

RacerPro User's Guide
Version 1.9.2

Racer Systems GmbH & Co. KG
<http://www.racer-systems.com>

October 18, 2007

Contents

1	Introduction	1
1.1	Views on RacerPro	1
1.1.1	RacerPro as a Semantic Web Reasoning System and Information Repository	1
1.1.2	RacerPro as a Description Logic System	2
1.1.3	RacerPro as a Combination of Description Logic and Specific Relational Algebras	4
1.2	Application Areas and Usage Scenarios	4
1.3	Racer Editions, Installation, Licenses, and System Requirements	5
1.3.1	Editions	5
1.3.2	Installation	6
1.3.3	Licenses	6
1.3.4	System Requirements	6
1.4	Acknowledgments	6
2	Using RacerPro	9
2.1	Sample Session with RacerPorter	9
2.2	The RacerPro Server	18
2.2.1	The File Interface	18
2.2.2	TCP APIs	20
2.2.3	Web Service Interface	22
2.2.4	HTTP Interface: DIG Interface	22
2.2.5	Options for the RacerPro Server	23
2.3	How to Send Bug Reports	24
2.4	RacerPorter	25
2.4.1	Preferences	26
2.4.2	RacerEditor	27
2.4.3	Tabs in RacerPorter	28

2.4.4	Known Problems	29
2.5	Other Graphical Client Interfaces for RacerPro	29
2.5.1	RICE	29
2.5.2	Protégé	31
2.5.3	Using Protégé and RacerPorter in Combination	38
2.5.4	TopBraidComposer	43
2.6	SWRL: Semantic Web Rule Language	44
3	RacerPro Knowledge Bases	55
3.1	Naming Conventions	55
3.2	Concept Language	56
3.3	Concept Axioms and T-boxes	59
3.4	Role Declarations	60
3.5	Concrete Domains	61
3.6	Concrete Domain Attributes	65
3.7	Individual Assertions and A-boxes	65
3.8	Inference Modes	66
3.9	Retraction and Incremental Additions	67
4	Description Logic Modeling with RacerPro	69
4.1	Representing Data with Description Logics (?)	69
4.2	Nominals or Concrete Domains?	70
4.3	Open-World Assumption	72
4.4	Closed-World Assumption	73
4.5	Unique Name Assumption	73
4.6	Differences in Expressivity of Query and Concept Language	73
4.7	OWL Interface	74
5	Knowledge Base Management	77
5.1	Configuring Optimization Strategies	77
5.2	The RacerPro Persistency Services	78
5.3	The Publish-Subscribe Mechanism	79
5.3.1	An Application Example	79
5.3.2	Using JRacer for Publish and Subscribe	84
5.3.3	Realizing Local Closed-World Assumptions	85

6	The New RacerPro Query Language - nRQL	87
6.1	The nRQL Language	93
6.1.1	Query Atoms, Objects, Individuals, and Variables	93
6.1.2	Query Head Projection Operators	110
6.1.3	Lambda Head Operators to Evaluate Expressions	118
6.1.4	Complex Queries	127
6.1.5	Defined Queries	147
6.1.6	ABox Augmentation with Simple Rules	153
6.1.7	Complex TBox Queries	160
6.1.8	Hybrid Representations with the Substrate Representation Layer . .	167
6.1.9	Formal Syntax of nRQL	193
6.2	The nRQL Query Processing Engine	203
6.2.1	The Query Processing Modi of nRQL	203
6.2.2	The Life Cycle of a Query	205
6.2.3	The Life Cycle of a Rule	207
6.2.4	How to Implement Your Own Rule Application Strategy	208
6.2.5	Configuring the Degree of nRQL Completeness	212
6.2.6	Automatic Deadlock Prevention	217
6.2.7	Reasoning with Queries	220
6.2.8	The Query Repository - The QBox	227
6.2.9	The Query Realizer	232
6.2.10	The nRQL Persistency Facility	234
7	Outlook	237
A	Another Family Knowledge Base	239
B	A Knowledge Base with Concrete Domains	241
C	SWRL Example Ontology	247
D	LUBM benchmark	251
	Index	261

Chapter 1

Introduction

RacerPro stands for **R**enamed **A**Box and **C**oncept **E**xpression **R**easoner **P**rofessional. As the name suggests, the origins of RacerPro are within the area of description logics. Since description logics provide the foundation of international approaches to standardize ontology languages in the context of the so-called semantic web, RacerPro can also be used as a system for managing semantic web ontologies based on OWL (e.g., it can be used as a reasoning engine for ontology editors such as Protégé). However, RacerPro can also be seen as a semantic web information repository with optimized retrieval engine because it can handle large sets of data descriptions (e.g., defined using RDF). Last but not least, the system can also be used for modal logics such as K_m .

1.1 Views on RacerPro

1.1.1 RacerPro as a Semantic Web Reasoning System and Information Repository

The semantic web is aimed at providing “machine-understandable” web resources or by augmenting existing resources with “machine-understandable” meta data. An important aspect of future systems exploiting these resources is the ability to process OWL (Web Ontology Language) documents (OWL KBs), which is the official semantic web ontology language. Ontologies may be taken off the shelf or may be extended for domain-specific purposes (domain-specific ontologies extend core ontologies). For doing this, a reasoning system is required as part of the ontology editing system. RacerPro can process OWL Lite as well as OWL DL documents (knowledge bases). Some restrictions apply, however. OWL DL documents are processed with approximations for nominals in class expressions and user-defined XML datatypes are not yet supported.

A first implementation of the semantic web rule language (SWRL) is provided with RacerPro 1.9 (see below for a description of the semantics of rules in this initial version).

The following services are provided for OWL ontologies and RDF data descriptions:

- Check the consistency of an OWL ontology and a set of data descriptions.

- Find implicit subclass relationships induced by the declaration in the ontology.
- Find synonyms for resources (either classes or instance names).
- Since extensional information from OWL documents (OWL instances and their inter-relationships) needs to be queried for client applications, an OWL-QL query processing system is available as an open-source project for RacerPro.
- HTTP client for retrieving imported resources from the web. Multiple resources can be imported into one ontology.
- Incremental query answering for information retrieval tasks (retrieve the next n results of a query). In addition, RacerPro supports the adaptive use of computational resource: Answers which require few computational resources are delivered first, and user applications can decide whether computing all answers is worth the effort.

Future extensions for OWL (e.g., OWL-E) are already supported by RacerPro if the system is seen as a description logic system. RacerPro already supports qualified cardinality restrictions as an extension to OWL DL.

1.1.2 RacerPro as a Description Logic System

RacerPro is a knowledge representation system that implements a highly optimized tableau calculus for a very expressive description logic. It offers reasoning services for multiple T-boxes and for multiple A-boxes as well. The system implements the description logic \mathcal{ALCQHI}_{R+} also known as \mathcal{SHIQ} (see [9]). This is the basic logic \mathcal{ALC} augmented with qualifying number restrictions, role hierarchies, inverse roles, and transitive roles. In addition to these basic features, RacerPro also provides facilities for algebraic reasoning including concrete domains for dealing with:

- min/max restrictions over the integers,
- linear polynomial (in-)equations over the reals or cardinals with order relations,
- equalities and inequalities of strings.

For these domains, no feature chains can be supported due to decidability issues.

RacerPro supports the specification of general terminological axioms. A T-box may contain general concept inclusions (GCIs), which state the subsumption relation between two concept *terms*. Multiple definitions or even cyclic definitions of concepts can be handled by RacerPro.

RacerPro implements the HTTP-based quasi-standard DIG for interconnecting DL systems with interfaces and applications using an XML-based protocol [4]. RacerPro also implements most of the functions specified in the older Knowledge Representation System Specification (KRSS), for details see [17].

Given a T-box, various kinds of queries can be answered. Based on the logical semantics of the representation language, different kinds of queries are defined as inference problems

(hence, answering a query is called providing inference service). As a summary, we list only the most important ones here:

- Concept consistency w.r.t. a T-box: Is the set of objects described by a concept empty?
- Concept subsumption w.r.t. a T-box: Is there a subset relationship between the set of objects described by two concepts?
- Find all inconsistent concept names mentioned in a T-box. Inconsistent concept names result from T-box axioms, and it is very likely that they are the result of modeling errors.
- Determine the parents and children of a concept w.r.t. a T-box: The parents of a concept are the most specific concept names mentioned in a T-box which subsume the concept. The children of a concept are the most general concept names mentioned in a T-box that the concept subsumes. Considering all concept names in a T-box the parent (or children) relation defines a graph structure which is often referred to as taxonomy. Note that some authors use the name taxonomy as a synonym for ontology.

Note that whenever a concept is needed as an argument for a query, not only predefined names are possible, instead concept expressions allow for adaptive formulation of queries that have not been anticipated at system construction time.

If also an A-box is given, among others, the following types of queries are possible:

- Check the consistency of an A-box w.r.t. a T-box: Are the restrictions given in an A-box w.r.t. a T-box too strong, i.e., do they contradict each other? Other queries are only possible w.r.t. consistent A-boxes.
- Instance testing w.r.t. an A-box and a T-box: Is the object for which an individual stands a member of the set of objects described by a specified concept? The individual is then called an instance of the concept.
- Instance retrieval w.r.t. an A-box and a T-box: Find all individuals from an A-box such that the objects they stand for can be proven to be a member of a set of objects described by a certain query concept.
- Retrieval of tuples of individuals (instances) that satisfy certain conditions w.r.t. an A-box and a T-box.
- Computation of the direct types of an individual w.r.t. an A-box and a T-box: Find the most specific concept names from a T-box of which a given individual is an instance.
- Computation of the fillers of a role with reference to an individual w.r.t. an A-box and a T-box.
- Check if certain concrete domain constraints are entailed by an A-box and a T-box.

RacerPro provides another semantically well-defined query language (nRQL, new Racer Query Language), which also supports negation as failure, numeric constraints w.r.t. attribute values of different individuals, substring properties between string attributes, etc. In order to support special OWL features such as annotation and datatype properties, special OWL querying facilities have been incorporated into nRQL. The query language OWL-QL [4] is the W3C recommendation for querying OWL documents. nRQL has been used as a basic engine for implementing a very large subset of the OWL-QL query language (see above).

1.1.3 RacerPro as a Combination of Description Logic and Specific Relational Algebras

For some representation purposes, e.g., reasoning about spatial relations such as contains, touching, etc., relational algebras and constraint propagation have proven to be useful in practice. RacerPro combines description logics reasoning with, for instance, reasoning about spatial (or temporal) relations within the A-box query language nRQL. Bindings for query variables that are determined by A-box reasoning can be further tested with respect to an associated constraint network of spatial (or temporal) relationships.

Although RacerPro is one of the first systems supporting this kind of reasoning in combination with description logics (or OWL), we expect that international standardization efforts will also cover these important representation constructs in the near future. Note also that the semantically well-founded treatment can hardly be efficiently achieved using rule systems.

1.2 Application Areas and Usage Scenarios

Description logic systems are no databases. Although one can use RacerPro for storing data using A-boxes, probably, databases provide better support with respect to persistency and transactions. However, databases can hardly be used if indefinite information is to be handled in an application (e.g., “John was seen playing with a ball, but I cannot remember, it was soccer or basket ball, so he must be a soccer player or a basket-ball player”). Being able to deal with indefinite information of this kind is important when information about data comes in from various sources, and in situations where sources are, for instance, used to exclude certain cases that are possible given the information at hand. Description logics are also important if data descriptions are to be queries with respect to varying T-boxes (or ontologies). Note that this is a common scenario in modern information technology infrastructure. Due to the rapid pace in technology evolution, also the vocabulary used to access data descriptions changes quite frequently. New concepts are introduced, and can be set into relation to older terminology using description logics (or semantic web ontologies).

There are numerous papers describing how description logic in general (and RacerPro in particular) can be used to solve application problems (see the [International Workshops on Description Logics](#) and the workshops on Applications of Description Logics [ADL-01](#), [ADL-02](#), or [ADL-03](#)). Without completeness one can summarize that applications come from the following areas:

- Semantic Web, Semantic Grid (ontology representation and logic-based information retrieval)
- Electronic Business (e.g, reason about services descriptions)
- Medicine/Bioinformatics (e.g., represent and manage biomedical ontologies)
- Natural Language Processing and Knowledge-Based Vision (e.g., exploit and combine multiple clues coming from low-level vision processes in a semantically meaningful way).
- Process Engineering (e.g., represent service descriptions)
- Knowledge Engineering (e.g., represent ontologies)
- Software Engineering (e.g., represent the semantics of UML class diagrams)

1.3 Racer Editions, Installation, Licenses, and System Requirements

1.3.1 Editions

Racer Systems provides several different editions of the Racer technology.

RacerPro RacerPro is a server for description logic or OWL inference services. With RacerPro you can implement industrial strength projects as well as doing research on knowledge basis and develop complex applications. If you do not have a valid license, you are allowed to use RacerPro but some restrictions apply.

RacerPorter The “Porter to RacerPro” (aka RacerPorter) is the graphical user client for RacerPro. RacerPorter uses the TCP/IP network interface to connect to one or more RacerPro servers and helps you manage them: You can load knowledge bases, switch between different taxonomies, inspect your instances, visualize T-Boxes and A-Boxes, manipulate the server and much more. RacerPorter is already included in the installer versions of RacerPro for Windows and Mac OS X and separately available for Linux systems with a graphic display.

RacerPlus To minimize the performance overhead due to network-based communication between RacerPorter and RacerPro as well as to utilize the computing power offered by a single workstation we introduce RacerPlus, an integrated workbench which includes RacerPro and the RacerPorter graphical user interface in a single application.

RacerMaster Do you develop complex applications for the Semantic Web? Are you researching in the area of description logic, knowledge basis, and implement your own systems and algorithms for inference problems using the Common Lisp programming language? Then you might be interested in RacerMaster which actually is RacerPro as an object code library (“fasl file”). You can develop your own application and use Racer technology without an external server application.

1.3.2 Installation

The RacerPro system can be obtained from the following web site:

<http://www.racer-systems.com>

A documentation for the RacerPro installation process is included in the installer file that you can download from your personal download page. The URL for your personal download page is sent to you via email.

1.3.3 Licenses

RacerPro is available as a desktop version in which the server and clients run on localhost. In addition, there are licenses available to run RacerPro on a powerful server machine whereas graphical interfaces or client applications can be executed on your personal computer (or via wireless connections on your portable computer). Your license file is available from your personal download page (the file is called `license.racerlicense`). Put this license file in your home directory. Do not edit or delete this file. If the installation process is executed successfully, the RacerPro application comes with an additional license file named `racerpro.lic` in the same directory as the RacerPro executable. Do not remove or edit this file, either.

1.3.4 System Requirements

RacerPro is available for all major operating systems in 32bit and 64bit modes (Linux, Mac OS X, Solaris 2, Windows). For larger knowledge bases in industrial projects we recommend at least 1 GB of main memory and a 1GHz processor. For large ontologies (> 100000 concept and role names) we recommend 64bit computer architectures.

1.4 Acknowledgments

RacerPro is based on scientific results presented in publications about theoretical and practical work on KRIS, FaCT, and other description logic systems. See, for instance, [3], [5], [7], [8], [14], [15], [10], [11], [12].

We would like to thank the team at **Franz Inc.** (<http://www.franz.com/>), for their collaboration and for the support in making RacerPro one of the fastest and most expressive commercial OWL/RDF and description logic reasoning systems. The graphical user interface RacerPorter is built with the development environment from **Lispworks Ltd.**

(<http://www.lispworks.com/>). RacerPro is also developed with [Macintosh Common Lisp](http://www.digitool.com/) (<http://www.digitool.com/>).

The XML/RDF-based part of the input interface for RacerPro is implemented using the XML/RDF parser Wilbur written by Ora Lassila. For more information on Wilbur and the Wilbur source code see <http://wilbur-rdf.sourceforge.net/>.

For most versions of RacerPro, the DIG server interface is implemented with AllegroServe. For some special versions of RacerPro, however, the HTTP server for the DIG interface of RacerPro is implemented with CL-HTTP, which is developed and owned by John C. Mallery. For more information on CL-HTTP see <http://www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html>.

Many users have directly contributed to the functionality and stability of the RacerPro system by giving comments, providing ideas and test knowledge bases, implementing interfaces, or sending bug reports. There are too many of them to mention the name them all. Many thanks for any hint, comment, and bug report.

Chapter 2

Using RacerPro

In this section we present a first example for the use of RacerPro. We use the so-called KRSS-based interface rather than the XML (or OWL/RDF) interface here in order to directly present the declaration and results of queries in a brief and human-readable form. Note, however, that all examples can be processed in a similar way with the XML-based interfaces.

2.1 Sample Session with RacerPorter

The file "family.racer" in the examples folder of the RacerPro distribution contains the T-box and A-box introduced in this section. The queries are in the file "family-queries.racer". In order to run the example, just start RacerPro by double clicking the program icon or, alternatively, type **RacerPro** as a command in a shell window.¹

We use the interactive graphical interface to demonstrate the result of queries in this sample session. However, you can also use the RacerPro executable in batch mode. If you use the RacerPro executable in batch mode just type **RacerPro -f family.racer -q family-queries.lisp** into a shell window in order to see the results (under Windows type **RacerPro -- -f family.racer -q family-queries.lisp**). See also Section 2.2 for details on how to use RacerPro from a shell).

You should see something similar to the following:

```
;;; Welcome to RacerPro Version 1.9.0 2005-11-21!

;;; Racer: Renamed Abox and Concept Expression Reasoner
;;; Supported description logic: ALCQHIR+(D)-
;;; Supported ontology web language: subset of OWL DL (no so-called nominals)

;;; Copyright (C) 2004, 2005 by Racer Systems GmbH & Co. KG
;;; All rights reserved. See license terms for permitted usage.
```

¹We assume that **RacerPro** is on the search path of your operating system.

```

;;; Racer and RacerPro are trademarks of Racer Systems GmbH & Co. KG
;;; For more information see: http://www.racer-systems.com
;;; RacerPro comes with ABSOLUTELY NO WARRANTY; use at your own risk.

;;; RacerPro is based on:
;;; International Allegro CL Enterprise Edition 7.0 (Oct 19, 2004 13:28)
;;; Copyright (C) 1985-2004, Franz Inc., Oakland, CA, USA. All Rights Reserved.
;;; The XML/RDF/RDFS/OWL parser is implemented with Wilbur developed
;;; by Ora Lassila. For more information on Wilbur see
;;; http://wilbur-rdf.sourceforge.net/.

;;; =====
;;; Found license file
;;; /Users/rm/ralf-mac-os-x.lic
;;; This copy of RacerPro is licensed to:
;;;
;;; Ralf Moeller
;;; Hamburg University of Technology (TUHH)
;;; Harburger Schlossstr. 20
;;; STS Group
;;; 21079 Hamburg
;;; Deutschland
;;;
;;; Initial license generated on 06-29-2005, 12:35 for 1.8.1.
;;; Site, Commercial, on Mac OS X.
;;; This license is valid up to version 9.9.99.
;;; This license is valid forever.
;;;
;;; This is RacerPro for Ralf Moeller
;;;
;;; =====

```

HTTP service enabled for: <http://localhost:8080/>

TCP service enabled for: <http://localhost:8088/>

If you do not have a valid license, you are allowed to use RacerPro but some restrictions apply (see the RacerPro web site for details). If you have a valid license, your own name will be mentioned in the output, of course.

The following forms are found in the file `family.racer` in the examples folder.

```

;;; initialize the T-box "family"
(in-tbox family)

;;; Supply the signature for this T-box

```

```

(signature
:atomic-concepts (person human female male woman man parent mother
                  father grandmother aunt uncle sister brother)
:roles ((has-child :parent has-descendant)
        (has-descendant :transitive t)
        (has-sibling)
        (has-sister :parent has-sibling)
        (has-brother :parent has-sibling)
        (has-gender :feature t))
:individuals (alice betty charles doris eve))

;;; Domain & range restrictions for roles
(implies *top* (all has-child person))
(implies (some has-child *top*) parent)
(implies (some has-sibling *top*) (or sister brother))
(implies *top* (all has-sibling (or sister brother)))
(implies *top* (all has-sister (some has-gender female)))
(implies *top* (all has-brother (some has-gender male)))

;;; Axioms for relating concept names
(implies person (and human (some has-gender (or female male))))
(disjoint female male)
(implies woman (and person (some has-gender female)))
(implies man (and person (some has-gender male)))
(equivalent parent (and person (some has-child person)))
(equivalent mother (and woman parent))
(equivalent father (and man parent))
(equivalent grandmother (and mother (some has-child (some has-child person))))
(equivalent aunt (and woman (some has-sibling parent)))
(equivalent uncle (and man (some has-sibling parent)))
(equivalent brother (and man (some has-sibling person)))
(equivalent sister (and woman (some has-sibling person)))

(instance alice mother)
(related alice betty has-child)
(related alice charles has-child)

(instance betty mother)
(related betty doris has-child)
(related betty eve has-child)

(instance charles brother)
(related charles betty has-sibling)
(instance charles (at-most 1 has-sibling))

(related doris eve has-sister)

```

(related eve doris has-sister)

Start RacerPorter by double-clicking the program icon or type **RacerPorter** as a command in your shell.² Press the button Connect to connect the graphical interface to the RacerPro reasoning engine. Make sure RacerPro is already started. Then, load a file into RacerPro by pressing the button Load... (or by selecting Load... in the menu File on Mac OS X).

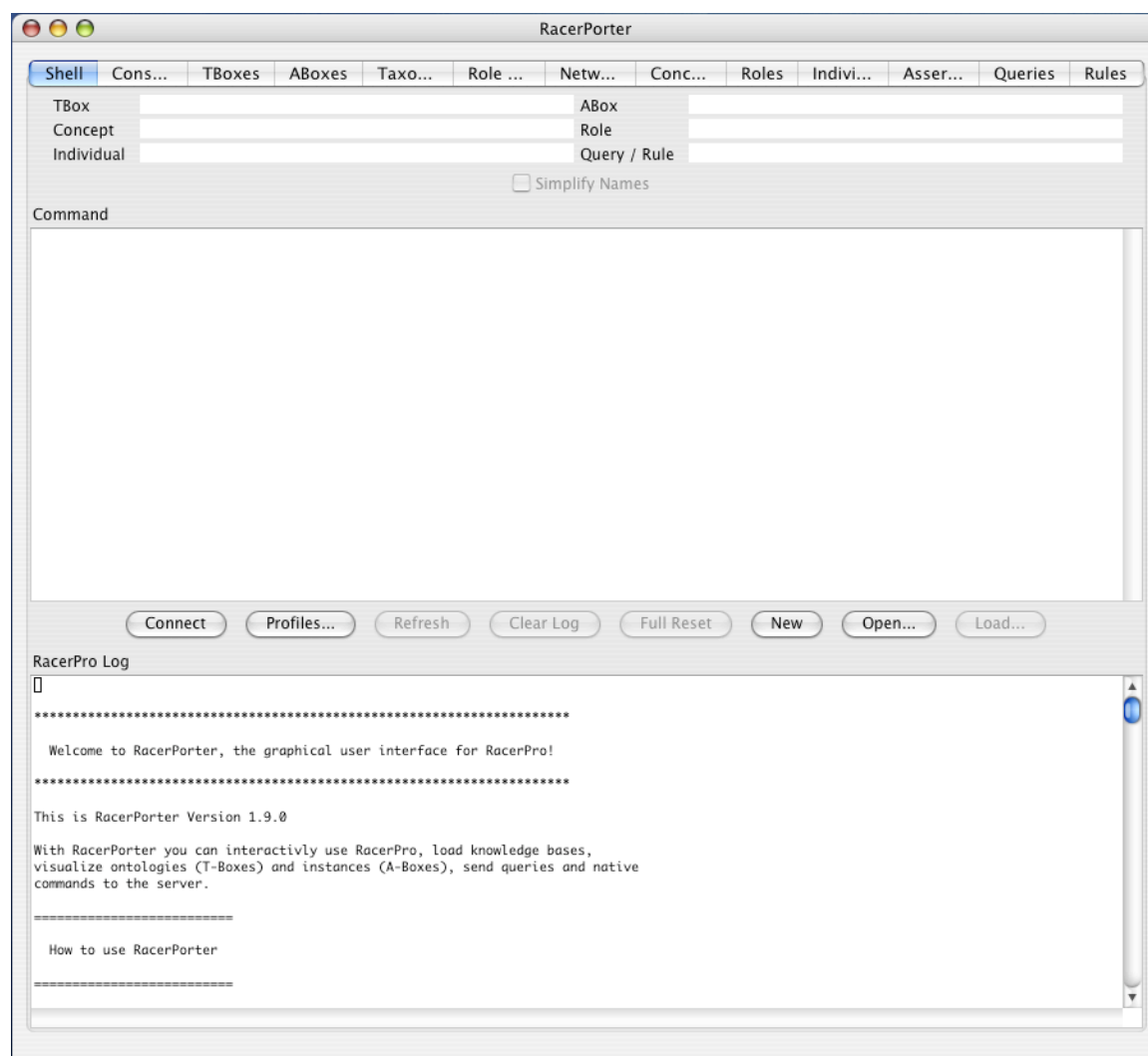


Figure 2.1: The RacerPorter interface for RacerPro.

²We assume that **RacerPorter** is on the search path of your operating system.

RacerPorter also allows you to edit knowledge bases (or ontologies and data descriptions) with the built-in editor. Just press the button Open... or select Open... in the File menu (Mac OS X). Under Mac OS X and Windows you can also double-click the file `family.racer` (make sure RacerPro is already running). If you have the file `family.racer` displayed in the editor (see Figure 2.2), you can send the statements to RacerPro by selecting Evaluate Racer Buffer in the Buffer menu (make sure the window RacerEditor is the active window).

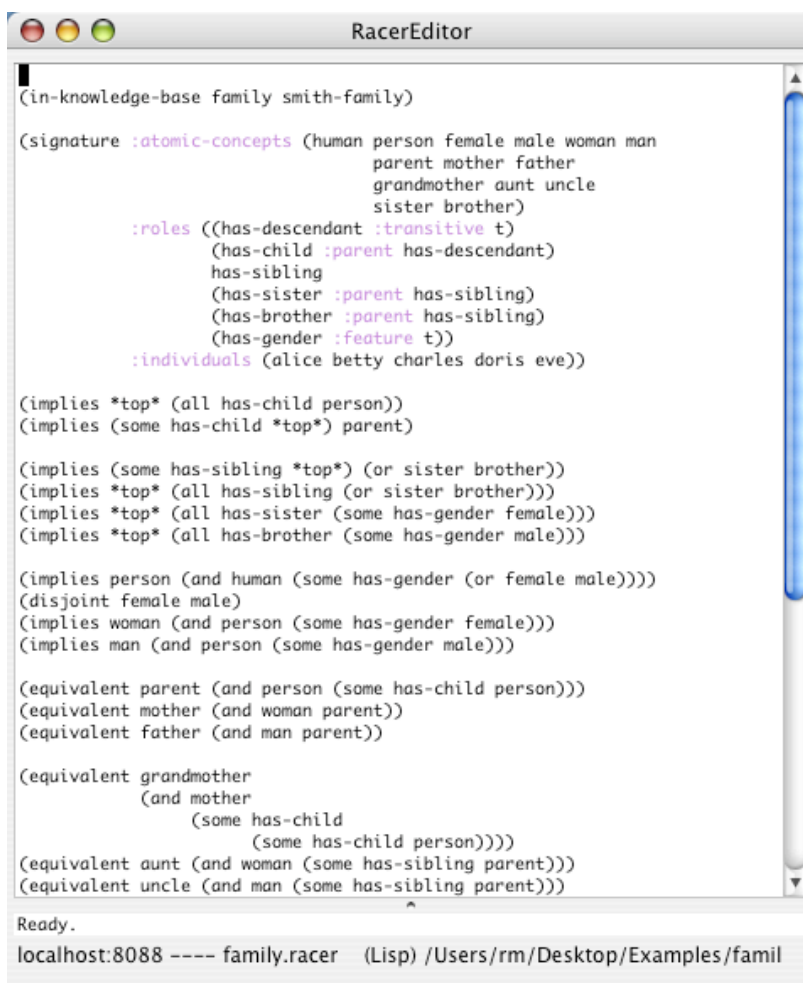


Figure 2.2: The RacerEditor interface for RacerPro.

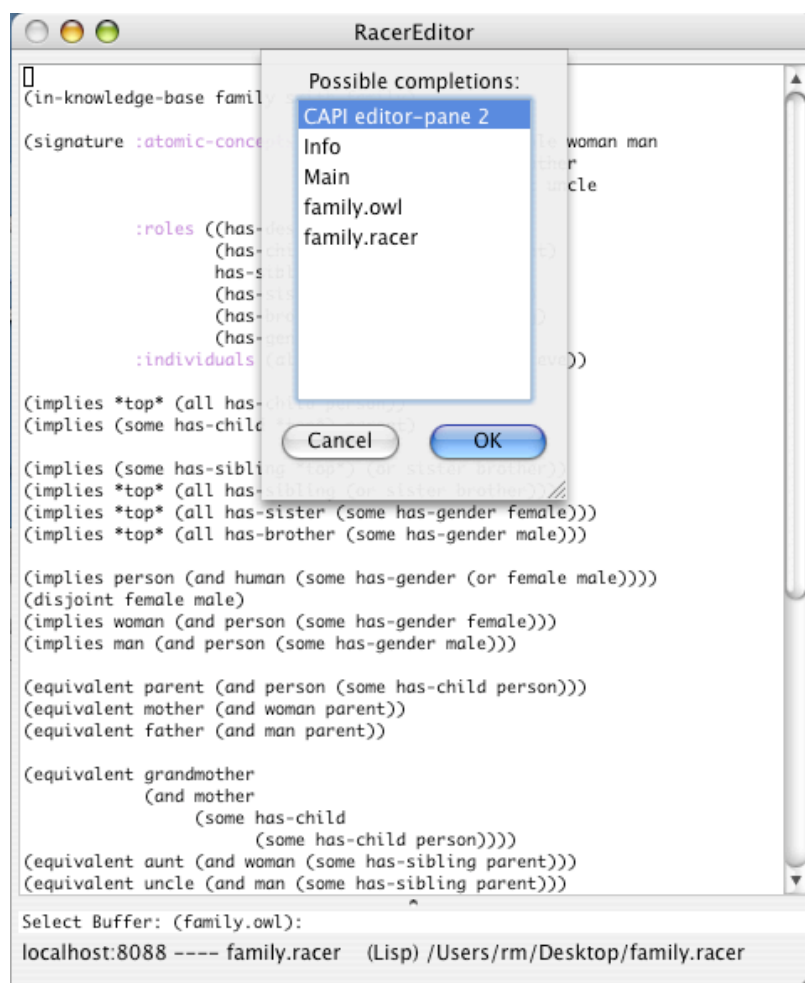


Figure 2.3: RacerEditor displaying the list of buffers.

RacerPro supports the standard Emacs commands such as, for instance, change buffer, etc. In addition, the standard key bindings are provided. E.g., change buffer is bound to `c-x b`. Press `tab` to see a list of possible completions (see Figure 2.3).

After having loaded the file `family.racer` into RacerPro, you can use RacerPorter to inspect the knowledge base. For instance, you might be interested in the hierarchy of concept names or roles using the tabs Taxonomy and Role Hierarchy, respectively (see Figures 2.4 and 2.5).

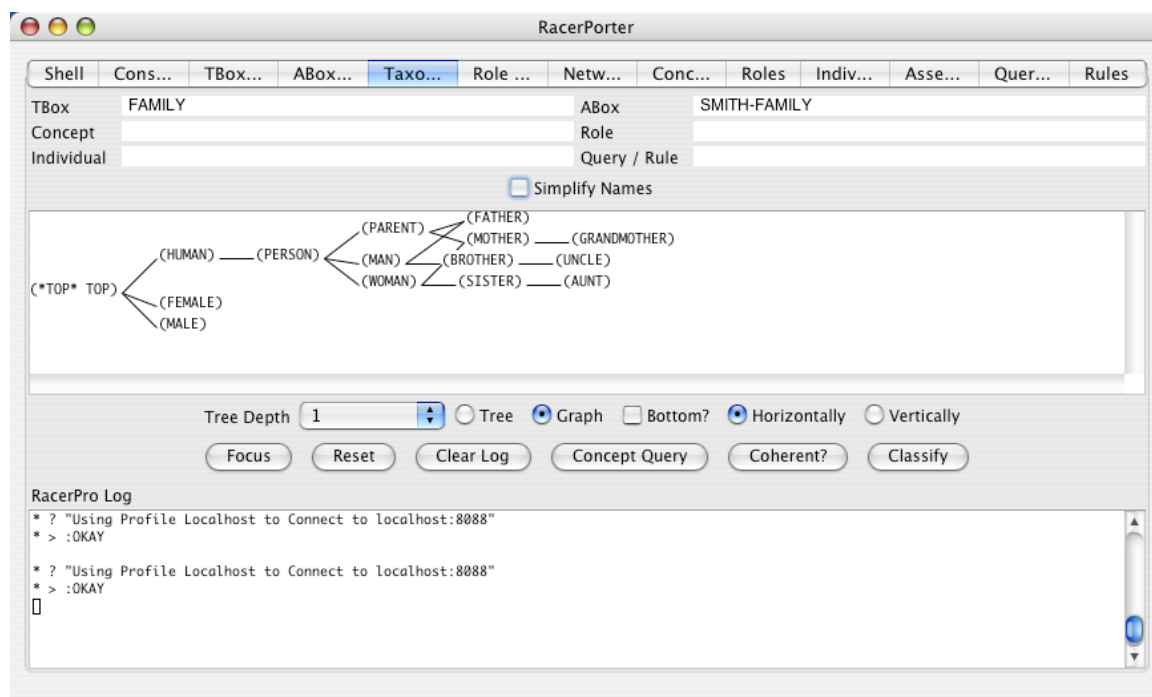


Figure 2.4: RacerPorter showing the taxonomy of the T-box `family`. You should select the button graph in order to see the full hierarchy.

You can switch between the shell, the taxonomy, and the role hierarchy by selecting the corresponding tabs in the RacerPorter window.

Next we will use RacerPorter to specify some queries and inspect the answers. You can use the Shell tab or, alternatively, you can use the Console tab to execute queries as we have done in Figure 2.6. Information about commands and key abbreviations is printed into the Shell window. With `meta-p` (or `alt-p`) and `meta-n` (or `alt-n`) you can get the previous and next command of the command history, respectively. You might want to use the `tab` key to complete a partially typed command. A command is executed once it is syntactically complete.

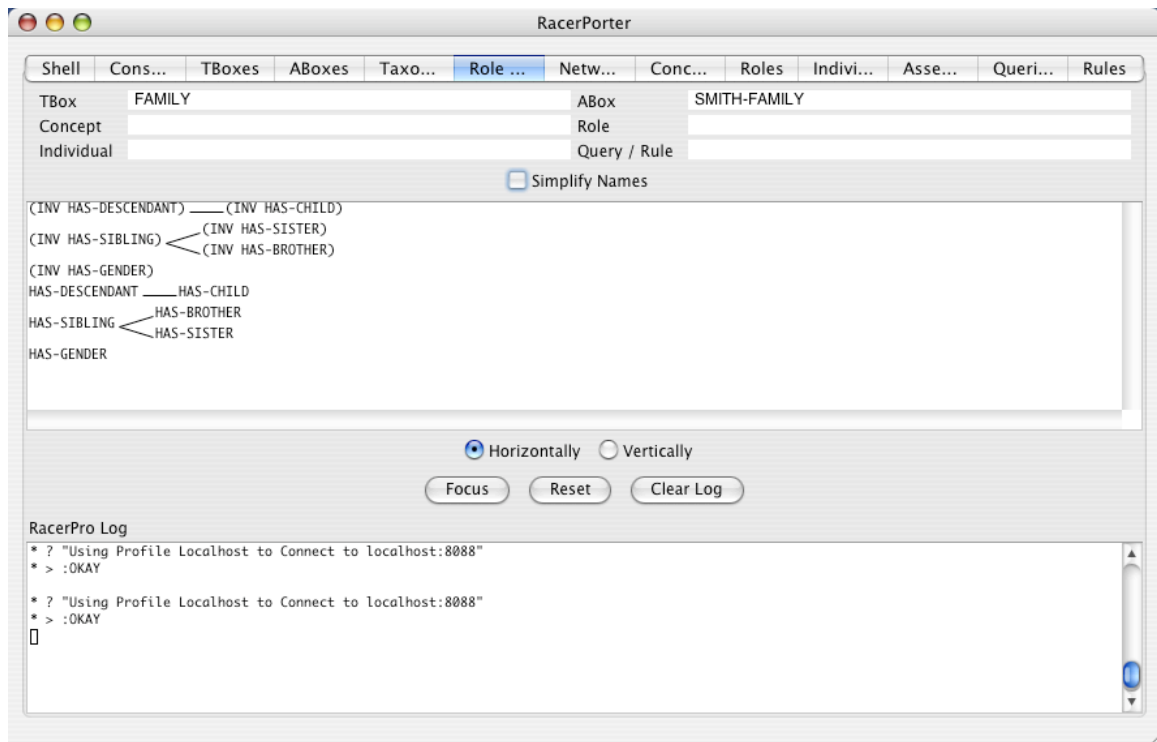


Figure 2.5: RacerPorter showing the role hierarchy of the T-box family.

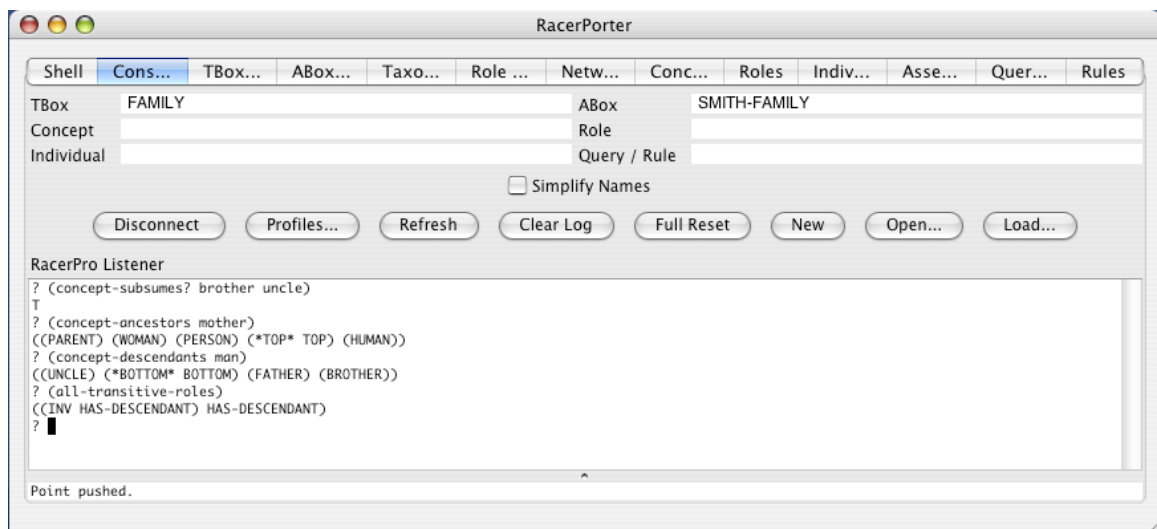


Figure 2.6: The Console tab of RacerPorter used for executing queries.

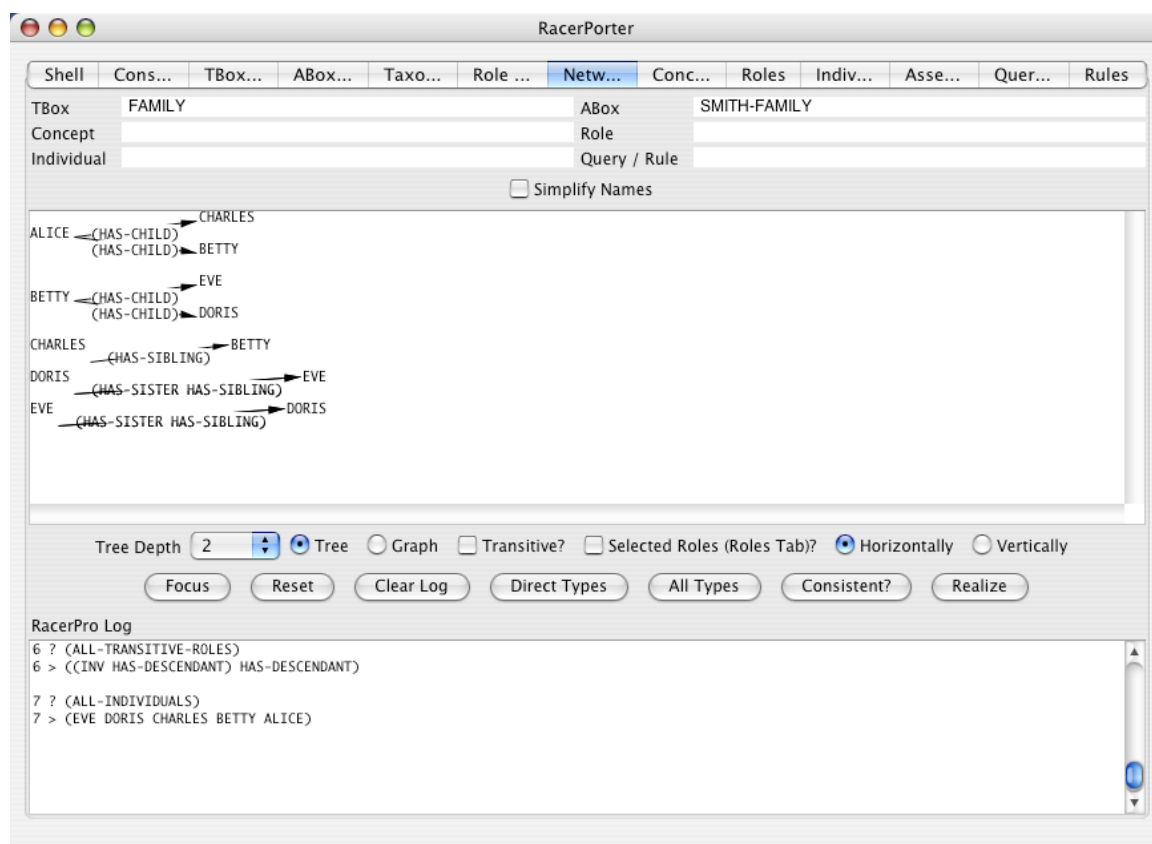
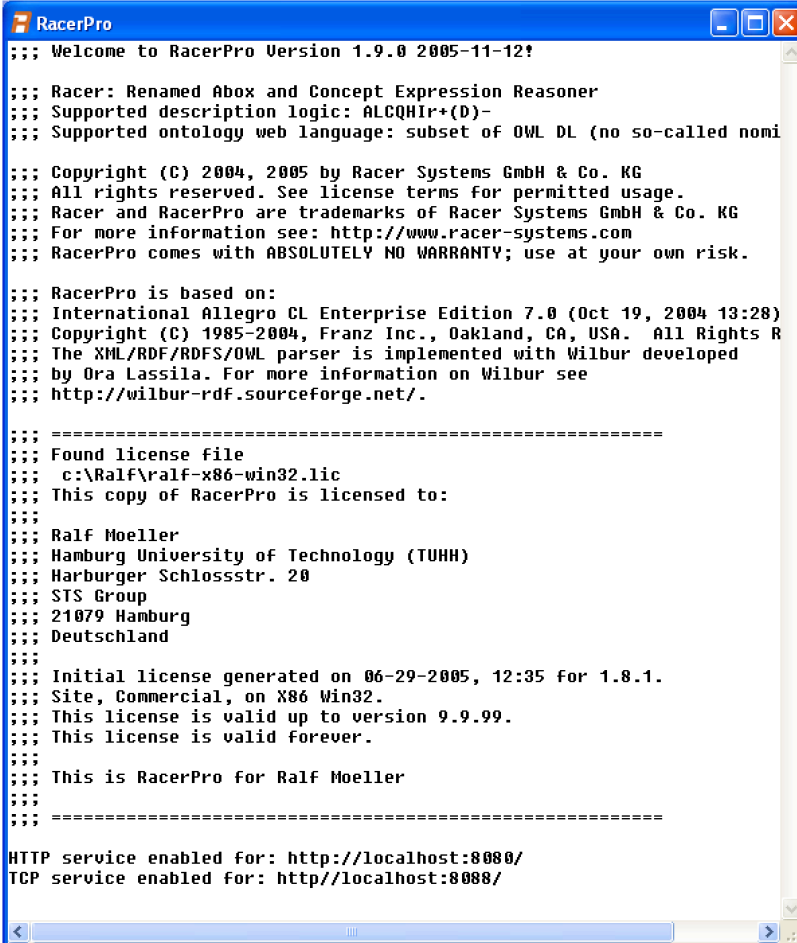


Figure 2.7: The Network tab shows the explicit relations for A-box individuals.

RacerPorter allows you to inspect data descriptions in an A-box. Just select the Network tab (see Figure 2.7). Adjust the display options to Tree and select Tree Depth 3 as shown in Figure 2.7.

2.2 The RacerPro Server

The RacerPro server is an executable file available for Linux, Mac OS X, Solaris 2, and Windows XP. It can be started from a shell or by double-clicking the corresponding program icon in a graphics-based environment. For instance, the Windows version is shown in Figure 2.8.



```

RacerPro
;;; Welcome to RacerPro Version 1.9.0 2005-11-12!

;;; Racer: Renamed Abox and Concept Expression Reasoner
;;; Supported description logic: ALCQHIr+(D)-
;;; Supported ontology web language: subset of OWL DL (no so-called nomi

;;; Copyright (C) 2004, 2005 by Racer Systems GmbH & Co. KG
;;; All rights reserved. See license terms for permitted usage.
;;; Racer and RacerPro are trademarks of Racer Systems GmbH & Co. KG
;;; For more information see: http://www.racer-systems.com
;;; RacerPro comes with ABSOLUTELY NO WARRANTY; use at your own risk.

;;; RacerPro is based on:
;;; International Allegro CL Enterprise Edition 7.0 (Oct 19, 2004 13:28)
;;; Copyright (C) 1985-2004, Franz Inc., Oakland, CA, USA. All Rights R
;;; The XML/RDF/RDFS/OWL parser is implemented with Wilbur developed
;;; by Ora Lassila. For more information on Wilbur see
;;; http://wilbur-rdf.sourceforge.net/.

;;; =====
;;; Found license file
;;; c:\Ralf\ralf-x86-win32.lic
;;; This copy of RacerPro is licensed to:
;;;
;;; Ralf Moeller
;;; Hamburg University of Technology (TUHH)
;;; Harburger Schlossstr. 20
;;; STS Group
;;; 21079 Hamburg
;;; Deutschland
;;;
;;; Initial license generated on 06-29-2005, 12:35 for 1.8.1.
;;; Site, Commercial, on X86 Win32.
;;; This license is valid up to version 9.9.99.
;;; This license is valid forever.
;;;
;;; This is RacerPro for Ralf Moeller
;;;
;;; =====

HTTP service enabled for: http://localhost:8080/
TCP service enabled for: http://localhost:8088/

```

Figure 2.8: A screenshot of the RacerPro server started under Windows.

Depending on the arguments provided at startup, the RacerPro executable supports different modes of operation. It offers a file-based interface, a socket-based TCP stream interface, and a HTTP-based stream interface. With the OWL-QL server comes a web service interface.

2.2.1 The File Interface

If your knowledge bases and queries are available as files use the file interface of RacerPro, i.e. start RacerPro with the option `-f`. In the following, we assume that you have RacerPro

on your search path. In your favorite shell on Unix-based systems just type:

```
$ RacerPro -f family.racer -q family-queries
```

Under Windows, you want to suppress the display of the RacerPro window. The command is slightly different:

```
$ RacerPro +c -- -f family.racer -q family-queries.lisp
```

The option `--` separates window management options from RacerPro options. A window option we use here is `+c` which suppress the display of the RacerPro window, which is not useful in batch mode. For debugging under Windows, the window option `+p` is useful.

```
$ RacerPro +p -- -f family.racer -q family-queries.lisp
```

If an error occurs you can read the error message in the console window. The option `-h` prints some information about the usage of the program RacerPro.

```
$ RacerPro +p -- -h
```

The `-f` RacerPro option has the following meaning. The input file is `family.racer` and the queries file is `family-queries.lisp`. The output of RacerPro is printed into the shell. If output is to be printed into a file, specify the file with the option `-o` as in the following example:

```
$ RacerPro -f family.racer -q family-queries.lisp -o ouput.text
```

Or use the following under Windows as explained above:

```
$ RacerPro +c -- -f family.racer -q family-queries.lisp -o ouput.text
```

The syntax for processing input files is determined by RacerPro using the file type (file extension). If `.lisp`, `.krss`, or `.racer` is specified, a KRSS-based syntax is used. Other possibilities are `.rdfs`, `.owl`, and `.dig`. If the input file has one of these extensions, the respective syntax for the input is assumed. The syntax can be enforced with corresponding options instead of `-f`: `-rdfs`, `-owl`, and `-dig`.

The option `-xml <filename>` is provided for historical reasons. The input syntax is the older XML syntax for description logic systems. This syntax was developed for the FaCT system [7]. In the RacerPro server, output for query results based on this syntax is also printed using an XML-based syntax. However, the old XML syntax of FaCT is now superseded by the DIG standard. Therefore, the RacerPro option `-xml` may no longer be supported once the file interface fully supports the DIG standard for queries (see above).

Currently, the file interface of RacerPro supports only queries given in KRSS syntax. However, DIG-based queries [4] can be specified indirectly with the file interface as well. Let us assume a DIG knowledge base is given in the file `kb.xml` and corresponding DIG queries are specified in the file `queries.xml`. In order to submit this file to RacerPro just create a file `q.racer`, say, with contents `(dig-read-file "queries.xml")` and start RacerPro as follows:

```
$ RacerPro -dig kb.xml -q q.racer
```

Under windows, you might want to suppress the console window:

```
$ RacerPro +c -- -dig kb.xml -q q.racer
```

Note the use of the option `-dig` for specifying that the input knowledge base is in DIG syntax. Since the file extension for the knowledge base is `.xml`, the option `-f` would assume the

older XML syntax for knowledge bases (see above). If the query file has the extension `.xml`, RacerPro assumes DIG syntax. For older programs this kind of backward compatibility is needed.

RacerPro opens one TCP port for the DIG protocol and one for RacerPro's native protocol. The DIG protocol is an HTTP based XML protocol and thus is based on plain text. The native protocol is proprietary to Racer but in "KRSS" syntax and is documented in the Reference Manual. Since both protocols are based on plain text format in a public available command language, anybody with knowledge about the existence of a RacerPro server can manipulate the server status, the data stored and the processing, since it is also possible to connect to one RacerPro server instance from multiple clients which may not be all under your control.

2.2.2 TCP APIs

There are two APIs based on TCP sockets, namely for the programming languages Common Lisp and Java. The socket interface of the RacerPro server can be used from application programs or graphical interfaces. Bindings for C++ and Prolog dialects have been developed as well.

If the option `-f` is not provided, the socket interface is automatically enabled. Just execute the following.

\$ RacerPro

The default TCP communication port used by RacerPro is 8088. In order to change the port number, the RacerPro server should be started with the option `-p`. For instance:

\$ RacerPro -p 8000 (or **\$ RacerPro -- -p 8000** under Windows).

In this document the TCP socket is also called the raw TCP interface. The functionality offered by the TCP socket interface is documented in the next sections.

RacerPro's server engine listens on the specified TCP port for incoming connections from client processes running on the local machine, the network or even the internet. Please note that RacerPro itself offers no fraud and privacy protection. You have to utilize network based security services such as firewalls, IP tunnels, VLAN switches and similar techniques if you are concerned about data privacy and security.

