$, ABoxes $\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_n$ $\mathcal{T}$-derivable from $\mathcal{A}$, and a concept Q, for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $o^{\mathcal{I}} \in Q^{\mathcal{I}}$ iff $o^{\mathcal{I}_i} \in Q^{\mathcal{I}_i}$ holds for every model $\mathcal{I}_i$ of every $\langle \mathcal{T}, \mathcal{A}_i \rangle$.*

Proof: "$\Rightarrow$" We prove this direction by contradiction. Thus assume that for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, we have $o^{\mathcal{I}} \in Q^{\mathcal{I}}$, and there exists one model $\mathcal{I}_i$ of some $\langle \mathcal{T}, \mathcal{A}_i \rangle$ such that $o^{\mathcal{I}_i} \notin Q^{\mathcal{I}_i}$. Since there exists one model $\mathcal{I}_i$ for some $\langle \mathcal{T}, \mathcal{A}_i \rangle$, we can see that there exists a extension $\mathcal{I}'_i$ of $\mathcal{I}_i$ that is a model of $\langle \mathcal{T}, \mathcal{A} \rangle$ (Proposition 4). Since $o^{\mathcal{I}_i} \notin Q^{\mathcal{I}_i}$ w.r.t. $\langle \mathcal{T}, \mathcal{A}_i \rangle$, we have $o^{\mathcal{I}'_i} \notin Q^{\mathcal{I}'_i}$ w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$. Since for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$ $o^{\mathcal{I}} \in Q^{\mathcal{I}}$ must hold, we derived a contradiction.

"$\Leftarrow$" Analogously, this direction can be proved by contradiction as well. Thus assume that for every model $\mathcal{I}_i$ of every $\langle \mathcal{T}, \mathcal{A}_i \rangle$, we have $o_i^{\mathcal{I}} \in Q_i^{\mathcal{I}}$, and there exists one model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$ such that $o^{\mathcal{I}} \notin Q^{\mathcal{I}}$. Since there exists one model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, we can see that $\mathcal{I}$ is also a model of some $\langle \mathcal{T}, \mathcal{A}_i \rangle$ (proposition 4). Since $o^{\mathcal{I}} \notin Q^{\mathcal{I}}$ w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$, we have $o^{\mathcal{I}} \notin Q^{\mathcal{I}}$ w.r.t. some $\langle \mathcal{T}, \mathcal{A}_i \rangle$. Since for every model $\mathcal{I}_i$ of every $\langle \mathcal{T}, \mathcal{A}_i \rangle$ $o^{\mathcal{I}_i} \in Q^{\mathcal{I}_i}$ must hold, we derived a contradiction. ■

Theorem 25 *The acyclic answering procedure returns individual x to concept Q with respect to a knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ (where $\mathcal{A}$ is acyclic) iff, for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $o^{\mathcal{I}} \in Q^{\mathcal{I}}$.*

Proof: Let o be an individual name in an acyclic ABox $\mathcal{A}$, let Q be a concept, let $\mathcal{L}'(o)$ be the extended label of o, let $\mathcal{A}_i$ be ABoxes $\mathcal{T}$-derivable from $\mathcal{A}$, and let $\mathcal{I}$ be a model of $\langle \mathcal{T}, \mathcal{A} \rangle$. We know that $o^{\mathcal{I}} \in Q^{\mathcal{I}}$ w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$ iff, for every model $\mathcal{I}_i$ of every $\langle \mathcal{T}, \mathcal{A}_i \rangle$,

$o^{\mathcal{I}_i} \in Q^{\mathcal{I}_i}$ (Lemma 24). We also know that $o^{\mathcal{I}_i} \in Q^{\mathcal{I}_i}$ w.r.t. $\langle \mathcal{T}, \mathcal{A}_i \rangle$ iff, $\mathcal{L}'(o)^{\mathcal{I}_i} \subseteq Q^{\mathcal{I}_i}$ (Proposition 5). Therefore, $o^{\mathcal{I}} \in Q^{\mathcal{I}}$ w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$ iff, for every $\langle \mathcal{T}, \mathcal{A}_i \rangle$, $\mathcal{L}'(o) \sqsubseteq Q$.

■

This means that we now have an instance retrieval algorithm: the acyclic answering procedure of Definition 23 provides an algorithmic way to compute the instance retrieval answers in a knowledge base.

6.6.5 Answering instance retrieval without acyclic restriction

When the ABox is cyclic, the idea of doing instance retrieval using extended label is not working anymore—the extended label generation process would not terminate because of the presence of cycles among role assertions. In the following we introduce an algorithm for retrieving instances in an $\mathcal{SHF}$ knowledge base without acyclicity restriction.

Definition 26 (locally-consistent) *Given a $\mathcal{T}$-precompleted ABox $\mathcal{A}$ and an individual name x, $\mathcal{A}$ is* locally-consistent *w.r.t. x if the following two conditions are satisfied:*

- *$\mathcal{L}(x)$ is satisfiable w.r.t. $\mathcal{T}$; and*
- *if $\langle x, y \rangle : R \in \mathcal{A}$, then $\mathcal{A}$ is* locally-consistent *w.r.t. y.*

■

Lemma 27 *Given a $\mathcal{T}$-precompleted ABox $\mathcal{A}$, $\mathcal{A}$ is inconsistent iff there exists some individual name o in $\mathcal{A}$ and $\mathcal{A}$ is* not *locally-consistent w.r.t. o.*

Definition 28 (boolean answering) *Given $\mathcal{T}$ a TBox, $\mathcal{A}$ a consistent $\mathcal{T}$-precompleted ABox, x an individual name in $\mathcal{A}$ and Q a concept, the* boolean answering *returns* True *for $(x, Q, \langle \mathcal{T}, \mathcal{A} \rangle)$ if there is no ABox $\mathcal{T}$-derivable from $\mathcal{A} \cup \{x : \neg Q\}$ that is locally-consistent w.r.t. x.* ■

Proposition 6 *Let $\mathcal{T}$ be a TBox, $\mathcal{A}$ be a consistent $\mathcal{T}$-precompleted ABox, Q be a concept. For every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $o^{\mathcal{I}} \in Q^{\mathcal{I}}$ iff the boolean answering returns* True *for $(x, Q, \langle \mathcal{T}, \mathcal{A} \rangle)$.*

Given the above boolean query answering definition, the procedure for answering instance retrieval queries without acyclic restriction is defined as follows:

Definition 29 (answering procedure) *The* answering procedure *returns an individual answer set*
$\{x_1, x_2, \cdots, x_n\}$ to query Q w.r.t. a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$ if, for each ABox $\mathcal{A}'$ $\mathcal{T}$-derivable from $\mathcal{A}$, the boolean query answering returns True *for $(x_i, Q, \langle \mathcal{T}, \mathcal{A}' \rangle)$.* ■

Theorem 30 *The answering procedure returns individual x to concept Q with respect to a knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ iff, for every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$, $o^{\mathcal{I}} \in Q^{\mathcal{I}}$.*

$Remarks$: Although the above answering procedure can be used to retrieve instances in an $\mathcal{SHF}$ knowledge base without acyclicity restriction, it is not efficient—the boolean query answering procedure only tests one individual name at a time. For implementation purpose, in the next section, we propose the query-oriented answering procedure.

6.6.6 Query-oriented answering

Definition 31 (literal, quantifier form, Disjunctive Normal Form) *An $\mathcal{SHF}$ concept is considered to be a* literal *iff it is either a concept name or the negation of a concept name.*

Given a role name R and a concept expression C, an $\mathcal{SHF}$ concept is considered to be in quantifier form *iff it is either in the form of $\forall R.C$ or in the form of $\exists R.C$.*

An $\mathcal{SHF}$ concept is considered to be in Disjunctive Normal Form *iff it is a disjunction of one or more conjunctions of one or more literals or one or more concept expressions in quantifier form.* ■

Given an $\mathcal{SHF}$ concept Q in Disjunctive Normal Form, without loss of generality we assume that each disjunct is in the following form:

$$(\bigsqcap_{1 \le j \le m_i} \exists S_j.V_j) \sqcap (\bigsqcap_{1 \le k \le n_i} C_k) \sqcap (\bigsqcap_{1 \le \ell \le o_i} \forall R_\ell.U_\ell)$$

We now introduce some new notation used, for convenience, in the following query-oriented answering procedure. For every disjunct D_i of Q:

- $\{\exists(D_i)\}$ for the set $\{\exists S_j.V_j \mid 1 \le j \le m_i\}$
- $\nexists(D_i)$ for the concept $(\bigsqcap_{1 \le k \le n_i} C_k) \sqcap (\bigsqcap_{1 \le \ell \le o_i} \forall R_\ell.U_\ell)$;

Definition 32 ($associated_{\mathcal{A}}(o_1, o_2, R)$) *Given a role name R, two individual names o_1 and o_2 in the ABox $\mathcal{A}$, IN the individual name set and RN the role name set in $\mathcal{A}$, the relation $associated \subseteq IN \times IN \times RN$ is defined inductively as follows:*

- *BASIS: if there exists a role name S such that $\langle o_1, o_2 \rangle : S$ in $\mathcal{A}$ and $S \preceq R$, then $associated_{\mathcal{A}}(o_1, o_2, R)$;*
- *INDUCTION: if there exists a transitive role name S, an individual name o' in $\mathcal{A}$ and $S \preceq R$, $associated_{\mathcal{A}}(o_1, o', S)$ and $associated_{\mathcal{A}}(o', o_2, S)$, then $associated_{\mathcal{A}}(o_1, o_2, R)$.*

■

CLAIM: [2] Given a clash-free $\mathcal{T}$-precompleted ABox $\mathcal{A}$, two individual names o_1,o_2 and two role names S,R in $\mathcal{A}$, if the relation $associated_{\mathcal{A}}(o_1, o_2, S)$ holds and $S \preceq R$, then $associated_{\mathcal{A}}(o_1, o_2, R)$.

Proof: Let $\mathcal{A}$ be a clash-free $\mathcal{T}$-precompleted ABox, let o_1,o_2 be individual names in $\mathcal{A}$, and let S,R be role names in $\mathcal{A}$. This claim can be proved using the following basis and induction steps:

- BASIS: The basis case is there exists a role name S' such that $\langle o_1, o_2 \rangle : S'$ in $\mathcal{A}$ and $S' \preceq S$. Since $S' \preceq S \preceq R$, we have $associated_{\mathcal{A}}(o_1, o_2, R)$.

- INDUCTION: Let $associated_{\mathcal{A}}(o_1, o_2, S)$ be the relation built by the inductive step of the definition, from $associated_{\mathcal{A}}(o_1, o', S')$ and $associated_{\mathcal{A}}(o', o_2, S')$ with S' a transitive role, o' an individual name and $S' \preceq S$. Since $S' \preceq S \preceq R$, we have $associated_{\mathcal{A}}(o_1, o_2, R)$.

■

CLAIM: [3] Given a clash-free $\mathcal{T}$-precompleted ABox $\mathcal{A}$, three individual names o_1,o_2,o_3 and a role name S in $\mathcal{A}$, if the relations $associated_{\mathcal{A}}(o_1, o_2, S)$ and $associated_{\mathcal{A}}(o_2, o_3, S)$ hold and S is a transitive role name, then $associated_{\mathcal{A}}(o_1, o_3, S)$.

Proof: Let $\mathcal{A}$ be a clash-free $\mathcal{T}$-precompleted ABox, let o_1,o_2,o_3 be individual names in $\mathcal{A}$, and let S be a role names in $\mathcal{A}$. This claim follows directly from the inductive step in the definition: if there exist a transitive role name S, three individual name o_1,o_2,o_3 in $\mathcal{A}$ and $S \preceq S$, $associated_{\mathcal{A}}(o_1, o2, S)$ and $associated_{\mathcal{A}}(o_2, o_3, S)$ hold, then the relation $associated_{\mathcal{A}}(o_1, o_3, S)$ holds. ■

Given a $\mathcal{T}$-precompleted ABox $\mathcal{A}$, a concept Q in disjunctive normal form, the instance of Q is computed as shown in Algorithm 1. The $retrieve(C)$ function[1] in the algorithm is taken from the Instance Store API [**?**] and it returns a set of individual names to a given concept C. The
$getSuccessors(x, R)$ function in the algorithm returns a set of individual names $\{y_1, y_2, \cdots, y_n\}$, for each y_i, the relation $associated_{\mathcal{A}}(x, R, y_i)$ holds in the ABox.

6.7 Related Work

As already mentioned, the idea of supporting DL style reasoning using databases is not new. One example is [BB93], where an architecture and algorithms are presented which can handle DL inference problems by converting them into a collection of SQL queries.

[1]Note that it only takes into account the individuals' concept assertions and ignores their role assertions.