JRacer

JRacer is the client library to access the services of a RacerPro server from Java client programs. The package `jracer` is provided with source code in your RacerPro distribution directory. See <http://www.racer-systems.com/products/download> for the latest version. An example client is provided with source code as well. The main idea of the socket interface is to open a socket stream, submit declarations and queries using strings and to parse the answer strings provided by RacerPro.

The following code fragment explains how to send message to a RacerPro server running on localhost `127.0.0.1` under port 8088.


```
import jracer.*;
import java.io.*;

public class Test {

    public static void main(String[] argv) {
        String ip = "127.0.0.1";
        int port = 8088;
        String filename = "\\Users\\rm\\Desktop\\Examples\\family.owl\\";

        Racerver racer1 = new Racerver(ip, port);
        try {
            FileInputStream fstream = new FileInputStream(filename);
            int n = 0;
            while (fstream.available() != 0) {
                int c = fstream.read();
                n += 1;
            }
            fstream.close();
            racer1.openConnection();
            System.out.println(racer1.send("owl-read-file " + filename + "));
            System.out.println(racer1.send("all-atomic-concepts"));
            racer1.closeConnection();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

We assume that you compile this file `Test.java` with the package `jracer` (see the JRacer directory in your RacerPro distribution) in the same directory (or on the Java CLASS-PATH).

LRacer

LRacer is the API for Common Lisp to access all services of RacerPro in a convenient way. LRacer is provided with source code. You can download the latest version from <http://www.racer-systems.com/products/download>. LRacer provides all functions (and macros) described in the RacerPro user's guide and reference manual directly from Common Lisp. Thus, from Common Lisp, you do not send strings to the server directly, but use stub functions (or macros) which internally communicate with the RacerPro server, interpret the results and provide them as Common Lisp data structures. In this sense, LRacer is not more powerful than JRacer but a little bit more convenient. Note the difference between LRacer and RacerMaster. Although LRacer provides the same functionality, all functions calls are sent to a RacerPro server via the TCP socket interface, whereas for RacerMaster there is no such overhead. The advantage of LRacer is that you can run RacerPro on a powerful

server machine and develop your application on a less expensive portable computer.

Let us assume the LRacer directory is stored under `~/Lracer/`. Start your favorite Common Lisp system, and evaluate `(load "~/Lracer/lracer-sysdc1.lisp")`. Then, evaluate `(compile-load-lracer "~/Lracer/")` to compile and load LRacer. You have to use the pathname that corresponds to your LRacer distribution directory, of course. Afterwards, you can import the package `RACER` into your own package or access RacerPro directly from the Common Lisp User package and the Lisp listener.

The variable `*default-racer-host*` can be set to a string denoting the host on which the RacerPro server run (the default is `"localhost"`). You can set the tcp port for the RacerPro server by setting the variable `*default-racer-tcp-port*` (default value is 8080).

You can explicitly open a server connection with the function `open-server-connection`. In order to close the server connection, use the function `close-server-connection`. See also the macro `with-server-connection`. However, explicitly opening a server is not necessary, it just reduces the network overhead for multiple server calls.

2.2.3 Web Service Interface

A web service interface is provided with the OWL-QL server. The documentation for this software can be found at <http://www.racer-systems.com>.

2.2.4 HTTP Interface: DIG Interface

The DIG Interface is a standardized XML interface to description logics systems developed by the DL Implementation Group (DIG), see <http://dig.sourceforge.net/> for details.

In a similar way as the socket interface the HTTP interface can be used from application programs (and graphical interfaces). If the option `-f` is not provided, the HTTP interface is automatically enabled. If you do not use the HTTP interface at all but the TCP interface only, start RacerPro with the option `-http 0`.

Clients can connect to the HTTP based RacerPro server using the POST method. For details see the DIG standard [4]. The default HTTP communication port used by RacerPro is 8080. In order to change the port number, the RacerPro server should be started with the option `-http`. For instance:

```
$ RacerPro -http 8000 under Unix and
```

```
$ RacerPro -- -http 8000 under Windows
```

Console logging of incoming POST requests is provided by default but can be switched off using the option `-nohttpconsolelog`. With the option `-httplogdir <directory>` logging into a file in the specified directory can be switched on.

The DIG standard as it is defined now is just a first step towards a communication standard for connecting applications to DL systems. RacerPro provides many more features that are not yet standardized. These features are offered only over the TCP socket interface. However, applications using RacerPro can be developed with DIG as a starting point. If other facilities of RacerPro are to be used, the raw TCP interface of RacerPro can be used in a seemingless way to access a knowledge base declared with the DIG interface. If, later

on, the standardization process make progress, users should be able to easily adapt their code to use the HTTP-based DIG interface.

2.2.5 Options for the RacerPro Server

Various options allow you to control the behavior of RacerPro. Under Windows we have to distinguish between RacerPro options and window manager options (see below). Under Windows, you have to use the separator `--` even if there are no window manager options used.

- Use the option `-h` to get the list of possible options and a short explanation. Use this option in combination with the window manager option `+p` (see below).
- As indicated above, the options `-f <filename>` can be used for reading knowledge bases from a file. The extension of `<filename>` is used to discriminate the syntax to be used (possible extensions for corresponding syntaxes are: `.racer`, `.owl`, `.rdfs`, `.rdfs`. In all other cases, Racer syntax is expected. If the extension of your input file does not have an appropriate extension, you can specify the syntax with the options `-owl`, `-dig`, `-rdfs` instead of `-f`.
- Use the option `-q <filename>` to specify a file with queries (extension `.racer` only).
- `-p <port-number>` specifies the port for TCP connections (e.g., for LRacer and JRacer, see above).
- `-http <port-number>` specifies the port for HTTP connections (e.g., for the DIG interface, see above).
- `-httplogdir <directory>` specifies the logging directory (see above).
- `-nohttpconsolelog` disables console logging for HTTP connections (see above).
- Processing knowledge bases in a distributed system can cause security problems. The RacerPro server executes statements as described in the sections below. Statements that might cause security problems are `save-tbox`, `save-abox`, and `save-kb`. Files may be generated at the server computer. By default these functions are not provided by the RacerPro server. If you would like your RacerPro server to support these features, startup RacerPro with the option `-u` (for unsafe). You also have to start RacerPro with the option `-u` if you would like to display (or edit) OWL files with RacerEditor and use the facility Evaluate OWL File.
- If RacerPro is used in the server mode, the option `-init <filename>` defines an initial file to be processed before the server starts up. For instance, an initial knowledge base can be loaded into RacerPro before clients can connect.
- The option `-n` allows for removing the prefix of the default namespace as defined for OWL files. See Chapter 4.7 for details.
- The option `-t <seconds>` allows for the specification of a timeout. This is particularly useful if benchmark problems are to be solved using the file interface.

- The option `-debug` is useful for providing bug reports. If an internal error occurs when RacerPro is started with `-debug` a stack backtrace is printed. See Section 2.3 about how to send a bug report.
- With `-una` you can force RacerPro to apply the unique name assumption.
- Sometimes, for debugging purposes it is useful to inspect the commands your application sends to RacerPro. Specify `-log <filename>` to print logging information into a file (see also the command `(logging-on)` to switch logging on dynamically).
- Specify `-temp <directory>` if you would like to change the default directory for temporary files (the default is `/temp` under Unix-based systems and the value of the environment variable `TEMP` or `TMP` under Windows).
- Sometimes using the option `-silent` is useful if you want to suppress any diagnostic output.
- RacerPro supports the DIG protocol with some extensions. For instance, RacerPro interprets the DIG specification generated by Protégé in such a way that DIG attributes are treated as datatype properties in order to match the semantics of OWL. Thus, by default DIG attributes do not imply at most one filler. If you have an application that relies on DIG-1.1 specify the option `-dig-1-1` to instruct RacerPro to obey the original semantics of DIG attributes.
- In case you have problems with the license file `license.racerlicense` you can start RacerPro with the option `-license <license-file>`. This prints information about the file `<license-file>`. More detailed information is printed with `-dump-license-info [<filename>]`.

Note again that under Windows, RacerPro options have to be separated with `--` from window manager options. The following options are important to control the behavior or the console window:

- The option `+p` make the console windows persistent, i.e., you have to explicitly close the window. This is useful to read error messages.
- The option `+c` instructs RacerPro not to open the console window. This option is useful for file-based operation of RacerPro (batch mode).

2.3 How to Send Bug Reports

Although RacerPro has been used in some application projects and version 1.9 has been extensively tested, it might be the case that you detect a bug with respect to a particular knowledge base. In this case, please send us the knowledge base together with the query as well as a description of the RacerPro version and operating system. It would be helpful if the knowledge base were stripped down to the essential parts to reproduce that bug. Before submitting a bug report please make sure to download the latest version of RacerPro.

Sometimes it might happen that answering times for queries do not correspond adequately to the problem that is to be solved by RacerPro. If you expect faster behavior, please do not hesitate to send us the application knowledge base and the query (or queries) that cause problems. In any case, get the latest version of RacerPro first.

As a registered RacerPro user you may send your questions and bug reports to the following e-mail address:

support@racer-systems.com

If you want to submit a bug report or a question about a certain behavior of RacerPro please attach the logfile to your e-mail. Of course the logfile should cover the session in which the error occurred.

Logging (including a stack backtrace in case of an error) is enabled by starting the executable as follows under Windows:

```
$ RacerPro.exe -- -log <filename> -debug
```

or under Unix:

```
$ RacerPro -log <filename> -debug
```

Please include at least this information in your correspondence:

- Your personal name and the name of your organization
- Your operating system
- Your contact data including telephone number
- The logfile (see above)
- The RacerPro build number
- Your transaction ID or the short license string
- A description of your problem and the environment where it occurred.

We may need additional information about your setup or some of the data files you process to simulate the reported error condition. You will greatly decrease our response time if you help us by providing such information on request. Of course we will try to work on your issue as soon as possible. However, due to the probably existing time lag between your location and ours or due to the existing work load our response may take up to two business days. We will acknowledge the receipt of your inquiry and in most cases give you further instructions or an estimate of the processing time.

2.4 RacerPorter

With RacerPorter you can interactively use RacerPro, load knowledge bases, visualize ontologies (T-Boxes) and instances (A-Boxes), send queries and native commands to RacerPro. Start RacerPro and start RacerPorter. Then, in RacerPorter, press the button Connect.

Enter your commands in RacerPro native syntax in the Command pane. The replies of RacerPro are put into the RacerLog pane. Just hit Return in case you need more than one line for your command. The shell will not send the input to Racer until the last parenthesis has been closed. If you press enter on an incomplete expression, RacerPorter just gives you a fresh line, starting with appropriate indentation.

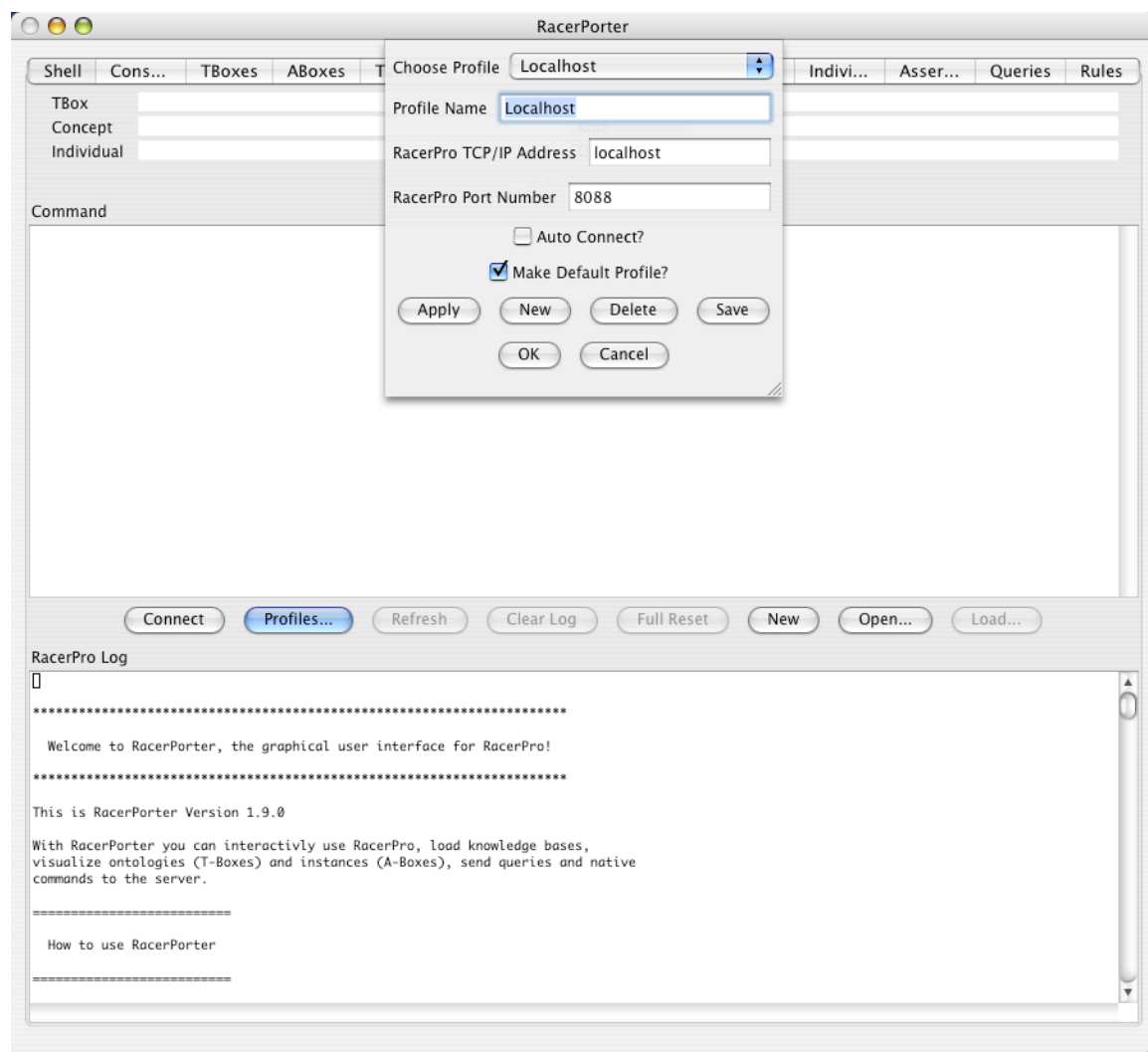


Figure 2.9: The RacerPorter interface to RacerPro.

2.4.1 Preferences

In order to specify preferences simply press the Profiles... button (Mac: choose Preferences... item from the menu), specify the RacerPro host and port to connect to, and then Connect to that RacerPro server. See also Figure 2.9. Note that you can manage different server settings with profiles. The file porter.pot in your home directory is used to store your

profiles. The default profile will be automatically used on startup. You will be connected automatically if “auto connect” is specified in the default profile.

If you are successfully connected you can use the shown tabs to access the functionality of RacerPro.

The six text fields at the top of the application are called the state display of RacerPorter. They indicate the current “state” of RacerPorter, NOT of RacerPro. Selecting, for example, a TBox from the TBoxes list panel updates the TBox field in the state display. The same applies to the other fields of the state display. The state is used to indicate the current object which is to be inspected or queried with RacerPorter - for example, the Individuals tab will always show the individuals of the ABox in the state display, the Taxonomy tab will always show the taxonomy of TBox in the state display etc.

Note that selecting an ABox (TBox) from the ABoxes (TBoxes) pane does NOT change the (**current-abox**) (resp. (**current-tbox**)) of RacerPro. Use the buttons Set Racer TBox and Set Racer ABox for this purpose if you “really” want to change the state of the RacerPro server. The (**current-abox**) and (**current-tbox**) of the RacerPro server is indicated in the TBoxes resp. ABoxes tab by marking the TBoxes resp. ABoxes name like this: >>> DEFAULT <<<.

2.4.2 RacerEditor

RacerPorter also includes an Emacs-style text editor (called RacerEditor) which you can use to view, edit and process your knowledge bases. To open the Editor just click on the New button or push the Open... button in the Shell tab and select a text file that should be read into the Editor.

You can open virtually any file with the Editor, from plain text to RDF/OWL files and those in RacerPro syntax. The editor offers a simple syntax sensitive support by displaying text in different colors and by indicating closing parenthesis.

At the bottom of the window, status lines are displayed that are typical to Emacs. They indicate a keystroke combination and give other feedback to user interaction in its first line. The bottom line shows more general information: the connected RacerPro server (if actively connected) is shown first. As usual for Emacs-like editors, a modification indicator is shown next (---- means buffer unmodified, -*- means buffer modified). Then, buffer-related information follows to the right: name of the edited file, recognized syntax of the content (“XML” for RDF/OWL files, “Lisp” for RacerPro files and “Fundamental” for all others) as well as the path of the imported file.

Please notice that the Editor is similar to Emacs and therefore not behaving like typical Windows-based text processor. For instance, browsing through the text with the help of the window scroll bar may move the cursor too. Also you can not select the text by holding down the shift key and move the cursor with the arrow keys. Rather, highlight the text by clicking and shift-dragging the mouse over the interesting passages: then right-click the mouse to select if you want to copy or cut the text from the window. Remember that the control key, especially the control-C keystroke combination has a different meaning in Emacs and analogously in RacerEditor.

In the following we list the most important key bindings.

- Return or Enter: fresh line or send command
- Tab: Completion key
- Ctrl-a: Beginning of line
- Ctrl-e: End of line
- Ctrl-k: Kill line
- Ctrl-left: Matching starting ”(”
- Ctrl-right: Matching ending ”)”
- Ctrl-d / Del: Delete
- Backspace: Backspace
- Ctrl-Space: Set Mark
- Meta-w: Copy
- Ctrl-y: Paste
- Ctrl-w: Cut
- Meta-p / Alt-p: Previous Command
- Meta-n / Alt-n: Next Command

2.4.3 Tabs in RacerPorter

In the Taxonomy tab, select a concept from the taxonomy, then use Concept Query to retrieve the instances of the selected concept. Note that the Concept field of the state display always shows the current Concept. Adapt the Tree Depth accordingly. Note that Tree Depth is ignored if Graph display mode is selected. If you push Focus while in Tree display mode, then only the subconcepts of the current Concept will be shown.

In the Individuals tab, select an individual, press Direct Types or All Types, then go to the Taxonomy tab. It will highlight the types of the selected individual (note the Individual field in the state display).

Select the Network tab afterwards in order to show the ABox structure focusing on the selected individual. Adapt the Tree Depth accordingly. Then push Focus. You can always change the focus individual by simply selecting it. The focus individual is shown in the Individual field at the top. To display the complete ABox structure, simply push the Reset button. Note that Tree Depth is ignored if Graph display mode is selected. You can also focus on a subset of the ABox edges - simply select the roles you are interested in from the Roles tab, then check out Selected Roles (Roles Tab) in the Network tab. Then, only the edges labeled with “selected roles” are shown.

Note that the Roles tab is the only tab which allows multiple selection of items. The last selected role is always the Current Role, as shown in the Role text field in the state display. The other list panes only allow a single selection.

In the Queries or Rules tab, select a nRQL query from the list, then use the buttons to apply a command on the selected query. Note that the current query or rule is shown in the Query or Rule field in the state display at the top.

2.4.4 Known Problems

Note that the JPEG image shown in the About tab will only work on Linux if you have `/user/lib/libImlib.so` installed (SuSE: `ln -s /opt/gnome/lib/libImlib.so.1.9.14 /user/lib/libImlib.so`).

2.5 Other Graphical Client Interfaces for RacerPro

In this section we present open-source Java-based graphical client interfaces for the RacerPro Server. The examples require that RacerPro be started with the option `-u`. The first interface is the RICE system. Afterwards, a short presentation of the Protégé system with RacerPro as a backend reasoner is given. Then, the coordinated use of Protégé and RICE or RacerPorter is sketched.

2.5.1 RICE

RICE is an acronym for RACER Interactive Client Environment and has been developed by Ronald Cornet from the Academic Medical Center in Amsterdam. RICE is provided with source code. The executable version is provided as a jar file. Newer versions of RICE can be found at <http://www.blg-systems.com/ronald/rice>.

In order to briefly explain the use of RICE let us consider the family example again. First, start the RacerPro server. As an example, the family knowledge base might be loaded into the server at startup.

```
RacerPro -u -init family.racer
```

Or, under windows, type:

```
RacerPro -- -u -init family.racer
```

Then, either double-click the jar file icon or type `java -jar rice.jar` into a shell. Connect RICE to RacerPro by selecting Connect from the Tools menu. In addition to the default T-box (named `default`) always provided by RacerPro, the T-box “FAMILY” is displayed in the left upper window which displays the parent-children relationship between concept names of different T-boxes. An example screenshot is shown in Figure 2.10. Users can interactively unfold and fold the tree display.

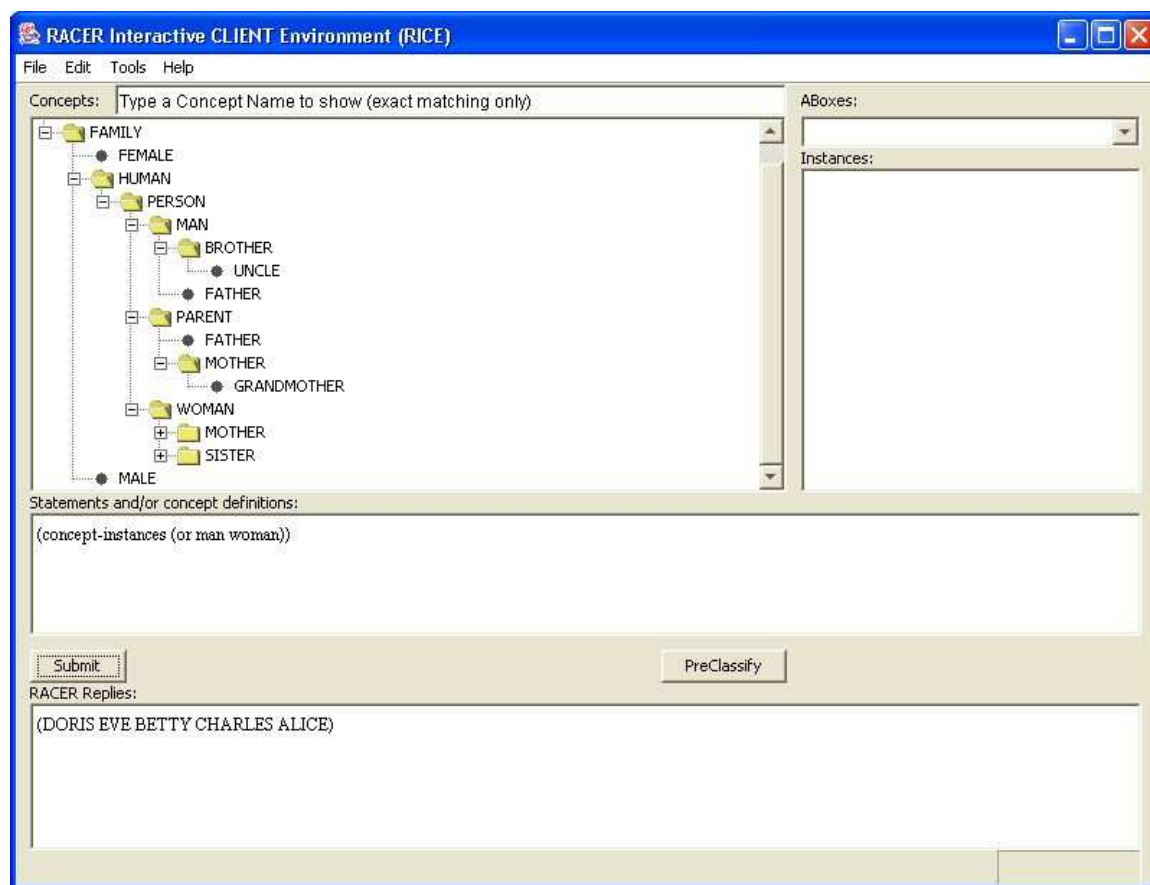


Figure 2.10: Screenshot of RICE.

Using the middle subwindow, statements, declarations, and queries can be typed and submitted to RacerPro. The answers are printed into the lower window. A query concerning the family knowledge base is presented in Figure 2.10. The query searches for instances of the concept (or man woman). Other queries (e.g., as those shown in the previous section) can be submitted to RacerPro in a similar way.

An example for submitting a statement is displayed in Figure 2.11. The family knowledge base is exported as an OWL file.

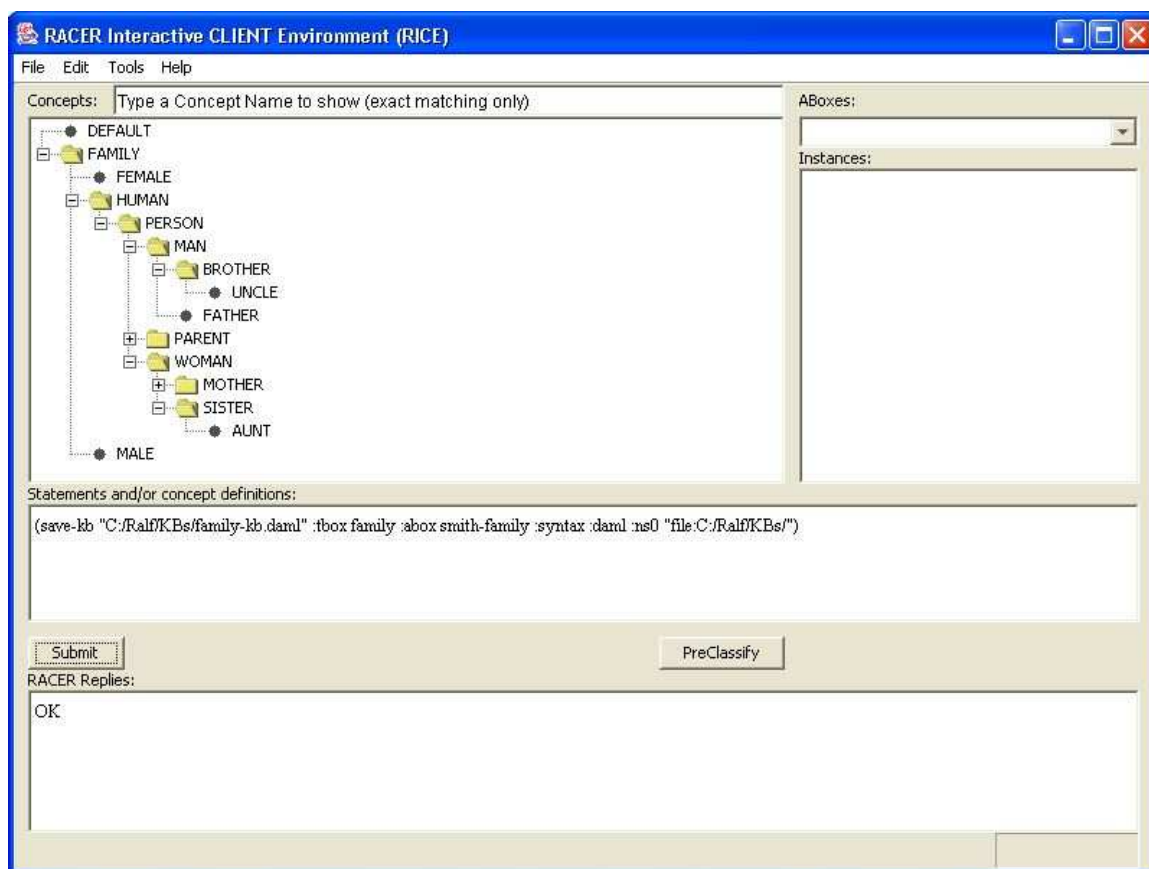


Figure 2.11: Screenshot of RICE.

2.5.2 Protégé

Protégé may be used as another graphical interface for RacerPro. Protégé is available from <http://protege.stanford.edu>. While RICE can be used to pose queries, in particular for A-boxes, Protégé can be used to graphically construct T-boxes (or ontologies) and A-boxes.

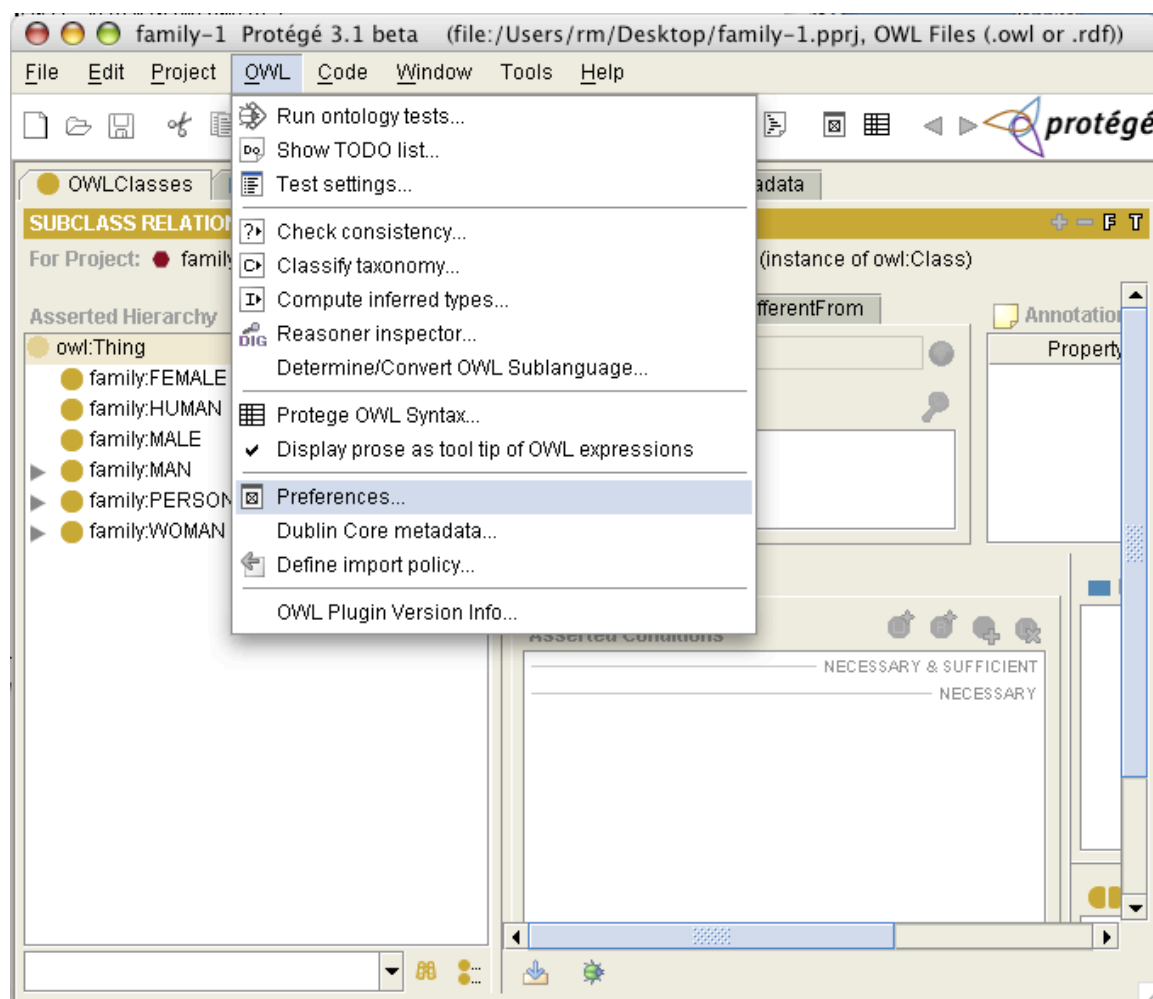


Figure 2.12: First step for declaring RacerPro as the reasoner used by Protégé.

In order to declare RacerPro as the standard reasoner used by Protégé, select the Preferences menu in Protégé (see Figure 2.12).

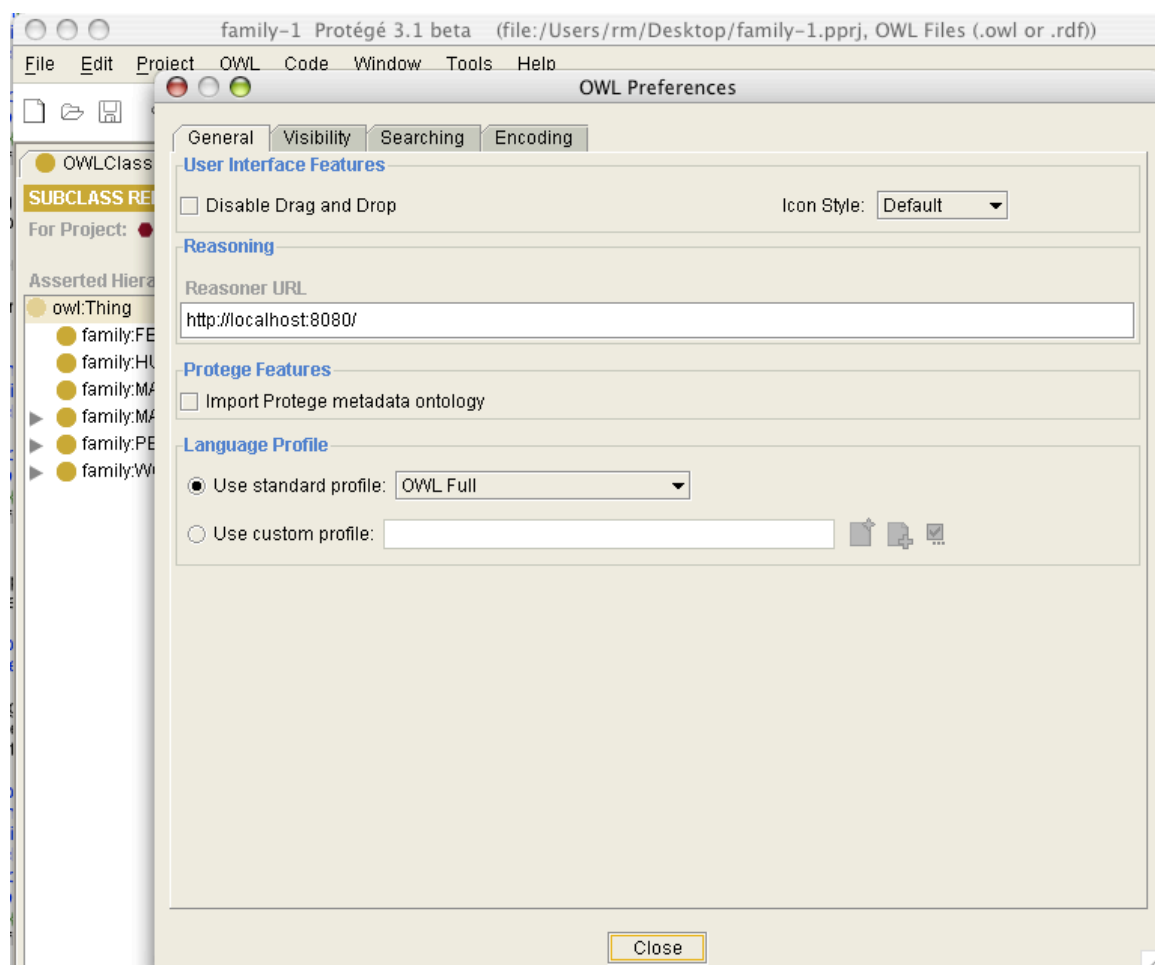


Figure 2.13: Second step for declaring RacerPro as the reasoner used by Protégé.

In the preferences dialog window specify host and port number (see Figure 2.13). Usually, the defaults (localhost:8080) are ok. Another example would be 135.28.70.104:8081. After making required changes press the close button.

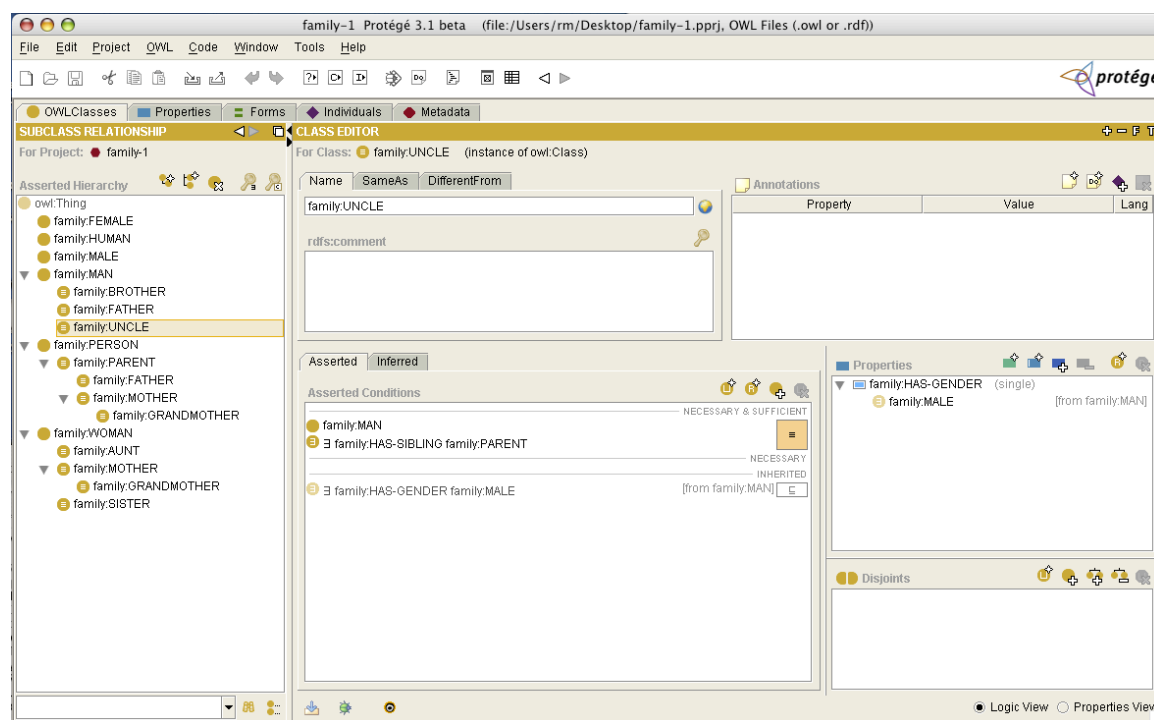


Figure 2.14: Protégé displaying the Family knowledge base. The class `UNCLE` is selected, and in the subwindow for asserted conditions, necessary and sufficient conditions are shown graphically.

The example presented in Figure 2.14 shows a screenshot of Protégé with the Family knowledge base. The knowledge base was exported from RacerPro using the OWL syntax (see the function `save-kb`). OWL files can be manipulated with Protégé.

In Figure 2.14 the concept `uncle` is selected (concepts are called classes in Protégé). See the restrictions displayed in the asserted conditions window and compare the specification with the KRSS syntax used above (see the axiom for `uncle`). Note that the class window displays only obvious subclass relationships. As we will see later, RacerPro can be used to also compute all implicit class subsumption relationships. Reasoning services are provided when Protégé is connected to RacerPro. This can be easily accomplished.

The role hierarchy of the Family example is shown Figure 2.15.

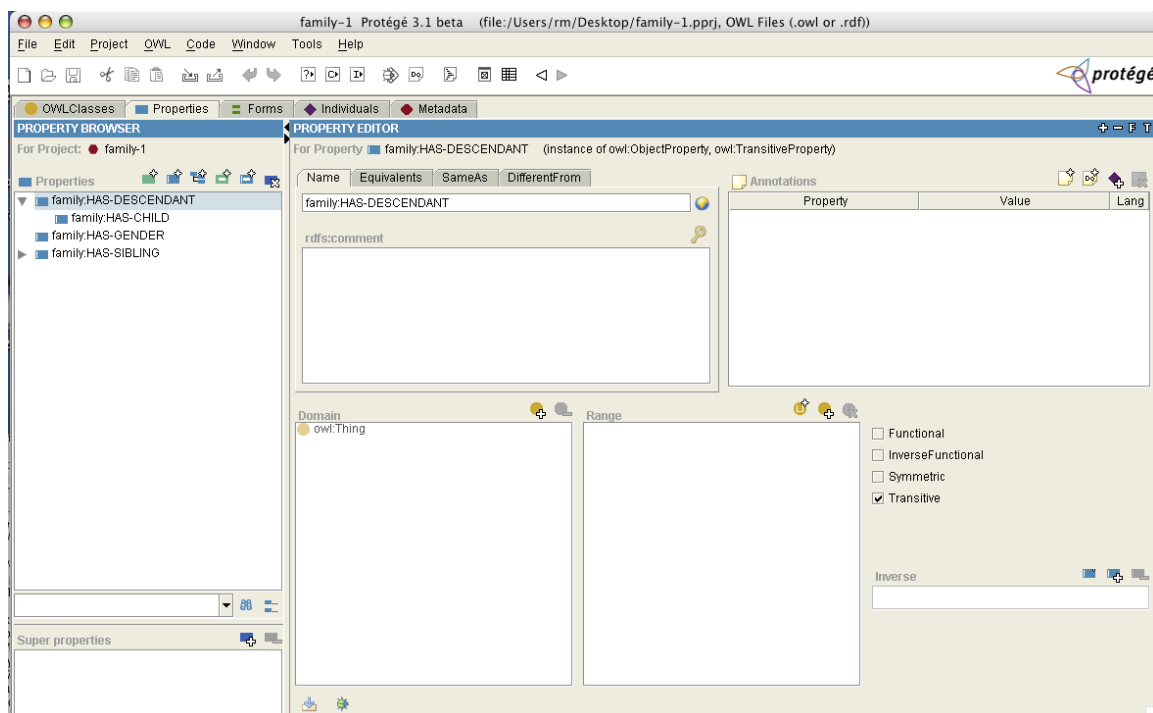


Figure 2.15: Display of the role hierarchy with Protégé.

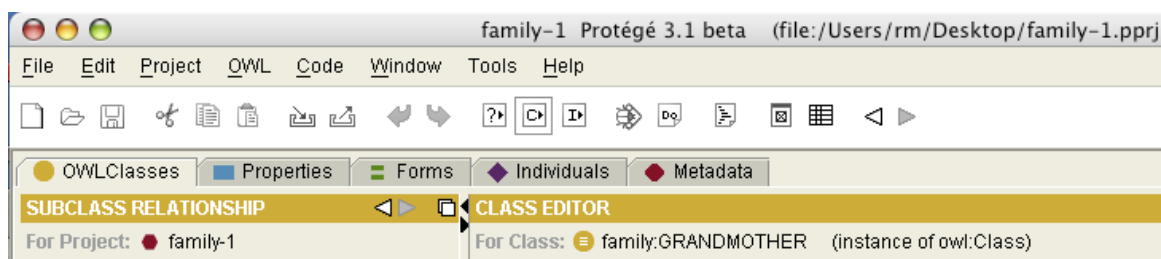


Figure 2.16: Protégé icon panel.

The Protégé icon panel provides for buttons labeled with **?**, **C**, **I**. They are used to check the “consistency” (coherence) of the ontology, classify the ontology, and to compute the inferred types of individuals, respectively.

Press the **C?** button in the tools bar to let RacerPro check for unsatisfiable concept names in the current knowledge base and find implicit subsumption relationships between concept names.

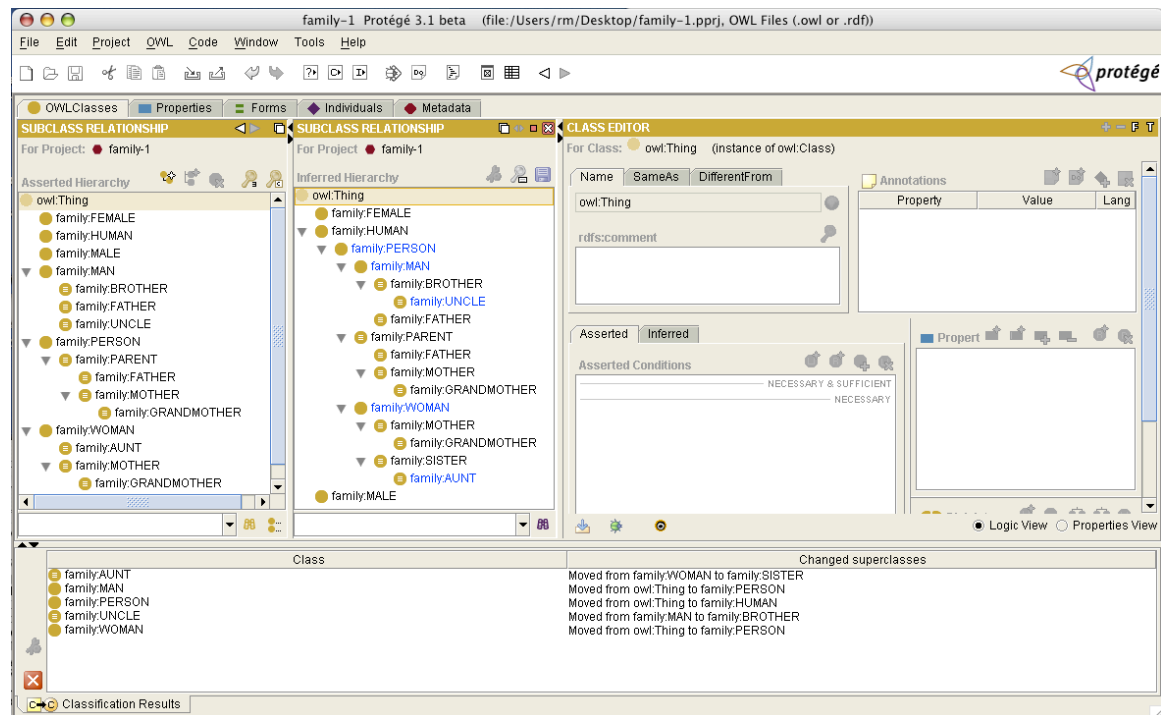


Figure 2.17: Protégé visualizes implicit subsumption relationships computed by RacerPro.

In Figure 2.17 it is indicated that RacerPro reveals implicit subsumption relationships. Implicit subsumption relationships are indicated in the Protégé window “Inferred Hierarchy” (see Figure 2.17). For instance, an **uncle** is also a **brother**. Protégé nicely summarizes classification results in the list of changed superclasses.

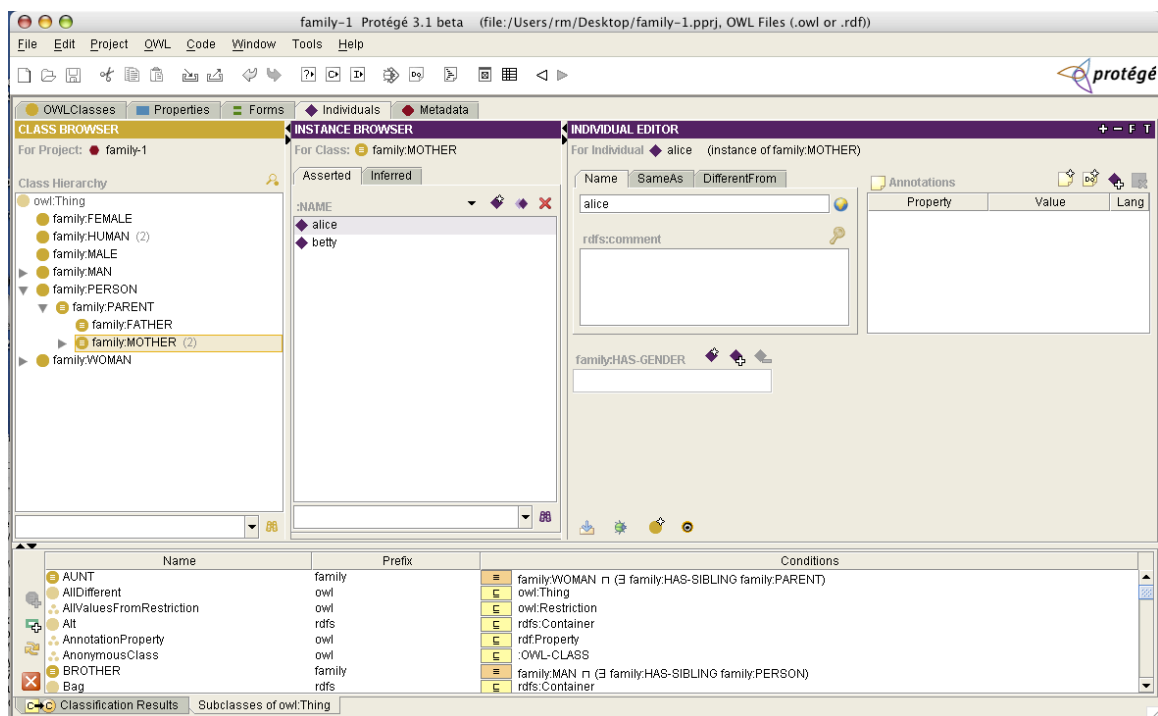


Figure 2.18: The individuals of the family example. Properties of the selected individual **alice** are displayed.

After the tab **Individuals** is selected, Protégé displays the individuals of the family knowledge base together with their properties (or relationships). This is shown here in Figure 2.18.

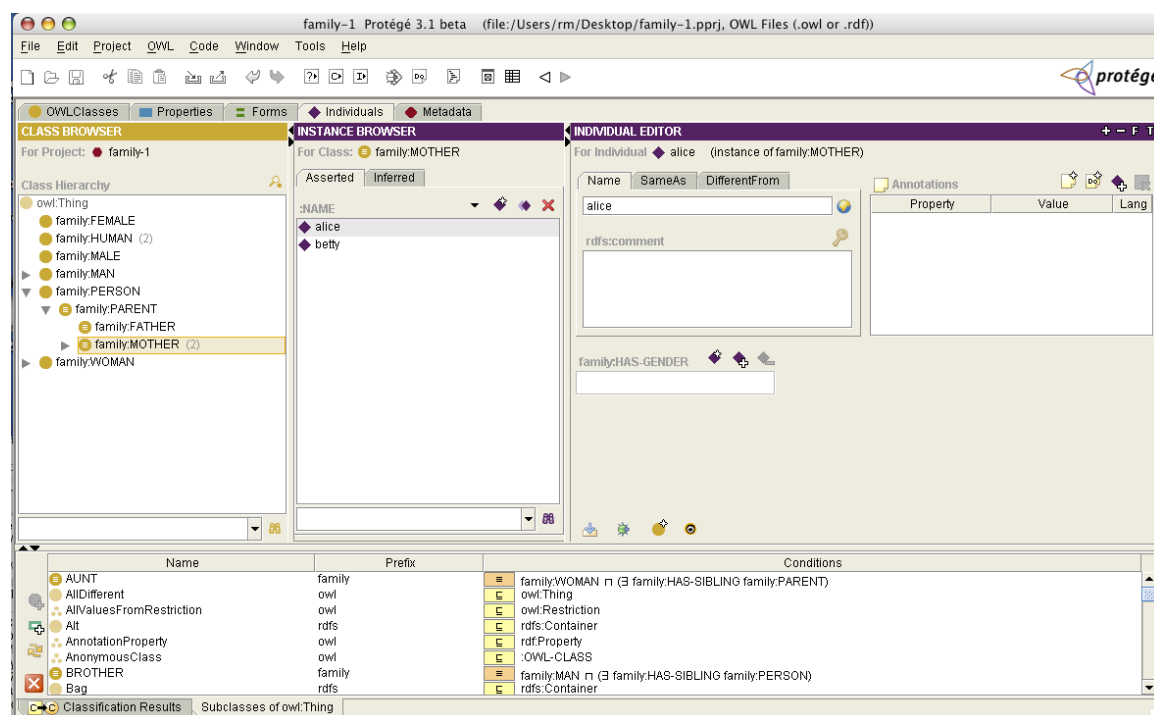


Figure 2.19: Screenshot of Protégé displaying inferred information about UNCLE.

If you press the I button in Protégé, the inferred types are computed for the individuals (aka instances or nominals) in the ontology (see also Figure 2.19).

Although Protégé can be used to display (and edit) A-boxes, let us return to RICE for a moment to examine the result of querying an A-box. In our example we assume that the RICE window is still open. Select the concept PERSON in the concept window. The instances of PERSON are displayed in the upper-right instance window.

2.5.3 Using Protégé and RacerPorter in Combination

Let us assume, RacerPro is started, Protégé is connected to RacerPro, and some knowledge base verifications on `pizza.owl` (a Protégé example knowledge base) are to be performed. In Figure 2.20 you can see the pizza ontology loaded into Protégé. Ontology verification is started by pressing the icon labeled with “C” (classify). The results are shown in Figure 2.21

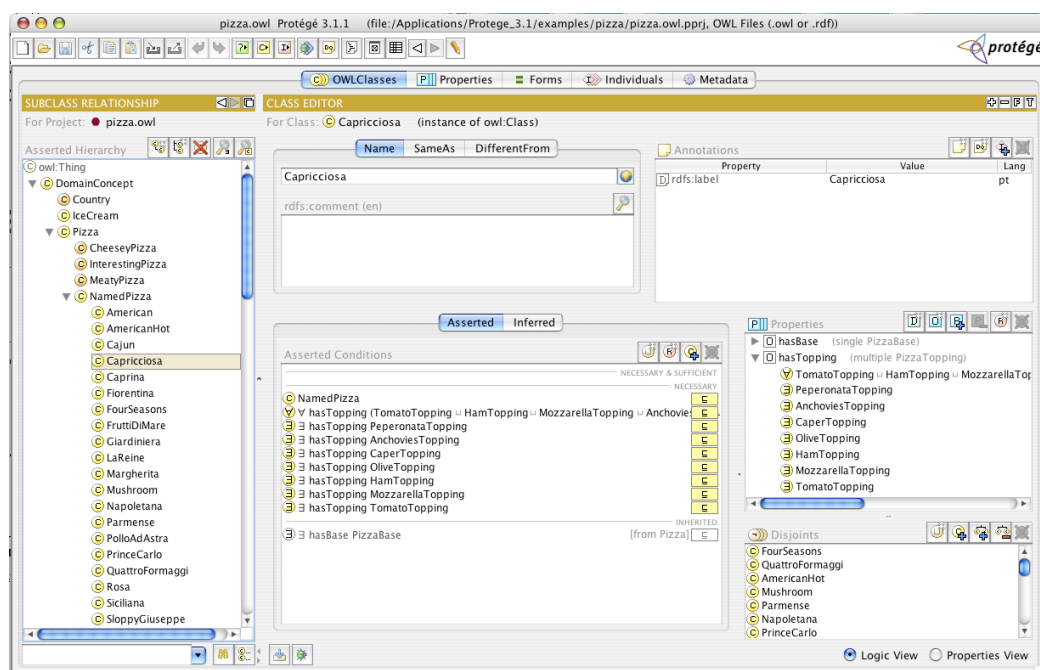


Figure 2.20: The pizza ontology loaded in Protégé.

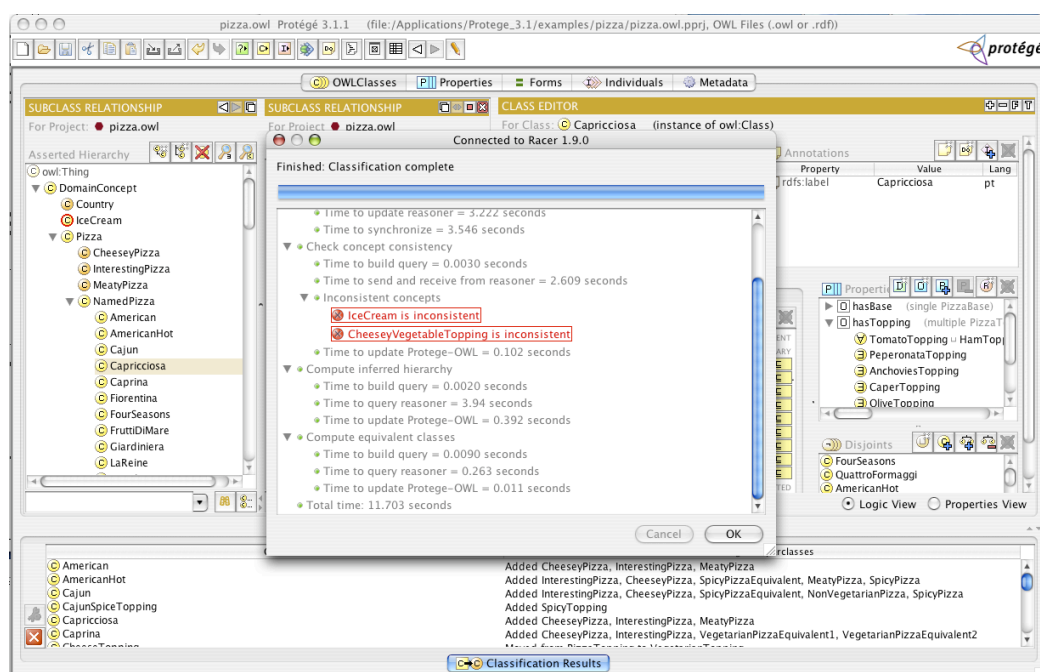


Figure 2.21: Some classes are inconsistent in pizza.owl.

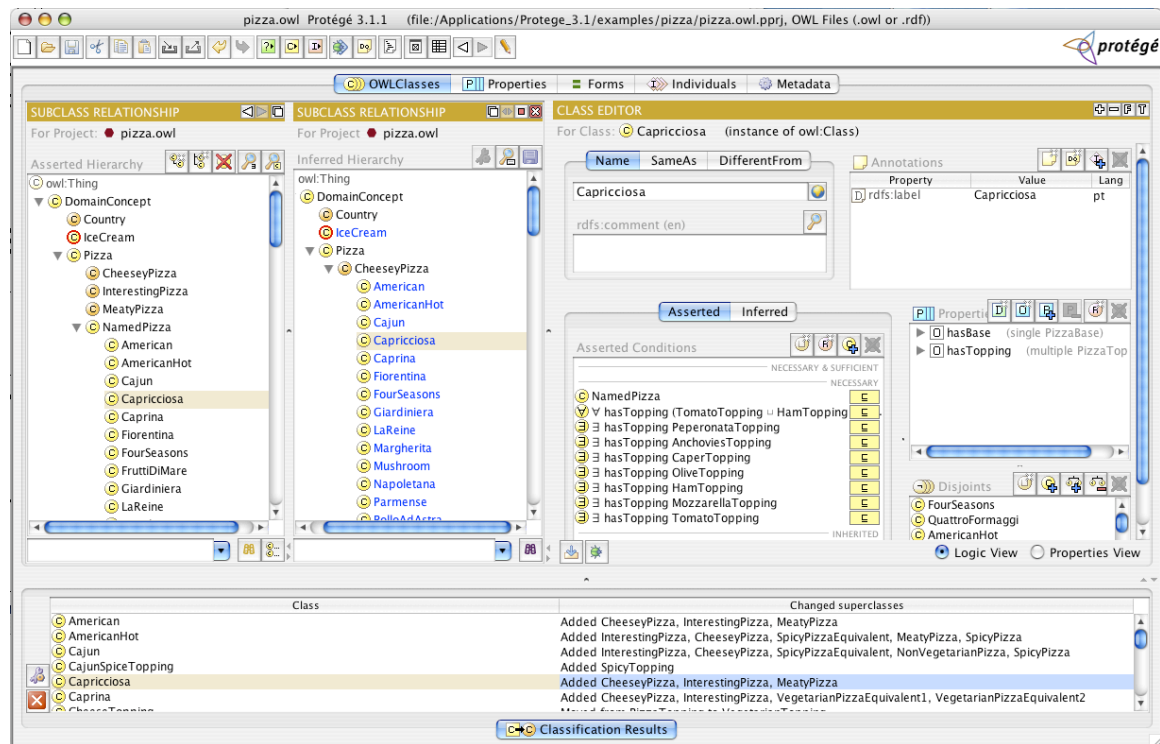


Figure 2.22: Inferred information about pizza classes is shown in the lower pane.



Racer Systems GmbH & Co. KG — <http://www.racer-systems.com>



Figure 2.24: RacerPorter used to query an ontology edited and verified with Protégé.

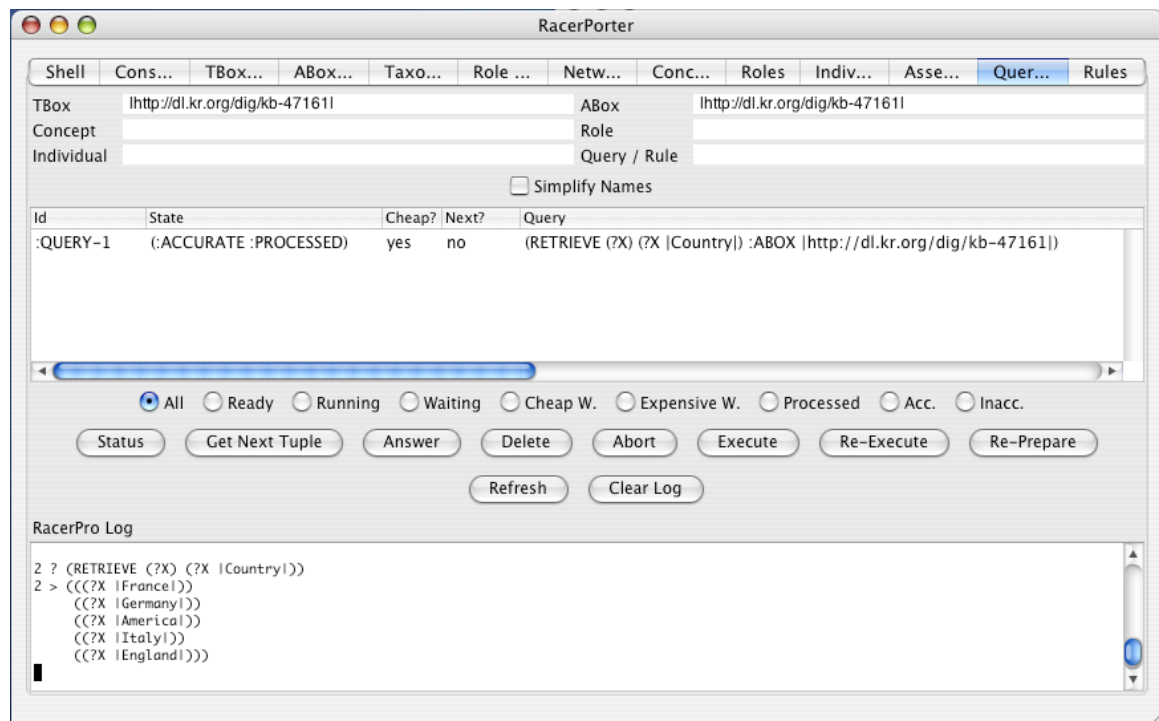


Figure 2.25: Using RacerPorter, queries can be inspected as objects.

2.5.4 TopBraidComposer

TopBraidComposer (www.topbraid.com) is a well-supported commercial ontology editor (see Figure 2.26). Currently, TopBraid Composer can be used via the DIG interface. In the near future, a more efficient dynamic link library will support a more seamless integration of RacerPro and TopBraid Composer.

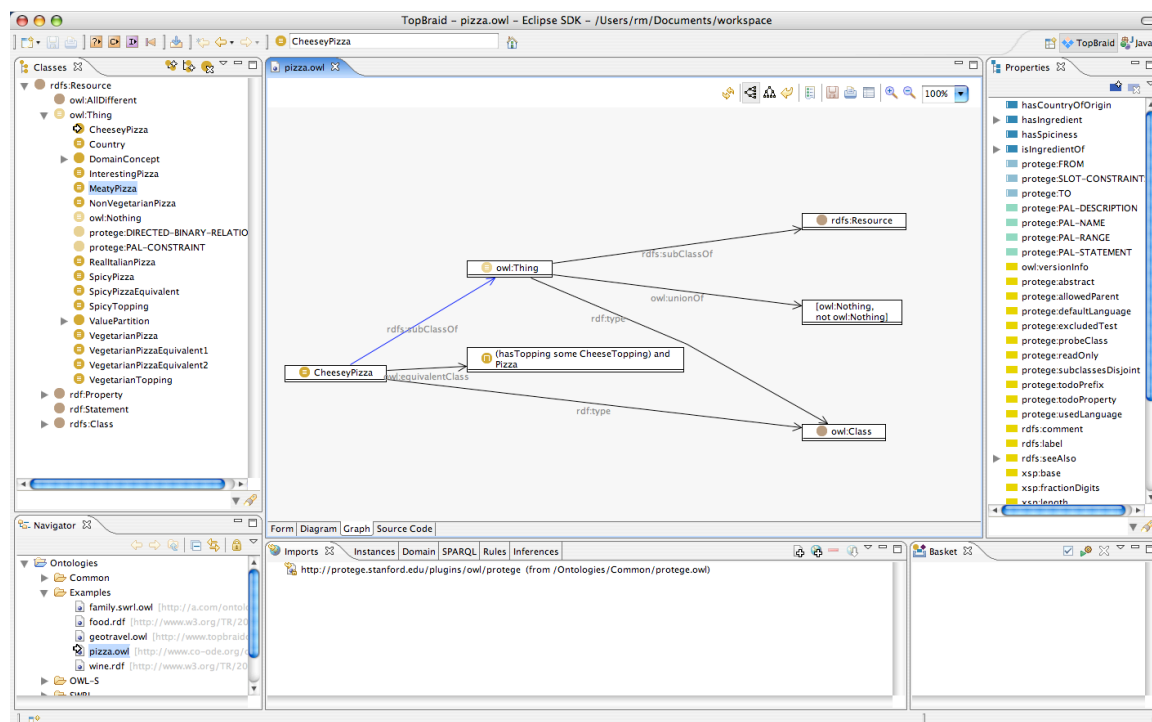


Figure 2.26: The TopBraid Composer interface for displaying and editing ontologies.

2.6 SWRL: Semantic Web Rule Language

The application of rules provides for optimized manipulation of structures, in particular in server-based environments such as RacerPro. Standards such as SWRL ensure the necessary consolidation such that industrial work becomes possible. Therefore, RacerPro has been extended with support for applying SWRL rules to instances mentioned in an OWL ontology or corresponding RDF data descriptions. The RacerPro SWRL rule engine is currently being extended to cope with OWL datatypes. A first experimental SWRL implementation is part of RacerPro 1.9.

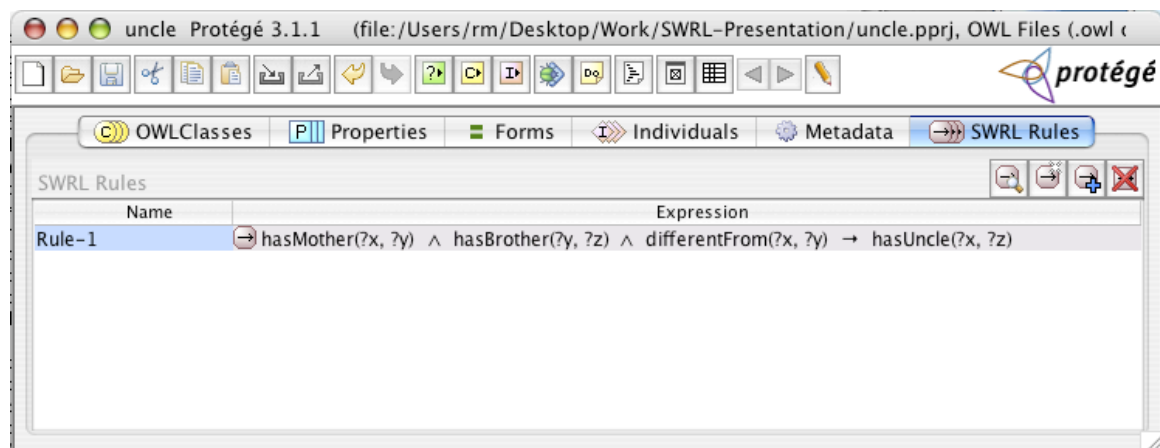


Figure 2.27: Snapshot of the famous “uncle rule” in Protégé.

Rules can be edited, for instance, with the graphical tool Protégé (see Figure 2.27). Use the menu Project in Protégé and select Configure. . . . Then, tick the check box SWRLTab in order to activate the SWRL editor in Protégé. After the ontology is saved, the OWL/SWRL specification can be interpreted by applications. For this purpose, a reasoner such as RacerPro is required. In Figure 2.28 the source code of the “uncle rule” is shown in the RacerPro Editor. The editor also shows some instances, MARY has a mother SUE who, in turn, has a brother JOHN. The rule is responsible for asserting that MARY has an uncle who is JOHN. With RacerEditor the OWL/SWRL file can be sent to RacerPro by selecting a menu item (or pressing a key combination).



Figure 2.28: RacerEditor showing the OWL/SWRL code of the example.

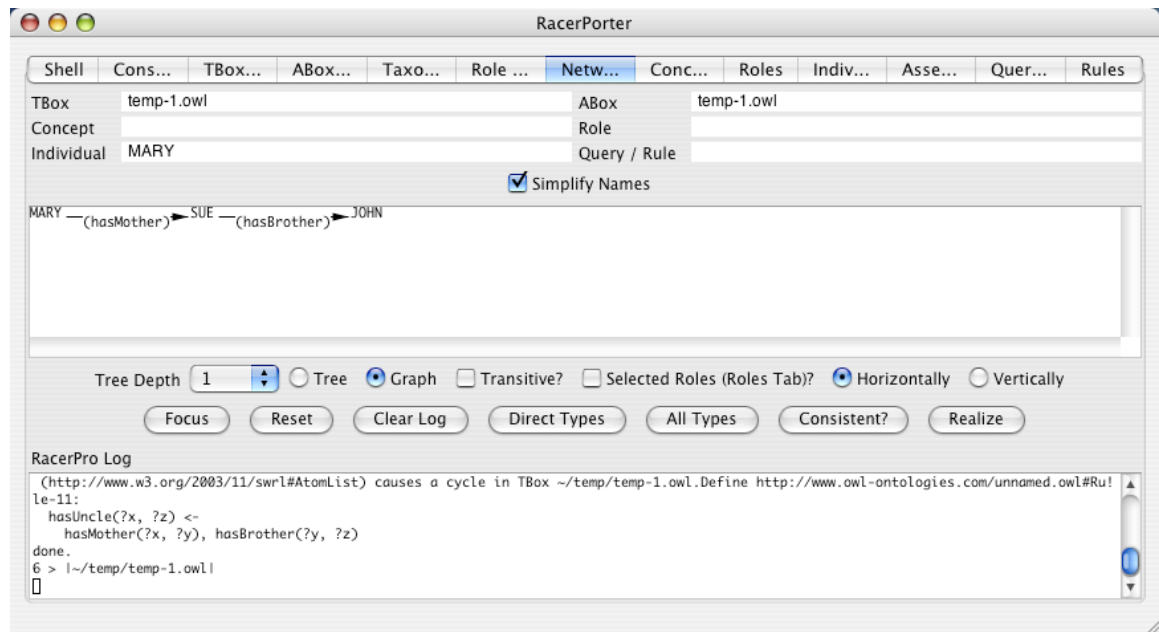


Figure 2.29: Graphical display of the instances and their relations.

In Figure 2.29 we use the network inspector of RacerPorter to have a look at the instances and their relations (properties).

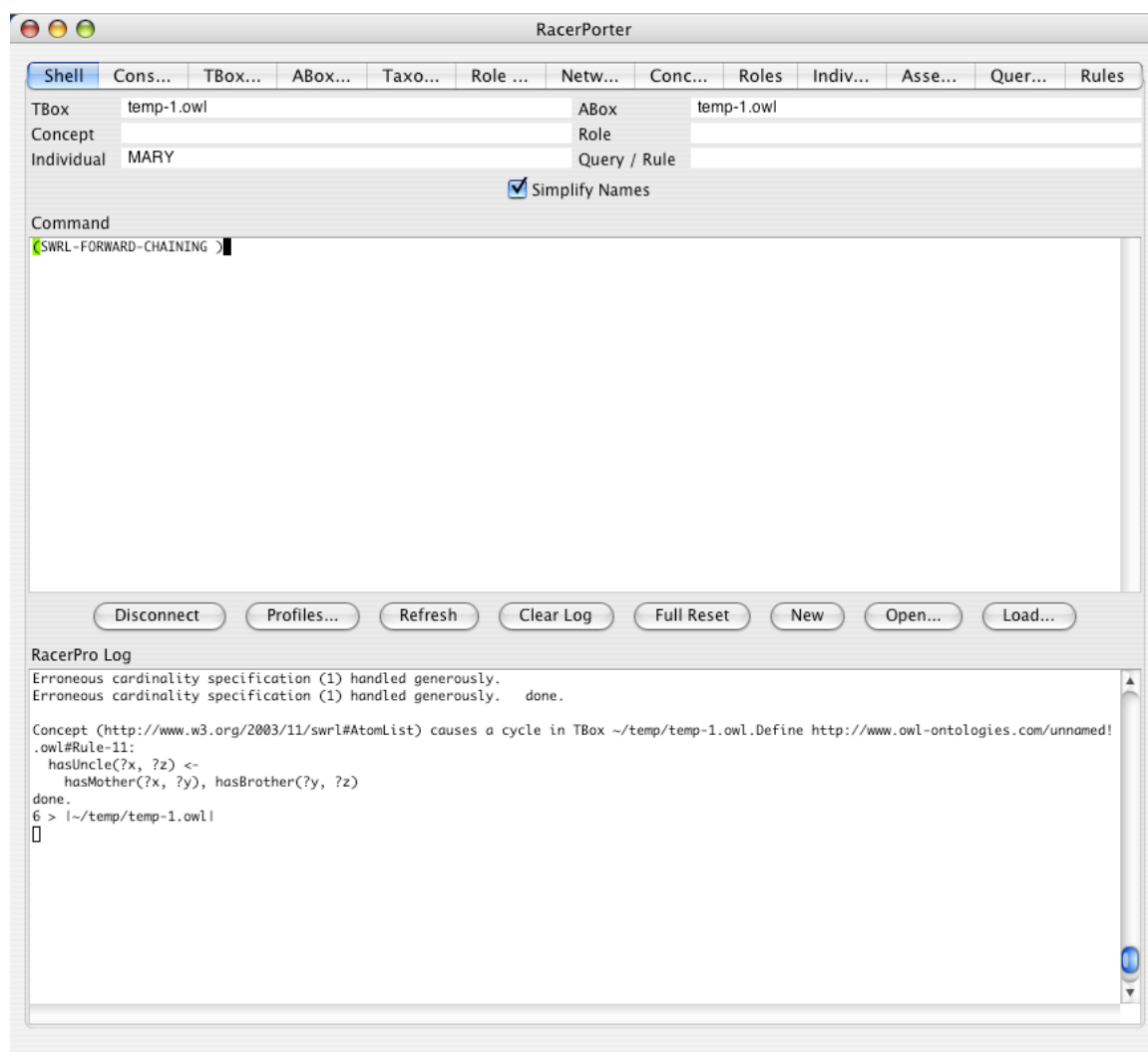


Figure 2.30: Shell window used for starting the rule engine.

For demonstration purposes we start the rule engine by typing a command into the RacerePorter shell window (Figure 2.30). RacerPro offers a forward chainer that applies SWRL rules until no new information is added.

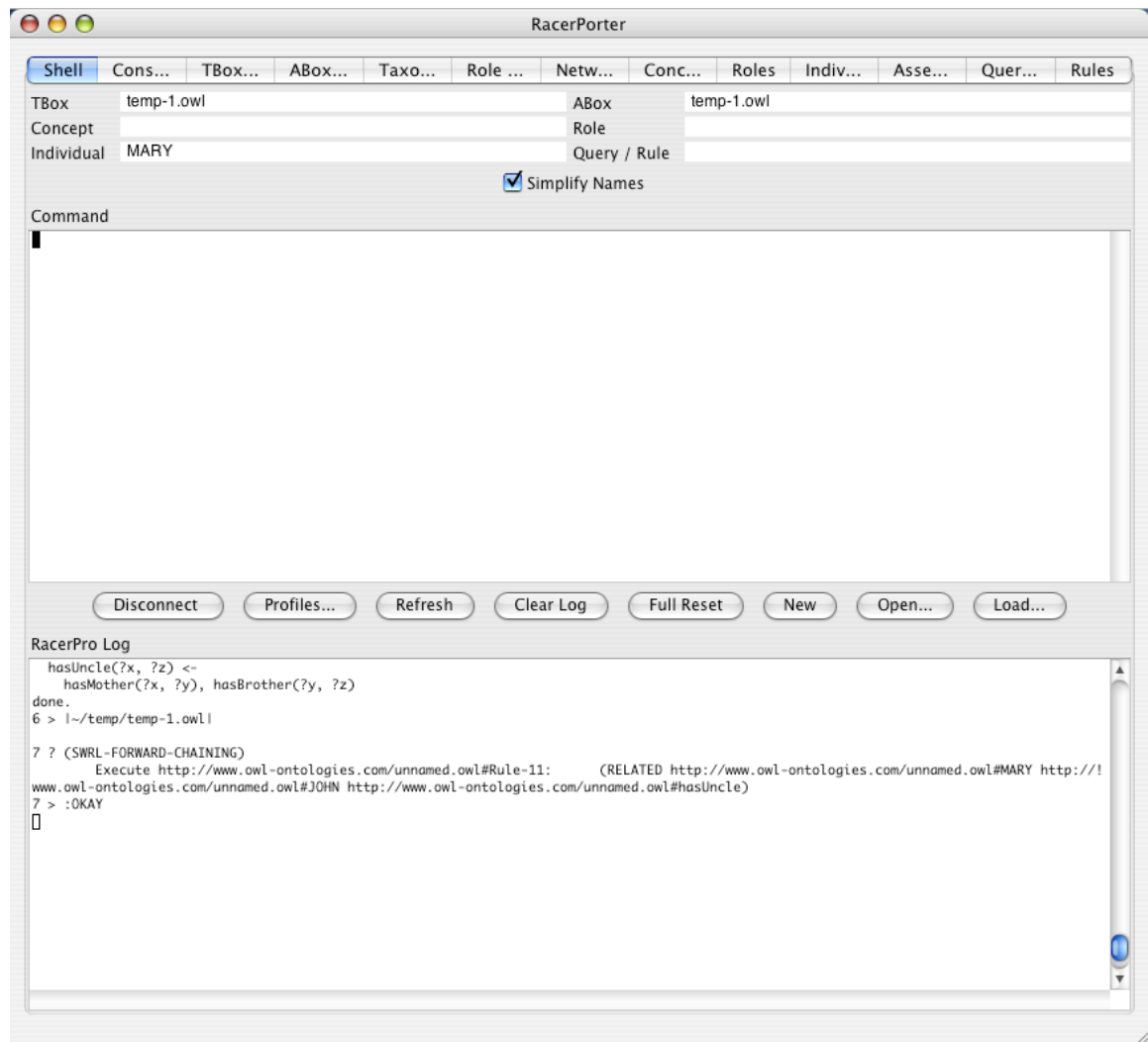


Figure 2.31: The uncle relation is established.

In Figure 2.31 we see that RacerPro fired the rule accordingly. The network tab of RacerePorter allows us to analyze the effect graphically (see Figure 2.32). With RacerePorter, rules can be inspected, rerun, etc. (see Figure 2.33).

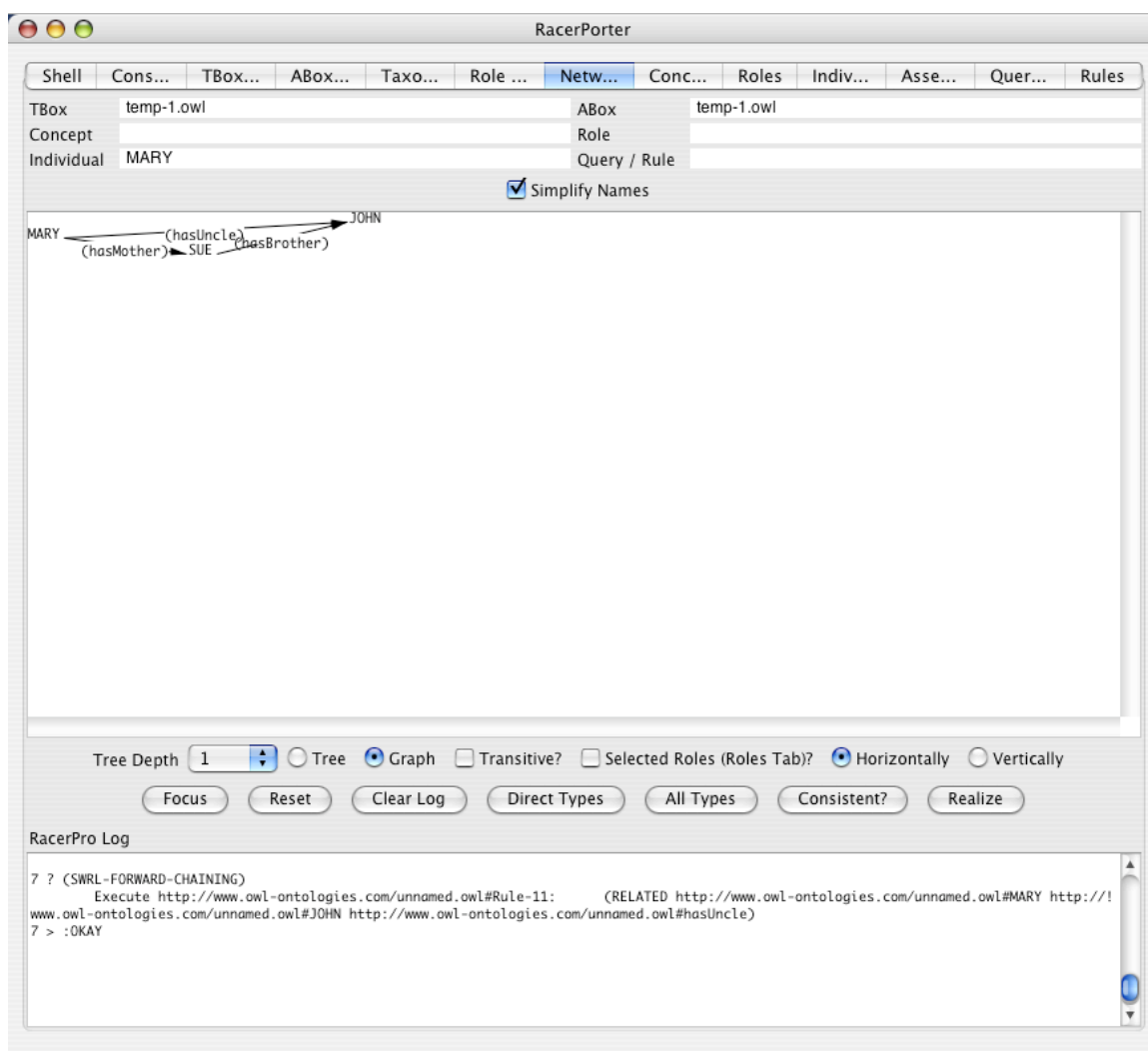


Figure 2.32: Network view indicating the asserted uncle relation between MARY and JOHN.

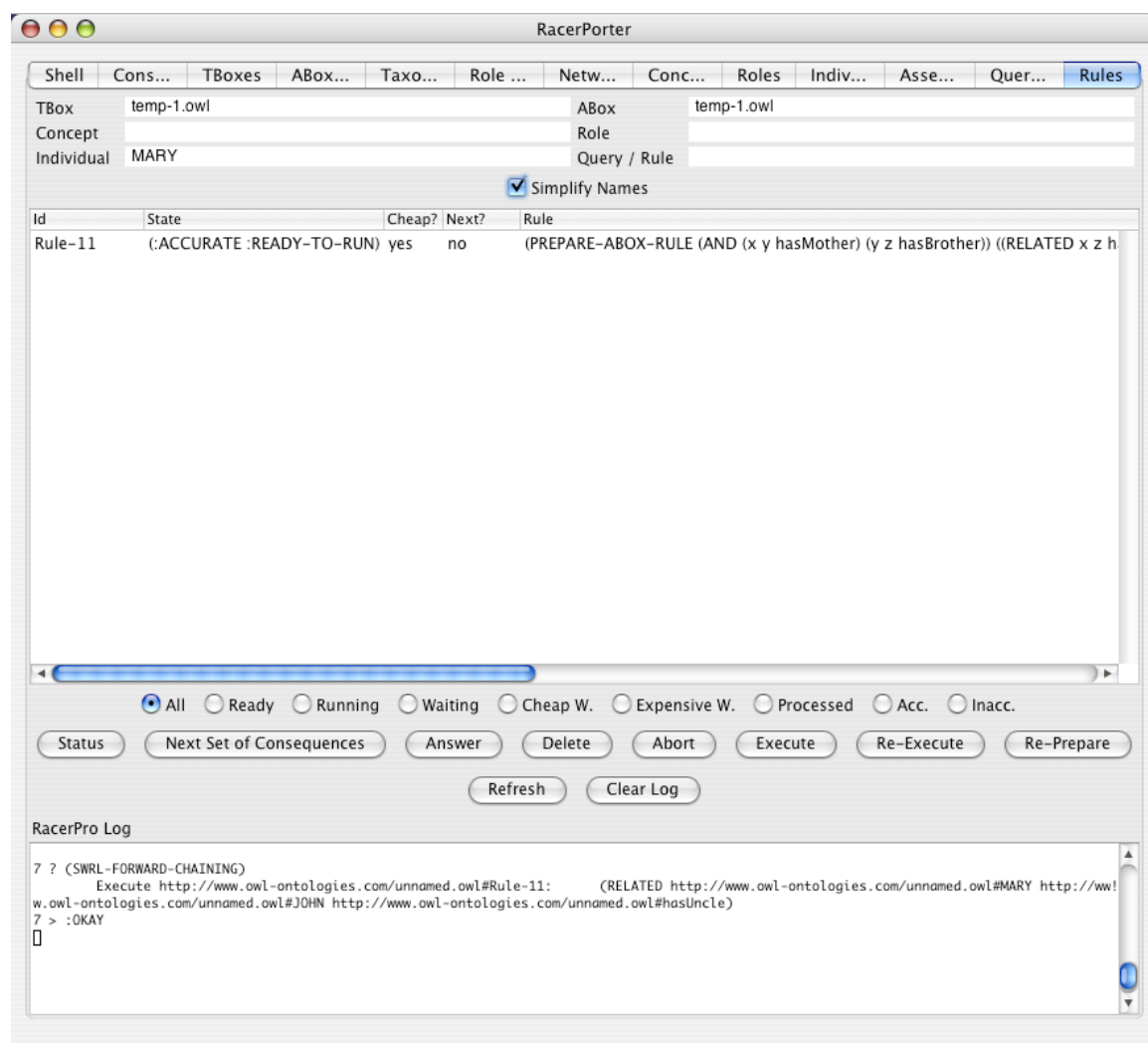


Figure 2.33: Rules tab in RacerPorter.

Various semantics have been proposed for rule languages. Rules have a body and a head. If there exists a binding for the variables in the body such that the predicates (either unary or binary predicates are possible in SWRL) are satisfied, then the predicate comprising the head also holds. The predicates in the body are also called the precondition and the head is the consequence. Variables are bound to a finite set of individuals, namely those, explicitly mentioned in the ontology (or A-box). The question is whether, given a specific binding for variables, the precondition has to be satisfied in *one* “world” or in *all* worlds. We call the former semantics the first-order semantics, whereas the latter is called the rule semantics.

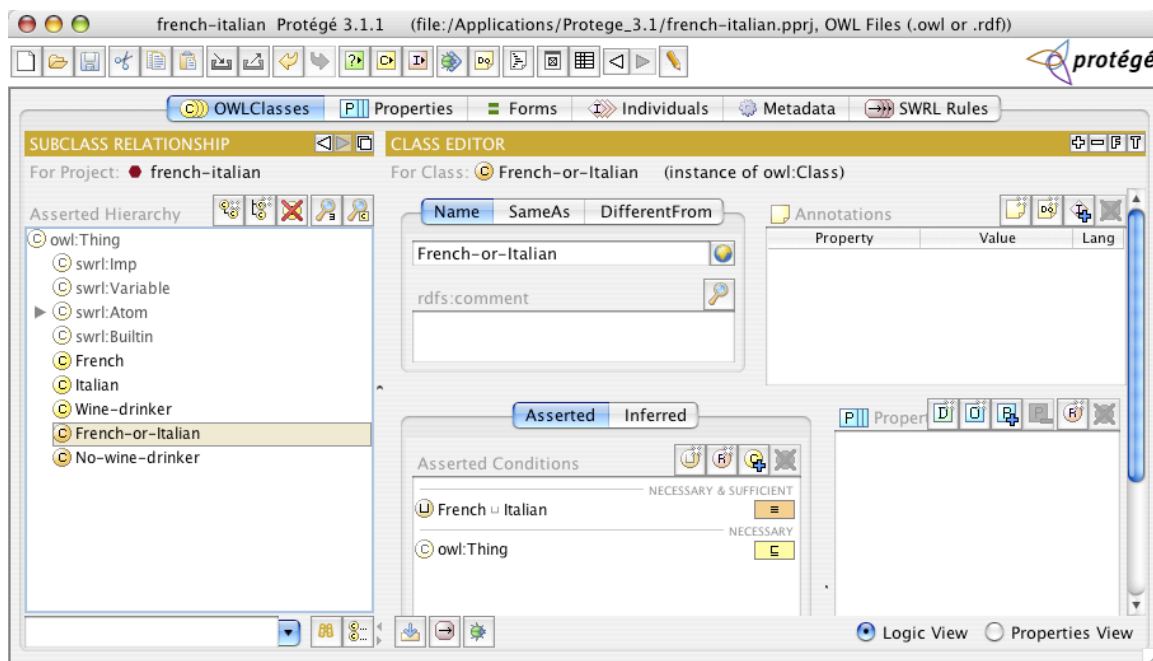


Figure 2.34: Necessary and sufficient conditions for French-or-italian.

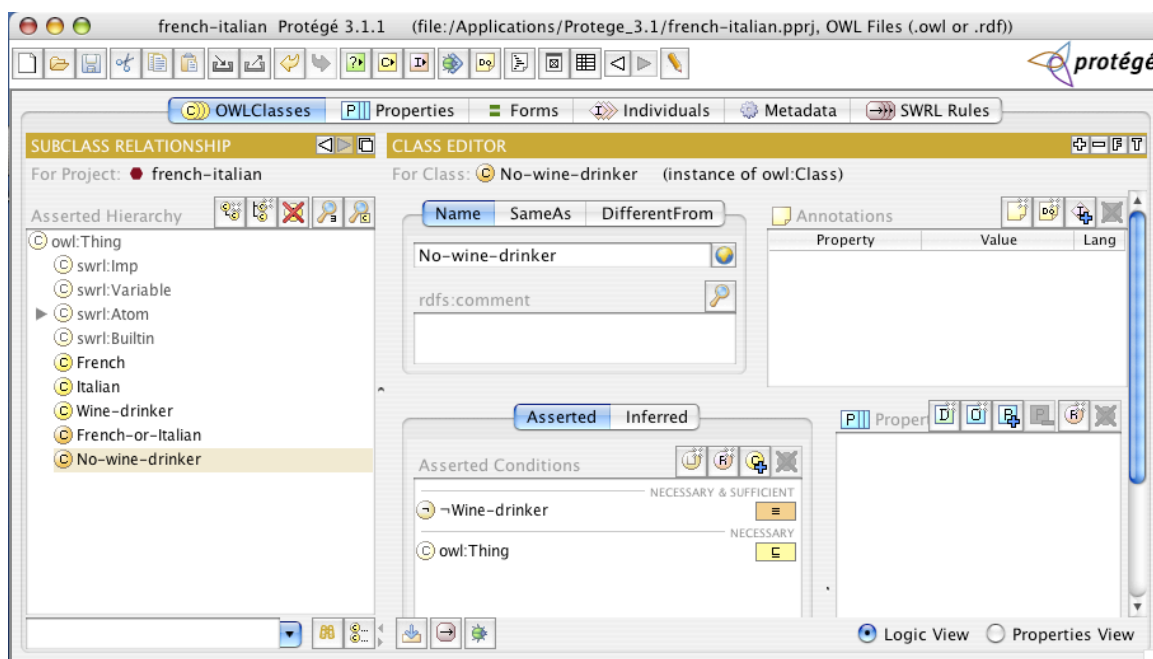


Figure 2.35: Necessary and sufficient conditions for Not-wine-drinker.

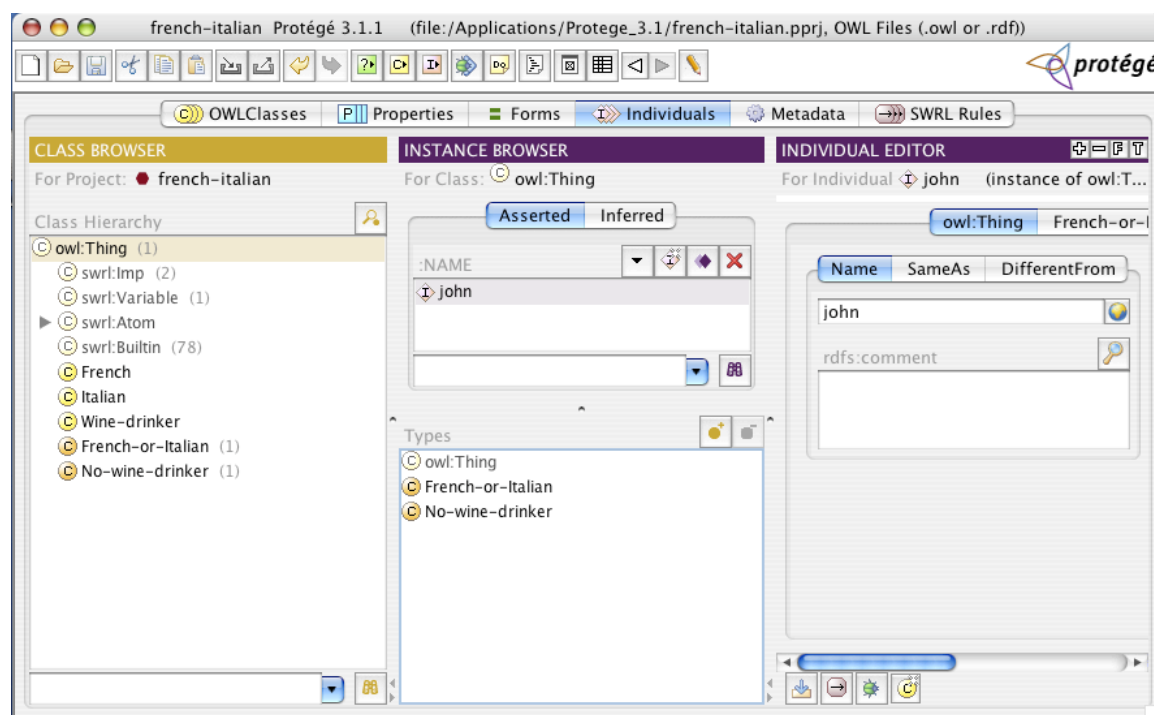
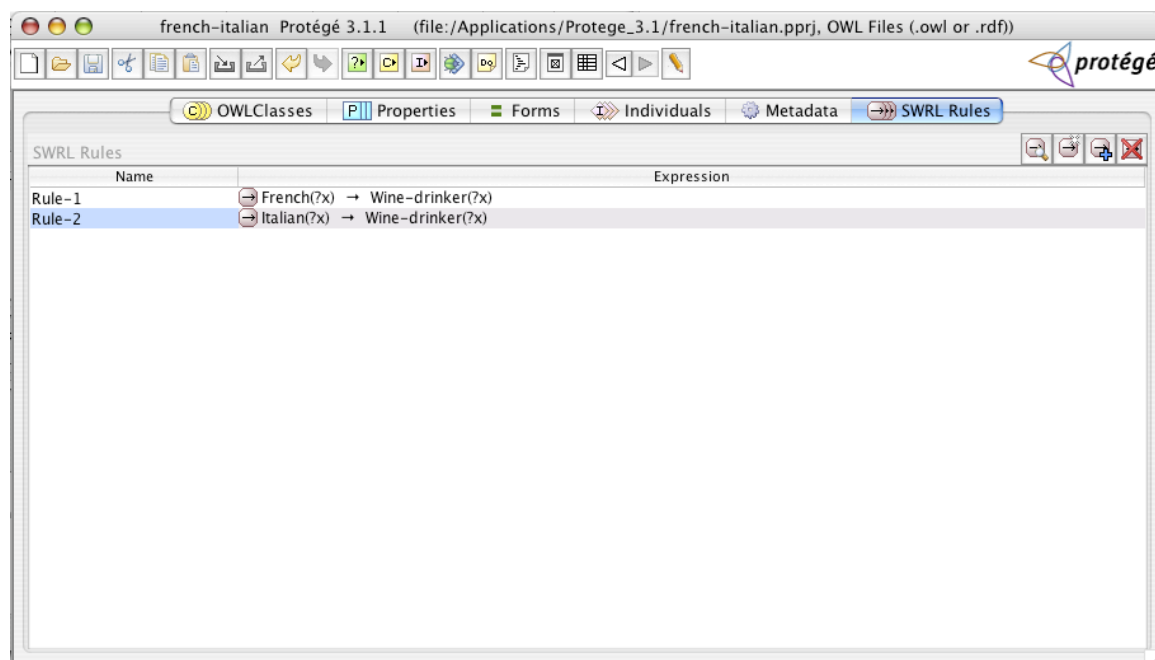


Figure 2.36: For *john* we have indefinite information. John is an instance of **French** or **Italian**.

In RacerPro 1.9, for a fixed variable binding to individual in an A-box the predicates in the body must be satisfied in all models. This has important consequences. Let us assume, we have class names **French**, **Italian**, **Wine-drinker**, **Not-wine-drinker**, and **French-or-italian**. For the latter two classes, necessary and sufficient conditions are specified (see Figures 2.34 and 2.35). The individuals *john* is an instance of **Not-wine-drinker** and **French-or-Italian**.

Figure 2.37: Rules for **French** and **Italian**.

If the rules in Figure 2.37 are specified, the ontology would be inconsistent with first-order semantics. There are two worlds to consider: Assume **john** is a French citizen, then he must be an instance of **Wine-drinker** due to Rule-1 (see Figure 2.37). But he is declared to be an instance of **Not-wine-drinker**. In the other possible world, **john** is an instance of **Italian**. Again, due to the rules, he must be a **Wine-drinker**, which results in a contradiction. In the rule semantics, one can neither prove that **john** is an instance of **Italian** nor can one prove that **john** is an instance of **French**. Thus, the rules are not applied and the ontology remains consistent.

RacerPro applies the rule semantics in version 1.9. This can be seen as an advantage or as a disadvantage. In order to achieve the same effect as in the first-order semantics, in this case a rule with precondition **Not-wine-drinker** could be added. Note also that “simple” rules such as those shown in Figure 2.37 can be represented as concept axioms in Protégé itself.

Chapter 3

RacerPro Knowledge Bases

In description logic systems a knowledge base is consisting of a T-box and an A-box. The conceptual knowledge is represented in the T-box and the knowledge about the instances of a domain is represented in the A-box. For more information about the description logic *SHIQ* supported by RacerPro see [9]. The extension of expressive description logics with concrete domains is discussed in [6].

3.1 Naming Conventions

Throughout this document we use the following abbreviations, possibly subscripted.

<i>C</i>	Concept term	<i>name</i>	Name of any sort
<i>CN</i>	Concept name	<i>S</i>	List of Assertions
<i>IN</i>	Individual name	<i>GNL</i>	List of group names
<i>ON</i>	Object name	<i>LCN</i>	List of concept names
<i>R</i>	Role term	<i>abox</i>	A-box object
<i>RN</i>	Role name	<i>tbox</i>	T-box object
<i>AN</i>	Attribute name	<i>n</i>	A natural number
<i>ABN</i>	A-box name	<i>real</i>	A real number
<i>TBN</i>	T-box name	<i>integer</i>	An integer number
<i>KBN</i>	knowledge base name	<i>string</i>	A string

The API is designed to the following conventions.¹ For most of the services offered by RacerPro, macro interfaces and function interfaces are provided. For macro forms, the T-box or A-box arguments are optional. If no T-box or A-box is specified, the value of (`current-tbox`) or (`current-abox`) is taken, respectively. However, for the functional counterpart of a macro the T-box or A-box argument is not optional. For functions which do not have macro counterparts the T-box or A-box argument may or may not be optional.

¹For RacerMaster or LRacer users: All names are Lisp symbols, the concepts are symbols or lists. Please note that for macros in contrast to functions the arguments should not be quoted.

$C \longrightarrow$	CN	
	<code>*top*</code>	
	<code>*bottom*</code>	
	<code>(not C)</code>	
	<code>(and $C_1 \dots C_n$)</code>	
	<code>(or $C_1 \dots C_n$)</code>	
	<code>(some $R C$)</code>	
	<code>(all $R C$)</code>	
	<code>(at-least $n R$)</code>	
	<code>(at-most $n R$)</code>	
	<code>(exactly $n R$)</code>	
	<code>(at-least $n R C$)</code>	
	<code>(at-most $n R C$)</code>	
	<code>(exactly $n R C$)</code>	
	<code>(a AN)</code>	
	<code>(an AN)</code>	
	<code>(no AN)</code>	
	CDC	
$R \longrightarrow$	RN	
	<code>(inv RN)</code>	

Figure 3.1: RacerPro concept and role terms.

Furthermore, if an argument *tbox* or *abox* is specified in this documentation, a name (a symbol) can be used as well.

Functions and macros are only distinguished in the Lisp version. Macros do not evaluate their arguments. If you use the RacerPro server, you can use functions just like macros. Arguments are never evaluated.

3.2 Concept Language

The content of RacerPro T-boxes includes the conceptual modeling of concepts and roles as well. The modeling is based on the signature, which consists of two disjoint sets: the set of concept names \mathcal{C} , also called the atomic concepts, and the set \mathcal{R} containing the role names².

Starting from the set \mathcal{C} complex concept terms can be build using several operators. An overview over all concept- and role-building operators is given in Figure 3.1.

²The signature does not have to be specified explicitly in RacerPro knowledge bases - the system can compute it from the all the used names in the knowledge base - but specifying a signature may help avoiding errors caused by typos!

$CDC \longrightarrow$	(min AN integer)	
	(max AN integer)	
	(equal AN integer)	
	(equal AN AN)	
	(divisible AN cardinal)	
	(not-divisible AN cardinal)	
	(> $aexpr$ $aexpr$)	
	(>= $aexpr$ $aexpr$)	
	(< $aexpr$ $aexpr$)	
	(<= $aexpr$ $aexpr$)	
	(<> $aexpr$ $aexpr$)	
	(= $aexpr$ $aexpr$)	
	(string= AN string)	
	(string<> AN string)	
	(string= AN AN)	
	(string<> AN AN)	
$string \longrightarrow$	" letter* "	
$aexpr \longrightarrow$	AN	AN must be of type real
	real	
	(+ $aexpr1$ $aexpr1^*$)	
	$aexpr1$	

Figure 3.2: RacerPro concrete domain concepts and attribute expressions.

Boolean terms build concepts by using the boolean operators.

	DL notation	RacerPro syntax
Negation	$\neg C$	(not C)
Conjunction	$C_1 \sqcap \dots \sqcap C_n$	(and $C_1 \dots C_n$)
Disjunction	$C_1 \sqcup \dots \sqcup C_n$	(or $C_1 \dots C_n$)

Qualified restrictions state that role fillers have to be of a certain concept. Value restrictions assure that the type of *all* role fillers is of the specified concept, while exist restrictions require that there be *a* filler of that role which is an instance of the specified concept.

	DL notation	RacerPro syntax
Exists restriction	$\exists R.C$	(some R C)
Value restriction	$\forall R.C$	(all R C)

Number restrictions can specify a lower bound, an upper bound or an exact number for the amount of role fillers each instance of this concept has for a certain role. Only roles that are not transitive and do not have any transitive subroles are allowed in number restrictions [9].

$aexpr1 \longrightarrow$	$aexpr2$	
	$aexpr3$	
$aexpr2 \longrightarrow$	$real$	
	AN	(AN of type real or complex)
	$(* real AN)$	(AN of type real)
$aexpr3 \longrightarrow$	$integer$	
	AN	(AN of type cardinal)
	$(* integer AN)$	(AN of type cardinal)

Figure 3.3: Specific expressions for predicates.

	DL notation	RacerPro syntax
At-most restriction	$\leq n R$	(at-most $n R$)
At-least restriction	$\geq n R$	(at-least $n R$)
Exactly restriction	$= n R$	(exactly $n R$)
Qualified at-most restriction	$\leq n R.C$	(at-most $n R C$)
Qualified at-least restriction	$\geq n R.C$	(at-least $n R C$)
Qualified exactly restriction	$= n R.C$	(exactly $n R C$)

Actually, the exactly restriction (**exactly** $n R$) is an abbreviation for the concept term (**and** (**at-least** $n R$) (**at-most** $n R$)) and (**exactly** $n R C$) is an abbreviation for the concept term (**and** (**at-least** $n R C$) (**at-most** $n R C$))

There are two concepts implicitly declared in every T-box: the concept “top” (\top) denotes the top-most concept in the hierarchy and the concept “bottom” (\perp) denotes the inconsistent concept, which is a subconcept to all other concepts. Note that \top (\perp) can also be expressed as $C \sqcup \neg C$ ($C \sqcap \neg C$). In RacerPro \top is denoted as ***top*** and \perp is denoted as ***bottom***³.

³For KRSS compatibility reasons RacerPro also supports the synonym concepts **top** and **bottom**.

Concrete domain concepts state concrete predicate restrictions for attribute fillers (see Figure 3.2). RacerPro currently supports three unary predicates for integer attributes (**min**, **max**, **equal**), six nary predicates for real attributes (**>**, **>=**, **<**, **<=**, **=**, **<>**), a unary existential predicate with two syntactical variants (**a** or **an**), and a special predicate restriction disallowing a concrete domain filler (**no**). The restrictions for attributes of type **real** have to be in the form of linear inequations (with order relations) where the attribute names play the role of variables. If an expression is built with the rule for *aexpr4* (see Figure 3.2), a so-called nonlinear constraint is specified. In this case, only equations and inequations (**=**, **<>**), but no order constraints (**>**, **>=**, **<**, **<=**) are allowed, and the attributes must be of type **complex**. If an expression is built with the rule for *aexpr5* (see Figure 3.2) a so-called cardinal linear constraint is specified, i.e., attributes are constrained to be a natural number (including zero). RacerPro also supports a concrete domain for representing equations about strings with predicates **string=** and **string<>**. The use of concepts with concrete domain expressions is illustrated with examples in Section 3.5. For the declaration of types for attributes, see Section 3.6.

	DL notation	RacerPro syntax
Concrete filler exists restriction	$\exists A.\top_{\mathcal{D}}$	(a <i>A</i>) or (an <i>A</i>)
No concrete filler restriction	$\forall A.\perp_{\mathcal{D}}$	(no <i>A</i>)
Integer predicate exists restriction with $z \in \mathbb{Z}$	$\exists A.\min_z$ $\exists A.\max_z$	(min <i>A</i> <i>z</i>) (max <i>A</i> <i>z</i>)
	$\exists A.=_z$	(equal <i>A</i> <i>z</i>)
Real predicate exists restriction with $P \in \{>, >=, <, <=, =\}$	$\exists A_1, \dots, A_n.P$	(<i>P</i> <i>aexpr</i> <i>aexpr</i>)

An all restriction of the form $\forall A_1, \dots, A_n.P$ is currently not directly supported. However, it can be expressed as a disjunction: $\forall A_1.\perp_{\mathcal{D}} \sqcup \dots \sqcup \forall A_n.\perp_{\mathcal{D}} \sqcup \exists A_1, \dots, A_n.P$.

3.3 Concept Axioms and T-boxes

RacerPro supports several kinds of concept axioms.

General concept inclusions (GCIs) state the subsumption relation between two concept terms.

DL notation: $C_1 \sqsubseteq C_2$

RacerPro syntax: (**implies** *C*₁ *C*₂)

Concept equations state the equivalence between two concept terms.

DL notation: $C_1 \doteq C_2$

RacerPro syntax: (**equivalent** *C*₁ *C*₂)

Concept disjointness axioms state pairwise disjointness between several concepts. Disjoint concepts do not have instances in common.

DL notation: $C_1 \sqsubseteq \neg(C_2 \sqcup C_3 \sqcup \dots \sqcup C_n)$

$C_2 \sqsubseteq \neg(C_3 \sqcup \dots \sqcup C_n)$

...

$$C_{n-1} \sqsubseteq \neg C_n$$

RacerPro syntax: `(disjoint C1 ... Cn)`

Actually, a concept equation $C_1 \doteq C_2$ can be expressed by the two GCIs: $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. The disjointness of the concepts $C_1 \dots C_n$ can also be expressed by GCIs.

There are also separate forms for concept axioms with just concept names on their left-hand sides. These concept axioms implement special kinds of GCIs and concept equations. But concept names are only a special kind of concept terms, so these forms are just syntactic sugar. They are added to the RacerPro system for historical reasons and for compatibility with KRSS. These concept axioms are:

Primitive concept axioms state the subsumption relation between a concept name and a concept term.

DL notation: $(CN \sqsubseteq C)$

RacerPro syntax: `(define-primitive-concept CN C)`

Concept definitions state the equality between a concept name and a concept term.

DL notation: $(CN \doteq C)$

RacerPro syntax: `(define-concept CN C)`

Concept axioms may be cyclic in RacerPro. There may also be forward references to concepts which will be “introduced” with `define-concept` or `define-primitive-concept` in subsequent axioms. The terminology of a RacerPro T-box may also contain several axioms for a single concept. So if a second axiom about the same concept is given, it is added and does not overwrite the first axiom.

3.4 Role Declarations

In contrast to concept axioms, role declarations are unique in RacerPro. There exists just one declaration per role name in a knowledge base. If a second declaration for a role is given, an error is signaled. If no signature is specified, undeclared roles are assumed to be neither a feature nor a transitive role and they do not have any superroles.

The set of all roles (\mathcal{R}) includes the set of features (\mathcal{F}) and the set of transitive roles (\mathcal{R}^+). The sets \mathcal{F} and \mathcal{R}^+ are disjoint. All roles in a T-box may also be arranged in a role hierarchy. The inverse of a role name RN can be either explicitly declared via the keyword `:inverse` (e.g. see the description of `define-primitive-role`) or referred to as `(inv RN)`.

Features (also called attributes) restrict a role to be a functional role, e.g. each individual can only have up to one filler for this role.

Transitive Roles are transitively closed roles. If two pairs of individuals IN_1 and IN_2 and IN_2 and IN_3 are related via a transitive role R , then IN_1 and IN_3 are also related via R .

Role Hierarchies define super- and subrole-relationships between roles. If R_1 is a superrole of R_2 , then for all pairs of individuals between which R_2 holds, R_1 must hold too.

In the current implementation the specified superrole relations may not be cyclic. If a role has a superrole, its properties are not in every case inherited by the subrole. The properties of a declared role induced by its superrole are shown in Figure 3.4. The table should be read as follows: For example if a role RN_1 is declared as a simple role and it has a feature RN_2 as a superrole, then RN_1 will be a feature itself.

		Superrole $RN_2 \in$		
		\mathcal{R}	\mathcal{R}^+	\mathcal{F}
Subrole RN_1	\mathcal{R}	\mathcal{R}	\mathcal{R}	\mathcal{F}
declared as	\mathcal{R}^+	\mathcal{R}^+	\mathcal{R}^+	-
element of:	\mathcal{F}	\mathcal{F}	\mathcal{F}	\mathcal{F}

Figure 3.4: Conflicting declared and inherited role properties.

The combination of a feature having a transitive superrole is not allowed and features cannot be transitive. Note that transitive roles and roles with transitive subroles may not be used in number restrictions.

RacerPro does not support role terms as specified in the KRSS. However, a role being the conjunction of other roles can as well be expressed by using the role hierarchy (cf. [5]). The KRSS-like declaration of the role (`define-primitive-role RN (and RN_1 RN_2)`) can be approximated in RacerPro by: (`define-primitive-role RN :parents (RN_1 RN_2)`).

KRSS	DL notation
<code>(define-primitive-role RN (domain C))</code>	$(\exists RN.\top) \sqsubseteq C$
<code>(define-primitive-role RN (range D))</code>	$\top \sqsubseteq (\forall RN.D)$
RacerPro Syntax	DL notation
<code>(define-primitive-role RN :domain C)</code>	$(\exists RN.\top) \sqsubseteq C$
<code>(define-primitive-role RN :range D)</code>	$\top \sqsubseteq (\forall RN.D)$

Figure 3.5: Domain and range restrictions expressed via GCIs.

RacerPro offers the declaration of domain and range restrictions for roles. These restrictions for primitive roles can be either expressed with GCIs, see the examples in Figure 3.5 (cf. [5]) or declared via the keywords `:domain` and `:range`.

3.5 Concrete Domains

RacerPro supports reasoning over natural numbers (\mathbb{N}), integers (\mathbb{Z}), reals (\mathbb{R}), complex numbers (\mathbb{C}), and strings. For different sets, different kinds of predicates are supported.

\mathbb{N}	linear inequations with order constraints and integer coefficients
\mathbb{Z}	interval constraints
\mathbb{R}	linear inequations with order constraints and rational coefficients
Strings	equality and inequality

For the users convenience, rational coefficients can be specified in floating point notation. They are automatically transformed into their rational equivalents (e.g., 0.75 is transformed into 3/4). In the following we will use the names on the left-hand side of the table to refer to the corresponding concrete domains.

Names for values from concrete domains are called *objects*. The set of all objects is referred to as \mathcal{O} . Individuals can be associated with objects via so-called *attributes names* (or attributes for short). Note that the set \mathcal{A} of all attributes must be disjoint to the set of roles (and the set of features). Attributes can be declared in the signature of a T-box (see below).

The following example is an extension of the family T-box introduced above. In the example, the concrete domains \mathbb{Z} and \mathbb{R} are used.