Algorithm 1 $queryOrientedRetrieve(Concept\ Q) : Set$

```
1: results ← $\emptyset$
2: results ← $retrieve(Q)$
3: for each $D_i \in Q$ do
4:    $candidateIndividualSet \leftarrow retrieve(\nexists(D_i)) \setminus$ results
5:    for each $x \in candidateIndividualSet$ do
6:       for each $\exists S.V \in \{\exists(D_i)\}$ do
7:          $successorSet$ = getSuccessors($x, S$)
8:          if isEmptySet($successorSet \cap queryOrientedRetrieve(V)$) then
9:             $candidateIndividualSet \leftarrow candidateIndividualSet \setminus x$
10:            Break
11:         end if
12:      end for
13:   end for
14:   results ← results $\cup\ candidateIndividualSet$
15: end for
16: return results
```

This approach is not limited to role-free ABoxes, but the DL language supported is much less expressive, and the database schema must be customised according to the structure of the given TBox.

Another example is the Parka system [ASH95]. Parka is not limited to role-free ABoxes and can deal with very large ABoxes. However, Parka also supports a much less expressive description language, and is not based on standard DL semantics, so it is not really comparable to the Instance Store.

Finally, [Sch94] describes a "semantic indexing" technique that is very similar to the approach used in the Instance Store except that files and hash tables are used instead of database tables, and optimisations such as the use of equivalence sets were not considered.

6.8 Discussion

Our experiments show that the Instance Store provides stable and effective reasoning for role-free ABoxes, even those containing very large numbers of individuals. In contrast, full ABox reasoning using the RACER system exhibited accelerating performance degradation with increasing ABox size, and was not able to deal with the larger ABoxes used in this test. (It may be possible to fix this problem by changing system parameters, but we had no way to investigate this.) The pseudo-individual approach to role-free ABox reasoning was more promising, and may be worth further investigation. It does not, however, have the Instance Store's advantage of ABox persistence, and it appears to be less likely to scale to even larger ABoxes: it does not cope well with large answer sets, and is inherently limited by the fact that DL reasoners (at least in current implementations) keep the entire TBox in memory. Moreover, it is not clear how the pseudo-individual approach could be extended to deal with ABoxes that are not role-free.

The acceptability of the Instance Store's performance would obviously depend on the nature of the application and the characteristics of the KB and of typical queries. It is likely that the performance of the Instance Store can be substantially improved simply by dealing with constant factors such as communication overheads—in the current implementation, communication overheads between the Instance Store and the DL reasoner account for nearly half the time taken to answer queries that require significant amounts of DL reasoning to compute the answer (i.e., when I_2 is large). It may also be possible to improve the performance of the database, e.g., using techniques such as indexing and clustering, or by reformulating queries.

As well as dealing with the above mentioned performance bottlenecks, future work will include the investigation of additional optimisations and enhancements. Possible optimisations include *semantic indexing feedback*—adding new indexing concepts to the ontology for the purpose of query optimisation; *description canonicalisation*—canonicalising the descriptions passed to the Instance Store, so that equivalent descriptions can be more effectively identified; *cardinality estimation*—estimating the cardinality of the result (and in particular of I_2) before executing a query, and giving users the chance to refine queries if the cost of answering them is likely to be very high; and *result caching*—caching the results of queries and of DL subsumption tests in order to avoid DL reasoning when answering subsequent queries. Possible enhancements include providing a more sophisticated query interface with support for, e.g., conjunctive queries [Tes97].

As discussed in Section 6.6, we are currently engaged in extending the Instance Store to deal with ABoxes that are not role-free. The impact that this will have on performance is likely to be heavily dependent on the structure of the given ABox. In particular, the Instance Store is not likely to perform well with ABoxes that result in highly non-deterministic precompletions. ABoxes that are highly interconnected and/or contain many cyclical connections are also likely to have an adverse affect on performance. An evaluation of the effectiveness of the extended Instance Store will therefore have to wait for the completion of the prototype, and on the development of application ontologies containing large numbers of individuals—currently these are in rather short supply, but we hope that development of such ontologies will be encouraged by the existence of the extended Instance Store.

Acknowledgements.

Thanks to Phil Lord for help with the implementation and to Chris Wroe for help with the GO ontology and the formulation of realistic queries.

Chapter 7

Optimising Instance Realisation — an Idea

In order to speed up the instance retrieval InstanceStore described in the previous chapter restricts the expressiveness of the A-Box to instances without any relationship to other instances in so-called role-free A-Box. The aim of this restriction is to be able to use database technologies for answering description logic queries.

However, InstanceStore can use database functionality not for every query. If a query is classified to the top element, for example, the proposed algorithm from InstanceStore must fall back into the traditional query answering procedure where every instance is checked deductively. This inference is known as the *instance realisation*. Instance realisation seems to be needed especially in the case where the query contains disjunctions — an separating feature of description logics. It is obvious that in such cases instance realisation is very inefficient even for very large sets of instances because every instance must be checked.

In this chapter we discuss the opportunity to optimise instance realisation. In cases where database technology can not be used the idea of our approach is different from traditional methods, where a specific goal — the implication between instance and query — is proven. Instead we propose a data-driven approach where all instances are assigned to the most specific concepts based on the available knowledge in the A-Box before the first query is sent to the system. The process of assigning instances to most specific concepts continues during the query answering. In this way the system is continuously optimised for instance realisation leading to a dynamic behaviour of the system.

In the following we explain this process using an example before we give details about the underlying theory.

7.1 A motivating example

For the example the domain of family relationship is used. We assume that the reader is familiar with the notion of description logics. Suppose the following small and simplified ontology together with the instances is given:

$$\begin{array}{rcl}
Woman & \equiv & Human \sqcap Female \\
Man & \equiv & Human \sqcap Male \\
Mother & \equiv & Woman \sqcap \exists Child\colon Human \\
MotherOfOnlySons & \equiv & Woman \sqcap \forall Child\colon Man \\
Father & \equiv & Man \sqcap \exists Child\colon Human \\
Parent & \equiv & Mother \sqcup Father \\
Grandmother & \equiv & Mother \sqcap \exists Child\colon Parent \\
Granduncle & \equiv & Man \sqcap \exists\, sibling\colon Grandmother \\
 & & \\
A_i & = & \{Woman(anja), Child(anja, nils), Man(nils), \\
 & & Father(fried), sibling(fried, anja)\}
\end{array}$$

Given the A-Box A_i above we can directly conclude that $anja$ must be a $Mother$ because she has a child, $nils$, and $nils$ is a $Human$ because every Man is a $Human$. Of course, a normal description logic reasoner (DLR) will not deduce it at the moment. Only if a query is sent to the system asking if $anja$ belongs to $Mother$ then the system will answer with yes (and perhaps store this result in its internal database). But normally such a query is a part of a sequence where the application tries to find out to which concepts $anja$ belongs. Apart from the question if $anja$ belongs to $Mother$ there must be further queries if $anja$ belongs to Man, $Father$, $Grandmother$, $MotherOfOnlySons$, etc. This leads to a uniformed search where the application tries to find out the most specific concept to which an instance belongs with the help of a sequence of queries. Furthermore if the application wants to retrieve all instances of $Mother$, all (relevant) instances must be checked if the could be assigned to $Mother$ before a DLR can answer the instance retrieval query. The uninformed search and instance retrieval in general makes instance realisation inefficient.