```
...
(signature
  :atomic-concepts (... teenager)
  :roles (... )
  :attributes ((integer age)))
...
(equivalent teenager (and human (min age 16)))
(equivalent old-teenager (and human (min age 18)))
...
```

Asking for the children of teenager reveals that **old-teenager** is a **teenager**. A further extensions demonstrates the usage of reals as concrete domain.

```
...
(signature
  :atomic-concepts (... teenager)
  :roles (... )
  :attributes ((integer age)
               (real temperature-celsius)
               (real temperature-fahrenheit)))
...
(equivalent teenager (and human (min age 16)))
(equivalent old-teenager (and human (min age 18)))
(equivalent human-with-fever (and human (>= temperature-celsius 38.5)))
(equivalent seriously-ill-human (and human (>= temperature-celsius 42.0)))
...
```

Obviously, RacerPro determines that the concept **seriously-ill-human** is subsumed by **human-with-fever**. For the reals, RacerPro supports linear equations and inequations.

Thus, we could add the following statement to the knowledge base in order to make sure the relations between the two attributes `temperature-fahrenheit` and `temperature-celsius` is properly represented.

```
(implies top (= temperature-fahrenheit
              (+ (* 1.8 temperature-celsius) 32)))
```

If a concept `seriously-ill-human-1` is defined as

```
(equivalent seriously-ill-human-1
             (and human (>= temperature-fahrenheit 107.6)))
```

RacerPro recognizes the subsumption relationship with `human-with-fever` and the synonym relationship with `seriously-ill-human`.

In an A-box, it is possible to set up constraints between individuals. This is illustrated with the following extended A-box.

```
...
(signature
 :atomic-concepts (... teenager)
 :roles (...)
 :attributes (...)
 :individuals (eve doris)
 :objects (temp-eve temp-doris))
...
(constrained eve temp-eve temperature-fahrenheit)
(constrained doris temp-doris temperature-celsius)
(constraints
 (= temp-eve 102.56)
 (= temp-doris 39.5))
```

For instance, this states that `eve` is related via the attribute `temperature-fahrenheit` to the object `temp-eve`. The initial constraint `(= temp-eve 102.56)` specifies that the object `temp-eve` is equal to 102.56.

Now, asking for the direct types of `eve` and `doris` reveals that both individuals are instances of `human-with-fever`. In the following A-box there is an inconsistency since the temperature of 102.56 Fahrenheit is identical with 39.5 Celsius.

```
(constrained eve temp-eve temperature-fahrenheit)
(constrained doris temp-doris temperature-celsius)
(constraints
 (= temp-eve 102.56)
 (= temp-doris 39.5)
 (> temp-eve temp-doris))
```

We present another example that might be important for many applications: dealing with dates. The following declarations can be processed with Racer. The predicates `divisible` and `not-divisible` are defined for natural numbers and are reduced to linear inequations internally.

```
(define-concrete-domain-attribute year :type cardinal)
(define-concrete-domain-attribute days-in-month :type cardinal)

(implies Month (and (>= days-in-month 28) (<= days-in-month 31)))

(equivalent month-inleapyear
  (and Month
    (divisible year 4)
    (or (not-divisible year 100)
        (divisible year 400))))

(equivalent February
  (and Month
    (<= days-in-month 29)
    (or (not month-inleapyear)
        (= days-in-month 29))
    (or month-inleapyear
        (= days-in-month 28)))))
```

Next, we assume some instances of February are declared.

```
(instance feb-2003 February)
(constrained feb-2003 year-1 year)
(constrained feb-2003 days-in-feb-2003 days-in-month)
(constraints (= year-1 2003))

(instance feb-2000 February)
(constrained feb-2000 year-2 year)
(constrained feb-2000 days-in-feb-2000 days-in-month)
(constraints (= year-2 2000))
```

Note that the number of days for both months is not given explicitly. Nevertheless, asking `(concept-instances month-inleapyear)` yields `(feb-2000)` whereas asking for `(concept-instances (not month-inleapyear))` returns `(feb-2003)`. In addition, one could check the number of days:

```
(constraint-entailed? (<> days-in-feb-2003 29))
(constraint-entailed? (= days-in-feb-2000 29))
```

In both cases, the answer is true.

3.6 Concrete Domain Attributes

Attributes are considered as “typed” since they can either have fillers of type `cardinal`, `integer`, `real`, `complex`, or `string`. The same attribute cannot be used in the same T-box such that both types are applicable, e.g., `(min has-age 18)` and `(>= has-age 18)` are not allowed. If the type of an attribute is not explicitly declared, its type is implicitly derived from its use in a T-box/A-box. An attribute and its type can be declared with the signature form (see above) or by using the KRSS-like form `define-concrete-domain-attribute`. If an attribute is declared to be of type `complex` it can be used in linear (in-)equations. However, if an attribute is declared to be of type `real` or `integer` it is an error to use this attribute in terms for nonlinear polynoms. In a similar way, currently, an attribute of type `integer` may not be used in a term for a linear polynoms, either. If the coefficients are integers, then `cardinal` (natural number, including 0) for the type of attributes may be used in a linear polynomial. Furthermore, attributes of type `string` may not be used on polynoms, and non-strings may not be used in constraints for strings.

3.7 Individual Assertions and A-boxes

An A-box contains assertions about individuals. The set of individual names (or individuals for brevity) \mathcal{I} is the signature of the A-box. The set of individuals must be disjoint to the set of concept names and the set of role names. There are four kinds of assertions:

Concept assertions with `instance` state that an individual IN is an instance of a specified concept C .

Role assertions with `related` state that an individual IN_1 is a role filler for a role R with respect to an individual IN_2 .

Attribute assertions with `constrained` state that an object ON is a filler for a role R with respect to an individual IN .

Constraints within `constraints` state relationships between objects of the concrete domain. The syntax for constraints is explained in Figure 3.2. Instead of attribute names, object names must be used.

In RacerPro the *unique name assumption* holds, this means that all individual names used in an A-box refer to distinct domain objects, therefore two names cannot refer to the same domain object. Note that the unique name assumption does not hold for object names.

In the RacerPro system each A-box refers to a T-box. The concept assertions in the A-box are interpreted with respect to the concept axioms given in the referenced T-box. The role assertions are also interpreted according to the role declarations stated in that T-box. When a new A-box is built, the T-box to be referenced must already exist. The same T-box may be referred to by several A-boxes. If no signature is used for the T-box, the assertions in the

A-box may use new names for roles⁴ or concepts⁵ which are not mentioned in the T-box.

3.8 Inference Modes

After the declaration of a T-box or an A-box, RacerPro can be instructed to answer queries. Processing the knowledge base in order to answer a query may take some time. The standard inference mode of RacerPro ensures the following behavior: Depending on the kind of query, RacerPro tries to be as smart as possible to locally minimize computation time (lazy inference mode). For instance, in order to answer a subsumption query w.r.t. a T-box it is not necessary to classify the T-box. However, once a T-box is classified, answering subsumption queries for atomic concepts is just a lookup. Furthermore, asking whether there exists an atomic concept in a T-box that is inconsistent (**tbox-coherent-p**) does not require the T-box to be classified, either. In the lazy mode of inference (the default), RacerPro avoids computations that are not required concerning the current query. In some situations, however, in order to globally minimize processing time it might be better to just classify a T-box before answering a query (eager inference mode).

A similar strategy is applied if the computation of the direct types of individuals is requested. RacerPro requires as precondition that the corresponding T-box has to be classified. If the lazy inference mode is enabled, only the individuals involved in a “direct types” query are realized.

We recommend that T-boxes and A-boxes should be kept in separate files. If an A-box is revised (by reloading or reevaluating a file), there is no need to recompute anything for the T-box. However, if the T-box is placed in the same file, reevaluating a file presumably causes the T-box to be reinitialized and the axioms to be declared again. Thus, in order to answer an A-box query, recomputations concerning the T-box might be necessary. So, if different A-boxes are to be tested, they should probably be located separately from the associated T-boxes in order to save processing time.

During the development phase of a T-box it might be advantageous to call inference services directly. For instance, during the development phase of a T-box it might be useful to check which atomic concepts in the T-box are inconsistent by calling **check-tbox-coherence**. This service is usually much faster than calling **classify-tbox**. However, if an application problem can be solved, for example, by checking whether a certain A-box is consistent or not (see the function **abox-consistent-p**), it is not necessary to call either **check-tbox-coherence** or **classify-tbox**. For all queries, RacerPro ensures that the knowledge bases are in the appropriate states. This behavior usually guarantees minimum runtimes for answering queries.

⁴These roles are treated as roles that are neither a feature, nor transitive and do not have any superroles. New items are added to the T-box. Note that this might lead to surprising query results, e.g. the set of subconcepts for \top contains concepts not mentioned in the T-box in any concept axiom. Therefore we recommend to use a **signature** declaration (see below).

⁵These concepts are assumed to be atomic concepts.

3.9 Retraction and Incremental Additions

RacerPro offers constructs for retracting T-box axioms (see the function `forget-statement`). However, complete reclassification may be necessary in order to answer queries. Retracting axioms is mainly useful if the RacerPro server is used. With retracting there is no need to delete and retransfer a knowledge base (T-box).

RacerPro also offers constructs for retracting A-box assertions (see `forget`, `forget-concept-assertion`, `forget-role-assertion`, and friends). If a query has been answered and some assertions are retracted, then RacerPro might be forced to compute the index structures for the A-box again (realization), i.e. after retractions, some queries might take some time to answer. Note that many queries are answered without index structures at all (see also Section 5.1).

RacerPro also supports incremental additions to A-boxes, i.e. assertions can be added even after queries have been answered. However, the internal data structures used for answering queries are recomputed from scratch. This might take some time. If an A-box is used for hypothesis generation, e.g. for testing whether the assertion $i : C$ can be added without causing an inconsistency, we recommend using the instance checking inference service. If `(individual-instance? i (not C))` returns `t`, $i : C$ cannot be added to the A-box. Now, let us assume, we can add $i : C$ and afterwards want to test whether $i : D$ can be added without causing an inconsistency. In this case it might be faster not to add $i : C$ directly but to check whether `(individual-instance? i (and C (not D)))` returns `t`. The reason is that, in this case, the index structures for the A-box are not recomputed.

Chapter 4

Description Logic Modeling with RacerPro

There are several excellent books and introductory papers on description logics (e.g., [1, 2]). In this Chapter we discuss additional issues that are important when RacerPro is used in practical applications.

4.1 Representing Data with Description Logics (?)

Almost nothing is required to use a description logic inference system to store data. In particular, there is no need to specify any kind of memory management information as in databases or even in object-oriented programming systems. In order to make this clear, a very simple example is given as follows:

```
(in-knowledge-base test)
(instance i a)
(instance i b)
(related i j r)
```

Given these declarations for “data”, query functions can be used for data retrieval.

```
(concept-instances (and a (some r b)))
```

returns `i`. You could also use the nRQL query language (see Chapter 6).

```
(retrieve (?x) (?x (and a (some a b))))
```

If you would like to know which fillers actually got names in the A-box, use the following query:

```
(retrieve (?x ?y) (and (?x a) (?x ?y r) (?y b)))
```

Note, however, that the latter query would not return results for the following A-box whereas the former query would return `i` (the query language uses the active domain semantics for variable bindings).

```
(in-knowledge-base test)
(instance i (and a (some r b)))
```

For a detailed explanation see Chapter 6. In any case, description logics (and ontology languages such as OWL) are important if concept names have definitions in the T-box or if the A-box contains indefinite descriptions (such as `(instance john (or french italian))`). Although it is possible to represent (an object-based view of) a database as an A-box, currently, description logic systems do not provide for transactions and persistency of data, and in order to ensure decidability in the general case, the query language is in some sense less expressive than, for instance, relational database query languages such as SQL. So, mass data (representing definite information) is better stored in databases right now. Description logic (and semantic web) technology comes into play when indefinite information (disjunctive information) is to be treated as well. See also the comment about the open-world and closed-world assumptions below. Databases employ the closed-world assumption. What is not explicitly stated in the database is assumed to be false. This is not the case for description logics.

4.2 Nominals or Concrete Domains?

The language OWL provides for means to address individuals in concepts. As a example, one could represent the concept `human` with the following axioms:

```
(define-primitive-role ancestor-of :transitive t :inverse has-descendant)
(define-primitive-role has-descendant)
(equivalent human (some ancestor-of (one-of adam)))
(instance john human)
(related kain john has-descendant)
```

The concept-forming operator `one-of` takes an individual and construct a concept whose extension contains just the semantic object to which the individual `adam` is mapped. Thus the extension of `(one-of adam)` is a singleton set. Individuals in concepts are also known a *nominals*.

Asking for the ancestors of `john` yields `kain` and `adam` although the latter is not explicitly stated in the A-box.

Currently, RacerPro does not support nominals in full generality. Only axioms of the form

```
(equivalent (one-of i) c)
(equivalent (one-of i) (some r (one-of j)))
```

are supported. These axioms are very important for practical purposes, and they directly correspond to the following A-box assertions

```
(instance i c)
(related i j r)
```

In many practically important cases, nominals are not required in concepts terms. The same effect can be achieved using concrete domain values. For instance, one might think of using nominals for representing colors of (simple) traffic lights: (one-of red green). The following example demonstrates the use of A-boxes and the string concrete domain to provide for a formal model of a crossing with (simple) traffic lights.

```
(in-knowledge-base traffic-lights)

(define-concrete-domain-attribute color :type string)

(define-concept colorful-object
  (or (string= color "red")
      (string= color "green")))

(define-concept traffic-light
  (and (a color) colorful-object))

(instance traffic-light-1 traffic-light)
(instance traffic-light-2 traffic-light)
(instance traffic-light-3 traffic-light)
(instance traffic-light-4 traffic-light)

(constrained traffic-light-1 ?color-traffic-light-1 color)
(constrained traffic-light-2 ?color-traffic-light-2 color)
(constrained traffic-light-3 ?color-traffic-light-3 color)
(constrained traffic-light-4 ?color-traffic-light-4 color)

(constraints (string= ?color-traffic-light-1 ?color-traffic-light-3))
(constraints (string= ?color-traffic-light-2 ?color-traffic-light-4))
(constraints (string<> ?color-traffic-light-1 ?color-traffic-light-2))

(constraints (string<> ?color-traffic-light-2 "red"))

(constraint-entailed? (string= ?color-traffic-light-2 "green"))
(constraint-entailed? (string= ?color-traffic-light-4 "green"))
(constraint-entailed? (string= ?color-traffic-light-1 "red"))
(constraint-entailed? (string= ?color-traffic-light-3 "red"))
```

The four queries at the end all return `t` (for true) although only indefinite information is explicitly stated (the color of `traffic-light-2` is not "red"). Thus, only "green" remains, and due to the other constraints, the colors of the other traffic lights are determined. It is obvious that instead of string values one could have used nominals. However, optimized reasoning algorithms for nominals will only be part of a future version of RacerPro.

Using nominals might be even tricky. Consider the following knowledge base:

```
(equivalent shabby-car (all has-color shabby-color))
(equivalent reddish-object (some has-color (one-of orange, red)))
(instance car-1 (and reddish-object (not (some has-color (one-of orange)))))
(instance car-2 (and reddish-object (not (some has-color (one-of orange)))))
```

Now assume, some information of the color of `car-1` is available, `car-1` is an old car.

```
(instance car-1 (all has-color shabby-color))
```

It is obvious that due to the use of nominals, `car-2` is a shabby car as well. This is probably unintended and is most likely a modeling error.

4.3 Open-World Assumption

As other description logic systems, RacerPro employs the Open World Assumption (OWA) for reasoning. This means that what cannot be proven to be true is not believed to be false. Given the T-box and A-box of the family example (see the previous chapters), a standard pitfall would be to think that RacerPro is wrong considering its answer to the following query:

```
(individual-instance? alice (at-most 2 has-child))
```

RacerPro answers NIL. However, NIL does not mean NO but just “cannot be proven w.r.t. the information given to RacerPro”. Absence of information w.r.t. a third child is not interpreted as “there is none” (this would be the Closed-World Assumption, CWA). It might be the case that there will be an assertion `(related alice william has-child)` added to the A-box later on. Thus, the answer NIL is correct but has to be interpreted in the sense of “cannot be proven”. Note that it is possible to add the assertion

```
(instance alice (at-most 2 has-child))
```

to the A-box. Given this, the A-box will become inconsistent if another individual (e.g., `william`) is declared to be a child of `alice`. Many users asked for a switch such that RacerPro automatically closes roles. However, this problem is ill-defined. A small example should suffice to illustrate why closing a role (or even a KB) is a tricky problem. Assume the following axioms:

```
(disjoint a b c)
(instance i (and (some r a) (some r b) (some r c) (some r d)))
(related i j r)
```

Now assume the task is to closed the role `r` for the individual `i`. Just determining the number of fillers of `r` w.r.t. `i` and adding a corresponding assertion `(at-most 1 r)` to the A-box is a bad idea because the A-box gets inconsistent. Due to the T-box, the minimum number of fillers is 3. But, should we add `(at-most 1 r)` or `(at-most 1 r (and a b c))`? The first one might be too strong (because of `i` being an instance of `(some r d)`). What about `d`? Would it be a good idea to also add `(at-most 1 r d)`? If yes, then the question arises how to determine the qualifier concepts used in qualified number restrictions.

4.4 Closed-World Assumption

Due to the non-monotonic nature of the closed-world assumption and the ambiguities about what closing should actually mean, in description logic inference systems usually there is no support for the closed-world assumption. However, with the publish and subscribe interface of RacerPro, users can achieve a local closed-world (LCW) assumption (see Section 5.3). The nRQL query language allows you to query A-boxes using negation as failure semantics (see Chapter 6).

4.5 Unique Name Assumption

In addition to the Open World Assumption, it is possible to instruct RacerPro to employ the Unique Name Assumption (UNA). This means that all individuals used in an A-box are assumed to be mapped to different elements of the universe, i.e. two individuals cannot refer to the same domain element. Hence, adding `(instance alice (at-most 1 has-child))` does not identify `betty` and `charles` but makes the A-box inconsistent. Due to our experience with users, we would like to emphasize that most users take UNA for granted but are astonished to learn that OWA is assumed (rather than CWA). However, in order to be compliant with the semantics of OWL RacerPro does not apply the unique name assumption by default. If you want this, use the following statement before asking any queries.

```
(set-unique-name-assumption t)
```

You might want to put this directive into a file `init.racer` and start RacerPro with the following option.

```
$ RacerPro -- -init init.racer
```

4.6 Differences in Expressivity of Query and Concept Language

It should be emphasized that the query language of RacerPro (nRQL, see Chapter 6) is different from the concept language. Feature chains and (pseudo) nominals are supported in queries but not in the concept language. In addition, specific concrete domain predicates may be used in query expressions (e.g., `substring`) that cannot be supported in the concept language.

4.7 OWL Interface

RacerPro can read RDF, RDFS, and OWL files, see the function `owl-read-file` and friends described below). Information in an RDF file is represented using an A-box in such a way that usually triples are represented as **related** statements, i.e., the subject of a triple is represented as an individual, the property as a role, and the object is also represented as an individual. The property `rdf:type` is treated in a special way. Triples with property `rdf:type` are represented as concept assertions. RacerPro does not represent meta-level knowledge in the theory because this might result in paradoxes (which are reported elsewhere).

The triples in RDFS files are processed in a special way. They are represented as T-box axioms. If the property is `rdf:type`, the object must be `rdfs:Class` or `rdfs:Property`. These statements are interpreted as declarations for concept and role names, respectively. Three types of axioms are supported with the following properties: `rdfs:subClassOf`, `rdfs:range`, and `rdfs:domain`. Other triples are ignored.

OWL files are processed in a similar way. The semantics of OWL is described elsewhere (see <http://www.w3.org/TR/owl-ref/>). There are a few restrictions in the RacerPro implementation. The UNA cannot be switched off and number restrictions for attributes (datatype properties) are not supported. Only basic datatypes of XML-Schema are supported (i.e., RacerPro cannot read files with datatype declarations right now).

Usually, the default namespace for concept and role name is defined by the pathname of the OWL file. If the OWL file contains a specification for the default namespace (i.e., a specification `xmlns="..."`) this URL is taken as a prefix for concept and role names.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns="http://www.mycompany.com/project#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
  ...
</rdf:RDF>
```

By default, RacerPro prepends the URL of the default namespace to all OWL names starting with the `#` sign. If you would like to instruct RacerPro to return abbreviated names (i.e., to remove the prefix again in output it produces), start the RacerPro server with the option `-n`.

Individual names (nominals) in class declarations introduced with `owl:oneOf` are treated as disjoint (atomic) concepts. This is similar to the behavior of other OWL inference engines. Currently, RacerPro can provide only an approximation for true nominals. Note that reasoning is sound but still incomplete if `owl:oneOf` is used. In RacerPro, individuals used in class declarations are also represented in the A-box part of the knowledge base. They are instances of a concept with the same name. An example is appropriate to illustrate the

idea. Although the KRSS syntax implemented by RacerPro does not include `one-of` as a concept-building operator we use it here for demonstration purposes.

```
(in-knowledge-base test)
(implies c (some r (one-of j)))
(instance i c)
```

Dealing with individuals is done by an approximation such that reasoning is sound but must remain incomplete. The following examples demonstrate the effects of the approximation.

Given this knowledge base, asking for the role fillers of `r` w.r.t. `i` returns `nil`. Note that OWL, names must be enclosed with bars (`|`).

```
? (individual-fillers |file:C:\\Ralf\\Ind-Examples\\ex1.owl#i|
    |file:C:\\Ralf\\Ind-Examples\\ex1.owl#R|)
NIL
```

Asking for the instances of `j` returns `j`.

```
? (concept-instances |file:C:\\Ralf\\Ind-Examples\\ex1.owl#j|)
(|file:C:\\Ralf\\Ind-Examples\\ex1.owl#j|)
```

The following knowledge base (for the OWL version see file `ex2.owl` in the examples folder) is inconsistent:

```
(in-knowledge-base test)
(implies c (all r (one-of j)))
(instance i c)
(related i k r)
```

Note again that, in general, reasoning is incomplete if individuals are used in concept terms. The following query is given w.r.t. the above-mentioned knowledge base given in the OWL file `ex2.owl` in the examples folder.

```
? (concept-subsumes? (at-most 1 |file:C:\\Ralf\\Ind-Examples\\ex1.owl#R|)
    |file:C:\\Ralf\\Ind-Examples\\ex1.owl#c|)
NIL
```

If dealing with nominals were no approximation, i.e., if reasoning were complete, then RacerPro would be able to prove a subsumption relationship because `(all r (one-of j))` implies `(at-most 1 r)`.

RacerPro can download imported ontology documents. Use the command `(owl-read-file <filename>)` to read an OWL file or use `(owl-read-document <url>)` to read an OWL resource given a URL.

You can manage multiple knowledge bases with RacerPro, and you can load multiple ontologies into a single knowledge base. For instance, try

```
(owl-read-document "http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf"
                  :kb-name dinner)
(owl-read-document "http://www.co-ode.org/ontologies/pizza/2005/05/16/pizza.owl"
                  :kb-name dinner :init nil)
```

for a delicious dinner.¹ Imported ontologies are automatically loaded from the corresponding web server. Make sure you are connected to the Internet or use the `mirror` functionality if you are offline.

```
(mirror <url-spec1> <another-url-or-local-filename>)
```

Examples:

```
(mirror "http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf"
        "/home/users/rm/wine.rdf")
(mirror "http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf"
        "http://localhost:8081/examples/wine.rdf")
```

Mirror statements might be placed in a RacerPro init file (see the previous subsection). In RacerPorter you can abbreviate the display of OWL names with the check box "Simplify Names".

There is a known limitation: The current version of the RDF reader of RacerPro ignores nodes with `parseType="Resource"` or `parseType="Literal"`. This will be fixed in the next version.

¹In LRacer or RacerMaster you have to quote the symbol dinner.

Chapter 5

Knowledge Base Management

5.1 Configuring Optimization Strategies

The standard configuration of RacerPro ensures that only those computations are performed that are required for answering queries. For instance, in order to answer a query for the parents of a concept, the T-box must be classified. However, for answering an instance retrieval query this is not necessary and, therefore, RacerPro does not classify the T-box in the standard inference mode. Nevertheless, if multiple instance retrieval queries are to be answered by RacerPro, it might be useful to have the T-box classified in order to be able to compute an index for query answering. Considering a single query RacerPro cannot determine whether computing an index is worth the required computational resources. Therefore, RacerPro can be instructed about answering strategies for subsequent queries.

- In order to ensure a T-box is classified, you can use the statement `(classify-tbox &optional tbox-name)`.
- To compute an index for fast instance retrieval for an A-box you can call `(compute-index-for-instance-retrieval &optional abox-name)`. Computing an index is usually a costly process, so this should be done offline. After having computed an index for an A-box one might use the persistency services of RacerPro to dump the internal A-box data structures for later reuse. The index computation process is also known as A-box realization, and you can also call the function `(realize-abox &optional abox-name)` to achieve the same effect.
- The function `(prepare-abox)` can be used to compute index structures for an A-box. You can call this function offline to save computational resources at query answering time. In addition, you can call `(prepare-racer-engine)` to compute index structures for query answering. Again, this function is to be used offline to save computational resource for answering the first query.
- If multiple queries are to be answered and each query will probably be more specific than previous ones, use the directive `(enable-subsumption-based-query-processing)`. The T-box is then classified once the first query is answered.

- If you use nRQL it is possible to instruct RacerPro to use less costly algorithms for query answering (see also Chapter 6). This can be done by calling `(set-nrql-mode 1)`. In this case, query answering is complete only for hierarchies (T-boxes with very simple axioms). If for every exists restrictions declared in the T-box there exists an explicit filler declared in the A-box, you can instruct RacerPro to speedup query answering even more by calling `(enable-optimized-query-answering)`. Note that it is necessary that you call `(enable-optimized-query-answering)` before you read the OWL files comprising the knowledge base.
- The function `(abox-consistent?)` can be explicitly called. Before the first instance retrieval query is answered, the A-box in question is checked for consistency. You might want to call this function offline to save computational resource when the first query is answered. The function `(abox-consistent?)` is not to be called if `(set-nrql-mode 1)` is called (see above).

The following sequence of statements provides for fastest execution times. Reasoning is only complete for $\mathcal{AL}\mathcal{E}$ with simple GCIs and A-boxes for which there exists a filler for every exists restriction.

```
;; offline phase
(full-reset)
(set-nrql-mode 1)
(enable-optimized-query-processing)
(set-unique-name-assumption t)
(time (load-data))
(time (prepare-abox))
(time (prepare-racer-engine))
;; online phase
(retrieve ...)
```

Nevertheless, it might be useful to use this mode for information retrieval problems (see also Chapter 6).

5.2 The RacerPro Persistency Services

If you load some knowledge bases into RacerPro and ask some queries, RacerPro builds internal data structure that enables the system to provide for faster response times. However, generating these internal data structures takes some time. So, if the RacerPro server is shut down, all this work is usually lost, and data structures have to be rebuilt when the server is restarted again. In order to save time at server startup, RacerPro provides a facility to “dump” the server state into a file and restore the state from the file at restart time. The corresponding functions form the persistency services of a RacerPro server. The persistency services can also be used to “prepare” a knowledge base at a specific server and use it repeatedly at multiple clients (see also the documentation about the RacerPro Proxy). For instance, you can classify a T-box or realize an A-box and dump the resulting data

structures into a file. The file(s) can be reloaded and multiple servers can restart with much less computational resources (time and space). Starting from a dump file is usually about ten times faster than load the corresponding text files and classifying the T-box (or realizing the A-box) again.

Since future versions of RacerPro might be supported by different internal data structures, it might be the case that old dump files cannot be loaded with future RacerPro versions. In this case an appropriate error message will be shown. However, you will have to create a new dump file again.

If you have a license for RacerMaster, dumping an image is possible with the underlying Common Lisp technology and is much, much faster.

5.3 The Publish-Subscribe Mechanism

Instance retrieval (see the function `concept-instances`) is one of the main inference services for A-boxes. However, using the standard mechanism there is no “efficient” way to declare so-called hidden or auxiliary individuals which are not returned as elements of the result set of instance retrieval queries.¹ Furthermore, if some assertions are added to an A-box, a previous instance retrieval query might have an extended result set. In this case some applications require that this might be indicated by a certain “event”. For instance, in a document retrieval scenario an application submitting an instance retrieval query for searching documents might also state that “future matches” should be indicated.

In order to support these features, RacerPro provides the publish-subscribe facility. The idea of the publish-subscribe system is to let users “subscribe” an instance retrieval query under a certain name (the subscription name). A subscribed query is answered as usual, i.e. it is treated as an instance retrieval query. The elements in the result set are by definition only those individuals (of the A-box in question) that have been “published” previously. If information about a new individuals is added to an A-box and these individuals are published, the set of subscription queries is examined. If there are new elements in the result set of previous queries, the publish function returns pairs of corresponding subscription and individual names.

5.3.1 An Application Example

The idea is illustrated in the following example taken from a document retrieval scenario. In some of the examples presented below, the result returned by RacerPro is indicated and discussed. If the result of a statement is not discussed, then it is irrelevant for understanding the main ideas of the publish-subscribe mechanism. First, a T-box `document-ontology` is declared.

```
(in-tbox document-ontology)
```

¹Certainly, hidden individuals can be marked as such with special concept names, and in queries they might explicitly be excluded by conjoining the negation of the marker concept automatically to the query concept. However, from an implementation point of view, this can be provided much more efficiently if the mechanism is built into the retrieval machinery of RacerPro.

```

(define-concrete-domain-attribute isbn)
(define-concrete-domain-attribute number-of-copies-sold)
(implies book document)
(implies article document)
(implies computer-science-document document)
(implies computer-science-book (and book computer-science-document))
(implies compiler-construction-book computer-science-book)
(implies (and (min number-of-copies-sold 3000) computer-science-document)
          computer-science-best-seller)

```

In order to manage assertions about specific documents, an A-box `current-documents` is defined with the following statements. The A-box `current-documents` is the “current A-box” to which subsequent statements and queries refer. The set of subscriptions (w.r.t. the current A-box) is initialized, i.e., with `init-subscription` you can instruct the server to delete all previous subscriptions.

```

(in-abox current-documents document-ontology)
(init-subscriptions)

```

With the following set of statements five document individuals are declared and published, i.e. the documents are potential results of subscription-based instance retrieval queries.

```

(state
  (instance document-1 article)
  (publish document-1)

  (instance document-2 book)
  (constrained document-2 isbn-2 isbn)
  (constraints (equal isbn-2 2234567))
  (publish document-2)

  (instance document-3 book)
  (constrained document-3 isbn-3 isbn)
  (constraints (equal isbn-3 3234567))
  (publish document-3)

  (instance document-4 book)
  (constrained document-4 isbn-4 isbn)
  (constraints (equal isbn-4 4234567))
  (publish document-4)

  (instance document-5 computer-science-book)
  (constrained document-5 isbn-5 isbn)
  (constraints (equal isbn-5 5234567))
  (publish document-5))

```

Now, we assume that a “client” subscribes to a certain instance retrieval query.

```
(state
  (subscribe client-1 book))
```

The answer returned by RacerPro is the following

```
((CLIENT-1 DOCUMENT-2)
 (CLIENT-1 DOCUMENT-3)
 (CLIENT-1 DOCUMENT-4)
 (CLIENT-1 DOCUMENT-5))
```

RacerPro returns a list of pairs each of which consists of a subscriber name and an individual name. In this case four documents are found to be instances of the query concept subscribed und the name `client-1`.

An application receiving this message from RacerPro as a return result can then decide how to inform the client appropriately. In future releases of RacerPro, subscriptions can be extended with information about how the retrieval events are to be signaled to the client. This will be done with a proxy which is currently under development.

The example is continued with the following statements and two new subscriptions.

```
(state
  (instance document-6 computer-science-document)
  (constrained document-6 isbn-6 isbn)
  (constraints (equal isbn-6 6234567))
  (publish document-6))

(state
  (subscribe client-2 computer-science-document)
  (subscribe client-3 computer-science-best-seller))
```

The last statement returns two additional pairs indicating the retrieval results for the instance retrieval query subscription of `client-2`.

```
((CLIENT-2 DOCUMENT-5)
 (CLIENT-2 DOCUMENT-6))
```

Next, information about another document is declared. The new document is published.

```
(state
  (instance document-7 computer-science-document)
  (constrained document-7 isbn-7 isbn)
  (constraints (equal isbn-7 7234567))
  (constrained document-7 number-of-copies-sold-7 number-of-copies-sold)
  (constraints (equal number-of-copies-sold-7 4000))
  (publish document-7))
```

The result of the last statement is:

```
((CLIENT-2 DOCUMENT-7)
 (CLIENT-3 DOCUMENT-7))
```

The new document `document-7` is in the result set of the query subscribed by `client-2` and `client-3`. Note that document can be considered as structured objects, not just names. This is demonstrated with the following statement whose result is displayed just below.

```
(describe-individual 'document-7)

(DOCUMENT-7
 :ASSERTIONS ((DOCUMENT-7 COMPUTER-SCIENCE-DOCUMENT))
 :ROLE-FILLERS NIL
 :TOLD-ATTRIBUTE-FILLERS ((ISBN 7234567)
                           (NUMBER-OF-COPIES-SOLD 4000))
 :DIRECT-TYPES ((COMPUTER-SCIENCE-BEST-SELLER)
                 (COMPUTER-SCIENCE-DOCUMENT)))
```

Thus, RacerPro has determined that the individual `document-7` is also an instance of the concept `computer-science-best-seller`. This is due to the value of the attribute `number-of-copies-sold` and the given sufficient conditions for the concept `computer-science-best-seller` in the T-box `document-ontology`.

Now, we have information about seven documents declared in the A-box `current-document`.

```
(all-individuals)

(DOCUMENT-1 DOCUMENT-2 DOCUMENT-3 DOCUMENT-4 DOCUMENT-5 DOCUMENT-6 DOCUMENT-7)
```

In order to delete a document from the A-box, it is possible to use RacerPro's forget facility. The instance assertion can be removed from the A-box with the following statement.

```
(forget () (instance document-3 book))
```

Now, asking for all individuals reveal that there are only six individuals left.

```
(all-individuals)

(DOCUMENT-1 DOCUMENT-2 DOCUMENT-4 DOCUMENT-5 DOCUMENT-6 DOCUMENT-7)
```

With the next subscription a fourth client is introduced. The query is to retrieve the instances of `book`. RacerPro's answer is given below.

```
(subscribe client-4 book)
```

```
((CLIENT-4 DOCUMENT-2) (CLIENT-4 DOCUMENT-4) (CLIENT-4 DOCUMENT-5))
```

The query of `client-4` is answered with three documents. Next, we discuss an example demonstrating that sometimes subscriptions do not lead to an immediate answer w.r.t. the current A-box.

```
(subscribe client-2 computer-science-best-seller)
```

The result is `()`. Although `document-7` is an instance of `computer-science-best-seller`, this individual has already been indicated as a result of a previously subscribed query. In order to continue our example we introduce two additional documents one of which is a `computer-science-best-seller`.

```
(state
  (instance document-8 computer-science-best-seller)
  (constrained document-8 isbn-8 isbn)
  (constraints (equal isbn-8 8234567))

  (instance document-9 book)
  (constrained document-9 isbn-9 isbn)
  (constraints (equal isbn-9 9234567)))
```

The publish-subscribe mechanism requires that these documents are published.

```
(state
  (publish document-8)
  (publish document-9))
```

The RacerPro system handles all publish statements within a `state` as a single `publish` statement and answers the following as a single list of subscription-individual pairs.

```
((CLIENT-1 DOCUMENT-9)
 (CLIENT-2 DOCUMENT-8)
 (CLIENT-3 DOCUMENT-8)
 (CLIENT-4 DOCUMENT-9))
```

Now `client-2` also get information about instances of `computer-science-best-seller`. Note that `document-8` is an instance of `computer-science-best-seller` by definition although the actual number of sold copies is not known to RacerPro.

```
(describe-individual 'document-8)

(DOCUMENT-8
 :ASSERTIONS ((DOCUMENT-8 COMPUTER-SCIENCE-BEST-SELLER))
 :ROLE-FILLERS NIL
 :TOLD-ATTRIBUTE-FILLERS ((ISBN 8234567))
 :DIRECT-TYPES ((COMPUTER-SCIENCE-BEST-SELLER)))
```

The following subscription queries indicate that the query concept must not necessarily be a concept name but can be a concept term.

```
(state
  (subscribe client-4 (equal isbn 7234567)))
```

RacerPro returns the following information:

```
((CLIENT-4 DOCUMENT-7))
```

Notice again that subscriptions might be considered when new information is added to the A-box.

```
(state
  (subscribe client-5 (equal isbn 10234567)))
```

The latter statement returns NIL. However, the subscription is considered if, at some time-point later on, a document with the corresponding ISBN number is introduced (and published).

```
(state
  (instance document-10 document)
  (constrained document-10 isbn-10 isbn)
  (constraints (equal isbn-10 10234567))
  (publish document-10))
```

```
((CLIENT-5 DOCUMENT-10))
```

This concludes the examples for the publish-subscribe facility offered by the RacerPro system. The publish-subscribe mechanism provided with the current implementation is just a first step. This facility will be extended significantly. Future versions will include optimization techniques in order to speedup answering subscription based instance retrieval queries such that reasonably large set of documents can be handled. Furthermore, it will be possible to define how applications are to be informed about “matches” to previous subscriptions (i.e. event handlers can be introduced).