Instead of testing we can reformulate the concept expressions into classification rules. The rule for $Mother$ would be:

$$Mother(X) \longleftarrow Woman(X) \wedge child(X, Y) \wedge Human(Y)$$

With such a rule we can directly conclude that $anja$ is a $Mother$. All conditions are satisfied, i.e. $anja$ is a $Woman$, has as child $nils$ (i.e. $child(anja, nils)$) and $nils$ is at least a $Human$. If similar rules exist for every concept and can be applied to the A-Box knowledge, the application must not guess which the possible concepts of $anja$ are.

The rules seem to be a direct translation of the concept definition into logical rules. However, for the instance realisation it is not possible — and not needed — to translate every (part of) concept definition into its corresponding logical rule. For example we can never infer in A_i that $anja$ is a $MotherOfOnlySons$. $nils$ is her only son and the conditions for a $MotherOfOnlySons$ might be satisfied at the moment. But in future $anja$ can get further children perhaps including a daughter. Or in other words the open world assumption (OWA) prevents the inference from the knowledge in the A-Box that $anja$ has only sons as children. The only way for instance realisation is to wait that $anja$ is assigned to concept term $\forall Child : Man$ — implicitly or explicitly.[1] Therefore the concept definition $MotherOfOnlySons$ can *not* be translated into the obvious *logical* rule which can be used for instance realisation but into a simplified one where the concept term $\forall Child\!:\!Man$ is replaced by the concept instantiation $FCM(X)$ and a new concept definition $FCM \equiv \forall Child\!:\!Man$ is added to the T-Box:

$$MotherOfOnlySons(X) \longleftarrow Woman(X) \wedge FCM(X)$$

The negation and the disjunction must be handled in the same way.

All the information that can be derived from A-Box A_i is now derived. But now the dynamic behaviour of the proposed idea is considered. For this purpose the A-Box will be extended to A_{i+1}:

$$A_{i+1} \quad = \quad A_i \cup \{Father(nils)\}$$

With new information about $nils$ we can trigger that rules which are affected by this new information:

$$Grandmother(X) \longleftarrow Woman(X) \wedge child(X,Y) \wedge Parent(Y)$$

Because $nils$ becomes a $Parent$ (to be precise, $nils$ become a $Father$ which is a specialisation of $Parent$) we now know that $anja$ must become a $Grandmother$. This derived information can be added to the A-Box and again trigger some rules. We can now conclude with the help of following rule

$$Granduncle(X) \longleftarrow Man(X) \wedge sibling(X,Y) \wedge Grandmother(Y)$$

that $fried$ become a $Granduncle$ because $anja$, his sibling, becomes a $Grandmother$. This "chain reaction" imagines the great benefit of the data-driven, dynamic approach: Every time when new information is arrived in the A-Box all consequences are tried to be directly computed and as much as possible inferences from the data is derived.

[1]This is not completely true. The problem can also be solved if an operator is available which says that no further instantiation of a role/property will exists in future.

The rule-oriented approach can further be optimised. In order to safe tests we can partly instantiate the rules even if not all knowledge for satisfying the conditions is present. That parts which are already satisfied can be omitted indicating that these tests are already passed and must not checked again. For example with the A-Box A_i the rules for $Grandmother$ and for $Granduncle$ can already be instantiated to

$$\begin{array}{llcl}
 & Grandmother(anja) & \longleftarrow & Woman(anja) \wedge child(anja, nils) \wedge Parent(nils) \\
\Longrightarrow & Grandmother(anja) & \longleftarrow & Parent(nils) \\
 & Granduncle(fried) & \longleftarrow & Man(fried) \wedge sibling(fried, anja) \wedge Grandmother(anja) \\
\Longrightarrow & Granduncle(fried) & \longleftarrow & Grandmother(anja)
\end{array}$$

Now $anja$ seems to “wait” for $nils$ to become itself a $Parent$ in order to become a $Grandmother$. $fried$ is waiting for $anja$ to become a $Grandmother$. However the rule instantiation may imply that a lot of rules must be instantiated for one instance. With large sets of instances a still larger set of instantiated rules must be stored and maintained which may have a detrimental effect to the optimisation with rule instantiation.

This small example demonstrates two main characteristics of the proposed idea. First it shows the data-driven behaviour. Instead of waiting for some queries the system directly computes the most specific concepts to which an instance can belong and prevents some uninformed search for the application which uses this system. Second it shows the dynamic behaviour. New information can (monotonically) be added when they appear and the consequences are tried to be derived directly.

7.2 The Representation Formalism

After this motivating example the representation formalism for the data-driven instance realisation will now be introduced. As already mentioned it has a strong relationship to rule formalism. However, the dynamic nature should also be reflected by the formalism. One adequate method to model dynamic behaviour is an event-driven approach. For the instance realisation such an event indicates the arrival of new information either by the application or by the instance realisation process itself. The new information that $anja$ becomes a $Grandmother$ and $fried$ a $Granduncle$ are two examples for such system-generated events.

System-generated events may also be interesting for the application which uses the DLR. In order to keep informed about the new derived information the events can also be sent to the application.[2] Then the application will be informed by the DLR if an instance was assigned to a more specific concept.

[2]Of course this functionality extends the current available interfaces like DIG.

In order to become a little bit more technical events are generated and consumed. For both the following notation is introduced:

$\lfloor . \rfloor$	checks if the event has appeared. $\lfloor X : C \rfloor$ looks for an instance X which is associated to concept C. $\lfloor R(X,Y) \rfloor$ is the corresponding event check that the instances X and Y is related through R.
$\lceil . \rceil$	generates an event if not already generated in the past. $\lceil X : C \rceil$ says that instance X now belongs to C. $\lceil R(X,Y) \rceil$ is the corresponding event generator for the relation R.

Both notations can be combined to more complex event terms by the usual logical connectives $\wedge$ and $\vee$. Furthermore the logical implication $\rightsquigarrow$[3] can be used to connect events in order to formulate conditions for event checking or generating. For example, $\lfloor R(X,Y) \rfloor \rightsquigarrow \lceil Y : C \rceil$ says, that the event $\lceil Y : C \rceil$ is only be generated if the instances X and Y are related trough R, i.e. the event $\lfloor R(X,Y) \rfloor$ was observed. Both complex event terms constitute both sides of an event rule. To be more precise:

$\psi \longleftarrow \phi$	is an *event rule* which generates the events in $\psi = ... \circledast \lceil \delta \rceil \circledast ...$ when the events in $\phi = ... \circledast \lfloor \delta \rfloor \circledast ...$ are observed. $\circledast$ represents one of the following connectives: $\wedge$, $\vee$, or $\rightsquigarrow$.