5.3.2 Using JRacer for Publish and Subscribe

The following code fragment demonstrates how to interact with a RacerPro server from a Java application. The aim of the example is to demonstrate the relative ease of use that such an API provides. In our scenario, we assume that the agent instructs the RacerPro system to direct the channel to computer "rm.sts.tu-harburg.de" at port 8080. Before the subscription is sent to a RacerPro server, the agent should make sure that at "rm.sts.tu-harburg.de", the assumed agent base station, a so-called listener process is started at port 8080. This can be easily accomplished by starting the following program on rm.sts.tu-harburg.de.


```

public class Listener {
    public static void main(String[] argv) {
        try {
            ServerSocket server = new ServerSocket(8080);
            while (true) {
                Socket client = server.accept();
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(client.getInputStream()));
                String result = in.readLine();
                in.close();
            }
        } catch (IOException e) {
            ...
        }
    }
}

```

If a message comes in over the input stream, the variable **result** is bound accordingly. Then, the message can be processed as suitable to the application. We do not discuss details here. The subscription to the channel, i.e., the registration of the query, can also be easily done using the JRacer interface as indicated with the following code fragment (we assume RacerPro runs at node "racer.racer-systems.com" on port 8088).

```

public class Subscription {
    public static void main(String[] argv) {
        RacerServer racer1 = new RacerServer("www.racer-systems.com",8088);
        String res;
        try {
            racer1.openConnection();
            res = racer1.send("(subscribe q_1 Book (:notification-method tcp \"rm.sts.tu-harburg.de\" 8088)");
            racer1.closeConnection();
            System.out.println(res);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

The connection to the RacerPro server is represented with a client object (of class **RacerSocketClient**). The client object is used to send messages to the associated RacerPro server (using the message **send**). Control flow stops until RacerPro acknowledges the subscription.

5.3.3 Realizing Local Closed-World Assumptions

Feedback from many users of the RacerPro system indicates that, for instance, instance retrieval queries could profit from possibilities to “close” a knowledge base in one way or

another. Due to the non-monotonic nature of the closed-world assumption and the ambiguities about what closing should actually mean, in description logic inference systems usually there is no support for the closed-world assumption. However, with the publish and subscribe interface of RacerPro, users can achieve a similar effect. Consider, for instance, a query for a book which does not have an author. Because of the open-world assumption, subscribing to a channel for (`and Book (at-most 0 has-author)`) does not make much sense. Nevertheless the agent can subscribe to a channel for `Book` and a channel for (`at-least 1 has-author`). It can accumulate the results returned by RacerPro into two variables A and B, respectively, and, in order to compute the set of books for which there does not exist an author, it can consider the complement of B wrt. A. We see this strategy as an implementation of a local closed-world (LCW) assumption.

However, as time evolves, authors for documents determined by the above-mentioned query indeed might become known. In others words, the set B will probably be extended. In this case, the agent is responsible for implementing appropriate backtracking strategies, of course.

The LCW example demonstrates that the RacerPro publish and subscribe interface is a very general mechanism, which can also be used to solve other problems in knowledge representation.

Chapter 6

The New RacerPro Query Language - nRQL

In this chapter of the user guide we describe an expressive ABox query language for RacerPro, called *nRQL*. nRQL is an acronym for *new Racer Query Language*, pronounce: *Nerckle*. Previous versions have been called *RQL*; in order to avoid confusion with an RDF query language we have changed the name.

nRQL can be used to

- query RacerPro ABoxes,
- query RacerPro TBoxes,
- query RDF(S) documents,
- query OWL documents.

Thus, nRQL is

- an expressive ABox query language for the very expressive DL $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$,
- an RDF(S) query language,
- an OWL query language.

nRQL allows for the formulation of so-called (*grounded*) *conjunctive queries*. A query contains *variables* which are bound to ABox (OWL / RDF(S)) individuals, concepts, or so-called substrate nodes. Thus, we distinguish between ABox variables, TBox variables, and substrate variables. In case of an ABox variable, the variable is bound to those ABox individuals that *satisfy* the *query body*. Within a query body, concept and role terms are used to specify retrieval conditions on the bindings of a query variable (e.g., retrieve only those *?x* which are instances of class *woman* which have a child).

The features of the nRQL language can be summarized as follows:

- nRQL has a *well-defined syntax and semantics*. Please refer to the list of research publications at <http://www.racer-systems.com> for the formal specification of the semantics of nRQL. The semantics is based on the notion of *logical entailment*.

From a theoreticians point of view, nRQL offers at least so-called *grounded conjunctive queries*, but goes beyond these by offering additional constructs. Moreover, the operational aspects of the query answering engine (see below) make it somehow unique.

nRQL only offers so-called *must-bind (or: distinguished) variables* which range over the individuals of an ABox (or individuals of an RDF(S) / OWL document). Thus, a *concept query atom* (see below) such as `(?x mother)` has the same semantics as `(concept-instances mother)`. A variable is bound to an ABox individual if (and only if) this individual *satisfies* the query. *Satisfies* means that the query resulting from substituting all variables with their current bindings (resulting in the so-called grounded query) is *logically entailed* by the knowledge base (KB). So, `?x` is bound to the individual `betty` if (and only if) the ground conjunctive query `(betty mother)` is entailed.

nRQL does not offer so-called *do not bind (or: non-distinguished) variables*. A variable is always distinguished, even if the bindings of that variables bindings are not included in the query answer (the query answer contains only the bindings of those variables which appear in the *head* of the query, see below).

However, purely *existentially quantified retrieval conditions* such as “there exists a filler of the `has-child` role of `?x` such that ...” can be formulated anyway by exploiting the expressive power of the concept expressions used in concept query atoms: Simply use concept query atoms such as `(?x (some has-child ...))`.

- nRQL offers a *variety of query atoms*; the most important ones are: concept query atoms, role query atoms, constraint query atoms, and **SAME-AS** query atoms. Complex queries / query bodies are built from query atoms using the operators **and**, **union**, **neg**, **project-to**.
- *Negation as failure (NAF)* as well as *true classical negation* is available.

NAF is especially useful for measuring the degree of completeness of a modeling of the domain of discourse in the KB.

- Special support for querying the *concrete domain* part of an ABox. nRQL allows for the formulation of complex retrieval conditions on concrete domain attribute fillers of ABox individuals by means of complex *concrete domain predicates*. Concrete domain predicates can appear in concept query atoms as well as in constraint query atoms. For example, we can retrieve those individuals whose fillers of the concrete domain attribute `has-age` satisfy the complex concrete domain predicate ≥ 18 using a concept query atom `(?x (min age 18))`.

Moreover, in case satisfying concrete domain values are known in the ABox (they have been “told” to the system), these known *told values* (if existing) can be returned as part of the query answer. Thus, in case the concrete age (being a filler of the `has-age` attribute) is given as a told value in the KB, it can be returned by the query together with the individual.

nRQL also supports binary concrete domain predicates (not that ≥ 18 is a unary concrete domain predicate). For example, we can retrieve all pairs of persons having the same age with the constraint query atom `(?x ?y (constraint age age =))`.

Please note that the available retrieval predicates are from the concrete domains offered by RacerPro. Thus, for the sake of decidability in the RacerPro concept language their number is rather limited, e.g., there is no concrete domain predicate which checks whether a string contains a certain (fixed) substring. However, such additional predicates are often needed for query formulation.

A pragmatic solution to this “missing predicates problem” is to enable the so-called (*mirror*) *data substrate* and use the additional retrieval facilities offered by the *data substrate query atoms* (see below). Here you will find many useful predicates. But, unlike the concrete domain predicates, these data substrate predicates only work on *told (data) values*.

Moreover, a third pragmatic solution to the “missing predicates problem” is to use the *MiniLisp expression language* (see below) to specify arbitrary custom retrieval (filter) predicates.

- nRQL offers a projection operator **project-to** for query bodies. In combination with negation as failure this operator can be used to express *closed world universal quantification* which is important for many retrieval tasks.
- nRQL is also a powerful OWL and RDF(S) query language.

In contrast to many other RDF(S) query languages (such as Sparql), nRQL uses description logic reasoning to retrieve also “inferred triples” and thus does not work on the syntactic level of triples, but on the level of *semantic models*.

Like RDF(s) query languages, nRQL allows for the specification of retrieval predicates for data values (which are instances of XML Schema datatypes). In Sparql, these retrieval conditions are specified using the **filter** construct.

There are two main options to specify such retrieval conditions:

1. The concrete domain constraint checking facilities also apply to datatype properties. Thus, a datatype property can be used as if it were a concrete domain attribute. This applies to concept as well as to constraint query atoms. For concept query atoms, we have extended the concept expression syntax to support this. However, the language which can be used for formulating such predicates is limited to the predicates of the different concrete domains offered by RacerPro. Moreover, certain predicates cannot be supported at all in order to guarantee decidability in the RacerPro concept language.
2. However, additional and more complex retrieval predicates can be specified with queries posed to the so-called data substrate. nRQL is a *hybrid* query language. If the data substrate is enabled, then the so-called data substrate query atoms can be used to query the data substrate associated with an ABox. The data substrate of an ABox is a representation layered on top of an ABox. A full set of retrieval predicates can be evaluated on that representation (substring searching etc.).

- So-called complex TBox queries are available. These can be used to search for certain patterns of sub/superclass relationships in a taxonomy.
- Optionally, nRQL is a hybrid query language which can query a hybrid representation consisting of an ABox associated / layered with a so-called substrate. To query such a hybrid ABox/substrate representation, a hybrid nRQL query may be used which contains query atoms which are evaluated on the ABox, as well as query atoms which are evaluated on the substrate.
- The language offers extensibility and flexibility by means of a *simple expression language called MiniLisp*. With MiniLisp a user can write a simple (termination safe!) “program” which is executed on the RacerPro server. MiniLisp programs give the flexibility to specify the format of the query answers, write query answers to a file, start subqueries, etc. Moreover, and most importantly, MiniLisp enables the formulation of aggregation operators (count, sum, average, ...). See below for examples.

The *nRQL language* must be distinguished from the *nRQL query processing engine*, which is an internal part of the RacerPro server. The main features of this query processing / answering engine are:

- A cost-based heuristic *optimizer*. The optimizer reorders the query atoms in a conjunction in order to find a performant query evaluation plan. In order to estimate the costs caused by a query atom, ABox statistics (cardinality information) are exploited.
- The engine is also a rule engine. A rule has an antecedence (precondition, left hand side) which is a nRQL query. The consequence (postcondition, right hand side) contains so-called generalized ABox assertions (ABox assertions referencing variables and/or individuals)
- Queries (and rules) are maintained as *objects* within the engine. Query objects understand a complex protocol (i.e., they can be asked to return their stored answers, be deleted, reexecuted, etc.). A query (rule) is identified with an Id. The engine offers a full *life-cycle management* for queries and rules.
- The engine supports *defined queries*. A defined query is a query which has been given a name (not to be confused with the Id of a query / rule, although Id and name can be identically). The named query can be used to refer to that query within other queries, similar to a macro mechanism. Definitions must be acyclic.
- The engine supports *multi-processing* of queries which means that more than one query can be answered “simultaneously” (concurrently). For each query, a query answering thread is created.
- Support for *different querying modi*:
 - “Set at a time” mode: In this mode, the answer to a query is delivered in one big bunch as a set. The nRQL API works in a *synchronous* fashion. This means that the API is *blocked* (not available) until the query answer has been computed and delivered.

- “Tuple at a time mode”: In this mode, the answer to a query is computed and retrieved incrementally, *tuple by tuple*. The nRQL API behaves in an *asynchronous* fashion. A client can incrementally request tuples from a query one by one; moreover, while the query is still running, another query can be started in parallel.

This mode comes in two disguises: *lazy* or *eager*. In *lazy* mode, the next tuple is only computed if requested (on demand). In *eager* mode, additional (even if not yet requested) tuples are computed in advance in the background. The *lazy* mode maximizes the availability of nRQL, whereas the *eager* mode has the advantage that a request for a next tuple can eventually be satisfied immediately if that tuple has already been computed.

- The *degree of completeness* is configurable. nRQL offers a *complete mode* as well as *various incomplete modi* which differ w.r.t. the *degree of completeness* they achieve. Eventually, an incomplete mode will only deliver a subset of the complete answer to a query (compared to the answer which would be computed using the complete mode). However, the incomplete modi can be much more performant and thus achieve better scalability for large ABoxes. Also note that the incomplete modi will be complete for “simple” KBs (KBs which do not required the full expressivity of $\mathcal{ALCQHI}_{\mathcal{R}}^+(\mathcal{D}^-)$ or OWL).

The advantages of the incomplete and complete modi can be combined in nRQL: The so-called *two-phase processing modi* are complete and take care that query processing works in *two phases*. In *phase one*, the so-called *cheap tuples* are computed and delivered to the client. After there are no more cheap tuples, the client / user can be notified about the upcoming phase transition to *phase two*. nRQL can be advised to deliver a so-called *warning token* before phase two starts. The client / user can then decide to retrieve these additional expensive tuples or not. The remaining tuples from phase two are also called *expensive tuples*; unlike for the tuples from phase one, full ABox reasoning is required for their computation.

- The *runtime resources are configurable*. The API allows to set an upper bound on the number of answer tuples computed, a timeout can be specified, and the incomplete modi are available. Moreover, permutations of answer tuples can be excluded. Internally, the engine uses a thread pool. The minimum / maximum number of threads in this pool can be specified; note that a query has to wait if it is about to be executed and there is no thread available in the pool. It must thus wait until a thread is released by another (currently running) query and put back into the pool.
- We already mentioned the *substrate representation layer*. The engine offers an extensive API for maintaining this layer. There are different kinds / classes of representation substrates available, tailored for the representation of different aspects. According to the substrate kind, different substrate query atoms apply.
- *Reasoning on queries*. This functionality is currently still experimental (and incomplete). nRQL offers (incomplete) query satisfiability and query containment (“subsumption”) checks.

nRQL can be advised to maintain and exploit a so-called *query repository (QBox)*, which is a DAG-structured query cache. nRQL uses the query containment check to

maintain and compute this DAG. This can be called an semantic optimization (unlike the heuristic optimizer, which uses only syntactic information).

A further semantic (but still experimental) optimization technique which relies on query reasoning called *query realization* is also implemented.

This Chapter is structured as follows:

- The first section introduces the nRQL query language in a tutorial-like style.
- The second section describes the nRQL engine and its features.
- The complete API of the nRQL engine is specified in the RacerPro Reference Manual.

6.1 The nRQL Language

In the following we introduce the nRQL language bottom up in a tutorial-like style. “Bottom up” means that we introduce the language by first considering the various available query atom, and then introduce complex (compound) queries build from these atoms.

Please note that the example files we are using in this section are contained in the subfolder called “nrql-user-guide-examples” in the “examples” archive which can be downloaded from your personal download page under <http://www.racer-systems.com>.

Moreover, it is assumed that you use RacerPorter for the tutorial interaction with RacerPro. In case you get different results as listed here, please push the button Full Server Reset (use the Server or Shell Tab) and retry the interaction.

6.1.1 Query Atoms, Objects, Individuals, and Variables

The basic expressions of the nRQL language are so called *query atoms*, or simply atoms. In this Subsection we are only considering ABox query atoms (nRQL offers TBox and Substrate atoms as well, see below).

Atoms are either *unary* or *binary*. A unary atom references one *object*, and a binary atom references two objects.

An *object* is either an *individual*, or a *variable*. In case of an ABox atom, this object is thus either an ABox individual or a ABox variable. For example, `?x` and `$?x` are variables, and `betty` is an individual. Variables are bound to those ABox individuals that satisfy the query expression.

nRQL offers (ordinary) variables as well as so-called injective variables:

- *Variables* are prefixed with “?”, e.g. `?x`, `?y`.
- *Injective variables* are prefixed with “\$?”, e.g. `$?x`, `$?y`. An injective variable can only be bound to an ABox individual which is not already bound to another injective variable - the mapping from variables to ABox individuals is thus *injective*. For example, if `$?x` is bound to `betty`, then `$?y` *cannot* be bound to `betty` as well. Note that this can also be achieved by means of `(neg (same-as ?x ?y))`, if ordinary variables are used.

There are only four types of atoms available:

- Unary atoms:
 - concept query atoms.
- Binary atoms:
 - role query atoms,
 - constraint query atoms,

- SAME-AS query atoms.

In principle, there is a fifth kind of atom, the *query reference atom* which is n -ary and used to refer to a defined query. However, this atom should be understood on a syntactic (macro) level and is thus not discussed here. So, this atom is in fact “syntactic sugar”, as some other additional auxiliary atoms as well, e.g. the **bind-individual** atoms.

We will now discuss each type of atom.

6.1.1.1 Concept Query Atoms

Concept query atoms are unary atoms. A concept query atom is used to retrieve the instances of a concept or an OWL (or RDF(S)) class, for example, the instances of the concept **woman** in **family.racer**. We use RacerPorter to read in that file into RacerPro; note that **?** denotes the input (prompt), and **>** the RacerPro result. Please use the Shell Tab of RacerPorter and enter the following commands, or use the button Load KB... (which will send the same command) to read in the **family.racer** KB:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file
  "nrql-user-guide-examples/family.racer")
> :OKAY
```

Now, we retrieve the instances of class **woman** using a query whose body consists of a single concept query atom, **(?x woman)**:

```
? (retrieve (?x) (?x woman))
> (((?x betty)) ((?x eve)) ((?x doris)) ((?x alice)))
```

The format of the *query answer* is specified by the query head (see below). In this case, the head is given as **(?x)**.

Note that this query has the same semantics as **(concept-instances woman)**, but the result set is not returned as a single set, but as a set of so-called *binding-lists* or (componentwise named) *tuples*.

Note that a *set* of tuples is returned. We emphasize *set* here because this implies that there is no guarantee on the order of the tuples in this set. However, the result tuples itself are ordered, of course.

Moreover, complex concept terms / expressions can be used in concept query atoms:

```
? (retrieve (?x) (?x (and woman (some has-child top))))
> (((?x betty)) ((?x alice)))
```

6.1.1.1.1 Terminology - Query Head and Query Body As already mentioned, the query `(retrieve (?x) (?x woman))` has the head `(?x)` and the body `(?x woman)`.

The *body of a query* specifies the retrieval conditions, and the *head* specifies the format of the query result / answer tuples.

Each binding-list lists a number of variable-value pairs. A binding-list can also be seen as a (component-wise) named tuple.

The head may also contain individuals. However, the set of objects mentioned in the query head must be a subset of the set of objects used in the query body.

An empty query head is permitted as well (see below); this will result in a boolean query (which only returns TRUE or FALSE).

6.1.1.1.2 Boolean Queries A boolean query only returns TRUE or FALSE. Simply use an empty head to check whether there are any `woman` and `aunts` at all:

```
? (retrieve nil (?x woman))
> t

? (retrieve nil (?x aunt))
> NIL
```

Note that `t` means TRUE and `NIL` means FALSE.

6.1.1.1.3 Queries with individuals Individuals can be used in queries as well:

```
? (retrieve nil (betty woman))
> t

? (retrieve nil (janice woman))
> (:ERROR Undefined individual name janice in ABox smith-family)

? (retrieve nil (betty aunt))
> NIL
```

Please note that certain names are reserved for nRQL and thus cannot be used for individuals (see Section 6.1.9). For example, in the query `(retrieve nil (and c))`, `and` is not recognized as an individual name and thus the query is not recognized as a concept query atom. A syntax error is raised instead.

Moreover, nRQL establishes a special semantics for individuals which are used in the head of a query. Consider the query `(retrieve (betty) (betty woman))`. One possible way to answer this query would be to simply return TRUE, as if the head was empty. However, nRQL returns:

```
? (retrieve (betty) (betty woman))
> (((?betty betty)))
```

The variable `?betty` is called the representative variable for `betty`. Internally, nRQL rewrites queries by replacing individuals with representative variables:

```
? (describe-query :last)
> (:query-XX
   (:accurate :processed)
   (retrieve
    (?betty)
    (and (same-as ?betty betty) (?betty woman))
    :abox
    smith-family)))
```

6.1.1.1.4 Querying OWL and RDF(S) Documents with Concept Query Atoms

OWL and RDF(S) KBs (documents) can be queried with concept query atoms as well. RacerPro represents OWL / RDF(S) individuals and their interrelationships in the ABox, OWL / RDF(S) classes as concepts, and properties as roles in the TBox.

Consider the following OWL document `nrql-user-guide-examples/owl-ex1.owl`, defining one instance `michael` of class `person` and one instance `book123` of class `book`:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">
  <owl:Ontology rdf:about=""/>

  <owl:Class rdf:ID="person"/>
  <owl:Class rdf:ID="book"/>

  <person rdf:ID="michael"/>
  <book rdf:ID="book123"/>

</rdf:RDF>
```

With nRQL we can easily retrieve the instances of the OWL classes `http://www.owl-ontologies.com/unnamed.owl#person` and `http://www.owl-ontologies.com/unnamed.owl#book`:

```
? (full-reset)
> :okay-full-reset

? (owl-read-file
```

```

"nrql-user-guide-examples/owl-ex1.owl")
> /home/mi.wessel/nrql-user-guide-examples/owl-ex1.owl

? (get-namespace-prefix)
> http://www.owl-ontologies.com/unnamed.owl#

? (retrieve
  (?x)
  (?x http://www.owl-ontologies.com/unnamed.owl#person))
> (((?x http://www.owl-ontologies.com/unnamed.owl#michael)))

? (retrieve (?x) (?x http://www.owl-ontologies.com/unnamed.owl#book))
> (((?x http://www.owl-ontologies.com/unnamed.owl#book123)))

? (retrieve (?x) (?x #!book))
> (((?x http://www.owl-ontologies.com/unnamed.owl#book123)))

```

Please note that it is important to use the correct concept names; the name `http://www.owl-ontologies.com/unnamed.owl#person` is a so called qualified name, and `http://www.owl-ontologies.com/unnamed.owl#` is the so-called XML namespace prefix. The namespace prefix can be retrieved with the function `get-namespace-prefix`. In order to avoid typing of these long names, the `#!` prefix may be used to abbreviate the current namespace prefix (see the last query). Note that this does not only apply to concept / class names, but also to individuals and roles (properties):

```

? (retrieve () (#!book123 #!book))
> t

```

If you are using an older version ($\leq 1.9.0$) of RacerPro / RacerPorter you must use full qualified names for the classes, properties and individuals (`#!` doesn't work), and also put bars `|` around names like in this example:

```

? (retrieve (?x) (?x |http://www.owl-ontologies.com/unnamed.owl#person|))
> (((?x |http://www.owl-ontologies.com/unnamed.owl#michael|)))

```

In case you are unsure about the correct names, use the following functions to find out:

```

? (all-atomic-concepts)
> (top
  bottom
  http://www.owl-ontologies.com/unnamed.owl#person
  http://www.owl-ontologies.com/unnamed.owl#book)

? (all-roles)
> ((inv http://www.w3.org/2000/01/rdf-schema#comment)

```

```

(inv http://www.w3.org/2002/07/owl#comment)
(inv http://www.w3.org/2000/01/rdf-schema#seeAlso)
(inv http://www.w3.org/2002/07/owl#seeAlso)
(inv http://www.w3.org/2000/01/rdf-schema#isDefinedBy)
(inv http://www.w3.org/2002/07/owl#isDefinedBy)
(inv http://www.w3.org/2000/01/rdf-schema#label)
(inv http://www.w3.org/2002/07/owl#label)
(inv http://www.w3.org/2002/07/owl#versionInfo)
http://www.w3.org/2000/01/rdf-schema#comment
http://www.w3.org/2002/07/owl#comment
http://www.w3.org/2000/01/rdf-schema#seeAlso
http://www.w3.org/2002/07/owl#seeAlso
http://www.w3.org/2000/01/rdf-schema#isDefinedBy
http://www.w3.org/2002/07/owl#isDefinedBy
http://www.w3.org/2000/01/rdf-schema#label
http://www.w3.org/2002/07/owl#label
http://www.w3.org/2002/07/owl#versionInfo)

? (all-individuals)
> (http://www.owl-ontologies.com/unnamed.owl#book123
    http://www.owl-ontologies.com/unnamed.owl#michael)

```

Alternatively you can also use the RacerPorter Concepts, Roles, and Individuals Tabs in RacerPorter and turn off the Simplify Names checkbox.

nRQL is more powerful than other RDF(S) query languages; for example, true (classical) disjunction is available:

```

? (retrieve (?x) (?x (or #!book #!person)))
> (((?x http://www.owl-ontologies.com/unnamed.owl#book123))
    ((?x http://www.owl-ontologies.com/unnamed.owl#michael)))

```

6.1.1.2 Role Query Atoms

The second type of query atoms are the role query atoms. Role query atoms are binary atoms. Role query atoms are used to retrieve role fillers from an ABox, or OWL / RDF(S) individuals which are related via a certain OWL object property.

For example, we can retrieve all individuals in the ABox of `family.racer` using the following query whose body consists of a single role query atom (`?x ?y has-child`):

```

? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

```

```
? (retrieve (?x ?y) (?x ?y has-child))
> (((?x betty) (?y doris))
  ((?x betty) (?y eve))
  ((?x alice) (?y betty))
  ((?x alice) (?y charles)))
```

The expression `(?mother ?child has-child)` is a *role query atom*.

Again, individuals can be used at any position where a variable is accepted – to retrieve the children of `betty`:

```
? (retrieve (?child-of-betty) (betty ?child-of-betty has-child))
> (((?child-of-betty eve)) ((?child-of-betty doris)))
```

Please note that *concrete domain attributes* cannot be used in role query atoms. Fillers of concrete domain attributes can be retrieved by means of the so-called *head-projection operators*, see below. `family.racer` contains the `age` attribute of type `cardinal`:

```
? (retrieve (?x ?y) (?x ?y age))
> (:ERROR Parser Error: Unrecognized expression (?x ?y age))
```

However, *features (functional roles)* are a special kind of roles and thus can be used; for example, `has-gender` is a feature in `family.racer`:

```
? (retrieve (?person ?gender) (?person ?gender has-gender))
> NIL
```

So, the answer set is empty! Why is this? The reason is that there are no explicit “gender” fillers of the `has-gender` feature in that ABox. Thus, `?gender` cannot be bound to an ABox individual. There are two solutions: Either new “gender” individuals could be added with a *rule* (see below), or a concept query atom can be used to retrieve, for example, those instances with female gender:

```
? (retrieve (?x) (?x (some has-gender female)))
> (((?x betty)) ((?x eve)) ((?x doris)) ((?x alice)))
```

6.1.1.2.1 Role Terms in Role Query Atoms Whereas arbitrary concept terms / expressions (not only atomic concepts, also called concept names) are admissible in concept query atoms, role terms are admissible in role query atoms (not only atomic roles, also called role names). nRQL admits inverse and negated roles:

```
? (retrieve (?child-of-betty) (?child-of-betty betty (inv has-child)))
> (((?child-of-betty eve)) ((?child-of-betty doris)))
```

Please note that `(inv has-child)` is a role term. The second kind of role term constructor is given by the `not` operator, which is used to construct a negated role. Let us discuss negated roles.

6.1.1.2.2 Classical Negated Roles in nRQL Availability of classical negated roles is another unique feature of nRQL. The semantics is analog to (classical) negated concepts. For example, RacerPro can prove that `male` persons can never be instances of the concept `mother` in `family.racer`, since `mother` implies `female`, and `male` and `female` are declared as disjoint in `family.racer`:

```
? (retrieve (?x) (?x (not woman)))
> (((?x charles)))
```

A *negated role* works analogously to a negated concept. Note that negated roles are only available in the nRQL query language, but not in the RacerPro concept language (to grant decidability). A negated role (`not has-sister`) allows us to “verify” that male persons can never be fillers of the `has-sister` role, since `family.racer` contains the axiom

```
(implies *top* (all has-sister (some has-gender female)))
```

This means that fillers of the `has-sister` role have a female gender, and thus they cannot be male.¹ This matches our intuition, and indeed, using a negated role (`not has-sister`), we see that `charles` cannot be a sister of anyone:

```
? (retrieve (?x ?y) (?x ?y (not has-sister)))
> (((?x eve) (?y charles))
  ((?x doris) (?y charles))
  ((?x charles) (?y charles))
  ((?x betty) (?y charles))
  ((?x alice) (?y charles)))
```

Please note that queries involving negated roles are rather expensive.

6.1.1.2.3 Querying OWL KBs with Role Query Atoms In the OWL realm, so-called *object properties* are the equivalent of roles. Consider the following OWL document `owl-ex2.owl` in the `nrql-user-guide-examples` folder:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">

  <owl:Ontology rdf:about=""/>
```

¹Note that `has-gender` has been declared as a so-called feature, a functional role. So, any person can have at most one gender, and male and female are disjoint.


```

<owl:Class rdf:ID="person"/>

<owl:ObjectProperty rdf:ID="hasChild">
  <rdfs:domain rdf:resource="#person"/>
  <rdfs:range rdf:resource="#person"/>
</owl:ObjectProperty>

<person rdf:ID="alice">
  <hasChild>
    <person rdf:ID="betty"/>
  </hasChild>
</person>
</rdf:RDF>

```

Thus, there are two instances of class `person`. The individual `michael` is the filler of the `hasChild` object property of the individual `margrit`. Since object properties are mapped to roles in RacerPro, role query atoms can be used:

```

? (full-reset)
> :okay-full-reset

? (owl-read-file "2 ? (owl-read-file "nrql-user-guide-examples/owl-ex2.owl")
> nrql-user-guide-examples/owl-ex2.owl

? (retrieve (?x ?y) (?x ?y #!hasChild))
> (((?x http://www.owl-ontologies.com/unnamed.owl#alice)
    (?y http://www.owl-ontologies.com/unnamed.owl#betty)))

```

Again, `#!` is bound to the current namespace prefix:

```

? (get-namespace-prefix)
> http://www.owl-ontologies.com/unnamed.owl#

? (retrieve (?x ?y) (?x ?y http://www.owl-ontologies.com/unnamed.owl#hasChild))
> (((?x http://www.owl-ontologies.com/unnamed.owl#alice)
    (?y http://www.owl-ontologies.com/unnamed.owl#betty)))

```

Moreover, `#!` also works for individual names:

```

? (retrieve (?x) (?x #!betty #!hasChild))
> (((?x http://www.owl-ontologies.com/unnamed.owl#alice)))

```

Please note that for older version of RacerPro ($\leq 1.9.0$) the `#!` abbreviation will not work, and moreover, bars must be used as follows:

```
? (retrieve (?x ?y) (?x ?y |http://www.owl-ontologies.com/unnamed.owl#hasChild|))
> (((?x http://www.owl-ontologies.com/unnamed.owl#alice)
    (?y http://www.owl-ontologies.com/unnamed.owl#betty)))
```

Note that nRQL also provides facilities to query for the fillers of *OWL annotation* and *OWL datatype properties*, see below.

6.1.1.2.4 Explicit and Implicit Role Fillers It is important to understand that only explicitly modeled role fillers can be retrieved with nRQL's role query atoms. However, some KBs may have *implicit / logically implied role fillers* whose presence is enforced in the logical models of the KB. Let us consider `family.racer` again to illustrate this effect:

```
? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

? (instance doris mother)
> :OKAY

? (retrieve (?x) (?x (some has-child top)))
> (((?x betty)) ((?x doris)) ((?x alice)))

? (retrieve (?x) (?x ?y has-child))
> (((?x betty)) ((?x alice)))
```

Now we have the situation for `doris`: Since we have added the assertion `(instance doris mother)` it is the case that `doris` must have some child; thus, in every model of the KB, there will be some successor of the `has-child` role. However, this child is not explicitly present in the ABox. Thus, no binding can be found for `?y` if `?x=doris` in `(?x ?y has-child)`.

It is possible to identify such individuals which have a certain logically implied / implicit role filler which is not explicitly modelled using the NAF negation `neg` and `project-to` (see below):

```
? (retrieve
    (?x)
    (and (?x mother) (neg (project-to (?x) (?x ?y has-child)))))
> (((?x doris)))
```

Using a rule (see below) it even becomes possible to “create” some name for the logically implied child and add appropriate assertions for `doris` to the ABox as follows:

```
? (firerule
    (and (?x mother) (neg (project-to (?x) (?x ?y has-child)))))
    ((related ?x (new-ind child-of ?x) has-child)))
> (((related doris child-of-doris has-child)))
```

```
? (retrieve (?x ?y) (?x ?y has-child))
> (((?x doris) (?y child-of-doris))
    ((?x betty) (?y doris))
    ((?x betty) (?y eve))
    ((?x alice) (?y betty))
    ((?x alice) (?y charles)))
```

6.1.1.3 Constraint Query Atoms

Constraint query atoms are binary atoms and address the concrete domain part of an ABox. Like role query atoms they are used to retrieve pairs of individuals which are in a certain relationship to one another. However, this relationship is not a role, but specified with a (possibly complex) binary concrete domain predicate, like `=`. For example, we can retrieve those pairs of individuals that have the same **age**. **age** is a so-called *concrete domain attribute* of type **cardinal**. The `family.racer` KB already contains such an **age** attribute, as well as assertions specifying the individual ages of the family members:

```
(instance alice (= age 80))
(instance betty (= age 50))
(instance charles (= age 55))
(instance eve (= age 18))
(instance doris (= age 24))
```

Using a concept query atom one can easily identify the adult family members:

```
? (retrieve (?x) (?x (>= age 18)))
> (((?x alice)) ((?x doris)) ((?x eve)) ((?x betty)) ((?x charles)))
```

However, in order to find out who is older than whom, a binary retrieval predicate must be used:

```
? (retrieve (?x ?y) (?x ?y (constraint age age <)))
> (((?x doris) (?y alice))
    ((?x doris) (?y betty))
    ((?x doris) (?y charles))
    ((?x eve) (?y alice))
    ((?x eve) (?y doris))
    ((?x eve) (?y betty))
    ((?x eve) (?y charles))
    ((?x betty) (?y alice))
    ((?x betty) (?y charles))
    ((?x charles) (?y alice)))
```

Using the `neg` and `project-to` operators (see below) it is even possible to retrieve the oldest person, the oldest female, or oldest male person as follows:

```
? (retrieve (?x) (neg (project-to (?x) (?x ?y (constraint age age <)))))
> (((?x alice)))
```

```
? (retrieve (?x)
  (and
    (?x woman)
    (neg
      (project-to
        (?x)
        (and (?y woman) (?x ?y (constraint age age <)))))
  ))
> (((?x alice)))
```

```
? (retrieve
  (?x)
  (and
    (?x man)
    (neg
      (project-to
        (?x)
        (and (?y man) (?x ?y (constraint age age <)))))
  ))
> (((?x charles)))
```

6.1.1.3.1 Role Chains in Constraint Query Atoms Not only attributes can be used in constraint query atoms, but *role chains ended by an attribute*. This means that a constraint query atoms such as

```
(?x ?y (constraint (has-child age) (has-child age) <))
```

can be used to retrieve those (not necessarily distinct) parents *?x*, *?y* such that *?x* has at least one child that is younger than one of *?y*'s children. Moreover, role terms can be used in these role chains – thus, negated and inverse roles are admissible.

6.1.1.3.2 Complex Predicates So far we have only used simple concrete domain predicate, e.g. *<*, *=*. Complex predicates are admissible as well – suppose you want to find out who is *at least 40 years older than whom*:

```
? (retrieve
  (?x ?y)
  (?x ?y (constraint (age) (age) (> age-1 (+ age-2 40)))))
> (((?x alice) (?y doris)) ((?x alice) (?y eve)))
```

The complex predicate *(> age-1 (+ age-2 40))* is best understood as a *lambda expression* with two formal parameters *age-1* and *age-2* which are bound to the actual arguments supplied by the *(age)* of *?x* and the *(age)* of *?y*:

$$((\lambda(age_1, age_2) \bullet age_1 > age_2 + 40) \ age(?x), \ age(?y))$$

Thus, $\lambda(age_1, age_2) \bullet age_1 > age_2 + 40$ is applied to the actual arguments $age(?x)$ and $age(?y)$ and either returns TRUE or FALSE.

Note that the formal parameters (here: age_1, age_2) are computed from the attribute names. The suffixes -1 and -2 are added just in case the same attribute is used, in order to be able to differentiate the two in the body of the predicate.

6.1.1.3.3 Querying OWL KBs with Constraint Query Atoms In the OWL realm, the equivalent of concrete domain attributes are called OWL *datatype properties*.

Suppose we add to `owl-ex2.owl` two datatype property declarations for the properties `age` and `name`, and specify appropriate fillers for `betty` and `alice` as follows (see file `owl-ex3.owl`):

```
<owl:DatatypeProperty rdf:ID="age">
  <rdfs:domain rdf:resource="#person"/>
  <rdfs:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>

<owl:FunctionalProperty rdf:ID="name">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#person"/>
  <rdfs:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<person rdf:ID="alice">
  <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">80</age>
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Alice</name>
  <hasChild>
    <person rdf:ID="betty">
      <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">50</age>
      <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Betty</name>
    </person>
  </hasChild>
</person>
</rdf:RDF>
```

First we like to mention that it is possible to query datatype properties as if they were concrete domain attributes; we have extended the RacerPro concept expression syntax appropriately:

```
? (retrieve (?x) (?x (> #!age 75)))
> (((?x http://www.owl-ontologies.com/unnamed.owl#alice)))

? (retrieve (?x) (?x (string= #!name "Betty")))
```

```
> (((?x http://www.owl-ontologies.com/unnamed.owl#betty)))

? (retrieve (?x) (?x (a #!name)))
35 > (((?x http://www.owl-ontologies.com/unnamed.owl#alice))
      ((?x http://www.owl-ontologies.com/unnamed.owl#betty)))
```

Constraint query atoms work as well for datatype properties, but some auxiliary role assertions must be added. nRQL must be put into a special mode as the following example demonstrates:

```
? (full-reset)
> :okay-full-reset

? (add-role-assertions-for-datatype-properties)
> :okay-adding-role-assertions-for-datatype-properties

? (owl-read-file "nrql-user-guide-examples/owl-ex3.owl")
> /home/mi.wessel/nrql-user-guide-examples/owl-ex3.owl

? (retrieve
  (?x ?y)
  (?x ?y (constraint (#!age) (#!age) <)))
> (((?x http://www.owl-ontologies.com/unnamed.owl#betty)
    (?y http://www.owl-ontologies.com/unnamed.owl#alice)))
```

Again, complex predicates can be used

```
? (retrieve (?x ?y)
  (?x ?y
    (constraint
      (#!age) (#!age)
      (< (+ #!age-1 30) #!age-2))))
> NIL

? (retrieve (?x ?y)
  (?x ?y
    (constraint
      (#!age) (#!age)
      (< (+ #!age-1 29) #!age-2))))
> (((?x http://www.owl-ontologies.com/unnamed.owl#betty)
    (?y http://www.owl-ontologies.com/unnamed.owl#alice)))
```

6.1.1.4 SAME-AS Query Atoms and Synonym Individuals

nRQL supports different notions of equality or sameness. The first notion is the syntactic notion of *name equality*. Using the binary `same-as` query atom (in a complex query) it is

possible to specify that two variables must be bound to the same individual, or must be bound to individuals with different names. Please note that we have not yet introduced the `neg` and `and` operators that appear in the following example, but an intuitive understanding is sufficient for the moment:

```
? (full-reset)
> :okay-full-reset

? (instance santa-claus good-man)
> :OKAY

? (instance weihnachtsmann good-man)
> :OKAY

? (retrieve (?x ?y) (and (?x good-man) (?y good-man)))
> (((?x santa-claus) (?y santa-claus))
    ((?x santa-claus) (?y weihnachtsmann))
    ((?x weihnachtsmann) (?y santa-claus))
    ((?x weihnachtsmann) (?y weihnachtsmann)))

? (retrieve
  (?x ?y)
  (and (?x good-man) (?y good-man) (same-as ?x ?y)))
> (((?x santa-claus) (?y santa-claus))
    ((?x weihnachtsmann) (?y weihnachtsmann)))

? (retrieve
  (?x ?y)
  (and (?x good-man) (?y good-man) (neg (same-as ?x ?y))))
> (((?x santa-claus) (?y weihnachtsmann))
    ((?x weihnachtsmann) (?y santa-claus)))
```

The second notion of sameness is *semantic equality*. Suppose you learn that `weihnachtsmann` is just the german name for `santa-claus`; thus, `weihnachtsmann` and `santa-claus` denote the same thing in the universe of discourse – they are *individual synonyms*. Let us add one more assertion that enforces that `weihnachtsmann` and `santa-claus` are synonyms:

```
? (same-as weihnachtsmann santa-claus)
> :OKAY

? (individual-synonyms santa-claus)
> (weihnachtsmann santa-claus)
```

In general, for each individual there is a set of cardinality at least one of synonym individuals, including the individual itself, forming the *synonym equivalence class* for that individual.

nRQL always removes redundant bindings from a query answer; i.e., you will never get a query answer which contains `((?x santa-claus))` twice. In order to detect redundant bindings, nRQL uses by default name equality. Thus, nRQL considers two variable bindings as different if the individuals to which the variables are bound have different names:

```
? (retrieve (?x) (?x good-man))
> (((?x santa-claus)) ((?x weihnachtsmann)))
```

However, sometimes one wants to ensure that only “semantically different” bindings are delivered; thus, either `(?x santa-claus)` or `(?x weihnachtsmann)` would be considered redundant. This is possible as well (see below).

In addition to the (negated) `same-as` query atom, nRQL offers a special role, the so-called `nrql-equal-role` which holds between two individuals if they are in the same synonym equivalence class. So it becomes possible to query for pairs of individuals which have a different name but are in fact synonyms:

```
? (retrieve
    (?x ?y)
    (and (neg (same-as ?x ?y)) (?x ?y nrql-equal-role)))
> (((?x weihnachtsmann) (?y santa-claus))
    ((?x santa-claus) (?y weihnachtsmann)))
```

Moreover, nRQL can be switched into a mode in which variables are not bound to different ABox individuals, but to *representative individuals from different synonym equivalence classes*:

```
? (use-individual-synonym-equivalence-classes)
> :okay-using-individual-equivalence-classes

? (retrieve (?x) (?x good-man))
> (((?x santa-claus)))
```

Using a so-called *head-projection operator* (see below) the synonyms can be delivered as well:

```
? (retrieve (?x (individual-synonyms ?x))
    (?x good-man)
    :dont-show-lambdas-p
    t)
> (((?x santa-claus) (santa-claus weihnachtsmann)))
```

In this mode, the `same-as` atom no longer denotes name equality, but semantic equality. Thus, `(same-as ?x ?y)` becomes equivalent to `(?x ?y nrql-equal-role)`, and the `nrql-equal-role` shouldn't be used:


```
? (retrieve (?x) (same-as ?x weihnachtsmann))
> (((?x santa-claus)))
```

Finally we want to mention that the `same-as` assertion is not the only way how to two individuals can be forced to become synonyms. Also features (functional roles) or so-called number restrictions can cause individuals to become synonyms, for example:

```
? (full-reset)
> :okay-full-reset
```

```
? (define-primitive-role f :feature t)
> :OKAY
```

```
? (related i j f)
> :OKAY
```

```
? (related i k f)
> :OKAY
```

```
? (related l m r)
> :OKAY
```

```
? (related l n r)
> :OKAY
```

```
? (instance l (at-most 1 r))
> :OKAY
```

```
? (retrieve
  (?x ?y)
  (and (?x ?y nrql-equal-role) (neg (same-as ?x ?y))))
> (((?x n) (?y m)) ((?x m) (?y n)) ((?x k) (?y j)) ((?x j) (?y k)))
```

Please note that the `same-as` assertion is also available in OWL.

6.1.1.5 Auxiliary Query Atoms

Some auxiliary query atoms are offered by nRQL. However, these atoms are not as important as the atoms we have already discussed.

6.1.1.5.1 HAS-KNOWN-SUCCESSOR Query Atoms Sometimes one just wants to retrieve individuals which have a certain role successor, but the actual successor itself is of no interest:

```
? (full-reset)
```

```
> :okay-full-reset
```

```
? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY
```

```
? (retrieve (?x) (?x (has-known-successor has-child)))
> (((?x betty)) (?x alice)))
```

This is in fact equivalent to:

```
? (retrieve (?x) (?x ?y has-child))
> (((?x betty)) (?x alice)))
```

However, the situation changes if the NAF negated variants of these atoms are considered:

```
? (retrieve (?x) (neg (?x ?y has-child)))
> (((?x eve)) (?x doris)) ((?x charles)) ((?x betty)) ((?x alice)))

? (retrieve (?x) (neg (?x (has-known-successor has-child))))
> (((?x eve)) (?x doris)) ((?x charles)))
```

The explanation for the difference is that for the first query the two-dimensional complement of `(?x ?y has-child)` is constructed with `(neg (?x ?y has-child))` and this set of pairs is projected to their first components / tuple positions. In the second query, a one-dimensional complement is constructed. The second query is in fact equivalent to

```
? (retrieve (?x) (neg (project-to (?x) (?x ?y has-child))))
> (((?x eve)) (?x doris)) ((?x charles)))
```

The atom `(?x (has-known-successor has-child))` is therefore just syntactic sugar for the body `(neg (project-to (?x) (?x ?y has-child)))`.

6.1.1.5.2 BIND-INDIVIDUAL Query Atoms A `(bind-individual <name>)` query atom for some individual `<name>` is just syntactic sugar for `(same-as ?<name> <name>)`, where `?<name>` is the representative variable for `<name>`. These atoms are provided for backward compatibility with older versions of nRQL only.

6.1.2 Query Head Projection Operators

So far we have only used simple objects (variables and individuals) in the head of a query. Additionally, so-called *head projection operators* are admitted as head entries as well. A head projection operator is simply a function (operator) which is applied to the current binding of the variable (the current individual), and the operator result is included in the answer tuple. There are various predefined operators available. nRQL also offers a simple (termination-safe) expression language called MiniLisp. Users can define their own projection operators as `lambda` head operators.

6.1.2.1 Retrieving Told Values from the Concrete Domain

We already mentioned that variables can only be bound to ABox individuals. However, an ABox may also contain so-called *concrete domain objects* (*CD objects*) as well as *told values* (data values / literals from the concrete domain). We already showed that a concept query atom such as `(?x (>= age 18))` can be used to specify retrieval conditions on concrete domain attributes (here `age`). However, no variable can be bound to the `age` attribute value, even if there is a *told value* given in the ABox.

Thus, head projection operators are provided in order to retrieve concrete domain values and/or concrete domain objects.

```
? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

? (retrieve (?x) (?x (>= age 18)))
> (((?x alice)) ((?x doris)) ((?x eve)) ((?x betty)) ((?x charles)))

? (retrieve (?x (told-value-if-exists (age ?x))) (?x (>= age 18)))
> (((?x alice) ((:existing-told-values (age ?x)) (80)))
  ((?x doris) ((:existing-told-values (age ?x)) (24)))
  ((?x eve) ((:existing-told-values (age ?x)) (18)))
  ((?x betty) ((:existing-told-values (age ?x)) (50)))
  ((?x charles) ((:existing-told-values (age ?x)) (55))))
```

Please note that the concrete age cannot always be determined. In this example, the told values have been gathered from the concept assertions in `family.racer` of the form `(instance (instance alice (= age 80)))` etc.

nRQL also considers constrained and constraints assertion in the ABox to identify told values; for example:

```
? (constrained peter age-of-peter age)
> :OKAY

? (constraints (= age-of-peter 36))
> :OKAY

? (retrieve
  (?x (told-value-if-exists (age ?x)))
  (?x (and (> age 30) (< age 40))))
> (((?x peter) ((:existing-told-values (age ?x)) (36))))
```

Note that `age-of-peter` is called a *concrete domain object (CD object)*, and `36` is a concrete domain value. Please note that concrete domain are in fact variables in a concrete domain constraint network; nRQL tries to identify and maximize as many told values as possible:

```
? (constrained mary age-of-mary age)
> :OKAY

? (constraints (= age-of-peter age-of-mary))
> :OKAY

? (retrieve
  (?x (told-value-if-exists (age ?x)))
  (?x (and (> age 30) (< age 40))))
> (((?x mary) ( (:existing-told-values (age ?x)) (36)))
  ((?x peter) ( (:existing-told-values (age ?x)) (36))))
```

However, in some case this is not possible; in this case, `told-value-if-exists` returns the CD object instead of the told value:

```
? (constrained paul age-of-paul age)
> :OKAY

? (constraints (< age-of-paul age-of-mary))
> :OKAY

9 ? (retrieve (?x (told-value-if-exists (age ?x))) (?x (< age 40)))
9 > (((?x peter) ( (:existing-told-values (age ?x)) (36)))
  ((?x paul) ( (:existing-told-values (age ?x)) (age-of-paul)))
  ((?x mary) ( (:existing-told-values (age ?x)) (36)))
  ((?x doris) ( (:existing-told-values (age ?x)) (24)))
  ((?x eve) ( (:existing-told-values (age ?x)) (18))))
```

Please note that `age-of-paul` is returned, since the told value could not be identified. This is not surprising, since Paul's age is underspecified (we only know that he is younger than Mary).

There are two more operators which address concrete domain objects. Operators of the form `(<attribute> <object>)`, e.g., `(age ?x)` return the concrete domain object which is the filler of `<attribute>` of the current binding of `<object>`:

```
? (retrieve (?x (age ?x)) (?x (an age)))
> (((?x peter) ((age ?x) (age-of-peter)))
  ((?x paul) ((age ?x) (age-of-paul)))
  ((?x mary) ((age ?x) (age-of-mary)))
  ((?x alice) ((age ?x) :no-cd-objects))
  ((?x doris) ((age ?x) :no-cd-objects)))
```

```
((?x eve) ((age ?x) :no-cd-objects))
((?x betty) ((age ?x) :no-cd-objects))
((?x charles) ((age ?x) :no-cd-objects)))
```

The `told-value` operator can be applied to CD objects (if existing) to retrieve told values for that CD object:

```
? (retrieve (?x (told-value (age ?x))) (?x (an age)))
> (((?x peter) ([:told-values (age ?x)] (36)))
  ((?x paul) ([:told-values (age ?x)] (:no-told-value)))
  ((?x mary) ([:told-values (age ?x)] (36)))
  ((?x alice) ([:told-values (age ?x)] :no-cd-objects))
  ((?x doris) ([:told-values (age ?x)] :no-cd-objects))
  ((?x eve) ([:told-values (age ?x)] :no-cd-objects))
  ((?x betty) ([:told-values (age ?x)] :no-cd-objects))
  ((?x charles) ([:told-values (age ?x)] :no-cd-objects))))
```

Please note that `told-value-if-exists` is more general.

6.1.2.2 Retrieving Told Fillers of OWL Datatype Properties

OWL KBs may contain datatype fillers of datatype properties. In OWL, these fillers are values / literals of XSD (XML Schema) datatypes. In RacerPro, these fillers are represented as told values in the concrete domain part of the ABox. With the help of the `datatype-fillers` head projection operator it is possible to retrieve these data fillers.

Let us consider `owl-ex3.owl` again, which contains two datatype properties `name` and `age`:

```
? (full-reset)
> :okay-full-reset

? (owl-read-file "nrql-user-guide-examples/owl-ex3.owl")
> nrql-user-guide-examples/owl-ex3.owl

? (retrieve
  (?x
    (datatype-fillers (!name ?x))
    (datatype-fillers (!age ?x))
    (?x (and !person (> !age 30)))))
> ((((?x http://www.owl-ontologies.com/unnamed.owl#alice)
  ([:datatype-fillers
    (http://www.owl-ontologies.com/unnamed.owl#name ?x)
    (Alice))
  ([:datatype-fillers
    (http://www.owl-ontologies.com/unnamed.owl#age ?x)
    (80)))))
```

```
((?x http://www.owl-ontologies.com/unnamed.owl#betty)
 (:datatype-fillers
  (http://www.owl-ontologies.com/unnamed.owl#name ?x))
 (Betty))
 (:datatype-fillers
  (http://www.owl-ontologies.com/unnamed.owl#age ?x))
 (50))))
```

Please note that it is possible to specify complex retrieval conditions on the fillers of such datatype properties, as the concept query atom (?x (> http://www.owl-ontologies.com/unnamed.owl#age 30)) demonstrates. The retrieval predicates can be rather complex, as the concept query atom (?x (some #!age (or (and (> 40) (< 60)) (= 80)))) demonstrates. Note that this concept is only permitted as a query concept in a concept query atom, but not elsewhere in a RacerPro KB.

In order to specify retrieval conditions on boolean datatype properties it is important to know how to specify TRUE and FALSE. In owl-ex4.owl we have added a datatype property married of type boolean to owl-ex3.owl:

```
? (full-reset)
> :okay-full-reset

? (owl-read-file "nrql-user-guide-examples/owl-ex4.owl")
> /home/mi.wessel/nrql-user-guide-examples/owl-ex4.owl

? (retrieve
  (?x
    (datatype-fillers (#!married ?x)))
  (?x (boolean= #!married #T)))
> (((?x http://www.owl-ontologies.com/unnamed.owl#alice)
  (:datatype-fillers
    (http://www.owl-ontologies.com/unnamed.owl#married ?x))
  (#T))))

? (retrieve
  (?x
    (datatype-fillers (#!married ?x)))
  (?x (boolean= #!married #F)))
> (((?x http://www.owl-ontologies.com/unnamed.owl#betty)
  (:datatype-fillers
    (http://www.owl-ontologies.com/unnamed.owl#married ?x))
  (#F))))
```

6.1.2.3 Retrieving Told Fillers of OWL Annotation Properties

In OWL, the so-called *annotation properties* are used to annotate resources with meta data, e.g., comments on authorship of an ontology, etc. Annotation properties are not used for

reasoning – they are simply ignored. Nevertheless, meta data is important and one should be able to retrieve this information from an OWL document.

OWL distinguishes two kinds of annotation properties:

1. annotation object properties, and
2. annotation datatype properties.

Fillers (told values) of annotation *datatype* properties are XML Schema datatype (XSD) values / literals, whereas fillers of annotation *object* properties are ordinary ABox (OWL / RDF) individuals. To retrieve a filler of an annotation datatype property, special head projection operators must be used (similar to the datatype property case): **annotation-datatype-fillers**. To retrieve fillers of annotation object properties, ordinary role query atoms can be used. Please note that no reasoning is performed on annotation properties. Thus, no *complex* retrieval conditions can be specified with the help of concept query atoms. However, with the help of the so-called *mirror data substrate* this becomes possible (see below).

Let us consider an example OWL (file `owl-ex5.owl`) document which contains all four kinds of properties:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.owl-ontologies.com/Ontology1159352693.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.owl-ontologies.com/Ontology1159352693.owl">

  <owl:Ontology rdf:about="">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >Demo Ontology</rdfs:comment>
  </owl:Ontology>

  <owl:Class rdf:ID="class">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >This is a test class</rdfs:comment>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="object-property">
    <rdfs:domain rdf:resource="#class"/>
  </owl:ObjectProperty>

  <owl:DatatypeProperty rdf:ID="datatype-property">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
```

```

    <rdfs:domain rdf:resource="#class"/>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="annotation-object-property">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
    <rdfs:domain rdf:resource="#class"/>
  </owl:ObjectProperty>

  <owl:DatatypeProperty rdf:ID="annotation-datatype-property">
    <rdfs:domain rdf:resource="#class"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
  </owl:DatatypeProperty>

  <class rdf:ID="test">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Comment for individual test</rdfs:comment>
    <object-property rdf:resource="#test"/>
    <datatype-property rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
    >123</datatype-property>
    <annotation-object-property rdf:resource="#test"/>
    <annotation-datatype-property rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
    >456</annotation-datatype-property>
  </class>

</rdf:RDF>

```

Thus, here we have

- an ontology which is annotated with the string "Demo Ontology" which is the filler of the (RDF(S) predefined) `rdfs:comment` annotation datatype property of type `XSD:string`,
- a class `#!class`, which is `rdfs:comment` annotated with "This is a test class",
- an ordinary object property `#!object-property`,
- an ordinary datatype property `#!datatype-property` of type `XSD:int`,
- an *annotation object property* `#!annotation-object-property`,
- an *annotation datatype property* `#!annotation-datatype-property` of type `int`,
- and an individual named `#!test` being an instance of `#!class`, which has
 - itself as a filler of `#!object-property`,
 - the integer 123 as a filler of `#!datatype-property`,

- the string "Comment for individual test" as a filler of the annotation datatype property `rdf:comment`,
- itself as a filler of `#!annotation-object-property`, and
- the integer 456 as as filler of the `#!annotation-datatype-property`.

The following queries demonstrate how the fillers of these different properties can be retrieved with nRQL; the queries should speak for themselves:

```
? (retrieve (?x) (?x #!class))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)))

? (retrieve (?x ?y) (?x ?y #!object-property))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  (?y http://www.owl-ontologies.com/Ontology1159352693.owl#test)))

? (retrieve (?x ?y) (?x ?y #!annotation-object-property))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  (?y http://www.owl-ontologies.com/Ontology1159352693.owl#test)))

? (retrieve (?x) (?x (some #!object-property #!class)))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)))

? (retrieve (?x) (?x (some #!annotation-object-property #!class)))
> NIL

;;; Note: no reasoning for annotation object properties!

? (retrieve (?x (datatype-fillers (#!datatype-property ?x)))
  (?x (a #!datatype-property)))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  (:datatype-fillers
    (http://www.owl-ontologies.com/Ontology1159352693.owl#datatype-property ?x))
  (123)))

? (retrieve (?x (datatype-fillers (#!datatype-property ?x)))
  (?x (some #!datatype-property integer)))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  (:datatype-fillers
    (http://www.owl-ontologies.com/Ontology1159352693.owl#datatype-property ?x))
  (123)))

? (retrieve (?x (datatype-fillers (#!datatype-property ?x)))
  (?x (some #!datatype-property (and (min 100) (max 200)))))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  (:datatype-fillers
    (http://www.owl-ontologies.com/Ontology1159352693.owl#datatype-property ?x))
  (123)))

? (retrieve (?x (annotation-datatype-fillers (#!annotation-datatype-property ?x)))
  (?x (an #!annotation-datatype-property)))
```

```

> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  ((:annotation-datatype-fillers
    (http://www.owl-ontologies.com/Ontology1159352693.owl#annotation-datatype-property
     ?x))
   (456))))

? (retrieve (?x (annotation-datatype-fillers (!annotation-datatype-property ?x)))
  (?x (some #!annotation-datatype-property integer)))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  ((:annotation-datatype-fillers
    (http://www.owl-ontologies.com/Ontology1159352693.owl#annotation-datatype-property
     ?x))
   (456))))

? (retrieve (?x (annotation-datatype-fillers (!annotation-datatype-property ?x)))
  (?x (some #!annotation-datatype-property (and (min 300) (max 500)))))
> NIL

;;; Note: no reasoning for annotation datatype properties!

```

Note that we can neither retrieve the `RDF(S):comment` annotation "Demo Ontology" of the whole ontology nor the class annotation "This is a test class" in that way, since nRQL is an ABox query language and there are simply no corresponding ABox individuals representing the whole ontology and/or the classes.

However, the so-called *mirror data substrate* will (among other things) contain “nodes” representing the *used XSD literals* as well as nodes representing the individual *OWL classes* and thus, using a mirror data substrate instead of an ABox, the annotation fillers of classes can be retrieved, see below.

6.1.3 Lambda Head Operators to Evaluate Expressions

The application of a head projection operator can be understood as a function application. So-called *lambda expressions* can denote (anonymous) functions. nRQL allows for the specification of lambda expressions in the head of a query; nRQL uses a Lisp dialect which we call MiniLisp. MiniLisp is a termination-safe expression language (not a programming language).

For example, consider an ABox representing material objects in the physical world, having width and length, and we want to compute and return the area of these objects with a query (see file `lambda-ex1.owl`):

```

? (define-concrete-domain-attribute width :type integer)
> :OKAY

? (define-concrete-domain-attribute length :type integer)
> :OKAY

```

```
? (instance i (and (equal width 10) (equal length 20)))
> :OKAY

? (retrieve
  (?x
    ((lambda (x y) (* (first x) (first y)))
      (told-value-if-exists (width ?x))
      (told-value-if-exists (length ?x))))
  (?x (and (a width) (a length))))
> (((?x i)
  (((:lambda (x y) (* (first x) (first y)))
    (:existing-told-values (width ?x))
    (:existing-told-values (length ?x))
    200)))
```

The function / lambda application is performed by substituting the formal parameters *x*, *y* to the actual arguments supplied by the two `told-value-if-exists` projection operators. These operators return lists of (told) values; thus, the *first* function is applied before *** is applied to yield the total size.

6.1.3.1 MiniLisp

MiniLisp is easy to understand and use for readers which have some Common Lisp (CL) experience. MiniLisp only supports symbols, strings, numbers, and lists. Many of the built-in operators are inherited from CL. All RacerPro API functions can be called from within a lambda body; RacerPro macros are treated as functions.

MiniLisp supports / `quote`, ' / `backquote`, , / `comma`, and @ / `bq-comma-at-sign`. This is very useful if nRQL (sub)queries with variable parts shall be executed from within a lambda body.

The lambda body is evaluated in an environment where `*current-abox*` is bound to the query ABox, and `*current-tbox*` to the query TBox. Moreover, `*output-stream*` is bound to the file output stream within the scope of `with-open-output-file`; this is an ordinary CL output stream, and can thus be used as argument to standard CL functions such as `format`. See below for examples for file output.

The following special forms are provided: `reduce`, `and`, `or`, `not`, `if`, `when`, `unless`, `cond`, `maptree`, `maplist`, `every`, `some`, `progn`, `progn1`, `let`, `let*`, `lambda`, `with-nrql-settings`. Note that functions like `maptree` take lambda bodies as arguments; however, lambda expressions are not first-order in MiniLisp in order to grant termination.

These head projection operators are available as functions: `describe-ind` instantiators `most-specific-instantiators` `retrieve-individual-synonyms`.

Some type conversion functions: `to-number` `to-string` `to-symbol`.

There are some predefined sorting functions: `sort-string-greaterp` `sort-string-lessp` `sort-string<` `sort-string>` `sort-symbol-name-greaterp` `sort-symbol-name-lessp` `sort<` `sort>`.

The following functions are borrowed from Common Lisp and work as expected:

```
* + - / 1+ 1- < <= = > >= append asin asinh atan atanh ceiling concat cons consp
cos cosh eighth ensure-list eq eql equal equalp evenp expt fifth find first
flatten float floor format fourth intersection length list listp log max member
min minusp ninth nth nth null numberp oddp plusp position princ rationalize
remove rest reverse round search second set-difference set-equal set-subset
seventh signum sin sinh sixth sqrt string-capitalize string-downcase string-equal
string-greaterp string-lessp string-upcase string< string<= string= string>
string>= stringp symbol-name symbolp tan tanh tenth terpri third tree-equal
type-of union with-open-output-file write zerop
```

6.1.3.2 MiniLisp Head-Projection Operators

We have already discussed the told value and attribute retrieval head-projection operators. nRQL offers some more head projection operators which are implemented in MiniLisp. The following head projections operators are implemented in MiniLisp and can be called MiniLisp macro head-projection operators: `:all-types` (also: `:types` `:instantiators` `:all-instantiators`), `:all-types-flat` (also: `:all-instantiators-flat` `:types-flat` `:instantiators-flat`), and `:direct-types` (also: `:most-specific-types` `:most-specific-instantiators` `:direct-instantiators`), `:direct-types-flat` (also: `:most-specific-types-flat` `:most-specific-instantiators-flat` `:direct-instantiators-flat`), and `:describe` as well as `:individual-synonyms`.

Here are some examples illustrating these macro operators:

```
? (full-reset)
> :okay-full-reset

? (instance i c)
> :OKAY

? (implies c d)
> :OKAY

? (retrieve (?x (describe ?x)) (?x top))
> (((?x i)
  ((:lambda (?x) (describe-ind ?x *current-abox*)) ?x)
  (i
   :assertions
   ((i c))
   :role-fillers
   NIL
   :told-attribute-fillers
   NIL
   :told-datatype-fillers
   NIL
   :annotation-datatype-property-fillers
   NIL
```

```

      :annotation-property-fillers
      NIL
      :direct-types
      :to-be-computed))))

? (describe-query :last)
> (:query-1
   (:accurate :processed)
   (retrieve
    (?x ((:lambda (?x) (describe-ind ?x *current-abox*)) ?x))
    (?x top)
    :abox
    default))

? (retrieve (?x (types ?x)) (?x top))
> (((?x i)
     (((:lambda (?x) (instantiators ?x *current-abox*)) ?x)
      ((c) (d) (*top* top)))))

? (retrieve (?x (all-types ?x)) (?x top))
> (((?x i)
     (((:lambda (?x) (instantiators ?x *current-abox*)) ?x)
      ((c) (d) (*top* top)))))

? (retrieve (?x (all-types-flat ?x)) (?x top))
> (((?x i)
     (((:lambda
          (?x)
          (sort-symbol-name-lessp
           (flatten (instantiators ?x *current-abox*))))
          ?x)
      ((c d *top* top)))))

```

Please note that the `retrieve` macro can be advised to not include the lambda expression in the query answer:

```

? (retrieve (?x (all-types-flat ?x)) (?x top) :dont-show-lambdas-p t)
> (((?x i) ((c d *top* top))))

```

It should be noted that the lambda expression can also be used in rule antecedences, see below.

6.1.3.3 MiniLisp Examples

We present four examples which make MiniLisp more concrete and demonstrate its usefulness.

6.1.3.3.1 Aggregation Operators First we show how queries with *aggregation operators* can be implemented. Consider the following KB which models the compositional structure of a car. A car has certain parts, and each part has a certain weight (see file `lambda-ex2.racer`):

```
(full-reset)
(define-primitive-role has-part :transitive t)
(define-concrete-domain-attribute weight :type real)

(instance car1 car)
(related car1 engine1 has-part)
(related engine1 cylinder-1-4 has-part)
(related car1 wheel-1-4 has-part)
(related car1 chassis1 has-part)
(instance engine1 (= weight 200.0))
(instance chassis1 (= weight 400.0))
(instance wheel-1-4 (= weight 30.0))
```

Using MiniLisp, we can compute the overall weight as well as identify its number of components (see file `nrql-user-guide-examples/lambda-ex2-query1.racer`):

```
? (retrieve1 (?car car)
  (((lambda (car)
    (let ((w
      (reduce '+
        (flatten
          (retrieve1 '(and (,car car)
            (,car ?part has-part)
            (?part (a weight)))
          '(((lambda (weight) weight)
            (told-value-if-exists
              (weight ?part)))))))
      (parts (length (retrieve '(?part) '(,car ?part has-part))))
      '(((?car ,car) (?no-of-parts ,parts) (?total-weight ,w))))
    ?car)))
  > ((((?car car1) (?no-of-parts 4) (?total-weight 630.0))))
```

Please note that `retrieve1` is like `retrieve`, but with head and body argument positions flipped. The body of the query consists of the concept query atom `(?x car)`. The `lambda` expression is then applied to the current binding of `?x`. So, within the `lambda`, `car` is bound to the bindings / value of `?x`. First, the total weight is computed: for this purpose, a subquery is constructed. If `?x = car1`, then the query `(retrieve1 '(and (car1 car) (car1 ?part has-part) (?part (a weight))) ...)` is constructed and posed, asking for the parts of `car1`. The head of the subquery consists of yet another `lambda`, which simply applies the `told-value-if-exists` head projection operator to retrieve the told values of the `weight` attribute of `?part`. The subquery result is returned as a nested list; the list is then flattened, and its items are summed using `(reduce '+ ...)`. The result is bound to the local variable

w. Similarly, the number of `parts` is computed (by posing yet another subquery). Finally, the result of the `lambda` expression is constructed and returned. The constructed and returned value will become the result tuple. So, if `car` is `car1`, and `no-of-parts` is 4, and `w` is 630.0, then the template `'((?car ,car) (?no-of-parts ,parts) (?total-weight ,w))` constructs the result tuple `((?car car1) (?no-of-parts 4) (?total-weight 630.0))`.

6.1.3.3.2 Efficient Aggregation Operators using Promises Although the previous query demonstrated the power and utility of MiniLisp, the aggregation is not computed efficiently. The reason is that for each binding of `?car`, two new subqueries are constructed. Each query has to be parsed, compiled, and is then maintained as a query object. Since the structure of the subqueries does not change during query execution it would be better to construct these subqueries in advance. This can be achieved using so-called *promises*. During the time of query preparation, a promise for a variable declares that this variable is treated as an individual. Moreover, it is promised that at the time when the query is executed, that variable is indeed bound to an individual.

Thus, a more efficient version looks as follows (see file `nrql-user-guide-examples/lambda-ex2-query2.racer`):

```
? (with-future-bindings (?car)
  (delete-all-queries)
  (prepare-abox-query (?part) (and (?car car) (?car ?part has-part))
    :dont-show-lambdas t
    :id :parts-of-car)

  (prepare-abox-query (((lambda (weight) weight)
    (told-value-if-exists (weight ?part))))
    (and (?car car) (?car ?part has-part) (?part (a weight)))
    :dont-show-lambdas t
    :id :weights-of-parts-of-car))
> ...
```

This has defined two queries named `:parts-of-car` and `:weights-of-parts-of-car`. The query optimizer has treated the `?car` variable as an individual due to `with-future-bindings`. Thus, we have promised nRQL that we will only execute these queries if we supply a binding for `agg`. We can establish such a binding during query execution using `with-nrql-settings` as follows:

```
? (retrieve1 (?car car)
  ( (lambda (car)
    (with-nrql-settings (:dont-show-lambdas t
      :bindings '((?car ,car)))

      (let ((w
        (reduce '+
          (flatten
            (execute-or-reexecute-query :weights-of-parts-of-car))))
```

```

      (parts (length
              (execute-or-reexecute-query :parts-of-car))))

      '(((?car ,car) (?total-weight ,w))))
    ?car)))
> (((((?car car1) (?no-of-parts 4) (?total-weight 630.0))))))

```

6.1.3.3.3 User Defined Query Result Format As already demonstrated, the value returned by a `lambda` body is included in the binding list. In the previous query, the `lambda` body returns a structured list using the template `'((?car ,car) (?total-weight ,w))`; this is equivalent to `(list (list '?car car) (list '?total-weight w))`. It is also easy to specify natural language output; simply replace

```

      '(((?car ,car) (?total-weight ,w))
        with
      (format nil "Car with name  A weights  A kg and has  A parts." car w parts)
        in the previous query, and you will get
      ((Car with name car1 weights 630.0d0 kg and has 4 parts.))

```

as the query result.

6.1.3.3.4 File Output MiniLisp also offers the `with-open-output-file` which allows to open an output file. Simply print to the stream `*output-stream*` (which is established by `with-open-output-file`) to add arbitrary content to the output file.

```

? (retrieve1 (?car car)
  (((lambda (car)
    (with-open-output-file ("minilisp-output.txt")
      (let ((w
        (reduce '+
          (flatten
            (retrieve1 '(and (,car car)
              (,car ?part has-part)
              (?part (a weight))))
          '(((lambda (weight) weight)
            (told-value-if-exists
              (weight ?part))))))))))
    (parts
      (length
        (retrieve ' (?part) '(,car ?part has-part))))))
    (format *output-stream*
      "Car with name ~A weights ~A kg and has ~A parts.~%"
      car w parts))))
  ?car)))
> ((NIL))

```


The output can be found in the file `minilisp-output.txt`; moreover, the result `((NIL))` is delivered because the `lambda` body returns `(NIL)`. Please note that `with-open-output-file` only works if `RacerPro` is running in unsafe mode.

6.1.3.3.5 Filtering Result Tuples `lambda` bodies can also work as *filters* as follows: if the special token `:reject` is returned from the `lambda` body, then the result is rejected, i.e., will not appear in the query answer. For example, using `:reject` we can easily reject all parts which have no subparts:

```
? (retrieve1 (?part top)
      (((lambda (part)
            (let ((parts
                  (length
                    (retrieve '(?part) '(,part ?part has-part))))
              (if (zerop parts) :reject '(,part has ,parts parts))))
          ?part)))
> (((engine1 has 1 parts)) ((car1 has 4 parts)))
```

6.1.3.3.6 Combined TBox / ABox Queries Finally, let us present an example with demonstrates how to combine TBox queries and ABox queries. Sometimes, one wants to retrieve all and only the *direct instances* of a concept / OWL class. An individual is called a direct instance of a concept / OWL class if there is no subconcept / subclass of which the individual is also an instance.

Let us create two concepts `c` and `d` such that `d` is a sub concept (child concept) of `c`:

```
? (full-reset)
> :okay-full-reset

? (define-concept c (some r top))
> :OKAY

? (define-concept d (and c e))
> :OKAY
```

We can verify that `d` is indeed a child concept of `c`, using a so-called TBox query, see below:

```
? (tbox-retrieve (?x) (c ?x has-child))
> (((?x d)))
```

Let us create two individuals so that `i` and `j` are instances of `c`; moreover, `j` is also an instances of `d`:

```
? (related i j r)
> :OKAY
```

```
? (related j k r)
> :OKAY
```

```
? (instance j e)
> :OKAY
```

```
? (retrieve (?x) (?x c))
> (((?x j)) ((?x i)))
```

```
? (retrieve (?x) (?x d))
> (((?x j)))
```

The previous queries demonstrated that both *i* as well as *j* are *c* instances. However, only *i* is a *direct* *c* instance. We can retrieve these direct instances of *c* as follows:

```
? (retrieve1 (?x c)
  ( ( (:lambda (x)
      (if (some (lambda (subclass)
        (retrieve () '(,x ,subclass)))
        (flatten
          (tbox-retrieve1 '(c ?subclass has-child)
            '( ( (:lambda (subclass) subclass) ?subclass))))))
      :reject
      '(?x ,x)))
    ?x)))
> (((?x i)))
```

6.1.4 Complex Queries

After having discussed the available query atoms and the structure of the query head (which may contain head projection operators and/or lambda expressions), let us discuss the structure of the (complex) query bodies. Please note that some of the so far presented example queries already used complex query bodies.

A nRQL query body is inductively defined as being either a single query atom, or a complex query body which is constructed from query bodies with the help of the following *query body constructors*. As usual, a prefix syntax is used:

- **and** is an n -ary constructor which is used for the formulation of conjunctive queries. The arguments of the **and** are called conjuncts. The conjuncts are query bodies.
- **union** is an n -ary constructor. The arguments of the **union** are called disjuncts. The constructor computes the union of the query answers of its disjuncts. The disjuncts are query bodies.
- **neg** is a unary constructor, the *negation as failure (NAF)* negation. The argument is a query body.
- **project-to** is a unary constructor. This is the projection operator for query *bodies* (not to be confused with with *head* projection operators). The argument is a query body; moreover, the constructor also needs a so-called *projection list* as argument. So, the first argument to this constructor is the *projection list*. This list is a list of objects (variables and/or individuals). Head projection operators are not permitted here. The second argument is a query body. Moreover, the set of objects mentioned in the projection list must be a subset of the set of objects mentioned in the query body.

Please look up the EBNF specification to learn more about the compositional syntax of nRQL (see Section 6.1.9).

We will now present and discuss each body constructor.

6.1.4.1 The AND Constructor – Conjunctive Queries

Suppose we want to retrieve all woman having a male child from `family.racer`. This is a classic (grounded) conjunctive query:

```
? (retrieve (?x ?y) (and (?x woman) (?x ?y has-child) (?y man)))
> (((?x alice) (?y charles)))
```

Here is another example query which searches for children having a common mother:

```
? (retrieve
  (?mother ?child1 ?child2)
  (and
```

```

    (?child1 human)
    (?child2 human)
    (neg (same-as ?child1 ?child2))
    (?mother ?child1 has-child)
    (?mother ?child2 has-child)))

> (((?mother betty) (?child1 doris) (?child2 eve))
    ((?mother betty) (?child1 eve) (?child2 doris))
    ((?mother alice) (?child1 betty) (?child2 charles))
    ((?mother alice) (?child1 charles) (?child2 betty)))

```

Note that the atom `(neg (same-as ?child1 ?child2))` prevents that `?child1` and `?child2` are bound to the same individual.

6.1.4.2 The UNION Constructor

nRQL also offers a `union` constructor (also `or` can be used):

```

? (retrieve (?x) (union (?x woman) (?x man)))
> (((?x alice)) ((?x betty)) ((?x eve)) ((?x doris)) ((?x charles)))

```

As the name suggests, `union` simply returns the union of the query answers returned by its argument bodies (disjuncts). A disjunct which references n objects (variables and/or individuals) denotes a set of n -ary tuples. Since the union operator is only meaningful if all disjuncts have the same arity, nRQL ensures that the disjuncts will always have the same arity. This is achieved by adding additional (`<variable> top`) conjuncts to disjuncts:

```

? (retrieve (?x ?y) (union (?x ?y has-child) (?x mother)))
> (((?x alice) (?y eve))
    ((?x alice) (?y doris))
    ((?x alice) (?y charles))
    ((?x alice) (?y betty))
    ((?x alice) (?y alice))
    ((?x betty) (?y eve))
    ((?x betty) (?y doris))
    ((?x betty) (?y charles))
    ((?x betty) (?y betty))
    ((?x betty) (?y alice)))

? (describe-query :last)
> (:query-10
    (:accurate :processed)
    (retrieve
      (?x ?y)
      (or (and (?x mother) (top ?y)) (?x ?y has-child))
    )

```

```
:abox
smith-family))
```

Note that nRQL has rewritten the original query body into `(or (and (?x mother) (top ?y)) (?x ?y has-child))`.

It is important to note that the *variable names matter*. Consider the following query:

```
? (retrieve (?x) (union (?x man) (?y woman)))
> (((?x charles)) ((?x eve)) ((?x doris)) ((?x betty)) ((?x alice)))
```

A common pitfall is to think that this query is equivalent to the query

```
(retrieve (?y) (?y man)).
```

However, it is not, since `?x` and `?y` are different individuals. The disjuncts of the `union` have thus arity 2:

```
? (describe-query :last)
> (:query-11
   (:accurate :processed)
   (retrieve
    (?x)
    (or (and (?x man) (top ?y-ano1)) (and (top ?x) (?y-ano1 woman)))
    :abox
    smith-family))
```

nRQL has rewritten this body into `(union (and (?x man) (top ?y-ano1)) (and (top ?x) (?y-ano1 woman)))`; the variable `?y` has been replaced with `?y-ano1` since `?y` is not mentioned in the query head. Since we are thus only interested in the bindings of `?x` it becomes clear that this query body is in fact equivalent to `(union (?x body) (?x top))`, or `(?x top)`.

Another pitfall is to think that the query

```
(retrieve (?x) (union (?x C) (?x (not C))))
```

is equivalent to

```
(retrieve (?x) (?x (or C (not C)))).
```

However, it is not, since the concept `(or C (not C))` is equivalent to `top`, and thus, all individuals will be returned. However, the `union` query returns only those individuals for which either `C` or `(not C)` can be proven. Due to the *Open World Semantics (OWA)* there can be individuals which are neither instances of `C` nor of its complement `(not C)`, but every individual is of course an instance of `top`:

```

? (full-reset)
> :okay-full-reset

? (instance i c)
> :OKAY

? (instance j (not c))
> :OKAY

? (instance k top)
> :OKAY

? (retrieve (?x) (union (?x c) (?x (not c))))
> (((?x i)) ((?x j)))

? (retrieve (?x) (?x (or c (not c))))
> (((?x k)) ((?x j)) ((?x i)))

```

6.1.4.3 NEG – The Negation As Failure Constructor

A NEG constructor is provided which provides *Negation as Failure*, short: *NAF* negation.

NAF is especially useful for measuring the completeness of the modeling in a KB. Many users are probably familiar with logic programming language the PROLOG, which also offers NAF.

NAF is quite different from classical "true" negation. The **neg** operator can be applied to an arbitrary query body.

6.1.4.3.1 Negation as Failure for Concept Query Atoms NAF is quite different from true negation. Consider the following example:

```

? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

? (retrieve (?x) (?x grandmother))
> (((?x alice)))

? (retrieve (?x) (neg (?x grandmother)))
> (((?x eve)) ((?x doris)) ((?x charles)) ((?x betty)))

```

RacerPro can prove that **alice** is an instance of **grandmother**. Thus, for the other present individual, it *cannot prove* membership in **grandmother**. In order to retrieve the individuals for which RacerPro cannot prove membership, the **neg** operator is provided: We can

simply retrieve these individuals with the NAF-negated concept query atom (`neg (?x grandmother)`). This atom simply returns the complement set of `(?x grandmother)`. For any concept `C`, the body `(union (?x C) (neg (?x C)))` is equivalent to `(?x top)`:

```
? (retrieve (?x) (union (?x grandmother) (neg (?x grandmother))))
> (((?x alice)) (?x eve)) ((?x doris)) ((?x charles)) ((?x betty)))

? (retrieve (?x) (?x top))
> (((?x alice)) (?x betty)) ((?x charles)) ((?x doris)) ((?x eve)))
```

Note that classical negation is of course also available, since RacerPro offers concept negation in its concept language:

```
? (retrieve (?x) (?x (not grandmother)))
> (((?x charles)))
```

RacerPro is able to prove that `charles` is an instance of `(not grandmother)`, since `charles` is a man, and grandmothers are woman. Moreover, man and woman are disjoint. Note that, for any concept `C`, the atom `(?x (not C))` always returns a subset of `(neg (?x C))`.

Thus, as already mentioned, the following query is *not* equivalent to `(?x top)`:

```
(retrieve (?x) (union (?x grandmother) (?x (not grandmother))))
> (((?x alice)) (?x charles)))
```

It even possible to combine classical with NAF negation in a query:

```
? (retrieve (?x) (neg (?x (not grandmother))))
> (((?x eve)) (?x doris)) ((?x betty)) ((?x alice)))
```

So, we have asked for those individuals for which RacerPro cannot prove that they are instances of `(not grandmother)`. Note that this is again the complement of

```
? (retrieve (?x) (?x (not grandmother)))
> (((?x charles)))
```

6.1.4.3.2 Negation as Failure for Unary Atoms with Individuals A unary atom and its NAF-negated variant are always complementary. This also holds if the atom references an individual and not a variable:

```
? (retrieve (?x) (?x grandmother))
> (((?x alice)))

? (retrieve (?x) (neg (?x grandmother)))
> (((?x eve)) (?x doris)) ((?x charles)) ((?x betty)))
```

We have already discussed that individuals in queries are in fact replaced by representative variables; e.g., `alice` is replaced with `?alice`, and a `(same-as ?alice alice)` conjunct is added. We call this the *standard transformation* in the following. This allows convenient NAF negation, but note that in the NAF-negated variant, `alice` behaves like a variable:

```
? (retrieve (alice) (alice grandmother))
> (((?alice alice)))

? (describe-query :last)
> (:query-7
   (:accurate :processed)
   (retrieve
    (?alice)
    (and (same-as ?alice alice) (?alice grandmother))
    :abox
    smith-family))

? (retrieve (alice) (neg (alice grandmother)))
> (((?alice eve))
   ((?alice doris))
   ((?alice charles))
   ((?alice betty)))

? (describe-query :last)
> (:query-8
   (:accurate :processed)
   (retrieve
    (?alice)
    (or
     (not (same-as ?alice alice))
     (and (not (?alice grandmother)) (top alice)))
    :abox
    smith-family))
```

However, **if this behavior is unwanted** for `(neg (alice grandmother))`, then simply ensure that no individuals appear in NAF-negated atoms. Use only variables in NAF-negated atoms. Additional `same-as` and NAF-negated `same-as` atoms may be added to further constrain the bindings of variables to certain individuals, or to rule out undesired bindings, for example:

```
? (retrieve (?x) (and (?x grandmother) (same-as ?x alice)))
> (((?x alice)))

? (describe-query :last)
> (:query-24
   (:accurate :processed)
```



```

(retrieve
  (?x)
  (and (same-as ?x alice) (?x grandmother))
  :abox
  smith-family))

? (retrieve (?x) (and (neg (?x grandmother)) (same-as ?x alice)))
> NIL

? (describe-query :last)
> (:query-25
   (:accurate :processed)
   (retrieve
    (?x)
    (and (same-as ?x alice) (not (?x grandmother)))
    :abox
    smith-family))

```

Also the `bind-individual` atom can be used, which is syntactic sugar:

```

? (retrieve
   (alice)
   (and (neg (alice grandmother)) (bind-individual alice)))
> NIL

? (describe-query :last)
> (:query-10
   (:accurate :processed)
   (retrieve
    (?alice)
    (or
     (and (same-as ?alice alice) (not (same-as ?alice alice)))
     (and
      (same-as ?alice alice)
      (not (?alice grandmother))
      (top alice))))
   :abox
   smith-family))

```

Please note that NAF negation can also be tricky in the presence of individual synonyms, since the standard individual transformation uses `same-as` atoms, and `same-as` works on a syntactic level (unless you turn on the `(use-individual-synonym-equivalence-classes)` mode). This means that synonym individuals will not be recognized as such:

```

? (same-as alice grandmother)

```

```

> :OKAY

? (retrieve (alice) (alice grandmother))
> (((?alice alice)))

? (retrieve (alice) (neg (alice grandmother)))
> (((?alice grandmother))
    ((?alice eve))
    ((?alice doris))
    ((?alice charles))
    ((?alice betty)))

? (describe-query :last)
> (:query-69
   (:accurate :processed)
   (retrieve
    (?alice)
    (or
     (not (same-as ?alice alice))
     (and (not (?alice grandmother)) (top alice)))
   :abox
   smith-family))

```

So, `grandmother` is returned here, even though `alice` and `grandmother` are synonyms. Due to the standard individual transformation, the query `(alice grandmother)` is rewritten into `(and (?alice grandmother) (same-as ?alice alice))`. Thus, `(neg (alice grandmother))` was consequently rewritten into `(union (?alice grandmother) (neg (same-as ?alice alice)))`, as shown by the result of `(describe-query :last)` (as mentioned, `not` can be used for `neg`, and `or` for `union`). Since `(neg (same-as ?alice alice))` holds for the binding `?alice=grandmother`, this binding is included in the answer.

What can be done to exclude this binding? We can simply replace the automatically added `same-as` atom with a role query atom referencing the `nrql-equal-role` (alternatively, we could also turn on the `semantic same-as interpretation` using the `(nrql-symbols:use-individual-synonym-equivalence-classes)` directive):

```

? (retrieve
   (?alice)
   (or
    (not (?alice alice nrql-equal-role))
    (not (?alice grandmother))))
> (((?alice eve))
    ((?alice doris))
    ((?alice charles))
    ((?alice betty)))

```

Please note that individuals in role query atoms which use the `nrql-equal-role` will never be replaced with their representative variables.

Moreover, we can also turn on the semantic `same-as` interpretation as follows:

```
? (use-individual-synonym-equivalence-classes)
> :okay-using-individual-equivalence-classes

? (retrieve (alice (individual-synonyms alice))
  (alice grandmother)
  :dont-show-lambdas-p t)
> (((?alice alice) (alice grandmother)))

? (retrieve
  (alice (individual-synonyms alice))
  (neg (alice grandmother))
  :dont-show-lambdas-p t)
> (((?alice eve) (eve))
  ((?alice doris) (doris))
  ((?alice charles) (charles))
  ((?alice betty) (betty)))
```

Note that this rule does not hold for binary atoms, see below.

6.1.4.3.3 Negation as Failure for Role Query Atoms The `neg` operator can be applied to role query atoms as well. Like for concept query atoms, a role query atom and its NAF-negated variant are complementary:

```
? (retrieve (?x ?y) (?x ?y has-child))
> (((?x betty) (?y doris))
  ((?x betty) (?y eve))
  ((?x alice) (?y betty))
  ((?x alice) (?y charles)))

? (retrieve (?x ?y) (neg (?x ?y has-child)))
> (((?x eve) (?y eve))
  ((?x eve) (?y doris))
  ((?x eve) (?y charles))
  ((?x eve) (?y betty))
  ((?x eve) (?y alice))
  ((?x doris) (?y eve))
  ((?x doris) (?y doris))
  ((?x doris) (?y charles))
  ((?x doris) (?y betty))
  ((?x doris) (?y alice))
  ((?x charles) (?y eve)))
```

```

((?x charles) (?y doris))
((?x charles) (?y charles))
((?x charles) (?y betty))
((?x charles) (?y alice))
((?x betty) (?y charles))
((?x betty) (?y betty))
((?x betty) (?y alice))
((?x alice) (?y eve))
((?x alice) (?y doris))
((?x alice) (?y alice)))

```

Note that $4+21=25=5*5$ (and we have 5 individuals in that KB).

Suppose we are now looking for people *without known children*. So we try:

```

? (retrieve (?x) (neg (?x ?y has-child)))
> (((?x eve)) ((?x doris)) ((?x charles)) ((?x betty)) ((?x alice)))

```

However, we already know that only **betty** and **alice** have children in the ABox:

```

? (retrieve (?x) (?x ?y has-child))
> (((?x betty)) ((?x alice)))

```

In fact we were looking for the answer `(((?x charles)) ((?x eve)) ((?x doris)))`. Didn't we just say that `(neg <atom>)` always returns the complement set of `<atom>`? So what went wrong? In fact, internally the correct complement set is constructed. However, these two-dimensional sets are no longer complementary to one another after the projection to `?x` has been performed. So, `(neg (?x ?y has-child))` returns the set of 21 pairs shown in the query `(retrieve (?x ?y) (neg (?x ?y has-child)))` above. The reader can verify that we get **eve**, **doris**, **charles**, **betty**, **alice** if this set is projected to its first components.

If we want to ensure that also the query results sets constructed by the final projection (the projection to `?x` as specified by the query head) of these queries are complementary, we must make sure that the complement operator is applied to a set of instances, and not to a set of pairs. The `project-to` operator is provided for this purpose (see below):

```

? (retrieve (?x) (neg (project-to (?x) (?x ?y has-child))))
> (((?x eve)) ((?x doris)) ((?x charles)))

```

This query now solves the specified retrieval task (retrieve all people without known children).

Analog as for concept query atoms, it is the case that `(?x ?y (not R))` always returns a subset of `(neg (?x ?y R))`, for an arbitrary role `R`. Moreover, classical negation and NAF negation can be mixed, like for concept query atoms:

```
? (retrieve (?x ?y) (neg (?x ?y (not has-sister))))
> (((?x eve) (?y eve))
    ((?x eve) (?y doris))
    ((?x eve) (?y betty))
    ((?x eve) (?y alice))
    ((?x doris) (?y eve))
    ((?x doris) (?y doris))
    ((?x doris) (?y betty))
    ((?x doris) (?y alice))
    ((?x charles) (?y eve))
    ((?x charles) (?y doris))
    ((?x charles) (?y betty))
    ((?x charles) (?y alice))
    ((?x betty) (?y eve))
    ((?x betty) (?y doris))
    ((?x betty) (?y betty))
    ((?x betty) (?y alice))
    ((?x alice) (?y eve))
    ((?x alice) (?y doris))
    ((?x alice) (?y betty))
    ((?x alice) (?y alice)))
```

Note that the atom without NAF returns 5 tuples (recall that `charles` cannot be a sister, since he is male):

```
? (retrieve (?x ?y) (?x ?y (not has-sister)))
> (((?x eve) (?y charles))
    ((?x doris) (?y charles))
    ((?x charles) (?y charles))
    ((?x betty) (?y charles))
    ((?x alice) (?y charles)))
```

Thus, we get our $20+5=25=5*5$ pairs again.

6.1.4.3.4 Negation as Failure for Constraint Query Atoms Also constraint query atoms can be NAF negated. However, for simple CD predicates, NAF negation and classical negation coincide, since the CDs in RacerPro are *complete theories*; thus, for two concrete domain objects i, j , either $\models P(i, j)$ or $\models \neg P(i, j)$ holds. This is demonstrated using the `=` CD predicate; please note that for each CD predicate, a negated predicate is available, in this case, the negated predicate for `=` is `<>`:

```
? (retrieve (?x ?y) (neg (?x ?y (constraint age age =))))
> (((?x eve) (?y doris))
    ((?x eve) (?y charles))
    ((?x eve) (?y betty)))
```

```

((?x eve) (?y alice))
((?x doris) (?y eve))
((?x doris) (?y charles))
((?x doris) (?y betty))
((?x doris) (?y alice))
((?x charles) (?y eve))
((?x charles) (?y doris))
((?x charles) (?y betty))
((?x charles) (?y alice))
((?x betty) (?y eve))
((?x betty) (?y doris))
((?x betty) (?y charles))
((?x betty) (?y alice))
((?x alice) (?y eve))
((?x alice) (?y doris))
((?x alice) (?y charles))
((?x alice) (?y betty)))

? (retrieve (?x ?y) (?x ?y (constraint age age <>)))
> (((?x alice) (?y doris))
  ((?x alice) (?y eve))
  ((?x alice) (?y betty))
  ((?x alice) (?y charles))
  ((?x doris) (?y alice))
  ((?x doris) (?y eve))
  ((?x doris) (?y betty))
  ((?x doris) (?y charles))
  ((?x eve) (?y alice))
  ((?x eve) (?y doris))
  ((?x eve) (?y betty))
  ((?x eve) (?y charles))
  ((?x betty) (?y alice))
  ((?x betty) (?y doris))
  ((?x betty) (?y eve))
  ((?x betty) (?y charles))
  ((?x charles) (?y alice))
  ((?x charles) (?y doris))
  ((?x charles) (?y eve))
  ((?x charles) (?y betty)))

```

Thus, both queries return the same answer.

However, the situation changes if *role chains* are used within constraint query atoms; in this case, the complement can no longer be retrieved by simply negating the CD predicate, since additional (implicit) constraints resp. retrieval conditions on the bindings are imposed by the role chains. Again, the semantics of **neg** is defined in such a way that a constraint query atom and its negated variant are always complementary:

```

? (retrieve
  (?x ?y)
  (?x ?y (constraint (has-child age) (has-child age) =)))
> (((?x betty) (?y betty)) ((?x alice) (?y alice)))

? (retrieve
  (?x ?y)
  (neg (?x ?y (constraint (has-child age) (has-child age) =))))
> (((?x eve) (?y eve))
  ((?x eve) (?y doris))
  ((?x eve) (?y charles))
  ((?x eve) (?y betty))
  ((?x eve) (?y alice))
  ((?x doris) (?y eve))
  ((?x doris) (?y doris))
  ((?x doris) (?y charles))
  ((?x doris) (?y betty))
  ((?x doris) (?y alice))
  ((?x charles) (?y eve))
  ((?x charles) (?y doris))
  ((?x charles) (?y charles))
  ((?x charles) (?y betty))
  ((?x charles) (?y alice))
  ((?x betty) (?y eve))
  ((?x betty) (?y doris))
  ((?x betty) (?y charles))
  ((?x betty) (?y alice))
  ((?x alice) (?y eve))
  ((?x alice) (?y doris))
  ((?x alice) (?y charles))
  ((?x alice) (?y betty)))

```

Thus, together we have our $2+23=25=5*5$ pairs again. But note that, as just explained, the following query doesn't return 23 pairs:

```

? (retrieve
  (?x ?y)
  (?x ?y (constraint (has-child age) (has-child age) <>)))
> (((?x betty) (?y betty))
  ((?x betty) (?y alice))
  ((?x alice) (?y betty))
  ((?x alice) (?y alice)))

```

6.1.4.3.5 Negated SAME-AS Query Atoms (NAF) A negated SAME-AS query atom simply enumerates the complement of its positive variant:

```
? (retrieve (?x) (same-as ?x alice))
> (((?x alice)))

? (retrieve (?x) (neg (same-as ?x alice)))
> (((?x doris)) ((?x charles)) ((?x betty)) ((?x eve)))
```

Note that `same-as` works on a syntactic level unless you turn on `(use-individual-synonym-equivalence-classes)` (see also the output of `(describe-query-processing-mode)`).

A semantic equality predicate is given by role query atoms using the `nrql-equal-role`.

6.1.4.3.6 Negation as Failure for Binary Atoms with Individuals Both role query atoms, `same-as` query atoms, and `constraint` query atoms are binary atoms. In principle, a binary atom and its NAF-negated variant are complementary. However, since these atoms reference two objects, the effect of the final projection must be taken into account. Moreover,

```
? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

? (retrieve (alice ?x) (alice ?x has-child))
> (((?alice alice) (?x betty)) ((?alice alice) (?x charles)))

? (retrieve (alice ?x) (neg (alice ?x has-child)))
> (((?alice eve) (?x eve))
  ((?alice eve) (?x doris))
  ((?alice eve) (?x charles))
  ((?alice eve) (?x betty))
  ((?alice eve) (?x alice))
  ((?alice doris) (?x eve))
  ((?alice doris) (?x doris))
  ((?alice doris) (?x charles))
  ((?alice doris) (?x betty))
  ((?alice doris) (?x alice))
  ((?alice charles) (?x eve))
  ((?alice charles) (?x doris))
  ((?alice charles) (?x charles))
  ((?alice charles) (?x betty))
  ((?alice charles) (?x alice))
  ((?alice betty) (?x eve))
  ((?alice betty) (?x doris))
  ((?alice betty) (?x charles)))
```



```
((?alice betty) (?x betty))
((?alice betty) (?x alice))
((?alice alice) (?x eve))
((?alice alice) (?x doris))
((?alice alice) (?x alice)))
```

These are $2+23=25=5*5$ pairs. Please note that the sets returned by the atom and its negated variant are complementary; however, if a projection is performed on a set, then the sets resulting from the projection are *not* necessarily complementary to one another. A projection is either specified explicitly by means of the **project-to** operator, or implicitly, as specified by the query head:

```
? (retrieve (?x) (alice ?x has-child))
> (((?x charles)) ((?x betty)))

? (retrieve (?x) (neg (alice ?x has-child)))
> (((?x eve)) ((?x doris)) ((?x charles)) ((?x betty)) ((?x alice)))
```

This seems odd, since also the children of **alice** are returned. However, the result is correct, since this is just the answer returned by `(retrieve (alice ?x) (neg (alice ?x has-child)))` projected to **?x**.

So, to retrieve the individuals which are not children of **alice**, the following query must be used:

```
? (retrieve (?x) (neg (project-to (?x) (alice ?x has-child))))
> (((?x eve)) ((?x doris)) ((?x alice)))
```

Note that here the **neg** operator is applied *after* the projection to **?x** has been carried out, so **neg** is applied to (and returned) a one-dimensional set, whereas in the pervious query, **neg** was applied to (and returned) a two-dimensional set. This is easy to verify using **describe-query** again:

```
? (retrieve (?x) (alice ?x has-child))
> (((?x charles)) ((?x betty)))

? (describe-query :last)
> (:query-7
   (:accurate :processed)
   (retrieve
    (?x)
    (and (same-as ?alice-ano1 alice) (?alice-ano1 ?x has-child))
    :abox
    smith-family))

? (retrieve (?x) (neg (alice ?x has-child)))
```

```

> (((?x eve)) ((?x doris)) ((?x charles)) ((?x betty)) ((?x alice)))

? (describe-query :last)
> (:query-8
   (:accurate :processed)
   (retrieve
    (?x)
    (or
     (and (not (same-as ?alice-ano1 alice)) (top ?x))
     (and (top alice) (not (?alice-ano1 ?x has-child)))))
   :abox
   smith-family))

```

Moreover, regarding individuals in query heads, we have the same situation as already discussed for the unary atoms: they are replaced with representative variables and **same-as** conjuncts. This is the standard transformation. Negated individuals thus behave like variables:

```

? (retrieve (alice) (alice ?x has-child))
> (((?alice alice)))

? (retrieve (alice) (neg (alice ?x has-child)))
> (((?alice eve))
   ((?alice doris))
   ((?alice charles))
   ((?alice betty))
   ((?alice alice)))

```

Again, if this behavior is not desired, simply avoid using individuals in NAF-negated atoms, and add **same-as** and/or **nrql-equal-role** role query atoms manually, e.g. we can ask if there is anyone who is not a child of alice:

```

? (retrieve
   (alice)
   (and (neg (?alice ?x has-child)) (same-as ?alice alice)))
> (((?alice alice)))

```

However, everyone is a descendant of alice:

```

? (retrieve
   (alice ?x)
   (and
    (neg (?alice ?x has-descendant))
    (same-as ?alice alice)
    (neg (?x alice nrql-equal-role))))
> NIL

```

6.1.4.4 A Note on Boolean Complex Queries

So far we have introduced the **and**, **union** and **neg** query constructors. nRQL allows for the orthogonal composition of *boolean* query bodies.

For processing purposes, nRQL transforms queries into *Negation Normal form (NNF)*. In a NNF query, the **neg** operator appears only in front of query *atoms*.

Moreover, for optimization purposes, the queries are then brought into *Disjunctive Normal Form (DNF)*. This transformation might result in an exponential blowup in query size. We would like to inform the user of this potential performance pitfall.

The semantics of the original query is of courses preserved by the transformations.

6.1.4.5 PROJECT-TO – The Projection Operator for Query Bodies

The projection operator is needed in combination with **neg**. We have already discussed some example in the previous sections which made use of the projection operator. Why is the operator needed? The NAF-negation operator **neg** is designed to compute the *n*-dimensional complement set of its *n*-dimensional input argument set. Thus, **neg** preserves the arity of its argument. However, in some cases, the dimensionality of the argument shall be reduced before **neg** is applied. This is what **project-to** achieves.

For example, suppose we are looking for the woman which have a male child; this is easy:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

? (retrieve (?x) (and (?x woman) (?x ?y has-child) (?y man)))
> (((?x alice)))
```

But how can we retrieve the complement set, i.e., the woman which do not have a male child? In a first attempt we try:

```
? (retrieve (?x) (neg (and (?x woman) (?x ?y has-child) (?y man))))
> (((?x charles)) ((?x eve)) ((?x doris)) ((?x betty)) ((?x alice)))
```

So, why is **alice** included in this set= If we also put **?y** in the query head, it becomes clear what happened:

```
? (retrieve
  (?x ?y)
  (neg (and (?x woman) (?x ?y has-child) (?y man))))
> (((?x charles) (?y eve))
  ((?x charles) (?y doris)))
```

```

((?x charles) (?y charles))
((?x charles) (?y betty))
((?x charles) (?y alice))
((?x eve) (?y eve))
((?x eve) (?y doris))
((?x eve) (?y charles))
((?x eve) (?y betty))
((?x eve) (?y alice))
((?x doris) (?y eve))
((?x doris) (?y doris))
((?x doris) (?y charles))
((?x doris) (?y betty))
((?x doris) (?y alice))
((?x betty) (?y charles))
((?x betty) (?y betty))
((?x betty) (?y alice))
((?x alice) (?y eve))
((?x alice) (?y doris))
((?x alice) (?y alice))
((?x betty) (?y eve))
((?x betty) (?y doris))
((?x alice) (?y betty))

```

These are 24 tuples; only `((?x alice) (?y charles))` is missing. Indeed, this tuple is returned by the non-negated variant of this query, if we also add `?y` to the head:

```

? (retrieve (?x ?y) (and (?x woman) (?x ?y has-child) (?y man)))
> (((?x alice) (?y charles)))

```

This shows that the two queries are indeed complementary to one another, but due to the final projection to `?x`, this no longer holds for the returned sets. The solution is to *first* apply the projection to `?x`, and then build a one-dimensional complement set with `neg` as follows:

```

? (retrieve
  (?x)
  (neg
    (project-to (?x) (and (?x woman) (?x ?y has-child) (?y man)))))
> (((?x eve)) ((?x doris)) ((?x charles)) ((?x betty)))

```

Please note that a `project-to` body can also be understood as a kind of “subquery”. In fact, `(project-to (?x) (and (?x woman) (?x ?y has-child) (?y man)))` can be understood as a subquery `(retrieve (?x) (and (?x woman) (?x ?y has-child) (?y man)))`. Only after the complete result set of that subquery has been computed, the complement can be computed with `neg`.

Please note that `neg` and `project-to` really add some expressive power to nRQL. For example, it is possible to retrieve the oldest person from `family.racer` as follows. How shall we formulate such a query? Obviously, a person is the oldest person if there is no other person that is older. This intuition gives us the following query:

```
? (retrieve
  (?x)
  (and
    (?x person)
    (?x (an age))
    (neg
      (project-to
        (?x)
        (and
          (?y person)
          (?y (an age))
          (?x ?y (constraint age age <)))))))

> (((?x alice)))
```

Please note that the following query does *not* work, since only the variables mentioned in the projection list of `project-to` are “shared” between queries:

```
? (retrieve
  (?x)
  (and
    (?x person)
    (?x (an age))
    (neg
      (project-to
        (?y)
        (and
          (?y person)
          (?y (an age))
          (?x ?y (constraint age age <)))))))

> (((?x alice)) ((?x doris)) ((?x eve)) ((?x betty)) ((?x charles)))
```

The reason is that `?y` is not used in the surrounding query, and `?x` is not shared either, since it is not mentioned in the projection list after `project-to`. Using `describe-query` it is to verify that the query body within `project-to` shares no variables with its surrounding query:

```
? (describe-query :last)
> (:query-3
  (:accurate :processed)
```

```

(retrieve
  (?x)
  (and
    (?x person)
    (?x (an age))
    (not
      (:project-to
        (?y-ano2)
        (and
          (?y-ano2 person)
          (?y-ano2 (an age))
          (?x-ano1-ano3 ?y-ano2 (:constraint age age <))))))
    :abox
    smith-family))

```

Instead of `project-to`, also `project` and `pi` can be used as keywords.

Please note that `neg` in combination with `project-to` allows to formulate some kind of *closed world / closed domain universal quantification*, as just demonstrated: Since there *exists no* (know) person that is older than `?x`, we know that *all (known) persons* are younger than `?x`. On the other hand, this universal quantification only considers the *known* individuals. Not that this query would be impossible to formulate without the NAF-negation in nRQL, since DLs (and RacerPro) use the OWA in general.

We already mentioned that the atom `(has-known-successor R)` is in syntactic sugar for `(project-to (?x) (?x ?y R))`, and thus, `(neg (has-known-successor R))` is equivalent to `(neg (project-to (?x) (?x ?y R)))`.

6.1.5 Defined Queries

nRQL offers a simple macro mechanism for the specification of *defined queries*. A defined query is just a named query; its name can then subsequently be used in other queries, so its definition is inserted at the place it is used (macro expansion):

```
? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

? (defquery mother-of (?x ?y) (and (?x woman) (?x ?y has-child)))
> mother-of

? (describe-definition mother-of)
> (defquery mother-of (?x ?y) (and (?x woman) (?x ?y has-child)))

? (all-queries)
> NIL
```

Note that a defined query is not created as a query object at definition time; thus, (all-queries) returns NIL.

The list (*?x ?y*) in the **mother-of** definition is called the list of *formal arguments* of the definition; it is comparable to a query head; however, it differs from a query head because only objects (variables and individuals) are permitted (no head projection operators). The same restriction applies to the object list of the **project-to** constructor.