The most interesting question now is how an ontology can be translated into this event-based rule formalism. The translation will be explained in the next section.

7.3 Translating the Ontology into the Formalism

The event rules are generated from the terminological axioms in the ontology where every axiom of the form $C \equiv D$ or $C \sqsubseteq D$ will be translated into a set of rules. Before the translation can begin the concept term must be transformed into disjunctive normal form. Furthermore like for the negation normal form it is assumed that the negation is propagated to the innermost terms, i.e. the negation only appears together with a concept name.[4], i.e. $C \equiv D_1 \sqcup ... \sqcup D_n$ resp. $C \sqsubseteq D_1 \sqcup ... \sqcup D_n$ with $D_i = D_{i1} \sqcap ... D_{im_i}$. For the case of $C \equiv D_1 \sqcup ... \sqcup D_n$ the translation function τ forms for each combination of C and D_i an event rule $\tau_{\lceil\rceil}(X : C) \leftarrow \tau_{\lfloor\rfloor}(X : D_i)$ as shown in table 7.1.

$C \sqsubseteq / \equiv D$	$\tau(.) =$
$C \equiv D_1 \sqcup ... \sqcup D_n$	$\tau_{\lceil\rceil}(X : C) \leftarrow \tau_{\lfloor\rfloor}(X : D_1), ..., \tau_{\lceil\rceil}(X : C) \leftarrow \tau_{\lfloor\rfloor}(X : D_n)$
$C \sqsubseteq D_1 \sqcup ... \sqcup D_n$	—

Table 7.1: Translation τ for axioms

If the events in the conditions $\tau_{\lfloor\rfloor}(X : D_i)$ of the event rule are observed then the events

[3]We use a different notion for implication in order to distinguish it from the rules

[4]The normal form preserves the satisfiability property and must not be visible for the application or an user.

$\tau_{\sqcap}(X : C)$ are generated for the common variable X. Which events must be observed resp. generated is determined by the translation function $\tau_{\sqcap}(.)$ resp. $\tau_{\sqcup}(.)$ depending on the variable X which is associated to D_i resp. C.

For the case of $C \sqsubseteq D_1 \sqcup ... \sqcup D_n$ no translation exists because the axioms only defines necessary but not sufficient conditions $D_1 \sqcup ... \sqcup D_n$ that an instance X must satisfy. Obviously only sufficient conditions can be translated into event rules (if you conclude from $D_1 \sqcup ... \sqcup D_n$ to C).

Data-driven instance realisation has one great advantage in opposite to inferences like satisfiability checking: it must not discuss all (hypothetical) cases during a proof by cases but can derive on explicitly known facts. Because of the inherent open world semantics a realisation of an instance to a more specific concept can only be done if all required information is available in the A-Box. For example a constraint $\forall R : C$ can never be guaranteed by a data-driven instance realisation itself because a relation can be added every time in the future which violates that concept term (see also section 7.1). To satisfy this concept term the only way is to tell the A-Box explicitly that the concept term is fulfilled. Because of the restricted A-Box formalism this can only be done for an instance X if X belongs to a concept FRC_{new} (i.e. $X : FRC_{new}$) which is defined as $FRC_{new} \equiv \forall R{:}C$.

FRC_{new} is a artificial concept definition added to the T-Box which normally is not seen by an user or application. It is not expected that the user explicitly associate an instance X to that concept FRC_{new} but to some subsumers of FRC_{new}. For example if $anja$ is told belonging to $MotherOfOnlySons$ (i.e. $anja : MotherOfOnlySons$) and $MotherOfOnlySons$ is a specialisation of $FRC_{new} \equiv \forall child{:}Man$ then we also know that the concept term $\forall child{:}Man$ is satisfied by $anja$.

Such artificial concept definitions will appear in several situations during the translation with $\tau_{\sqcup}(.)$ resp. $\tau_{\sqcap}(.)$. Table 7.2 shows the translation of concept terms by $\tau_{\sqcup}(.)$ and defines the events which must be observed in order to satisfy an A-Box expression. C and D are concept terms, CN is a concept name, R a role and F a attribute. Note that currently the translation is restricted to ALC, an expressive but restricted subset of OWL DL. The extension of the translation to OWL DL needs further investigation.

$X : CN$ and $X : C \sqcap D$ is translated in obvious way to $\lfloor X : CN \rfloor$ and $\tau_{\sqcup}(X : C) \wedge \tau_{\sqcup}(X : D)$. Because the concept terms are not unfold (cf. [BMNPS02]) we must ensure that an event $\lceil X : EN \rceil$ which is generated for a specialisation EN of CN can also be caught by $\lfloor X : CN \rfloor$. Therefore the following event rules are virtually added to the system:

$$\lceil X : CN \rceil \leftarrow \lfloor X : EN \rfloor \quad \text{where } EN, CN \text{ are concept names with} \quad EN \sqsubseteq CN$$

For $X : C \sqcup D$ and $X : \neg C$ there is no possibility to check these conditions directly and an artificial concept definition must be generated. When X is associated to a subsumer of $C \sqcup D$ resp. $\neg C$ then the events can be caught. $\exists R : C$ resp. $\exists F : C$ are also translated

	$\tau_{\lfloor\rfloor}(.) =$	Remarks
$X : CN$	$\lfloor X : CN \rfloor$	
$X : C \sqcap D$	$\tau_{\lfloor\rfloor}(X : C) \wedge \tau_{\lfloor\rfloor}(X : D)$	
$X : C \sqcup D$	$\lfloor X : COD_{new} \rfloor$	$COD_{new} \equiv C \sqcup D$
$X : \neg CN$	$\lfloor X : DN_{new} \rfloor$	$DN_{new} \equiv \neg CN$
$X : \exists R{:}C$	$\lfloor R(X,Y) \rfloor \wedge \tau_{\lfloor\rfloor}(Y : C)$	
$X : \exists F{:}C$	$\lfloor F(X) = Y \rfloor \wedge \tau_{\lfloor\rfloor}(Y : C)$	
$X : \forall R{:}C$	$\lfloor X : FRC_{new} \rfloor$	$FRC_{new} \equiv \forall R{:}C$
$X : \forall F{:}C$	$\lfloor X : FFC_{new} \rfloor \vee (\lfloor F(X) = Y \rfloor \wedge \tau_{\lfloor\rfloor}(Y : C))$	$FFC_{new} \equiv \forall F{:}C$

Table 7.2: Event checking translation by $\tau_{\lfloor\rfloor}(.)$

obviously: two events must be observed telling that X is related to Y, i.e. $\lfloor R(X,Y) \rfloor$ resp. $\lfloor F(X) = Y \rfloor$, and instance Y belongs to C, i.e. $\tau_{\lfloor\rfloor}(Y : C)$. $X : \forall R{:}C$ is translated with help of an artificial concept definition but $X : \forall F : C$ for the attribute/function F can also be checked directly. In difference to the role R there is a number restriction for attributes: they can only be instantiated for one value. So if there is such a value Y for that attribute F and if this value Y belongs to the concept term C then $\forall F{:}C$ is satisfied.