The defined query named **mother-of** can subsequently be used as follows:

```
? (retrieve (?x ?y) (?x ?y mother-of))
> (((?x alice) (?y betty))
    ((?x alice) (?y charles))
    ((?x betty) (?y doris))
    ((?x betty) (?y eve)))

? (describe-query :last)
> (:query-2
   (:accurate :processed)
   (retrieve
    (?x ?y)
    (and (?x woman) (?x ?y has-child))
    :abox
    smith-family))
```

Another way to reuse a defined query is by means of the **substitute** keyword:

A defined query can be reused resp. referenced and its body inserted with the **substitute** keyword:

```
? (retrieve (?a ?b) (substitute (mother-of ?a ?b)))
> (((?a alice) (?b charles))
    ((?a alice) (?b betty))
    ((?a betty) (?b eve))
    ((?a betty) (?b doris)))

? (describe-query :last)
> (:query-4
    (:accurate :processed)
    (retrieve
     (?a ?b)
     (and (?a woman) (?a ?b has-child))
     :abox
     smith-family))
```

Please note that the *formal objects/arguments* used in the query definition, here ?x, ?y, are renamed to match the *actual objects/arguments*, here ?a, ?b. Obviously, the number of actual parameters supplied must always match the number of formal parameters in the definition. However, if one is not interested in the bindings for a certain formal parameter, then one can simply use NIL as an actual parameter:

```
? (retrieve (?a) (substitute (mother-of ?a nil)))
> (((?a alice)) ((?a betty)))

? (retrieve (?a) (?a nil mother-of))
> (((?a alice)) ((?a betty)))

? (describe-query :last)
> (:query-5
    (:accurate :processed)
    (retrieve
     (?a)
     (and (?a woman) (?a ?y-ano1-ano2 has-child))
     :abox
     smith-family))
```

A defined query may be referred to at any position in a query body; for example:

```
? (retrieve (?a) (neg (?a nil mother-of)))
> (((?a charles)) ((?a eve)) ((?a doris)) ((?a betty)) ((?a alice)))

? (describe-query :last)
```



```
> (:query-8
  (:accurate :processed)
  (retrieve
    (?a)
    (or
      (and (not (?a woman)) (top ?y-ano1-ano2))
      (not (?a ?y-ano1-ano2 has-child)))
    :abox
    smith-family))
```

However, in most cases you probably want

```
? (retrieve (?a) (neg (project-to (?a) (?a nil mother-of))))
> (((?a eve)) ((?a doris)) ((?a charles)))
```

instead (see discussion above).

Please note that defined queries can also be used within TBox queries (see below for more information on TBox queries resp. `tbox-retrieve`):

```
? (defquery subconcepts-of-mother (?x) (mother ?x has-descendant))
> subconcepts-of-mother

? (tbox-retrieve
  (?descendant-concept)
  (?descendant-concept subconcepts-of-mother))
> (((?descendant-concept grandmother))
  ((?descendant-concept *bottom*)))
```

Please note that defined queries can also be used in rules as well (see below for more information on rules).

6.1.5.1 Ambiguous Queries

Please note that the query

```
(retrieve (?a ?b) (?a ?b mother-of))
```

is *syntactically ambiguous*, since `mother-of` might be a role name as well. The same problem occurs for concept query atoms.

In case the query is ambiguous, nRQL will output a warning:

```
? (define-primitive-role mother-of)
> :OKAY
```

```

? (retrieve (?x ?y) (?x ?y mother-of))
*** NRQL WARNING: ROLE mother-of EXISTS IN TBOX family.
    ASSUMING YOU ARE REFERRING TO THE ROLE mother-of!
> NIL

? (defquery mother (?x) (and (?x woman) (?x ?y has-child)))
> mother

? (retrieve (?x) (?x mother))
*** NRQL WARNING: CONCEPT mother EXISTS IN TBOX family.
    ASSUMING YOU ARE REFERRING TO THE CONCEPT mother!
> (((?x alice)) ((?x betty)))

```

So, nRQL assumes you are referring to the role resp. concept then. If you *really* want to refer to the defined query, please use the `substitute` keyword as illustrated above.

6.1.5.2 Using Defined Queries in Query Definitions

Of course it is possible to use defined queries in query definitions. However, cyclic definitions are prohibited:

```

? (defquery
  mother-of-male-child
  (?m)
  (and (substitute (mother-of ?m ?c)) (?c man)))
> mother-of-male-child

? (retrieve (?x) (substitute (mother-of-male-child ?x)))
> (((?x alice)))

```

The alternative (but ambiguous) syntax can be used as well.

6.1.5.3 The API for Defined Queries

There are various nRQL API functions for accessing and manipulating the defined queries. Please refer to the Reference Manual.

6.1.5.4 Defined Queries Belong to the TBox

Please note that the definitions are local to the (`current-tbox`).

Thus, if the current TBox changes, the definitions are changing, too. If you want to put a definition into a TBox other than the current one, you can supply an optional keyword argument `:tbox` to `defquery` and related API functions and macros:

```
? (full-reset)
> :okay-full-reset

? (in-tbox a)
> a

? (in-abox a)
> a

? (defquery test (?x) (?x a))
> test

? (instance a a)
> :OKAY

? (retrieve (?x) (?x test))
> (((?x a)))

? (describe-all-definitions)
> ((defquery test (?x) (?x a)))

? (in-tbox b)
> b

? (in-abox b)
> b

? (associated-tbox b)
> b

? (instance b b)
> :OKAY

? (defquery test (?x) (?x b))
> test

? (retrieve (?x) (?x test))
> (((?x b)))

? (describe-all-definitions)
> ((defquery test (?x) (?x b)))

? (defquery test2 (?x) (?x top) :tbox a)
> test2

? (describe-all-definitions)
```

```
> ((defquery test (?x) (?x b)))

? (describe-all-definitions :tbox a)
> ((defquery test2 (?x) (?x top)) (defquery test (?x) (?x a)))

? (retrieve (?x) (substitute (test2 ?x)))
> (:ERROR
    *** NRQL ERROR: Can't find definition test2 in DBox for TBox b!)

? (in-tbox a)
> a

? (in-abox a)
> a

? (describe-all-definitions)
> ((defquery test2 (?x) (?x top)) (defquery test (?x) (?x a)))

? (retrieve (?x) (substitute (test2 ?x)))
> (((?x a)))
```

6.1.6 ABox Augmentation with Simple Rules

nRQL offers a simple ABox augmentation mechanism: rules. nRQL rules are termination-safe, since there is no automatic rule application strategy. The application strategy is completely under control of the client application / user. However, nRQL supports rule API functionality which will be needed for the client-side implementation of a rule application strategy (see below).

6.1.6.1 Terminology - Rule Antecedent and Consequence

nRQL rules have an *antecedent* (or *precondition*) and a *consequence* (or *postcondition*). An *antecedent* is just an ordinary nRQL query body. The *consequence* is a set of so-called *generalized ABox assertions*. A generalized ABox assertion can also reference variables which are bound in the rule antecedent (query body).

A rule consequence can contain generalized concept assertions, generalized role assertions, generalized **constrained**, generalized **constraint** assertions, generalized **same-as** assertions. Moreover, assertions can also be removed.

6.1.6.2 A Simple nRQL Rule

The following simple nRQL rule “promotes” `woman` which are not yet `mothers` to `mothers`:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family.racer")
> :OKAY

? (firerule
  (and (?x woman) (neg (?x mother)))
  ((instance ?x mother)))
> (((instance doris mother)) ((instance eve mother)))
```

The antecedence is the query body `(and (?x woman) (neg (?x mother)))`, and the consequence is a single generalize instance assertion `(instance ?x mother)`. As shown, the variables in the generalized assertions in a rule consequence are instantiated with the bindings satisfying the rule antecedence, i.e., the bindings which have been computed by the query body in the antecedence. The constructed assertions are returned as the result of the rule application. We can easily verify with a query that `eve` and `doris` have become `mothers` as well:

```
? (retrieve (?x) (?x mother))
> (((?x alice)) ((?x betty)) ((?x doris)) ((?x eve)))
```

Note that rules using NAF-negated atoms (like `(neg (?x mother))`) behave *non-monotonically*. Because now there are no more woman which are not also mothers, the rule can't be applied a second time:

```
? (firerule
  (and (?x woman) (neg (?x mother)))
  ((instance ?x mother)))
> NIL
```

6.1.6.3 Controlling what to add to an ABox

RacerPro can be advised to automatically add created rule consequences to an ABox if `(add-rule-consequences-automatically)` is enabled; you must add rule consequences manually (see below) if `(dont-add-rule-consequences-automatically)` has been evaluated; use `(describe-query-processing-mode)` to learn which mode is currently enabled. Moreover, the rule API allows a fine-grained control over what and when something is added.

6.1.6.4 Creating New ABox Individuals with Rules

A rule can also create new individuals in an ABox. For example, there are individuals in the ABox which are mothers, but there are no known children modeled:

```
? (retrieve
  (?x)
  (and (?x mother) (neg (?x (has-known-successor has-child)))))
> (((?x doris)) ((?x eve)))
```

Thus, `doris` and `eve` are known to be mothers, but they have no known children. If you are getting NIL, please ensure that the rule `(firerule (and (?x woman) (neg (?x mother))) ((instance ?x mother)))` was fired in advance).

So let's add these children with a rule:

```
? (firerule
  (and (?x mother) (neg (?x (has-known-successor has-child)))))
  ((instance (new-ind first-child-of ?x) human)
   (related ?x (new-ind first-child-of ?x) has-child)))
> (((instance first-child-of-eve human)
   (related eve first-child-of-eve has-child))
  ((instance first-child-of-doris human)
   (related doris first-child-of-doris has-child)))
```

Note that the operator `(new-ind <name> <ind-or-var>*)` creates a new ABox individual prefixed with `<name>` for each different binding possibility of the argument objects in `<ind-or-var>*`, and `(new-ind <name>)` will simply create a new ABox individual `<name>`.

Note that a rule with an empty antecedence `nil` or `()` is conceptually a conjunction without arguments and thus interpreted as `TRUE` (thus, also `true-query` could be used). A rule with tautological (always true) precondition can be used to add assertions. But please note that the objects mentioned in the consequence must also appear in the precondition; thus, for

```
(firerule () ((instance new-ind person)))
```

you get:

```
*** NRQL ERROR: Parser Error: Object ?new-ind not mentioned in query body nil
```

Please use

```
(firerule () ((instance (new-ind new-ind) person)))
```

instead.

6.1.6.5 Rules and the Concrete Domain

Rules can also be used to address the concrete domain part of a RacerPro ABox. This means you can add `constrained` as well as `constraints` assertions with a rule. The syntax for the constraints which is inherited from the RacerPro API function `constrained-entailed-p` has been extended so that also *head projection operators* are valid as parameters in the constraints. The following RacerPorter log demonstrates these possibilities:

```
? (full-reset)
> :okay-full-reset

? (define-concrete-domain-attribute age :type cardinal)
> :OKAY

? (firerule nil ((instance (new-ind a) (an age))))
> (((instance a (an age))))

? (firerule nil ((instance (new-ind b) (an age))))
> (((instance b (an age))))

? (firerule
  (and (?x (an age)) (?y (an age)) (neg (same-as ?x ?y)))
  ((constrained ?x (new-ind age-of ?x) age)
   (constrained ?y (new-ind age-of ?y) age))
  :how-many 1)
> (((constrained a age-of-a age) (constrained b age-of-b age)))

? (firerule
```

```

      (and (a (an age)) (b (an age)))
      ((constraints (= (age b) (+ 30 (age a))))))
> (((constraints (= age-of-b (+ 30 age-of-a))))))

? (retrieve (?x) (?x (= age 30)))
> NIL

? (constraints (= age-of-a 30))
> :OKAY

? (retrieve (?x) (?x (= age 30)))
> (((?x a)))

? (retrieve (?x (told-value (age ?x))) (?x (= age 60)))
> (((?x b) (:told-values (age ?x) (60))))

? (firerule (a top) ((constraints (= (told-value (age a)) (age a))))))
> (((constraints (= 30 age-of-a))))

? (firerule (b top) ((constraints (= (told-value (age b)) (age b))))))
> (((constraints (= 60 age-of-b))))

? (firerule
  (and (a (an age)) (b (an age)))
  ((constraints
    (= (told-value (age b)) (+ 30 (told-value (age a)))))))
> (((constraints (= 60 (+ 30 30))))))

? (abox-consistent?)
> t

```

6.1.6.6 Using Lambda Expressions in Rule Consequences

Another unique feature of the nRQL rules is their support for `lambda` head projection operators in rule consequences.

In [6.1.3.3](#) we showed how MiniLisp `lambda` expressions can be used in a query head; we used the `lambda-ex1.racer` to compute the size of a the object `i`. The `lambda-ex1.racer` file contains:

```

(full-reset)

(define-concrete-domain-attribute width :type integer)

(define-concrete-domain-attribute length :type integer)

(instance i (and (equal width 10) (equal length 20)))

```


Implicitly, we already know that `i` has a size of $10 \times 20 = 200$; however, we would like to add this computed value with a rule to the ABox as well to make the value explicit:

```
? (racer-read-file "nrql-user-guide-examples/lambda-ex1.racer")
> :OKAY

? (all-attributes)
> (width length)

? (firerule
  (?x (and (a width) (a length)))
  ((instance
    ?x
    (= size
      ((lambda (w l) (* (first w) (first l)))
       (told-value-if-exists (width ?x))
       (told-value-if-exists (length ?x)))))))
> (((instance i (= size 200))))

? (retrieve (?x (told-value-if-exists (size ?x))) (?x (a size)))
> (((?x i) ( (:existing-told-values (size ?x)) (200)))))
```

Note that RacerPro does not even require a `(define-primitive-attribute size ...)` declaration here.

Note that we have modeled the `size` filler implicitly, with the concept assertion `(instance i (= size 200))` which has been added by the rule. Sometimes, one also wants to have CD objects, e.g. `size-of-i`, in order to be able to formulate further constraints with other CD objects. A rule can also add `constrained` and `constraints` assertions. Please consider the following rule which creates a new CD object called `size-of-i` representing `i`'s `size`; moreover, we (unnecessarily) also add a `constraints` assertion on `size-of-i`:

```
? (firerule
  (?x (and (a width) (a length)))
  ((constrained ?x (new-ind size-of ?x) size)
   (constraints
    (= (new-ind size-of ?x)
      ((lambda (w l) (* (first w) (first l)))
       (told-value-if-exists (width ?x))
       (told-value-if-exists (length ?x)))))))
> (((constrained i size-of-i size) (constraints (= size-of-i 200))))

? (retrieve (?x (size ?x) (told-value (size ?x))) (?x (a size)))
> (((?x i) ((size ?x) (size-of-i) ( (:told-values (size ?x)) (200)))))
```

6.1.6.7 Adding Pseudo-Nominals with Rules

Suppose you need one new unique concept name for each ABox individual, e.g., you want to add `(instance betty concept-for-betty)` for `betty`, `(instance doris concept-for-doris)` for `doris`, and so on. Sometimes, these instance assertion resp. concepts are called “pseudo-nominals”.

Pseudo-nominals can be added with a rule as follows; please note that we must use `nrql-user-guide-examples/family-no-signature.racer` since the `signature` mechanism would prevent the creation of new concept names with the rule. So, `family-no-signature.racer` is the same KB as `family.racer`, but without the `signature` declaration:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (firerule (?x top) ((instance ?x (new-concept concept-for ?x))))
> (((instance eve concept-for-eve))
   ((instance doris concept-for-doris))
   ((instance charles concept-for-charles))
   ((instance betty concept-for-betty))
   ((instance alice concept-for-alice)))

? (abox-consistent?)
> t
```

Conceptually, `new-concept` is just another syntax for `new-ind`.

Due to the added instance assertions, queries referencing the pseudo-nominals can be answered:

```
? (retrieve (?x) (?x (some has-child concept-for-doris)))
> (((?x betty)))
```

6.1.6.8 OWL Rules

Let us demonstrate how to fire rules on OWL KBs. We are using the `people-pets.owl` KB:

```
? (full-reset)
> :okay-full-reset

? (owl-read-file "~/nrql-user-guide-examples/people-pets.owl")
> ~/nrql-user-guide-examples/people-pets.owl
```

So let us ask which persons have no pet:

```
? (retrieve
  (?x)
  (and
    (?x #!person)
    (neg (project-to (?x) (?x ?y #!has_pet))))))
> (((?x http://cohse.semanticweb.org/ontologies/people#Kevin)))
```

Let us create #!cat pets for those people with a rule:

```
? (firerule
  (and
    (?x #!person)
    (neg
      (project-to
        (?x)
        (?x ?y #!has_pet))))
  ((related ?x (new-ind cat-of ?x) #!has_pet)
    (instance (new-ind cat-of ?x) #!cat)))
> (((related
  http://cohse.semanticweb.org/ontologies/people#Kevin
  http://cohse.semanticweb.org/ontologies/people#cat-of-Kevin
  http://cohse.semanticweb.org/ontologies/people#has_pet)
  (instance
    http://cohse.semanticweb.org/ontologies/people#cat-of-Kevin
    http://cohse.semanticweb.org/ontologies/people#cat)))
```

Let us verify that #!kevin has a #!cat now:

```
? (retrieve
  (?pet-of-kevin (direct-types ?pet-of-kevin))
  (#!Kevin ?pet-of-kevin #!has_pet)
  :dont-show-lambdas-p t)
> (((?pet-of-kevin
  http://cohse.semanticweb.org/ontologies/people#cat-of-Kevin)
  ((http://cohse.semanticweb.org/ontologies/people#cat)
    (http://cohse.semanticweb.org/ontologies/people#pet))))
```

6.1.6.9 The API of the Rule Engine

There is an extensive rule API. Please consult the reference manual. Moreover, rules can be used in a “add all consequences at once”, or in an incremental “add one set of consequences at a time” style.

nRQL doesn’t offer a predefined rule application strategy. However, API functions are provided which enable the client- / user-side definition of (different) rule application strategies. See Section 6.2.4.

6.1.7 Complex TBox Queries

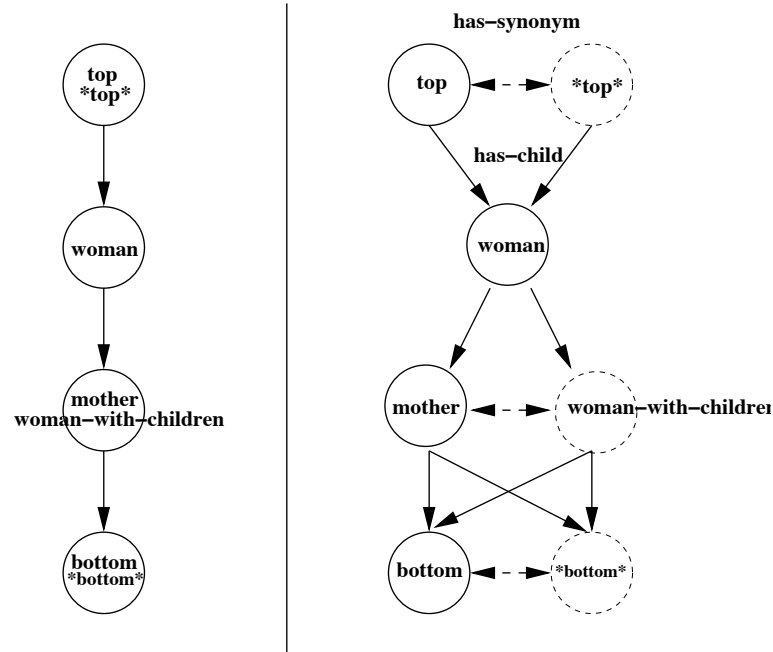


Figure 6.1: Left: Taxonomy. Right: Taxonomy ABox. Dashed nodes represent non-representative concepts. Dashed arrows represent the **has-synonym** relationship, the other arrows the **has-child** relationship.

Complex TBox queries are used to query the taxonomy of a TBox for certain sub/superclass relationships. The idea is simple: whereas the ABox queries we have used so far bind variables against ABox individuals, the variables in a TBox query are bound against taxonomy nodes. Note that a taxonomy node represents a set of synonym atomic concept names.

6.1.7.1 The Relational Structure of a TBox Taxonomy

The taxonomy of a TBox is a *directed acyclic graph (DAG)*. A node in this DAG represents an *equivalence class of synonym concept (names)*. An edge between two nodes represents a *direct-subsumer-of relationship* between the corresponding concepts. For example, the taxonomy produced by the following example is depicted on the left hand side of Figure 6.1:

```
? (full-reset)
> :okay-full-reset

? (implies mother woman)
> :OKAY

? (define-concept mother woman-with-children)
```

```

> :OKAY

? (atomic-concept-synonyms mother)
> (woman-with-children mother)

? (taxonomy)
> ((top NIL (woman))
    ((woman-with-children mother) (woman) (bottom))
    (woman (top) ((mother woman-with-children)))
    (bottom ((mother woman-with-children)) NIL))

```

We construct a *special relational structure* from the taxonomy which is then queried with nRQL, see right hand side of Figure 6.1. This relational structure can also be seen as “virtual” *taxonomy ABox*. Virtual means that this is not a real ABox. However, under this perspective, TBox queries become easy to understand. So, a TBox query is just an ordinary ABox query which is evaluated on the (virtual) taxonomy ABox.

The taxonomy ABox mirrors the taxonomy graph, see Figure 6.1. In the taxonomy, there is *one* node, representing a whole set of synonym concepts, e.g., **mother** and **woman-with-children** is represented as one node. One of these synonym concepts is called the *representative concept*. The representative concepts are shown as bold (non-dashed) circles in Figure 6.1. Thus, the taxonomy ABox contains one individual / node for each concept name. The synonym individuals are then linked by means of **has-synonym** edges / role assertions. Moreover, the individuals which correspond to the *representative concepts* of the taxonomy will be instances of the (virtual) concept **taxonomy-node**. The edges of the taxonomy are represented using (virtual) **has-child** edges / role assertions. In queries, also the inverse of **has-child** can be used, **has-parent**, as well as the transitive closures **has-descendant** and **has-ancestor**.

In principle, the taxonomy ABox could also contain additional edges, e.g., representing disjointness information between concepts. This will be supported in a future nRQL version.

6.1.7.2 Using TBOX-RETRIEVE to Query the Taxonomy

We will now illustrate the TBox queries using the above constructed taxonomy.

First, let us retrieve all child concepts (subsumed concepts) of the concept **woman**:

```

? (tbox-retrieve (?x) (woman ?x has-child))
> (((?x woman-with-children)) ((?x mother)))

```

The inverse role is called **has-parent**:

```

? (tbox-retrieve (?x) (?x woman has-parent))
> (((?x mother)) ((?x woman-with-children)))

```

Nodes in the taxonomy ABox can be accessed by name, or by concept:

```
? (tbox-retrieve (?x) (?x mother))
> (((?x woman-with-children)) ((?x mother)))
```

```
? (tbox-retrieve (mother) (mother mother))
> (((?mother mother)))
```

The set of *representative concepts* can be retrieved using the (virtual query) concept taxonomy-node:

```
? (tbox-retrieve (?x) (?x taxonomy-node))
> (((?x top)) ((?x woman-with-children)) ((?x woman)) ((?x bottom)))
```

Note that nRQL has selected *woman-with-children*, and not *mother* as the representative concept. However, representative concepts are only needed in order to reduce the number of query answer:

```
? (tbox-retrieve (?x ?y) (?x ?y has-child))
> (((?x woman) (?y mother))
   ((?x woman) (?y woman-with-children))
   ((?x mother) (?y *bottom*))
   ((?x mother) (?y bottom))
   ((?x woman-with-children) (?y *bottom*))
   ((?x woman-with-children) (?y bottom))
   ((?x top) (?y woman))
   ((?x *top*) (?y woman)))

? (tbox-retrieve
  (?x ?y)
  (and (?x ?y has-child) (?x taxonomy-node) (?y taxonomy-node)))
> (((?x top) (?y woman))
   ((?x woman-with-children) (?y bottom))
   ((?x woman) (?y woman-with-children)))
```

It is easy to retrieve the synonyms of concept:

```
? (tbox-retrieve (?y) (mother ?y has-synonym))
> (((?y mother)) ((?y woman-with-children)))
```

```
? (tbox-retrieve
  (?y)
  (and (mother ?y has-synonym) (neg (same-as mother ?y))))
> (((?y woman-with-children)))
```

We can also ask for the concepts which have no synonyms:

```
? (tbox-retrieve
  (?x)
  (neg
    (project-to
      (?x)
      (and (?x ?y has-synonym) (neg (same-as ?x ?y))))))
> (((?x woman)))
```

Moreover, the two head projection operator `concept-synonyms` and `describe` are understood:

```
? (tbox-retrieve
  (?x (concept-synonyms ?x))
  (?x taxonomy-node)
  :dont-show-lambdas-p
  t)
> (((?x top) (*top* top))
  ((?x woman-with-children) (woman-with-children mother))
  ((?x woman) (woman))
  ((?x bottom) (*bottom* bottom)))
> (tbox-retrieve
  (?x (describe ?x))
  (?x woman)
  :dont-show-lambdas-p
  t)
> (((?x woman)
  (woman :told-primitive-definition nil
    :synonyms (woman)
    :parents ((*top* top))
    :children ((woman-with-children mother)))))
```

To retrieve all subsumer or subsumees, use the `has-descendant` and/or `has-ancestor` role:

```
? (tbox-retrieve (?y) (and (?x woman) (?x ?y has-descendant)))
> (((?y woman-with-children)
  ((?y bottom))
  ((?y *bottom*))
  ((?y mother)))

? (tbox-retrieve
  (?y)
  (and (?x woman) (?x ?y has-descendant) (?y taxonomy-node)))
> (((?y bottom)) ((?y woman-with-children)))

? (tbox-retrieve
```

```

      (?y (concept-synonyms ?y))
      (and (?x woman) (?x ?y has-descendant) (?y taxonomy-node))
      :dont-show-lambdas-p
      t)
> (((?y bottom) (*bottom* bottom))
    ((?y woman-with-children) (woman-with-children mother)))

```

Also `neg` work as expected; for example, it is easy to retrieve all concepts different from `woman`:

```

? (tbox-retrieve (?x) (neg (?x mother)))
> (((?x bottom))
    ((?x *bottom*))
    ((?x woman))
    ((?x top))
    ((?x *top*)))

```

Note that also `woman-with-children` satisfies `mother`; therefore, it is excluded from the answer. However, if we refer to this concept using its name, we can retrieve all concepts which have a name different from `mother`:

```

? (tbox-retrieve (mother) (neg (mother mother)))
> (((?mother bottom))
    ((?mother *bottom*))
    ((?mother woman))
    ((?mother woman-with-children))
    ((?mother top))
    ((?mother *top*)))

```

Please recall that the standard transformation replaces variables with representative variables.

6.1.7.3 How to Retrieve All Concept Names

The query `(?x top)` will retrieve those nodes which satisfy `top`. Unlike in a “real” ABox, there are only two nodes which satisfy `top`:

```

? (tbox-retrieve (?x) (?x top))
> (((?x *top*)) ((?x top)))

```

To retrieve all individuals from the taxonomy ABox, use either the special *top atom*, or, if you only want to retrieve the representative concepts, use `taxonomy-node` (as already demonstrated):


```
? (tbox-retrieve (?x) (top ?x))
> (((?x bottom))
    ((?x *bottom*))
    ((?x woman))
    ((?x mother))
    ((?x woman-with-children))
    ((?x top))
    ((?x *top*)))

? (tbox-retrieve (?x) (?x taxonomy-node))
> (((?x top)) ((?x woman-with-children)) ((?x woman)) ((?x bottom)))
```

6.1.7.4 Complex TBox Queries

Finally, let us consider a more complex query which searches for diamond-shaped superclass relationships in the taxonomy. Thus, we are looking for a node *?x* that has descendants *?y* and *?z* that both have *?u* as a descendant. Moreover, we must ensure that *?y* and *?z* are “not on the same path”, i.e., *?z* is neither a descendant nor an ancestor of *?y*:

```
? (tbox-retrieve
  (?x ?y ?z ?u)
  (and
    (?x ?y has-descendant)
    (?x ?z has-descendant)
    (?y ?u has-descendant)
    (?z ?u has-descendant)
    (neg (same-as ?y ?z))
    (neg (?y ?z has-descendant))
    (neg (?y ?z has-ancestor))))
7 > (((?x woman) (?y mother) (?z woman-with-children) (?u *bottom*))
    ((?x woman) (?y mother) (?z woman-with-children) (?u bottom))
    ((?x woman) (?y woman-with-children) (?z mother) (?u *bottom*))
    ((?x woman) (?y woman-with-children) (?z mother) (?u bottom))
    ((?x top) (?y mother) (?z woman-with-children) (?u *bottom*))
    ((?x top) (?y mother) (?z woman-with-children) (?u bottom))
    ((?x top) (?y woman-with-children) (?z mother) (?u *bottom*))
    ((?x top) (?y woman-with-children) (?z mother) (?u bottom))
    ((?x *top*) (?y mother) (?z woman-with-children) (?u *bottom*))
    ((?x *top*) (?y mother) (?z woman-with-children) (?u bottom))
    ((?x *top*) (?y woman-with-children) (?z mother) (?u *bottom*))
    ((?x *top*) (?y woman-with-children) (?z mother) (?u bottom)))
```

So, nRQL has identified the synonym concepts *mother* and *woman-with-children* as possible bindings for *?y* and *?z*. However, we want to exclude synonym nodes here:

```
? (tbox-retrieve
```

```

(?x ?y ?z ?u)
(and
  (?x ?y has-descendant)
  (?x ?z has-descendant)
  (?y ?u has-descendant)
  (?z ?u has-descendant)
  (neg (same-as ?y ?z))
  (neg (?y ?z has-descendant))
  (neg (?y ?z has-ancestor))
  (neg (?y ?z has-synonym))))
> NIL

```

So, there are no non-trivial “diamonds” in the taxonomy. We can add one more node to the taxonomy which produces some diamonds; in order to avoid getting all the synonym matches as well we constrain the variable variables to **taxonomy-node** individuals only:

```

? (tbox-retrieve
  (?x ?y ?z ?u)
  (and
    (?x ?y has-descendant)
    (?x ?z has-descendant)
    (?y ?u has-descendant)
    (?z ?u has-descendant)
    (neg (same-as ?y ?z))
    (neg (?y ?z has-descendant))
    (neg (?y ?z has-ancestor))
    (neg (?y ?z has-synonym))
    (?x taxonomy-node)
    (?y taxonomy-node)
    (?z taxonomy-node)
    (?u taxonomy-node)))
> (((?x top) (?y man) (?z woman) (?u bottom))
  ((?x top) (?y man) (?z woman-with-children) (?u bottom))
  ((?x top) (?y woman-with-children) (?z man) (?u bottom))
  ((?x top) (?y woman) (?z man) (?u bottom)))

```

6.1.8 Hybrid Representations with the Substrate Representation Layer

The so-called *substrate representation layer* is used to associate a RacerPro ABox with an additional representation layer. This additional representation layer is called a *substrate*.

The coupling of a RacerPro ABox with a substrate results in a *hybrid representation*. The substrate layer is useful for the representation of *semi-structured data*.

nRQL offers various types of substrates:

- the basic *data substrate*,
- the *mirror data substrate*, as well as
- the *RCC substrate*.

We will discuss each substrate briefly. More types of substrates will be added in the future (e.g., the *database substrate* for coupling an ABox with a relational database is in preparation).

6.1.8.1 The Data Substrate

Similar to an ABox, a *data substrate* is a *relational structure* which can be viewed as a *node- and edge-labeled directed graph*. Nodes are named and have an optional description (the node label). Edges must always have a description (the edge label).

6.1.8.1.1 Data Substrate Labels These descriptions of the nodes and/or edges are called *data substrate labels*. The exact syntax can be found on Page 193, syntax rule `<data-substrate-label>`.

A data substrate label is either

- a simple data literal, or
- a list of data literals, or
- a list of list of data literals.

Data literals are taken from the host language (Common LISP). Symbols, strings, numbers as well as character are supported.

Such a data substrate label is similar to a boolean formula in *Conjunctive Normal Form (CNF)*. For example, the data substrate label `(("foo" "bar") 123.3 foobar)` somehow “represents” the positive boolean “formula” $(\text{"foo"} \vee \text{"bar"}) \wedge 123.3 \wedge \text{foobar}$. Note that `"foo"` and `"bar"` are strings, `123.3` is a floating point number, and `foobar` is a symbol. Unlike boolean formulas, data literals are always positive and thus cannot be negated.

6.1.8.1.2 Data Substrate Nodes, Edges and Labels To populate a data substrate, nRQL's substrate functionality must be enabled first. At its creation, a data substrate is associated with a certain ABox, in this case, the **default** ABox:

```
? (full-reset)
> :okay-full-reset

? (in-data-box default)
> default

? (describe-substrate :abox default)
> ((:type data-substrate)
    (:abox default)
    (:no-of-nodes 0)
    (:no-of-edges 0))
```

The output of **describe-substrate** shows us that there are yet no nodes or edges in that data substrate.

We can now create a node in the substrate as follows:

```
? (data-node betty ("Betty" age 50) (and woman (= age 50) (some has-child human)))
> betty
```

This does three things:

1. Creates a data substrate node named **betty**;
2. labels this node with the description **("Betty" age 50)** which is a “conjunction” of three data literals;
3. creates an ABox individual called **betty**, and finally,
4. adds the concept assertion **(instance betty (and woman (= age 50) (some has-child human)))** to the **default** ABox associated with this data substrate.

Note that not only symbols are valid as names for data substrate nodes, but also numbers, characters, and even strings.

We easily verify that **betty** is an instance in the substrate as well as in the ABox:

```
? (all-concept-assertions)
> ((betty (and woman (= age 50))))

? (get-data-node-description betty)
> ((:node-name betty)
    (:node-label ((Betty) (age) (50))))
```

```

      (:node-successors NIL)
      (:node-predecessors NIL))

? (describe-all-nodes)
> ((((:node-name betty)
      (:node-label ((Betty) (age) (50)))
      (:node-successors NIL)
      (:node-predecessors NIL))))

? (describe-substrate)
> ((:type data-substrate)
   (:abox default)
   (:no-of-nodes 1)
   (:no-of-edges 0))

```

Note that **betty** in the ABox and **betty** in the substrate are associated since the same name is used. However, it is possible to create substrate individuals without associated ABox individual; simply don't supply a concept (a third argument) to **data-node**.

In general, RacerPro's reasoning is completely unaffected by the presence of an associated data substrate layer. Thus, you might ask, what is it good for at all? The answer is: You can use nRQL to query the hybrid representation. nRQL is a hybrid query language.

To demonstrate this, we create one more node, as well as a data substrate edge:

```

? (data-node
   alice
   ("Alice" age 80)
   (and woman (= age 80) (some has-child human)))
> alice

? (data-edge
   alice
   betty
   (has-child "Alice this mother of Betty")
   has-child)
> (alice betty)

```

Note that **data-edge** accepts, similar to **data-node**, an optional argument. in case of **data-node**, the optional argument is a RacerPro concept which adds a concept assertion to the associated ABox, and in case of **data-edge**, a role assertion is created. Note that we have attached the label (**has-child "Alice this mother of Betty"**) to the edge, and used the role **has-child** for the role assertion. Thus:

```

? (describe-all-edges)
> ((((:from-node alice)
      (:to-node betty)

```

```

      (:edge-label ((has-child) (Alice is mother of Betty))))))

? (get-data-edge-description alice betty)
> ((:from-node alice)
   (:to-node betty)
   (:edge-label ((has-child) (Alice this mother of Betty))))

> (all-role-assertions)
> (((alice betty) has-child))

? (all-concept-assertions)
> ((alice (and woman (= age 80) (some has-child human)))
   (betty (and woman (= age 50) (some has-child human))))

? (node-label alice)
> ((Alice) (age) (80))

? (edge-label alice betty)
> ((has-child) (Alice is mother of Betty))

```

Let us add some trivial background knowledge to make the following queries more interesting:

```

? (implies woman human)
> :OKAY

? (define-concept mother (and woman (some has-child top)))
> :OKAY

```

Please note that nodes and edges can also be deleted (see `delete-data-node`, `delete-data-edge`). Please consult the API description of the data substrate facility in the Reference Manual.

6.1.8.1.3 Querying the Hybrid Representation nRQL can now be used to query the hybrid ABox+Substrate representation. A *hybrid query* consists of atoms of two kinds: atoms for the data substrate, and atoms for the ABox. The reader is already familiar with the ABox query atoms. The *data substrate atoms* are introduced now. Both kinds of atoms are combined using the already introduced query body constructors `and`, `union`, `neg`, `project-to`.

So, a hybrid query consists of two kinds of atoms. In order to distinguish substrate from ABox atoms, a simple idea is used: The objects (variables / individuals) in the substrate atoms are simply prefixed with an asterix “*” in order to distinguish them from the atoms which reference the ABox:

```

? (retrieve

```

```

    (?x ?*x ?y ?*y)
    (and (?x woman) (?x ?y has-child) (?*x "Alice") (?*y (age 50))))
> (((?x alice) (?*x alice) (?y betty) (?*y betty)))

```

So, the variables prefixed with `?*` are bound to substrate nodes, and the remaining variables to ABox individuals. This means that `(?x woman)` and `(?x ?y has-child)` are ABox query atoms, and `(?*x "Alice")` and `(?*y (age 50))` are data substrate query atoms. We will discuss these new atoms in a minute. The `?*x`-prefixed variables are called *substrate variables* (see rule `<data-substrate-query-object>`, Page 193).

Moreover, the variables in these atoms are bound pairwise, in parallel: If `?x` is bound to `alice`, then `?*x` is automatically bound to the corresponding substrate node `alice`, and vice versa. In case there is no corresponding node or individual, the pairwise binding fails (and the query returns `NIL` resp. `FALSE`).

Query atoms can reference individuals resp. substrate nodes as well. The same naming convention is used to refer to substrate nodes from within query atoms: Substrate nodes are prefixed with `*` as well. Considering the result delivered by the query above, we see that `?x=alice`, and `?*x=*alice`, even though the name of the substrate node is in fact `alice`:

```

? (retrieve nil (*alice "Alice"))
> t

? (retrieve nil (alice "Alice"))
> (:ERROR
    *** NRQL ERROR: Parser Error: Unrecognized concept expression Alice)

```

Not only the substrate nodes can be queried, but also the edges. Let us add one more atom, a so called *data substrate edge query atom*:

```

? (retrieve
    (?x ?*x ?y ?*y)
    (and
      (?x woman)
      (?x ?y has-child)
      (?*x "Alice")
      (?*y (age 50))
      (?*x ?*y has-child)))
> (((?x alice) (?*x alice) (?y betty) (?*y betty)))

```

There are also *injective data substrate variables*. All variables beginning with `$?*` are injective data substrate variables:

```

? (retrieve (?*x ?*y) (and (top ?*x) (top ?*y)))
> (((?*x alice) (?*y alice))
    ((?*x alice) (?*y betty))
    ((?*x betty) (?*y alice)))

```

```

((?*x betty) (*y betty)))

? (retrieve ($?*x $?*y) (and (top $?*x) (top $?*y)))
> (((?*x alice) ($?*y betty)) (($?*x betty) ($?*y alice)))

```

6.1.8.1.4 Data Substrate Query Atoms A data substrate variable is bound to a substrate node having a certain node label if its label *satisfies* the given *query expression*. We distinguish *data node query expressions* and *data edge query expressions*. Both are commonly called *query expressions*. Query expressions have a structure very similar to the structure of the labels. We already said that labels are positive boolean formulas in CNF. Query expressions are CNF formulas with variables.

Consider the node query expressions (age 50) used in the data substrate query atom (*y (age 50)) in the example query above. Intuitively, this atom matches all nodes which have (age 50) on their node labels. As for the usual ABox concept query atoms, a notion of *logical entailment* is employed for the matching process.

A data node *matches* a given data substrate node query atom iff the label of this node *logically implies* the node query expression in the query atom.

So, **betty** matches the query atom (*y (age 50)) because the label of **betty** is given as ("Betty" age 50) resp. "Betty" \wedge age \wedge 50 and thus *logically implies* the weaker query expression (age 50) resp. age \wedge 50, in signs: "Betty" \wedge age \wedge 50 \models age \wedge 50.

Similarly, the node description (("a" b) 1 2.3) will match the query atom (*x (("a" b #\c 123) (1 2) 2.3)), since (("a" b) 1 2.3) implies (("a" b #\c 123) (1 2) 2.3) (note that (1 2) means "1 or 2", and #\c denotes the character "c"). Since data query expressions are conjunctions (in CNF), the query

```
(retrieve (*x) (*x (age 50)))
```

is equivalent to

```
(retrieve (*y) (and (*x age) (*x 50))).
```

Note that the data substrate edge query expressions have a very similar structure.

Since disjunctions are available in the labels as well as in the queries, some flexibility is achieved (this is why we call it a semi-structured data model). For example, it is easy to retrieve the nodes whose label includes age and which are either 50 or 80:

```

? (retrieve (*x) (*x (age (50 80))))
> (((?*x betty)) ((*x alice)))

? (retrieve (*x) (*x (age (50 60))))
> (((?*x betty)))

? (retrieve (*x) (*x (age (50 "Alice"))))
> (((?*x betty)) ((*x alice)))

```

Note that (50 80) represents a disjunction. Moreover, of course, also **union** is available.

6.1.8.1.5 Using Data Query Predicates *Predicates* can be used in substrate query atoms. So far we have only used literals. Although disjunctions are available (even without **union**), the matching conditions which can be specified for nodes are somewhat limited. Predicates offer much more flexibility and querying power.

nRQL offers so-called *data predicates*. The recognized predicates are summarized on Page 193, see rule <data-query-predicate>. Here is query which uses a data predicate:

```
? (retrieve (?*x) (?*x (age (:predicate (< 40)))))
> NIL
```

```
? (retrieve (?*x) (?*x (age (:predicate (< 60)))))
> (((?*x betty)))
```

Whereas the item **age** is interpreted as before, the conjunct / list entry **(:predicate (< 60))** matches those nodes whose labels contain a numeric literal that satisfies the predicate. Predicates can be used in node atoms as well as in edge atoms.

Please note that the **(:predicate ...)** expressions plays the same role as a literal in the and/or list structure:

```
? (retrieve
  (?*x)
  (?*x (age ((:predicate (= 50)) (:predicate (= 80)))))
> (((?*x betty)) ((?*x alice)))
```

```
? (retrieve
  (?*x)
  (?*x (age (:predicate (= 50)) (:predicate (= 80)))))
> NIL
```

So, these queries are somehow equivalent to

```
? (retrieve (?*x) (?*x (age (50 80))))
> (((?*x betty)) ((?*x alice)))
```

```
? (retrieve (?*x) (?*x (age 50 80)))
> NIL
```

Another example demonstrates that predicates are available which are not offered in the concrete domain of RacerPro, for example, predicates which can search for substrings:

```
18 ? (retrieve (?*x) (?*x (:predicate (search "Bet"))))
18 > (((?*x betty)))
```

Moreover, it is even possible to find pairs of nodes **?*x** and **?*y** which *satisfy* a certain predicate, even if there is no explicit edge between these nodes:

```
? (retrieve (?*x ?*y) (?*x ?*y (:satisfies (:predicate <))))
> (((?*x betty) (?*y alice)))
```

Here we have been searching for substrate nodes $?*x$, $?*y$ which *satisfy* the binary predicate "<" - this means that their node labels must contain numeric literals a and b such that $a < b$ holds. A `:satisfies` predicate must always have arity two. Please refer to 193, rule `data-edge-query-satisfies-expression` for a list of currently supported predicates.

So, the query is somehow the equivalent of this concrete domain query:

```
? (retrieve (?x ?y) (?x ?y (constraint (age) (age) <)))
> (((?x betty) (?y alice)))
```

6.1.8.2 The Mirror Data Substrate

Most users will prefer a facility which automatically creates associated data substrate objects for ABox individuals. This is what the *mirror data substrate* does.

The mirror data substrate is also very useful for users who want to query OWL KBs with nRQL. If the facility is enabled, RacerPro will automatically create data substrate objects for all elements in an OWL KB.

6.1.8.2.1 Populating and Querying the Mirror Data Substrate Let us consider an example to demonstrate the benefits of the mirror data substrate; for this purpose we must “augment” `family-no-signature.racer` a little bit. We also need some `constrained` as well as `constraint` ABox assertions: 7

```
? (full-reset)
> :okay-full-reset

? (racer-read-file "nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (define-concrete-domain-attribute name :type string)
> :OKAY

? (constrained betty betty-age age)
> :OKAY

? (constrained alice alice-age age)
> :OKAY

? (constrained betty betty-name name)
> :OKAY

? (constrained alice alice-name name)
> :OKAY
```

```
? (constraints (string= betty-name "Betty"))
> :OKAY
```

```
? (constraints (string= alice-name "Alice"))
> :OKAY
```

```
? (constraints (= betty-age 50))
> :OKAY
```

```
? (constraints (= alice-age 80))
> :OKAY
```

```
? (abox-consistent?)
> t
```

Let us enable the data substrate mirroring:

```
? (enable-data-substrate-mirroring)
> :okay-data-substrate-mirroring-enabled
```

```
? (describe-substrate)
> (:ERROR
   *** NRQL ERROR: Can't find mirror-data-substrate of ABox default
   (TRY TO CALL "PREPARE-NRQL-ENGINE" FIRST))
```

However, the data substrate is not created before the first query has to be answered:

```
? (retrieve (?x) (?x top))
> (((?x alice)) ((?x betty)) ((?x charles)) ((?x eve)) ((?x doris)))
```

```
? (describe-substrate)
> (:type mirror-data-substrate
   (:abox smith-family)
   (:no-of-nodes 16)
   (:no-of-edges 24))
```

We can already see that the mirror substrate contains 16 nodes, as well as 24 edges. A data substrate node is created in the mirror

- for each ABox individual,
- for each concrete domain object, and
- for each concrete domain value.

Edges are created in the mirror for

- related as well as for
- constrained

axioms.

Let us inspect some nodes:

```
? (get-data-node-description 80)
> ((:node-name 80)
   (:node-label ((:abox-object) (:abox-concrete-domain-value) (80)))
   (:node-successors NIL)
   (:node-predecessors (alice alice-age)))

? (get-data-node-description alice)
> ((:node-name alice)
   (:node-label ((top) (mother) (:abox-object) (:abox-individual)))
   (:node-successors (charles betty 80 alice-name alice-age))
   (:node-predecessors (charles betty)))

? (get-data-node-description alice-name)
> ((:node-name alice-name)
   (:node-label ((:abox-object) (:abox-concrete-domain-object)))
   (:node-successors (Alice))
   (:node-predecessors (alice)))

? (get-data-node-description "Alice")
> ((:node-name Alice)
   (:node-label
    ((:abox-object) (:abox-concrete-domain-value) (Alice)))
   (:node-successors NIL)
   (:node-predecessors (alice-name)))
```

We see that there are various kinds of nodes: nodes for ABox individuals (`alice`), nodes for concrete domain objects (`alice-name`), and nodes for told concrete domain / told datatype values (`80`, `"Alice"`). Appropriate markers are added to the node labels in order to make them “self describing” (see `:node-label` entries in the above descriptions). Note that this information is available to queries.

There are three kinds of nodes:

1. The label of a node representing an ABox individual (e.g., `alice`) contains the conjuncts / marker `:abox-object`, `:abox-individual`.
The node also contains concept membership information, e.g. for `alice`: `(top)` (`mother`). This information has been extracted from the given concept assertions in the KB.
2. The label of a node representing a concrete domain object (e.g., `alice-name`) contains the conjuncts / marker `:abox-object`, `:abox-concrete-domain-object`.

3. The label of a node representing a told concrete domain value / datatype value (e.g., 80, "Alice"), contains the conjuncts / marker `:abox-object`, `:abox-concrete-domain-value`.

Moreover, the node has the name of its value and also contains its value (contains itself) in the label.

The **related**, **constrained**, and **constraints** ABox assertions give rise to appropriately labeled edges in the substrate, connecting the different types of substrate nodes. Let us inspect some edges:

```
? (get-data-edge-description alice betty)
> ((:from-node alice)
   (:to-node betty)
   (:edge-label
    ((has-child)
     (:abox-relationship)
     (:abox-role-relationship)
     (has-descendant))))

? (get-data-edge-description alice alice-age)
> ((:from-node alice)
   (:to-node alice-age)
   (:edge-label
    ((:abox-relationship) (:abox-attribute-relationship) (age))))

? (get-data-edge-description alice-age 80)
> ((:from-node alice-age)
   (:to-node 80)
   (:edge-label
    ((:abox-relationship) (:abox-told-value-relationship))))

? (get-data-edge-description alice-name "Alice")
> ((:from-node alice-name)
   (:to-node Alice)
   (:edge-label
    ((:abox-relationship) (:abox-told-value-relationship))))
```

Please consider the `:edge-label` information. The meaning of the markers should be obvious, given the previous explanations for the `:node-labels`. Again, there are three types of edges:

1. Edges connecting nodes representing ABox individuals (e.g., `alice` and `betty`); such an edge is created due to a role membership ABox assertion.
2. Edges connecting a node representing an ABox individual with a node representing a concrete domain (CD) object which is an attribute filler of the ABox individual (e.g., `alice` and `alice-name`). Such an edge is caused by a **constrained** ABox assertion.

3. Edges connecting a node representing a CD object with node representing a told CD / datatype value (e.g., `alice-age` and `80`). Such an edge is caused by a `constraints` assertion.

Please also try `(describe-all-nodes)`, `(describe-all-edges)`.

We can now exploit the more expressive representation using hybrid nRQL. The same data query atoms as already introduced in the previous section are used.

For example:

```
? (retrieve
  (?*x ?*y ?*name-of-*y ?*age-of-*y ?*tv-1 ?*tv-2)
  (and
    (?*x (:abox-individual))
    (?x ?y has-child)
    (?y woman)
    (?*y ?*age-of-*y (:abox-attribute-relationship age))
    (?*y ?*name-of-*y (:abox-attribute-relationship name))
    (?*age-of-*y (:abox-concrete-domain-object))
    (?*name-of-*y (:abox-concrete-domain-object))
    (?*age-of-*y ?*tv-1 (:abox-told-value-relationship))
    (?*name-of-*y ?*tv-2 (:abox-told-value-relationship))
    (?*tv-1 (:abox-concrete-domain-value (:predicate (< 90))))
    (?*tv-2
      (:abox-concrete-domain-value (:predicate (search "tty"))))))

> (((?*x alice)
  (?*y betty)
  (?*name-of-*y betty-name)
  (?*age-of-*y betty-age)
  (?*tv-1 50)
  (?*tv-2 Betty)))
```

Moreover, *the amount of information* which is mirrored can be controlled. Let us consider the node `alice` again:

```
? (get-data-node-description alice)
> ((:node-name alice)
  (:node-label ((top) (mother) (:abox-object) (:abox-individual)))
  (:node-successors (charles betty 80 alice-name alice-age))
  (:node-predecessors (charles betty)))
```

So, the `:node-label` contains the (told) concept membership information `(top)` `(mother)`. This information is computed and added to the labels according to the *current nRQL mode*, see the documentation of `set-nrql-mode` in the Reference Manual. The nRQL mode controls the amount of completeness of the representation: For example, using nRQL mode

0, only the *told syntactic information* from the ABox is exploited. Thus, only the instance assertions are analyzed and added to the label. In nRQL mode 1, also taxonomic information can be added to the labels:

```
? (full-reset)
> :okay-full-reset

? (enable-data-substrate-mirroring)
> :okay-data-substrate-mirroring-enabled

? (racer-read-file "nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (enable-smart-abox-mirroring)
> :okay-smart-abox-mirroring-enabled

? (retrieve nil (?x top))
> t

? (get-data-node-description alice)
> ((:node-name alice)
   (:node-label
    ((top)
     (woman)
     (parent)
     (human)
     (mother)
     (person)
     (:abox-object)
     (:abox-individual)))
   (:node-successors (charles betty 80))
   (:node-predecessors (charles betty)))
```

Much more information about `alice` is available now. Of course, in a query one can always refer to the associated ABox individual of such a node, so full description logic reasoning is always available.