	$\tau_{\lceil\rceil}(.) =$	Remarks
$X : CN$	$\lceil X : CN \rceil$	
$X : C \sqcap D$	$\tau_{\lceil\rceil}(X : C) \wedge \tau_{\lceil\rceil}(X : D)$	
$X : C \sqcup D$	$\lceil X : COD_{new} \rceil$	$COD_{new} \equiv C \sqcup D$
$X : \neg CN$	$\lceil X : DN_{new} \rceil$	$DN_{new} \equiv \neg CN$
$X : \exists R{:}C$	—	
$X : \exists F{:}C$	$\lfloor F(X) = Y \rfloor \rightsquigarrow \tau_{\lceil\rceil}(Y : C)$	
$X : \forall R{:}C$	$\lceil X : FRC_{new} \rceil \wedge (\lfloor R(X,Y) \rfloor \rightsquigarrow \tau_{\lceil\rceil}(Y : C))$	$FRC_{new} \equiv \forall R{:}C$
$X : \forall F{:}C$	$\lceil X : FFC_{new} \rceil \wedge (\lfloor F(X) = Y \rfloor \rightsquigarrow \tau_{\lceil\rceil}(Y : C))$	$FFC_{new} \equiv \forall F{:}C$

Table 7.3: Event generation translation by $\tau_{\lceil\rceil}(.)$

Table 7.3 defines how events are generated from complex terms by the translation function $\tau_{\lceil\rceil}(.)$. The first four conditions, $X : CN$, $X : C \sqcap D$, $X : C \sqcup D$, and $X : \neg CN$ now generate these events that may be observed by the event checking translation. But it may surprise that the translation of $X : \exists R{:}C$ will be empty. Because the role R has no number restriction it is always possible to generate an instance which belongs to C. The new generated instance can be absolutely independent from the instances Y which are currently related with X trough R. Therefore no event must be generated for existing instances. However, the attribute/function F has an implicit number restriction. Therefore the translation of $X : \exists F{:}C$ looks for the value Y of the function F and if Y exists then an event is generated that Y now belongs to C; if the value Y does not exists then the event will not be generated. A mixture is the translation for the $\forall$ constructor in $X : \forall R{:}C$ and $X : \forall F{:}C$. First they generate an event for the artificial concept definition in order to be able to catch them. Second they generate events for all values Y of R and F if they exist.

The rules in table 7.1 are generated from the concept axiom and formulate conditions D when an instance X can be classified to C. However these rules do not express all possibilities for an instance realisation. Suppose that the A-Box A_{i+1} in the example of section 7.1 is further extended to A_{i+2} by the information that $anja$ has an further child $jens$, i.e. $A_{i+2} = A_{i+1} \cup \{Child(anja, jens)\}$. When we further add that $anja$ becomes a $MotherOfOnlySons$, i.e. $A_{i+3} = A_{i+2} \cup \{anja : MotherOfOnlySons\}$, then we can conclude that $jens$ must be a Man. However, we can extend the A-Box in the opposite order, i.e. $A'_{i+2} = A_{i+1} \cup \{anja : MotherOfOnlySons\}$ and $A'_{i+3} = A'_{i+2} \cup \{Child(anja, jens)\}$, which also has as consequence that $jens$ must be a Man.

Both ways of extensions demonstrate two further possibilities for instance realisation based on events in the A-Box and are not covered by the rules in table 7.1. For the extension of A_{i+2} and A_{i+3} the realisation of $jens$ is triggered by the new information about $anja$ because $jens$ is related to $anja$. In more general terms the new classification of X to C lead to the realisation to some part of D. Now the events are propagated in the opposite order from C to D. Such event rules are illustrated in table 7.4. For both kind of axioms the same event rule is generated.

$C \sqsubseteq / \equiv D$	$\tau(.) =$
$C \equiv D_1 \sqcap ... \sqcap D_n$	$\tau_{\sqcap}(X : D_1) \wedge ... \wedge \tau_{\sqcap}(X : D_n) \longleftarrow \tau_{\sqcup}(X : C)$
$C \sqsubseteq D_1 \sqcap ... \sqcap D_n$	

Table 7.4: Further translation τ for axioms

Please note that for this translation the concept term D must be transformed into a different normal for, the conjunctive normal form.

The second way of extensions the realisation of $jens$ is invoked by introducing a new relationship $Child$ to $anja$. Because of this new relationship in general $anja$ and $jens$ may be subject of an instance realisation. The check that $anja$ needs further refinement is covered by the event rules in table 7.1. But the possibility of realisation for $jens$ is not checked by any rule. Therefore further rules are needed which are shown in table 7.5.

$C \sqsubseteq / \equiv D$	$\tau(.) =$
$C \sqsubseteq D_1 \sqcup ... \sqcup D_n$ $C \equiv D_1 \sqcup ... \sqcup D_n$	$\tau_{\sqcap}(Y : E) \longleftarrow \tau_{\sqcup}(X : C) \wedge \lfloor f(X) = Y \rfloor$ with $D_i = ... \sqcap \exists f : E \sqcap ...$ and f is a property or $\tau_{\sqcup}(X : C) \wedge \lfloor R(X, Y) \rfloor \rightarrow \tau_{\sqcap}(Y : E)$ with $D_i = ... \sqcap \forall R : E \sqcap ...$ for roles and properties

Table 7.5: Further translation τ for axioms

The first two translations of an axiom are not surprising. They have their correspondence in the logical translation. From a logical point of view the third translation is not needed and redundant. But together with the event mechanism such rules are desirable. The logical reason is that free variables in the heads of the rules are all-quantified. Instead realising it with the help of the inference mechanism the third kind of rules is introduced in order to simplify the implementation of the reasoning service.

With the artificial concept definitions some helper constructions are introduced which put some additional knowledge into the subsumption hierarchy. The use that knowledge from the subsumption hierarchy is characteristics for the proposed idea of optimising instance realisation. The subsumption hierarchy must be computed by the normal description logic reasoner — perhaps before any instance realisation can be performed. The a priori computation can be interpreted as a pre-compilation of the knowledge base in order to perform specialised reasoning, e.g. in this case instance realisation. Furthermore the need for a complete DLR indicates that the proposed data-driven instance realisation is not suitable to replace any normal DLR reasoning. Instead it can only optimise

Chapter 8

Conclusion

In this report, we have investigated the problems of query answering for Semantic Web query languages (such as RDF, OWL DL and OWL-E) in the OWL-QL specification. Key features of the OWL-QL specification are summarised in Section 3.1.

Theoretical results are mainly on query answering with RDF graphs and OWL-E ontologies. In Chapter 2, we recast the RDF model theory in a more classical logic framework. Given an RDF graph S and a query Q, the answer set of Q to S (as defined by [Hay04b]) is the same as the certain answer of Q to S given the empty KB. In other words, an RDF graph can be transformed to a DL ABox; therefore, OWL-QL servers (such as the one described in Chapter 3) can be used to support query answering w.r.t. RDF graphs. In Chapter 4, we extend OWL-QL to OWL-E-QL, so as to support conjunctive queries with datatype expression atoms. We have shown that, under certain restrictions, query answering w.r.t. OWL-E ontologies can be reduced to ABox reasoning (such as knowledge base satisfiability, instance checking or instance retrieval).

In addition to the above theoretical results, we also present some implementation results. Chapter 3 escribes how to implement query answering for the $\mathcal{SHIQ}$ DL in an OWL-QL server. As query answering can be reduced to ABox reasoning and instance retrieval is the expensive problem (among the three ABox reasoning problems mentioned above), we have further investigated optimisation techniques for instance retrieval based on a hybrid DL/Database architecture (Chapter 6).

As for future work, we will further look at how to apply our result to support the SPARQL language. In addition, we would like to investigate further implementation and optimisation issues on query answering for Semantic Web query languages (such as RDF, OWL DL and OWL-E) in the OWL-QL specification.

Bibliography

[ANS92] Information technology — database languages — SQL: ISO/IEC 9075:1992, 1992. New York.

[AP00] A.Morris and P.Jankowski. Combining fuzzy sets and databases in multiple criteria spatial decision making. pages 103–116, 2000.

[ASH95] W. A. Andersen, K. Stoffel, and J. A. Hendler. Parka: Support for extremely large knowledge bases. In G. Ellis, R. A. Levinson, A. Fall, and V. Dahl, editors, *Knowledge Retrieval, Use and Storage for Efficiency: Proceedings of the First International KRUSE Symposium*, pages 122–133, 1995.

[BB93] Alexander Borgida and Ronald J. Brachman. Loading data into description reasoners. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 217–226, 1993.

[BCM$^+$03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[BE01] Paul V. Biron and Ashok Malhotra (Eds.). XML schema part 2: Datatypes, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Pub Co, Nov 8 2002.

[Bec03a] Sean Bechhofer. The DIG Description Logic interface: DIG/1.1. Technical report, University of Manchester, Oxford Road, Manchester M13 9PL, Feb 7 2003. `http://dl-web.man.ac.uk/dig/2003/02/interface.pdf`.

[Bec03b] Sean Bechhofer. The DIG Description Logic Interface: DIG/1.1. URL `http://dl-web.man.ac.uk/dig/2003/02/interface.pdf`, Feb 2003.

[Bec04] RDF/XML syntax specification. URL, Feb 10 2004. `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`.

[Ber98] RFC 2396: Uniform Resource Identifiers (URI): Generic syntax. URL, August 1998.

[BF82] B.P Buckles and F.E.Petry. A fuzzy representation of data for relational databases. *Fuzzy Sets and Systems*, 7:213–226, 1982.

[Bir01] XML schema part 2: Datatypes. URL, May 2 2001. `http://www.w3.org/TR/xmlschema-2/`.

[BMNPS02] F. Baader, D. L. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *Description Logic Handbook: Theory, implementation and applications*. Cambridge University Press, 2002.

[Boa03] XQuery 1.0: An XML query language. URL, Nov 12 2003. `http://www.w3.org/TR/2003/WD-xquery-20031112/`.

[Bra04] Extensible markup language (XML) 1.0 (third edition). URL, Feb 04 2004. `http://www.w3.org/TR/2004/REC-xml-20040204`.

[BTMS04] M. Bada, D. Turi, R. McEntire, and R. Stevens. Using Reasoning to Guide Annotation with Gene Ontology Terms in GOAT. *SIGMOD Record (special issue on data engineering for the life sciences)*, June 2004. To appear – available at `http://www.cs.man.ac.uk/~dturi/papers/goat.pdf`.

[CDT04] Olga Caprotti, Mike Dewar, and Daniele Turi. Mathematical service matching using Description Logic and OWL. Technical Report IST-2001-34145, MONET Consortium, March 2004. Available at `http://www.cs.man.ac.uk/~dturi/papers/monet_onts.ps`.

[CGM$^+$04] 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`.

[CKW93] Weidong Chen, Michael Kifer, and David Warren. HILOG: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[Cor92] OMRON Corporation. *Fuzzy LUNA-Fuzzy Database System Library User's Manual and Fuzzy LUNA-Fuzzy Database Reference manual*. 1992.

[DCv$^+$02] Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language 1.0 reference, July 2002. Available at `http://www.w3.org/TR/owl-ref/`.

[DD90] D.Li and D.Liu. A fuzzy prolog database system. 1990.

[DEG$^+$03] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R. Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A. Tomlin, and Jason Y. Zien. Semtag and seeker: Bootstrapping the semantic web via automated semantic annotation. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, 2003.

[Dic04] 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`.

[DND99] D.papadias, N.Karacapilidis, and D.Arkoumanis. Processing fuzzy spatial queries: A configuration similarity approach. 1999.

[FHH03] Richard Fikes, Patrick Hayes, and Ian Horrocks. OWL-QL - a language for deductive query answering on the semantic web. Technical report, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 2003.

[Fik03] DAML Query Language (DQL) abstract specification. URL, Apr 2003. http://www.daml.org/2003/04/dql/.

[FJF03] Richard Fikes, Jessica Jenkins, and Gleb Frank. JTP: A system architecture and component library for hybrid reasoning. Technical report, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 2003. ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-03-01.pdf.

[FMM$^+$04] Mary Fernndez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh (Eds.). XQuery 1.0 and XPath 2.0 data model, Oct 2004. http://www.w3.org/TR/2004/WD-xpath-datamodel-20041029/.

[fSI96] 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.

[GO] GO project. European Bioinformatics Institute. http://www.ebi.ac.uk/go.

[Go 03] Gene Ontology Database, 2003. Available at http://www.godatabase.org/dev/database/.

[Hay04a] Patrick Hayes. RDF semantics. Technical report, W3C, February 2004. W3C recommendation, URL http://www.w3.org/TR/rdf-mt/.

[Hay04b] Patrick Hayes. RDF Semantics. Technical report, W3C, Feb 2004. W3C recommendation, URL http://www.w3.org/TR/rdf-mt/.

[HBEV04] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of RDF query languages. Oct 2004.

[HM01a] V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, 2001.

[HM01b] Volker Haarslev and Ralf Möller. RACER system description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 701–705. Springer, 2001.

[HMW04] Volker Haarslev, Ralf Möller, and Michael Wessel. RACER user's guide and reference manual, version 1.7.19. URL, http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-manual-1-7-19.pdf, April 2004.

[Hol96] Bernhard Hollunder. Consistency checking reduced to satisfiability of concepts in terminological systems. *Ann. of Mathematics and Artificial Intelligence*, 18(2–4):133–157, 1996.

[Hor97] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[Hor98] I. Horrocks. Using an Expressive Description Logic: FaCT or Fiction? pages 636–647, 1998.

[Hor03] I. Horrocks. Implementation and optimisation techniques. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press, 2003.

[HPS03] Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In *Proc. of the 2nd International Semantic Web Conference (ISWC)*, 2003.

[HPSB$^+$04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML, Apr 2004. http://www.daml.org/2004/04/swrl//.

[isw] Instance Store website. `http://instancestore.man.ac.uk`.

[JZ86] J.Kacprzyk and A Ziolkowski. Data base queries with fuzzy linguistic quantifiers. *IEEE Trans Systems, Man and Cybernetics*, 16:474–478, 1986.

[KAC$^+$02] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A declarative query language for RDF. In *Proceedings of the eleventh international conference on World Wide Web*, pages 592–603. ACM Press, New York, USA, May 7–11 2002.

[LH03] Lei Li and Ian Horrocks. A Software Framework For Matchmaking Based on Semantic Web Technology. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 331–339. ACM, 2003.

[MCG$^+$93] M.Zhang, C.Yu, G.Wang, T.Phamand, and H.Nakajima. A relational modelfor imprecise queries. *International Symposium on Methodologies in Inteligent Systems*, 1993.

[Min75] Marvin Minsky. A framework for representing knowledge. In Patrick J. Winston, editor, *The psychology of computer visions*, pages 211–277. McGraw-Hill, New York, 1975.

[MK85] M.Zemankova and A. Kandel. Implementing imprecision in information systems. *Information Sciences*, pages 107–141, 1985.

[MME04] Ashok Malhotra, Jim Melton, and Philip Wadler (Eds.). XQuery 1.0 and XPath 2.0 functions and operators, Jul 2004. http://www.w3.org/TR/2004/WD-xpath-functions-20040723/.

[Pan04] Jeff Z. Pan. *Description Logics: Reasoning Support for the Semantic Web*. PhD thesis, School of Computer Science, The University of Manchester, Oxford Rd, Manchester M13 9PL, UK, 2004.

[PFT$^+$04] Jeff Z. Pan, Enrico Franconi, Sergio Tessaris, Giorgos Stamou, Vassilis Tzouvaras, Luciano Serafini, Ian Horrocks, and Birte Glimm. Specification of Coordination of Rule and Ontology Languages. Technical report, The Knowledge Web project, June 2004.

[PH88] MGalibourg P.Bosc and G Hamon. Fuzzy quering with sql: Extensions and implementation aspects. *Fuzzy Sets and Systems*, 28:333–349, 1988.

[PH04] Jeff Z. Pan and Ian Horrocks. OWL-E: Extending OWL with Expressive Datatype Expressions. Technical report, School of Computer Science, the University of Manchester, April 2004.

[PS04] Eric Prud'hommeaux and Andy Seaborne(Eds.). SPARQL query language for RDF, Oct 2004. http://www.w3.org/TR/2004/WD-rdf-sparql-query-20041012/.

[QCJ$^+$95] Q.Yang, C.Liu, J.Wu, C.Yu, S.Dao, and H.Nakajima. Efficient processing of nested fuzzy sql queries. *In proceedings of the 11th International Conference on Data Engineering*, 1995.

[Qui68] M. R. Quillian. Semantic memory. In Marvin Minsky, editor, *Semantic Information Processing*, pages 216–270. The MIT Press, 1968.

[QWC$^+$93] Q.Yang, W.Zhanng, C.Luo, C.Yu, and H.Nakajima. Unnesting fuzzy sql queries in fuzzy databases. *Workshop on Incompleteness and Uncertainty in Information Systems*, pages 68–73, 1993.

[Rec04] A. Rector. Re: [UNITS, OEP] FAQ : Constraints on data values range. URL `http://lists.w3.org/Archives/Public/public-swbp-wg/2004Apr/0216.html`, Apr. 2004. Discussion in the `public-swbp-wg@w3.org` mailing list.

[SA90] S.Shenoi and A.Melton. An extended version of the fuzzy relational database model. *Information Sciences*, 52:35–52, 1990.

[Sch94] A. Schmiedel. Semantic indexing based on description logics. In F. Baader, M. Buchheit, M.A. Jeusfeld, and W. Nutt, editors, *Reasoning about structured objects: knowledge representation meets databases. Proceedings of the KI'94 Workshop KRDB'94*. CEUR (`http://ceur-ws.org/`), September 1994.

[SSS91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[Tes97] Sergio Tessaris. *Questions and answers: reasoning and querying in Description Logic.* PhD thesis, The University of Manchester, 1997. URL`http://www.cs.man.ac.uk/~tessaris/papers/phd-thesis.ps.gz`.

[Tes01] Sergio Tessaris. *Questions and answers: reasoning and querying in Description Logic*. Phd thesis, University of Manchester, 2001.

[The00] The Gene Ontology Consortium. Gene ontolgy: Tool for the unification of biology. *Nature Genetics*, 25(1):25–29, 2000.

[Tho01] XML schema part 1: Structures. URL, May 2 2001. `http://www.w3.org/TR/xmlschema-1/`.

[UCD$^+$03] Michael Uschold, Peter Clark, Fred Dickey, Casey Fung, Sonia Smith, Stephen Uczekaj Michael Wilke, Sean Bechhofer, and Ian Horrocks. A semantic infosphere. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, number 2870 in Lecture Notes in Computer Science, pages 882–896. Springer, 2003.

[WCBN95] W.Zhang, C.Yu, B.Reagan, and H Nakajima. Context -dependent interpretations of linguistic terms in fuzzy relational databases. *In proceedings of the 11th International Conference on Data Engineering*, 1995.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM archive*, 20:822–823, November 1977.

[Zad65] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.