Regarding the data edges created due to role membership assertions, also the role hierarchy is taken into account:

```
? (get-data-edge-description doris eve)
> ((:from-node doris)
   (:to-node eve)
   (:edge-label
    ((has-sister)
     (:abox-relationship))
```

```
(:abox-role-relationship)
(has-sibling))))
```

Note that the ABox only contains the following role assertions:

```
? (all-role-assertions)
> (((eve doris) has-sister)
    ((doris eve) has-sister)
    ((charles betty) has-sibling)
    ((betty eve) has-child)
    ((betty doris) has-child)
    ((alice charles) has-child)
    ((alice betty) has-child))
```

Thus, the labeling information `has-sibling` has been added according to the role hierarchy, since `((doris eve) has-sister)` implies `((doris eve) has-sibling)`.

6.1.8.2.2 Markers Used in the Mirror Data Substrate Let us summarize the markers which are used.

For the nodes:

- `:abox-object`,
- `:abox-individual`,
- `:abox-concrete-domain-object`,
- `:abox-concrete-domain-value`.

For the edges:

- `:abox-object`,
- `:abox-relationship`,
- `:abox-role-relationship`,
- `:abox-attribute-relationship`,
- `:abox-told-value-relationship`.

In addition, some additional markers are added in case the ABox has been created from an OWL KB:

- `:owl-object`,
- `:owl-relationship`,

- :owl-individual,
- :owl-class,
- :owl-literal,
- :owl-object-property-relationship,
- :owl-datatype-property-relationship,
- :owl-object-annotation-property-relationship,
- :owl-datatype-annotation-property-relationship.

6.1.8.2.3 Using the Mirror Data Substrate on OWL Documents Especially in the case of OWL KBs, the additional querying functionality provided by the mirror substrate is valuable. OWL KBs tend to contain a lot of annotations and told (XML Schema datatype) data values; thus, the query predicates (e.g., `search`) are helpful to retrieve certain resources based on information which is not available for reasoning (note that the fillers / values of annotation properties are not used for reasoning).

Note that substrate variables can now be bound to told values of OWL datatype or annotation fillers! So, you no longer have to use the head projection operators in order to retrieve told values. Moreover, retrieval conditions can be specified which couldn't be specified without the mirror substrate.

We will use `nrql-user-guide-examples/owl-ex5.owl` which contains all kinds of OWL properties:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.owl-ontologies.com/Ontology1159352693.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.owl-ontologies.com/Ontology1159352693.owl">

  <owl:Ontology rdf:about="">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Demo Ontology</rdfs:comment>
  </owl:Ontology>

  <owl:Class rdf:ID="class">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >This is a test class</rdfs:comment>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="object-property">
    <rdfs:domain rdf:resource="#class"/>
  </owl:ObjectProperty>
```

```

<owl:DatatypeProperty rdf:ID="datatype-property">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  <rdfs:domain rdf:resource="#class"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="annotation-object-property">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
  <rdfs:domain rdf:resource="#class"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="annotation-datatype-property">
  <rdfs:domain rdf:resource="#class"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
</owl:DatatypeProperty>

<class rdf:ID="test">
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >Comment for individual test</rdfs:comment>
  <object-property rdf:resource="#test"/>
  <datatype-property rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
  >123</datatype-property>
  <annotation-object-property rdf:resource="#test"/>
  <annotation-datatype-property rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
  >456</annotation-datatype-property>
</class>

</rdf:RDF>

```

We have already listed the markers which are added for the various OWL elements. Lets us briefly inspect the node and edge labels:

```

? (full-reset)
> :okay-full-reset

? (enable-data-substrate-mirroring)
> :okay-data-substrate-mirroring-enabled

? (owl-read-file "nrql-user-guide-examples/owl-ex5.owl")
> nrql-user-guide-examples/owl-ex5.owl

? (describe-all-nodes)
> (((:node-name 456)
    (:node-label ((:owl-object) (:owl-literal) (456)))
    (:node-successors NIL)
    (:node-predecessors
     (http://www.owl-ontologies.com/Ontology1159352693.owl#test)))
  ((:node-name
    http://www.owl-ontologies.com/Ontology1159352693.owl#class)
   (:node-label ((:owl-class)))

```

```

(:node-successors (This is a test class))
(:node-predecessors NIL))

((:node-name This is a test class)
 (:node-label
  ((:owl-object) (:owl-literal) (This is a test class)))
 (:node-successors NIL)
 (:node-predecessors
  (http://www.owl-ontologies.com/Ontology1159352693.owl#class)))

((:node-name Comment for individual test)
 (:node-label
  ((:owl-object) (:owl-literal) (Comment for individual test)))
 (:node-successors NIL)
 (:node-predecessors
  (http://www.owl-ontologies.com/Ontology1159352693.owl#test)))

((:node-name 123)
 (:node-label ((:owl-object) (:owl-literal) (123)))
 (:node-successors NIL)
 (:node-predecessors
  (http://www.owl-ontologies.com/Ontology1159352693.owl#test)))

((:node-name
  http://www.owl-ontologies.com/Ontology1159352693.owl#test)
 (:node-label
  ((top)
   (http://www.owl-ontologies.com/Ontology1159352693.owl#class)
   (:abox-object)
   (:abox-individual)
   (:owl-object)
   (:owl-individual)))
 (:node-successors
  (123
   Comment for individual test
   456
   http://www.owl-ontologies.com/Ontology1159352693.owl#test))
 (:node-predecessors
  (http://www.owl-ontologies.com/Ontology1159352693.owl#test))))

? (describe-all-edges)
> (((:from-node
      http://www.owl-ontologies.com/Ontology1159352693.owl#test)
   (:to-node 123)
   (:edge-label
    ((:owl-relationship)
     (:owl-datatype-property-relationship)
     (http://www.owl-ontologies.com/Ontology1159352693.owl#datatype-property))))

  ((:from-node
    http://www.owl-ontologies.com/Ontology1159352693.owl#test)
   (:to-node

```

```

http://www.owl-ontologies.com/Ontology1159352693.owl#test)
(:edge-label
  ((:owl-object-annotation-property-relationship)
    (http://www.owl-ontologies.com/Ontology1159352693.owl#annotation-object-property)
    (:abox-relationship)
    (:abox-role-relationship)
    (:owl-relationship)
    (:owl-object-property-relationship)
    (http://www.owl-ontologies.com/Ontology1159352693.owl#object-property))))

((:from-node
  http://www.owl-ontologies.com/Ontology1159352693.owl#class)
 (:to-node This is a test class)
 (:edge-label
  ((:owl-relationship)
    (:owl-datatype-property-relationship)
    (:owl-datatype-annotation-property-relationship)
    (http://www.w3.org/2000/01/rdf-schema#comment))))

((:from-node
  http://www.owl-ontologies.com/Ontology1159352693.owl#test)
 (:to-node 456)
 (:edge-label
  ((:owl-relationship)
    (:owl-datatype-property-relationship)
    (:owl-datatype-annotation-property-relationship)
    (http://www.owl-ontologies.com/Ontology1159352693.owl#annotation-datatype-property))))

((:from-node
  http://www.owl-ontologies.com/Ontology1159352693.owl#test)
 (:to-node Comment for individual test)
 (:edge-label
  ((:owl-relationship)
    (:owl-datatype-property-relationship)
    (:owl-datatype-annotation-property-relationship)
    (http://www.w3.org/2000/01/rdf-schema#comment))))

```

It is now easy to retrieve the OWL classes which have an annotation, since also the classes and its annotation are represented as nodes in the substrate (note that there are no ABox individual, though). So, we can ask for the datatype property annotation of all OWL classes which contain "test" as a substring:

```

? (retrieve
  (?*x ?*y)
  (and
    (?*x :owl-class)
    (?*x ?*y :owl-datatype-annotation-property-relationship)
    (?*y (:predicate (search "test")))))
> (((?*x http://www.owl-ontologies.com/Ontology1159352693.owl#class)
  (?*y This is a test class)))

```

We already know that the `#!` prefix always expands into the current default namespace of the OWL KB; thus, `#!test` means `http://www.owl-ontologies.com/Ontology1159352693.owl#test`. However, we already mentioned that *nodes in the substrate must be prefixed with an additional **. So, to access the node `http://www.owl-ontologies.com/Ontology1159352693.owl#test` from a substrate query atom, please either remember to add a `*` in front of the qualified name, **or use the `#&` prefix instead**; the following queries demonstrate this:

```
? (retrieve (?*x)
  (same-as ?*x http://www.owl-ontologies.com/Ontology1159352693.owl#test))
> (:ERROR
  *** NRQL ERROR: Parser Error: Unknown expression
  (same-as ?*x http://www.owl-ontologies.com/Ontology1159352693.owl#test) found)

? (retrieve (?*x)
  (same-as ?*x *http://www.owl-ontologies.com/Ontology1159352693.owl#test))
> (((?*x http://www.owl-ontologies.com/Ontology1159352693.owl#test)))

? (retrieve (?*x)
  (same-as ?*x #!test))
> (:ERROR
  *** NRQL ERROR: Parser Error: Unknown expression
  (same-as ?*x http://www.owl-ontologies.com/Ontology1159352693.owl#test) found)

? (retrieve (?*x)
  (same-as ?*x #&test))
> (((?*x http://www.owl-ontologies.com/Ontology1159352693.owl#test)))
```

Some more queries should be sufficient to demonstrate the utility of the mirror substrate:

```
? (retrieve
  (?*x ?*y)
  (and
    (?*x :owl-object)
    (?*x ?*y :owl-datatype-property-relationship)
    (?*y
      (:owl-literal
        (:predicate numberp)
        (:predicate (>= 100))
        (:predicate (<= 200))))))
> (((?*x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  (?*y 123)))

? (retrieve (?*x) (?*x :owl-object))
> (((?*x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
  (?*x 123)))
```

```

((?x Comment for individual test))
((?x This is a test class))
((?x 456)))

? (retrieve (?x ?y) (?x ?y :owl-relationship))
> (((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
    (?y http://www.owl-ontologies.com/Ontology1159352693.owl#test))
  ((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
   (?y 123))
  ((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
   (?y Comment for individual test))
  ((?x http://www.owl-ontologies.com/Ontology1159352693.owl#test)
   (?y 456))
  ((?x http://www.owl-ontologies.com/Ontology1159352693.owl#class)
   (?y This is a test class)))

```

6.1.8.3 The RCC Substrate - Querying RCC Networks

Many applications have to represent domain objects that also have spatial characteristics. However, the applicability of standard DLs in spatial domains is limited. To support such applications, we are offering a spatial representation layer - the RCC substrate.

The so-called *Region Connection (RCC) Calculus* provides a well-known and widely used set of *qualitative spatial relationships (QSRs)*. The QSRs offered by RCC are used to describe the relative location of spatial objects to one another. They are closely related to natural language prepositions such as *contains*, *overlaps*, *adjacent* etc.

6.1.8.3.1 The Purpose of the RCC Substrate The RCC substrate is a special kind of data substrate which offers representation and querying facilities for so-called RCC networks.

The *edges* of an RCC network are labeled with *RCC relationships*, denoting the relative qualitative spatial arrangement of two objects. The RCC substrate offers the following sets of RCC relationships:

- The set of *RCC5 relationships* distinguishes 5 relationships: DR = *discrete from* (meaning that the two objects neither overlap nor are adjacent), PO = *partial overlap* (the two objects overlap partially), EQ = *equal* (the two objects are congruent), PP = *proper part* (meaning the first object is contained within the second one), and its inverse, PPI = *proper part inverse* (denoting containment).
- The set of *RCC8 relationships* is a finer version of RCC5. It is finer than RCC5 because it also considers the relationships of the borders of the objects to refine the qualitative spatial description. RCC8 includes the RCC5 relationships PO and EQ. RCC8 distinguishes in the DR case whether two non-overlapping objects are adjacent / *externally connected* (EC), or whether they are *disconnected* (DC). Thus, the RCC5 relation DR is refined into the EC and DC case. Moreover, the PP and PPI relations are

refined by considering whether the contained object is a *a tangential proper part* (is tangentially contained, i.e., touches the border from the inside) or not. In the former case, TPP is used, and in the latter case the *non-tangential proper part* relationship NTPP. The inverse relations are called TPPI and NTPPI.

Moreover, *disjunctions* of these relations can be used to represent coarser or indefinite knowledge regarding the spatial relationship.

In general, the RCC edges labels have the same CNF structure as ordinary edge labels. However, certain symbols denote RCC base relations. The following edge labels will be recognized as RCC relations:

- either a *single symbol in lowercase starting with a colon* (":"), e.g., :dr, :ppi, ..., or
- a list of lists of such RCC symbols denoting a conjunction of disjunctions of RCC relations, e.g., ((:ec :dc)). Note that :dr=((:ec :dc)), :pp=((:ntpp :tpp)), and :ppi=((:ntppi :tpi)).

Please also note that an edge label such as (:ntpp :tpp) would actually denote a *conjunction*, and *not* a disjunction. Since the RCC base relations have the so-called JEPD (jointly exhaustive, pairwise disjoint) property, such a conjunction denotes the empty relation.

The RCC substrate supports is aware of the “spatial characteristics” / “inherent properties” of the RCC relationships. For example, suppose you want to represent the spatial situation in which object *a* contains object *b*, and object *b* contains object *c*, but you also specify that *a* and *c* are not overlapping (disjoint). Obviously, such a spatial situation can never be realized in the physical world: The RCC substrate will thus become *inconsistent*.

Moreover, in an RCC network / substrate, logically implied / entailed relationships will hold. These entailed relationships can be retrieved with nRQL. For example, suppose you construct a network in which *a* contains *b* (PPI(*a*, *b*)), and in which *a* and *c* are disconnected (DC(*a*, *c*)). A nRQL query asking for disconnected objects will now not only return (*a*, *c*), but also (*b*, *c*), since DC(*b*, *c*) is logically entailed.

Usually, you will not create an “isolated” RCC network, but a network in which the nodes also have *associated (corresponding) ABox individuals*. We have described this functionality when we discussed the ordinary data substrate. Thus, the RCC substrate can serve an ABox as an additional representation medium which is aware of the special characteristics of RCC relationships and thus can be used the answer (qualitative) spatial queries.

Let us consider the example file `rcc-substrate.racer` (the `;;`'s are comments):

```
(full-reset)
(in-tbox geo-example)

(define-concrete-domain-attribute inhabitants :type cardinal)
(define-concrete-domain-attribute has-name :type string)
(define-primitive-attribute has-language)
```

```
(in-abox geo-example)

;;; Create an RCC5 substrate which is associated with the ABox
;;; geo-example

(in-rcc-box geo-example :rcc5)

;;; Create some RCC substrate nodes.
;;; Note that "data-node" can be used
;;; as well.

(rcc-instance europe)

(rcc-instance germany
  (country germany)
  (and country
    (string= has-name "Germany")
    (all has-language german)
    (some has-language language)
    (= inhabitants 82600000)))

(rcc-instance france
  (country france)
  (and country
    (string= has-name "France")
    (all has-language french)
    (some has-language language)
    (= inhabitants 60656178)))

(rcc-instance hamburg
  (city hamburg)
  (and city
    (some in-country germany)
    (string= has-name "Hamburg")))

(rcc-instance paris
  (city paris)
  (and city
    (some in-country france)
    (string= has-name "Paris")))

;;; Create some RCC substrate edges.
;;; Europe contains Germany

(rcc-related europe germany :ppi)
(rcc-related europe france :ppi)
```



```
(rcc-related germany hamburg :ppi)
(rcc-related france paris :ppi)
(rcc-related france germany :dr)
```

In this example, 5 RCC nodes are created (`europa`, `germany`, `france`, `paris`, `hamburg`), and it is stated that `europa` contains `france` and `germany`, which do not overlap. Moreover, `paris` is contained in `france`, and `hamburg` in `germany`. These are the only spatial relationships given. For all nodes but `europa`, also corresponding ABox individuals are created and instance assertions are added.

This spatio-thematic representation can now be queried with nRQL, as the following examples demonstrate:

```
? (retrieve
  (?*x ?*y ?*z)
  (and
    (?x (and (string= has-name "Hamburg") (some in-country germany)))
    (?y (and country (> inhabitants 8000000)))
    (?*x ?*y :pp)
    (?*y ?*z :pp)))
> (((?*x hamburg) (?*y germany) (?*z europa)))

? (retrieve
  (?*x ?*y)
  (and
    (?*x ?*y :dr)
    (?y (and city (some in-country germany)))
    (?x (and city (some in-country france)))))
> (((?*x paris) (?*y hamburg)))

? (retrieve (?x) (and (*europa ?*x :ppi) (?x city)))
> (((?*x paris)) ((?*x hamburg)))
```

Note that nRQL has *deduced* that the `:dr` relation holds between `paris` and `hamburg`, and that `:pp` holds between `hamburg` and `europa`, etc.

6.1.8.3.2 OWL and the RCC Substrate The RCC substrate can be used on OWL KBs as well. The *RCC mirror substrate* has both the functionality of the *mirror data substrate* as well as of the *RCC substrate* just discussed. Thus, it automatically creates and populates the substrate from the (OWL) ABox. Moreover, certain OWL object properties will be recognized as RCC relations, given they have been declared as *RCC synonyms* in advance.

Consider the following OWL KB `nrql-user-guide-examples/owl-rcc.owl`:

```

<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://www.owl-ontologies.com/Ontology1162148702.owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.owl-ontologies.com/Ontology1162148702.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="city">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="geo-thing"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="continent">
    <rdfs:subClassOf rdf:resource="#geo-thing"/>
  </owl:Class>
  <owl:Class rdf:ID="country">
    <rdfs:subClassOf rdf:resource="#geo-thing"/>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="contains">
    <rdfs:domain rdf:resource="#geo-thing"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="adjacent">
    <rdfs:domain rdf:resource="#geo-thing"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="disjoint">
    <rdfs:domain rdf:resource="#geo-thing"/>
  </owl:ObjectProperty>
  <country rdf:ID="germany">
    <contains>
      <city rdf:ID="hamburg">
      </city>
    </contains>
  </country>
  <continent rdf:ID="europe">
    <contains>
      <country rdf:ID="france">
        <contains>
          <city rdf:ID="paris"/>
        </contains>
        <adjacent rdf:resource="#germany"/>
      </country>
    </contains>
    <contains rdf:resource="#germany"/>
  </continent>
</rdf:RDF>

```

So, the object properties `#!contains`, `#!adjacent` and `#!disjoint` are meant to denote qualitative spatial relationships in the OWL KB. In RCC8 terminology, these QSRs can be expressed as the (disjunctive) relations `(:ntppi :tppi)`, `(:EC)`, and `(:DC)`. If we declare

these object properties as *RCC synonyms*, then the RCC substrate will recognize and process them accordingly:

```
? (owl-read-file "nrql-user-guide-examples/owl-rcc.owl")
> nrql-user-guide-examples/owl-rcc.owl

? (enable-rcc-substrate-mirroring)
> :okay-rcc-substrate-mirroring-enabled

? (rcc-synonym #!contains (:ntppi :tppi))
> ((http://www.owl-ontologies.com/Ontology1162148702.owl#contains
    (:ntppi :tppi)))

? (rcc-synonym #!adjacent (:ec))
> ((http://www.owl-ontologies.com/Ontology1162148702.owl#adjacent
    (:ec))
    (http://www.owl-ontologies.com/Ontology1162148702.owl#contains
    (:ntppi :tppi)))

? (rcc-synonym #!disjoint :dc)
> ((http://www.owl-ontologies.com/Ontology1162148702.owl#disjoint
    :dc)
    (http://www.owl-ontologies.com/Ontology1162148702.owl#adjacent
    (:ec))
    (http://www.owl-ontologies.com/Ontology1162148702.owl#contains
    (:ntppi :tppi)))
```

These RCC synonyms can be used in queries as well:

```
? (retrieve
    (?*x ?*y)
    (and
      (?*x ?*y #!adjacent)))
> (((?*x
    http://www.owl-ontologies.com/Ontology1162148702.owl#france)
    (?*y
    http://www.owl-ontologies.com/Ontology1162148702.owl#germany))
    ((?*x
    http://www.owl-ontologies.com/Ontology1162148702.owl#germany)
    (?*y
    http://www.owl-ontologies.com/Ontology1162148702.owl#france)))
\begin{verbatim}
```

It is interesting to observe that the RCC substrate cannot prove that `{\tt \#!paris}` and `{\tt \#!france}` are in `{\tt :dc}` relation - actually, they could be `{\tt :ec}` as well. A coarser RCC relation specified with synonyms will also return

{\tt \#!paris} and {\tt \#!france}, try

```
\begin{verbatim}
(retrieve
  (?*x ?*y)
  (and
    (?*x ?*y (\#!adjacent \#!disjoint))))
```

Moreover, note that the full annotations added by the mirror data substrate for the OWL objects is available for retrieval as well:

```
? (get-data-node-description
  http://www.owl-ontologies.com/Ontology1162148702.owl#paris)
> ((:node-name
  http://www.owl-ontologies.com/Ontology1162148702.owl#paris)
  (:node-label
  ((top)
  (http://www.owl-ontologies.com/Ontology1162148702.owl#city)
  (:abox-object)
  (:abox-individual)
  (:owl-object)
  (:owl-individual)))
  (:node-successors NIL)
  (:node-predecessors
  (http://www.owl-ontologies.com/Ontology1162148702.owl#hamburg
  http://www.owl-ontologies.com/Ontology1162148702.owl#germany
  http://www.owl-ontologies.com/Ontology1162148702.owl#europe
  http://www.owl-ontologies.com/Ontology1162148702.owl#france))))

? (get-data-edge-description
  http://www.owl-ontologies.com/Ontology1162148702.owl#france
  http://www.owl-ontologies.com/Ontology1162148702.owl#germany)
> ((:from-node
  http://www.owl-ontologies.com/Ontology1162148702.owl#france)
  (:to-node
  http://www.owl-ontologies.com/Ontology1162148702.owl#germany)
  (:rcc-relation ((:ec)))
  (:edge-label
  ((:abox-relationship)
  (:abox-role-relationship)
  (:owl-relationship)
  (:owl-object-property-relationship)
  (http://www.owl-ontologies.com/Ontology1162148702.owl#adjacent))))
```

Please note that the RCC synonym mechanism also applies to KBs / ABoxes which are not in OWL format.

6.1.9 Formal Syntax of nRQL

Here we give an EBNF syntax definition of nRQL. * means zero or more occurrences; "X" denotes a literal; { "X" | "Y" | "Z" } means chose exactly one from the given literals "X", "Y", "Z".

For example, the list (?x betty (age ?y)) is a valid <abox-query-head> (if age is a concrete domain attribute, see <cd-attribute-name>), as well as (?x betty betty \$?y |http://a.com/ontology#a|). (), (nil) or (t) are invalid <abox-query-head>s.

The top level syntax is defined as follows:

```
(retrieve <query-head> <query-body>)
(retrieve-under-premise <query-premise> <abox-query-head> <abox-query-body>)
(tbox-retrieve <tbox-query-head> <tbox-query-body>)
(defquery <query-name> <def-query-head> <query-body>)
(firerule <rule-antecedence> <rule-consequence>)
```

Syntax of the query heads and premises:

```
<query-premise>      -> List of RacerPro ABox assertions
<query-name>         -> <symbol> (naming a defined query)
<symbol>             -> any Lisp symbol
                     (e.g., huhu, foobar, |http://a.com/ontoly#a|)
```

WITH THE FOLLOWING EXCEPTIONS:

```
all-types types instantiators all-instantiators all-types-flat
all-instantiators-flat types-flat instantiators-flat direct-types
most-specific-types most-specific-instantiators direct-instantiators
direct-types-flat most-specific-types-flat most-specific-instantiators-flat
direct-instantiators-flat describe individual-synonyms concept-synonyms describe
nrql-equal-role bind-individual inv not neg and or one-of racer satisfies top
bottom true-query false-query constraint has-known-successor substitute insert
same-as equal = intersection union cap cup lambda told-value told-values
told-value-if-exists told-values-if-exists datatype-filler datatype-fillers
annotation-datatype-filler annotation-datatype-fillers unknown-operator
bindings-from no-told-value no-known-cd-objects project project-to pi real-top
real-bottom has-child has-parent has-descendant has-ancestor has-synonym
```

```
<def-query-head>      -> "(" <def-head-entry>* ")"
<abox-query-head>     -> "(" <abox-head-entry>* ")"
<tbox-query-head>     -> "(" <tbox-head-entry>* ")"
```

<abox-head-entry>	->	<query-object>	
		<data-query-object>	
		<abox-head-projection-operator>	
		<lambda-application>	
<tbx-head-entry>	->	<query-object>	
		<tbx-head-projection-operator>	
		<lambda-application>	
<def-head-entry>	->	<query-object>	
		<data-query-object>	
<query-object>	->	<query-variable>	
		<query-individual>	
<lambda-application>	->	"(" "(lambda" "(" <lambda-var>* ")" <lambda-body> ")" <lambda-argument>* ")"	
<lambda-var>	->	<symbol>	
<lambda-body>	->	A MiniLisp S-expression, see the Manual	
<lambda-argument>	->	<query-object>	
		<data-query-object>	
		<head-projection-operator>	
<query-variable>	->	"? "<symbol> "\$? "<symbol>	
<query-individual>	->	<symbol> (naming an ABox individual or concept)	
<data-query-object>	->	<data-query-variable>	
		<data-query-individual>	
<data-query-variable>	->	"?* "<symbol> "\$?* "<symbol>	
<data-query-individual>	->	"* "<symbol>	

Data query objects can only be used if nRQL is used in hybrid mode, i.e., a data substrate mode is enabled.

The syntax of the *head* projection operators:

```

<abox-head-projection-operator ->
    "(" <attribute-name> <query-object> ")" |
    "(" { "told-value" | "told-values" }
        "(" <attribute-name> <query-object> ")" |
    "(" { "told-value-if-exists" | "told-values-if-exists" }
        "(" <attribute-name> <query-object> ")" |
    "(" { "datatype-filler" | "datatype-fillers" }
        "(" <OWL-datatype-property> <query-object> ")" |
    "(" { "annotation-datatype-filler" | "annotation-datatype-fillers" }
        "(" <OWL-annotation-datatype-property> <query-object> ")" |
    "(" { "all-types" | "types" | "instantiators" | "all-instantiators" |
        "all-types-flat" | "types-flat" | "all-instantiators-flat" |
        "instantiators-flat" | "direct-types" | "most-specific-types" |
        "most-specific-instantiators" | "direct-instantiators" |
        "direct-types-flat" | "most-specific-types-flat" |
        "most-specific-instantiators-flat" | "direct-instantiators-flat" |
        "describe" | "individual-synonyms" }
        <query-object> ")"

<tbox-head-projection-operator ->
    "(" { "concept-synonyms" | "describe" } <tbox-query-object> ")"

```

Query bodies and atoms are defined as follows:

```

<abox-query-body> ->
    <empty-query-body> |
    <abox-query-atom> |
    <substrate-query-atom> |
    "(" { "project-to" | "project" | "pi" }
        <def-query-head> <abox-query-body> ")" |
    "(" { "and" | "cap" | "intersection" } <abox-query-body>* ")" |
    "(" { "or" | "cup" | "union" } <abox-query-body>* ")" |
    "(" { "not" | "neg" } <abox-query-body> ")" |
    "(" "inv" <abox-query-body> ")"

<tbox-query-body> ->
    <empty-query-body> |

```

```

<tbody-query-atom>                                     |
"(" { "project-to" | "project" | "pi" }
      <def-query-head> <tbody-query-body> ")"          |
"(" { "and" | "cap" | "intersection" } <tbody-query-body>* ")" |
"(" { "or" | "cup" | "union" } <tbody-query-body>* ")" |
"(" { "not" | "neg" } <tbody-query-body> ")" |
"(" "inv" <tbody-query-body> ")"

<empty-query-body> -> ""

<abox-query-atom> ->
  "true-query"                                         |
  "false-query"                                       |
  "(" { "not" | "neg" } <abox-query-atom> ")"          |
  "(" "inv" <abox-query-atom> ")"                    |
  "(" <query-object> <concept-expression> ")"          |
  "(" <query-object> <query-object>
    { <role-expression> | "nrql-equal-role" } ")"      |
  "(" <query-object> "nil" <role-expression> ")"        |
  "(" "nil" <query-object> <role-expression> ")"        |
  "(" top <query-object> ")"                          |
  "(" bottom <query-object> ")"                        |
  "(" <query-object> <query-object>
    "(" "constraint"
      <role-chain-followed-by-attribute>
      <role-chain-followed-by-attribute>
      <predicate-expression> ")"                      |
  "(" "bind-individual" <abox-query-individual> ")"    |
  "(" { "substitute" | "insert" }
    "(" <query-name>
      { <query-object> | <data-query-object> | "nil" }* ")" |
  "(" { <query-object> | <data-query-object> | "nil" }*
    <query-name> ")"                                     |

```



```

    "(" { "same-as" | "=" | "equal" }
      <query-object> <query-object> ")" |

    "(" <query-object>
      "(" "has-known-successor" <role-expression> ")" )"

<tbbox-query-atom> ->
  "true-query" |

  "false-query" |

  "(" { "not" | "neg" } <tbbox-query-atom> ")" |

  "(" "inv" <tbbox-query-atom> ")" |

  "(" <query-object> <concept-name> ")" |

  "(" <query-object> <query-object>
    { "has-child" | "has-parent" |
      "has-descendant" | "has-ancestor" |
      "has-synonym" } ")" |

  "(" <query-object> "nil" <role-expression> ")" |

  "(" "nil" <query-object> <role-expression> ")" |

  "(" top <query-object> ")" |

  "(" bottom <query-object> ")" |

  "(" "bind-individual" <abox-query-individual> ")" |

  "(" { "substitute" | "insert" }
    "(" <query-name>
      { <query-object> | <data-query-object> | "nil" } * ")" |

  "(" { <query-object> | <data-query-object> | "nil" } *
    <query-name> ")" |

  "(" { "same-as" | "=" | "equal" }
    <query-object> <query-object> ")" |

  "(" <query-object>
    "(" "has-known-successor"
      { "has-child" | "has-parent" |
        "has-descendant" | "has-ancestor" |
        "has-synonym" } ")" )"

```

nRQL data substrate query language:

<data-query-atom> ->

```

    "(" { "not" | "neg" } <data-query-atom> ")" |
    "(" <data-query-object> <data-node-query-expression> ")" |
    "(" <data-query-object> <data-query-object> <data-edge-query-expression> ")" |
    "(" <data-query-object> "nil" <data-edge-query-expression> ")" |
    "(" "nil" <data-query-object> <data-edge-query-expression> ")" |
    "(" top <data-query-object> ")" |
    "(" bottom <data-query-object> ")" |
    "(" <data-query-object> <data-query-object>
        <data-edge-query-satisfies-expression> ")" |
    "(" "bind-individual" <data-query-individual> ")" |
    "(" { "same-as" | "=" | "equal" }
        <data-query-object> <data-query-object> ")" |
    "(" <data-query-object>
        "(" "has-known-successor" <data-edge-query-expression> ")" |
    "(" { "substitute" | "insert" }
        "(" <query-name>
            { <query-object> | <data-query-object> | "nil" }* ")" |
    "(" { <query-object> | <data-query-object> | "nil" }* <query-name> ")"

```

Syntax of labels of data nodes and edges, used in (data-node <symbol> <data-substrate-label>), (data-edge <symbol> <data-substrate-label>):

```

<data-substrate-label>    -> <conjunction-of-data-items>

<conjunction-of-data-items> -> <data-literal> |
                                "(" <disjunction-of-data-items>+ ")"

<disjunction-of-data-items> -> <data-literal> |
                                "(" <data-literal>+ ")"

<data-literal>           -> <symbol-data-literal> |
                                <character-data-literal> |

```

```

<string-data-literal>      |
<numeric-data-literal>

<symbol-data-literal>      -> a Common LISP symbol, e.g. symbol
<character-data-literal>   -> a Common LISP character, e.g. #\a
<string-data-literal>      -> a Common LISP string, e.g. "string"
<numeric-data-literal>     -> a Common LISP number, e.g. 123, 3/2, 123.23

```

Syntax of nRQL data substrate queries:

```

<data-node-query-expression>      -> <conjunction-of-data-query-items>
<data-edge-query-expression>      -> <conjunction-of-data-query-items>
<data-edge-query-satisfies-expression> ->
    "(" "satisfies"
      { <2-ary-data-query-predicate> |
        "(" <disjunction-of-data-query-predicates>+ ")"
      } ")"
<data-query-item>                 -> <data-literal> |
                                     <data-query-predicate>
<conjunction-of-data-query-items> -> <data-query-item> |
                                     "(" <disjunction-of-data-query-items>+ ")"
<disjunction-of-data-query-items> -> <data-query-item> |
                                     "(" <data-query-item>+ ")"
<data-query-predicate>            ->
    "(" "predicate"
      { <0-ary-data-query-predicate> |
        <1-ary-data-query-predicate> } ")"
<disjunction-of-data-query-predicate-items> -> <2-ary-data-query-predicate> |
                                                "(" <2-ary-data-query-predicate>+ ")"
<0-ary-data-query-predicate> -> "stringp" | "zerop" | "integerp" |
    "numberp" | "consp" | "symbolp" |
    "keywordp" | "rationalp" | "floatp" |
    "minusp"

```

```

<1-ary-data-query-predicate> -> "(" { "string="          | "string/="          |
                                     "string-equal"       | "string-not-equal"  |
                                     "string<"           | "string<="          |
                                     "string>"           | "string>="          |
                                     "string-lessp"       | "string-greaterp"   |
                                     "string-not-lessp"    | "string-not-greaterp" |
                                     "search"             |
                                     }
                                     <string-data-literal> ")" |

                                     "(" { "=" | "/"= |
                                           ">" | "<" |
                                           ">=" | "<=" }
                                     <numeric-data-literal> ")" |

                                     "(" "find" <character-data-literal> ")"

<2-ary-data-query-predicate> -> "string="          | "string/="          |
                                "string-equal"       | "string-not-equal"  |
                                "string<"           | "string<="          |
                                "string>"           | "string>="          |
                                "string-lessp"       | "string-greaterp"   |
                                "string-not-lessp"    | "string-not-greaterp" |
                                "=" | "/"= |
                                ">" | "<" |
                                ">=" | "<="

```

Rule consequences:

```

<rule-consequence>          -> "(" <generalized-abox-assertion>* ")"

<rule-antecedence>         -> <query-body>

<generalized-abox-assertion> -> <instance-assertion> |
                                <related-assertion>  |
                                <same-as-assertion>   |
                                <constrained-assertion> |
                                <constraints-assertion> |
                                <forget-concept-assertion> |
                                <forget-role-assertion> |
                                <forget-constrained-assertion> |
                                <forget-constraint-assertion>

```

[illegible]

Note that you can also use `<abox-head-projection-operator>` and `<lambda-application>` within constraint expression; thus, the syntax has been extended by allowing also expression such as `(constraints (= (age b) (+ 30 (age a))))` or `(constraints (= (told-value (age ?y)) (age ?y)))` etc.

```
<forget-concept-assertion>    -> "(" "forget-concept-assertion"
                                <query-object>
                                <concept-expression2> ")"

<forget-role-assertion>      -> "(" "forget-role-assertion"
                                <query-object>
                                <query-object>
                                <role-expression1> ")"

<forget-constrained-assertion> -> "(" "forget-constrained-assertion"
                                <query-object>
```

```

    <cd-object>
    <cd-attribute-name> ")"

```

<forget-constraint-assertion> -> see RacerPro syntax. However, the extensions mentioned in <constraints-assertion> cannot be used here.

Auxiliary syntax rules:

```

<role-chain-followed-by-attribute> -> <cd-attribute-name> |

    "(" <role-expression>* <cd-attribute-name> ")"

<concept-expression>                -> see <C> on page 56, Fig. 3.1

<cd-attribute-name>                 -> see <AN> on page 57, Fig. 3.2

<cd-object-name>                    -> a RacerPro concrete domain object

<role-expression1>                  -> see <R> on page 56, Fig. 3.1 |

    "(inv" <role-expression1> ")"

```

Note: a feature can be used for <R> as well.

```

<concept-expression>                -> see <C> on page 56, Fig. 3.1

<role-expression>                   -> <role-expression1> |

    "(not" <role-expression1> ")"

<OWL-datatype-property>             -> see <R> on page 56, Fig. 3.1

```

Note: role-used-as-datatype-property-p must return t (true) for <R>.

```

<OWL-annotation-property>           -> see <R> on page 56, Fig. 3.1

```

Note: role-used-as-annotation-property-p must return t (true) for <R>.

```

<prediate-expression>               -> see <CDC> on page 57, Fig. 3.2 |

    <CD-predicate>

<CD-prediate> -> "equal" | "unequal" | "string=" | "string<>" |
    ">" | "<" | ">=" | "<=" |
    "<>" | "=" | "boolean=" | "boolean<>" |
    "min" | "max" | "divisible" | "not-divisible"

```

6.2 The nRQL Query Processing Engine

The nRQL query processing engine implements the nRQL language and is integrated with RacerPro. The nRQL engine offers various querying modi. In this section we describe the core functionality of this engine.

6.2.1 The Query Processing Modi of nRQL

nRQL can be used in different querying modi. The two major modi are the *set-at-a-time mode* and the *tuple-at-a-time mode*.

6.2.1.1 The Set-at-a-Time Mode

This is the mode we have used so far in our examples. You are already familiar with the behavior of the nRQL API in this mode:

```
? (describe-query-processing-mode)
> ((:creating-substrates-of-type :racer-dummy-substrate)
   :using-$$-prefix-for-injective-variables
   :check-abox-consistency
   :same-as-is-syntactic-same-as
   :query-optimization-enabled
   :optimizer-uses-cardinality-heuristics
   :automatically-adding-rule-consequences
   :warnings
   :complete-mode
   :mode-3
   :set-at-a-time-mode
   :deliver-kb-has-changed-warning-tokens)

? (retrieve (?x) (?x woman))
> (((?x alice)) ((?x betty)) ((?x eve)) ((?x doris)))
```

Note the `:set-at-a-time-mode` token in the result of `(describe-query-processing-mode)`.

In this mode, a call to `retrieve` returns the whole query answer at once, in one bunch. Most users and/or client applications will be happy with this mode.

6.2.1.2 The Tuple-at-a-Time Modi – Incremental (Iterator-Based) Query Processing

Sometimes it is preferable to compute and retrieve the answer tuples in an *incremental, iterator-based fashion*. The user or application requesting the tuples can look at the tuples already retrieved and decide whether yet another tuple is needed or not.

The nRQL API provides the function `get-next-tuple` for this purpose. Applications can also check for the availability of a next tuple by calling `next-tuple-available-p`:

```
? (process-tuple-at-a-time)
> :okay-processing-tuple-at-a-time

? (retrieve (?x) (?x woman))
> (:query-3 :running)

? (get-next-tuple :query-3)
> ((?x alice))

? (get-next-tuple :last)
> ((?x betty))

? (next-tuple-available-p :last)
> t

? (get-next-tuple :last)
> ((?x eve))

? (next-tuple-available-p :last)
> t

? (get-next-tuple :last)
> ((?x doris))

? (next-tuple-available-p :last)
> NIL

? (get-next-tuple :last)
> :exhausted

? (get-answer :last)
> (((?x alice)) ((?x betty)) ((?x eve)) ((?x doris)))

? (describe-query :last)
> (:query-3
   (:accurate :processed)
   (retrieve (?x) (?x woman) :abox smith-family))
```

In the incremental mode, a call to `retrieve` does not return the query answer, but instead returns a so-called *query identifier (ID)*, which can also be understood as a kind of “iterator”. This ID is used as argument to API functions such as `get-next-tuple`. Moreover, the identifier `:last` always refers to the last submitted query (or rule). In order to differentiate queries from rules, also `:last-query` and `:last-rule` can be used instead of `:last`.

6.2.1.2.1 Concurrent Incremental Query Processing The nRQL engine is a multi-threaded query answering engine which allows you to run several queries concurrently (in parallel).

Thus, it is possible to submit a number of calls to `retrieve` to the engine, and then request tuples from these queries in a random order:

```
? (retrieve (?x) (?x man))
> (:query-4 :running)

? (retrieve (?x) (?x uncle))
> (:query-5 :running)

? (get-next-tuple :query-4)
> ((?x charles))

? (get-next-tuple :query-5)
> ((?x charles))

? (get-next-tuple :query-4)
> :exhausted

? (get-next-tuple :query-5)
> :exhausted
```

6.2.1.2.2 Lazy and Eager Tuple-at-a-Time Modi The incremental mode comes in two forms – *lazy* and *eager*:

- In the so-called *lazy incremental mode*, the next tuple of a query is not computed before it is actually requested by the user or application (unless `get-next-tuple` is called). The thread computing the tuples – which is called the query answering thread in the following – is put to sleep, and `get-next-tuple` re-awakes it.
- In the so-called *eager incremental mode*, the query answering thread is not put to sleep. Instead, it continues to pre-compute tuples, even if the application has not yet requested these “future tuples”. These tuples are put into a queue for future requests.

The API behaves identically in both modi. In any case, the query answering thread dies after all tuples have been computed, or if the query was aborted, or a timeout was reached, or the upper bound on the number of requested tuples was reached.

6.2.2 The Life Cycle of a Query

A query has a life cycle: It is created and prepared (parsed and compiled), then made active (computes tuples), eventually goes to sleep, will be awoken again (compute some more tuples), and eventually becomes inactive (its query answering thread dies), see Figure 6.2.

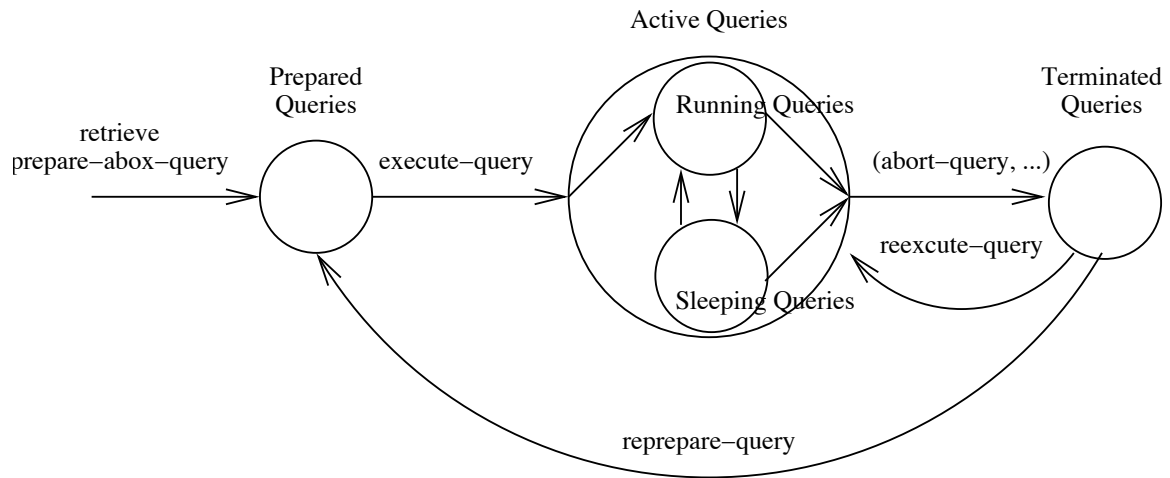


Figure 6.2: Lifecycle of a Query

Even after its tread is terminated, the query is still present as a *query object*, and can be reexecuted.

RacerPorter provides the “Queries” tab which can be used to inspect and manage the queries as well as their current states. Moreover, the function **describe-all-queries** and **describe-query** provide status information about queries.

We can distinguish the following lists / sets of queries according to their status:

1. *The list of all queries.* The corresponding API function **all-queries** returns this list. As long as a query appears on this list, its ID is recognized by the API functions. Use **delete-query**, **delete-all-queries**, etc. to remove queries from this list.
2. *Queries which are ready to run, but have not been started yet.* The corresponding API function **ready-queries** returns this list. The function **query-ready-p** is used to check whether a query is on this list.

The queries on this list are called *ready queries* or *prepared queries*. The queries on this list have been parsed and compiled (prepared), but have not been started yet, i.e., no query answering thread have been assigned to them.

The API function **execute-query** can be applied to queries on this list. In order to put a query to this list *without* also starting its query answering thread, use the API function/macro **prepare-abox-query** instead of **retrieve**.

3. *The list of active queries.* These queries have been started, and there is an active query answering thread associated with them. The corresponding API function **active-queries** returns this list. The function **query-active-p** is used to check whether a query is on this list.

According to the mode in which the query had been started, this list is further partitioned into the following two list:

- (a) *The list of queries which are currently running.* The corresponding API function `running-queries` returns this list. The function `query-running-p` is used to check whether a query is on this list.

A query which appears on this list is actively consuming CPU time; its thread is not sleeping.

- (b) *The list of queries which are currently waiting (sleeping).* The corresponding API function `sleeping-queries` returns this list. The function `query-sleeping-p` is used to check whether a query is on this list.

The thread of a query which appears on this list is currently sleeping; the query had been started in *lazy incremental mode*. Note that queries started in the set-at-a-time mode will never appear on this list.

- 4. *The list of terminated (inactive) queries.* The corresponding API function `terminated-queries` (also: `processed-queries`) returns this list. The function `query-terminated-p` (also: `query-processed-p`) is used to check whether a query is on this list.

A query is put on this list when its query answering thread terminates. Note that different events can cause a query answering thread to terminate, e.g., all query answers are computed, the query is aborted, the timeout is reached, etc.

The function `reprepare-query` puts an already processed query back on the list of *ready queries*, and so a fresh life cycle can begin (“reincarnation”).

It is also possible to directly re-execute and thus put a query back on the list of *active queries* by using the API function `reexecute-query`. Note that this function only applies to terminated queries, not to ready queries. However, there is the function `execute-or-reexecute-all-queries`.

6.2.3 The Life Cycle of a Rule

The same six different lists are also maintained for nRQL rules. RacerPorter provides the “Rules” tab which can be used to inspect and manage the rules as well as their current states.

The corresponding API functions are named

1. `all-rules`
2. `ready-rules` (equivalent)
3. `active-rules`
 - (a) `running-rules`
 - (b) `sleeping-rules`
4. `terminated-rules` (all equivalent)

As for the queries, there are various synonym functions (please refer to the Reference Manual).

Note that there is no facility which checks for applicable rules and applies them automatically. The rules have to be fired by hand. However, the API function `applicable-rules` returns the rules which *can* fire.

6.2.4 How to Implement Your Own Rule Application Strategy

A rule application strategy consists of two things: First the applicable rules have to be identified. Then a subset of these rules must be applied / fired. Finally, in case any of these rules produces more than one set of consequences, it must be decided which consequences to add to the ABox.

nRQL only provides a function which identified the applicable rules: `applicable-rules`. All other just mentioned aspects of a rule application strategy must be implemented by the client / user.

Regarding the third step (deciding which rule consequences to add to an ABox), nRQL provides an *add all* as well as a selected *add only certain chosen sets of consequences* to the ABox functionality. In the latter case, nRQL must be advised which consequences to add – the appropriate consequences must be “chosen”.

- In the *set-at-a-time mode*, all computed (produced) rule consequences can be added automatically to an ABox if nRQL is in `add-rule-consequences-automatically` mode. All computed rule consequences will be added to the ABox after the rule has terminated.

In case `dont-add-rule-consequences-automatically` mode is used, the computed set of rule consequences will be memorized, and later `add-chosen-sets-of-rule-consequences` can be called on this rule to add the memorized rule consequences (although nothing was really “chosen” here, please read further).

- In the *(incremental) tuple-at-a-time mode* the nRQL API provides appropriate functions which can be used to implement you own rule application strategy.

As in the incremental query answering modi, instead of returning the whole set of sets of rule consequences at once in a bunch, nRQL will incrementally construct and return one set of rule consequences after the other, one by one. The next set of rule consequences must be explicitly requested using `get-next-set-of-rule-consequences`. Then nRQL can be advised to *memorize* this computed set of rule consequences with `choose-current-set-of-rule-consequences`.

In case nRQL is in `add-rule-consequences-automatically` mode, the chosen sets of rule consequences are added automatically to the ABox after the last set had been delivered. Otherwise (for example, in case the rule was aborted), the API function `add-chosen-sets-of-rule-consequences` can be called to add the chosen sets of consequence manually. However, `add-chosen-sets-of-rule-consequences` *cannot be called if the rule is still active*. A rule which had been aborted never adds rule consequences automatically to an ABox.

Let us consider an example session which demonstrates these aspects of the API:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file
  "/home/mi.wessel/nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (process-tuple-at-a-time)
> :okay-processing-tuple-at-a-time

? (describe-query-processing-mode)
> ((:creating-substrates-of-type :racer-dummy-substrate)
   :using- $\$?$ -prefix-for-injective-variables
   :check-abox-consistency
   :same-as-is-syntactic-same-as
   :query-optimization-enabled
   :optimizer-uses-cardinality-heuristics
   :automatically-adding-rule-consequences
   :warnings
   :complete-mode
   :mode-3
   :tuple-at-a-time-mode
   :eager
   :deliver-kb-has-changed-warning-tokens)

;;; Please note that ":automatically-adding-rule-consequences" is enabled

? (prepare-abox-rule
  (?x woman)
  ((instance ?x mother)
   (related ?x (new-ind child-of ?x) has-child))))
> (:rule-1 :ready-to-run)

? (ready-rules)
> (:rule-1)

? (applicable-rules)
> (:rule-1)

? (get-next-set-of-rule-consequences :rule-1)
> :not-found

? (execute-rule :rule-1)
> (:rule-1 :running)
```

```
? (active-rules)
> (:rule-1)

? (get-next-set-of-rule-consequences :rule-1)
> ((instance alice mother) (related alice child-of-alice has-child))

? (get-current-set-of-rule-consequences :rule-1)
> ((instance alice mother) (related alice child-of-alice has-child))

? (choose-current-set-of-rule-consequences :rule-1)
> (((instance alice mother)
      (related alice child-of-alice has-child)))

? (get-next-set-of-rule-consequences :rule-1)
> ((instance betty mother) (related betty child-of-betty has-child))

? (get-next-set-of-rule-consequences :rule-1)
> ((instance eve mother) (related eve child-of-eve has-child))

? (get-next-set-of-rule-consequences :rule-1)
> ((instance doris mother) (related doris child-of-doris has-child))

? (get-next-set-of-rule-consequences :rule-1)
> :exhausted

? (active-rules)
> NIL

? (processed-rules)
> (:rule-1)

? (all-role-assertions)
> (((alice child-of-alice) has-child)
    ((eve doris) has-sister)
    ((doris eve) has-sister)
    ((charles betty) has-sibling)
    ((betty eve) has-child)
    ((betty doris) has-child)
    ((alice charles) has-child)
    ((alice betty) has-child))

? (applicable-rules)
> (:rule-1)

? (execute-or-reexecute-all-queries :rule-1)
```

```
> NIL

? (execute-or-reexecute-rule :rule-1)
> (:rule-1 :running)

? (execute-rule :rule-1)
> :not-found

? (get-next-set-of-rule-consequences :rule-1)
> ((instance betty mother) (related betty child-of-betty has-child))

? (applicable-rules)
> NIL

? (choose-current-set-of-rule-consequences :rule-1)
> (((instance betty mother)
      (related betty child-of-betty has-child)))

? (get-next-set-of-rule-consequences :rule-1)
> ((instance alice mother) (related alice child-of-alice has-child))

? (add-chosen-sets-of-rule-consequences :rule-1)
*** NRQL WARNING: DENIED DUE TO DEADLOCK PREVENTION!
    THE FOLLOWING RULES WILL NOT TERMINATE AUTOMATICALLY,
    SINCE THEY HAVE BEEN STARTED IN LAZY INCREMENTAL MODE:
    (rule-1).
> :denied-due-to-deadlock-prevention

? (get-next-n-remaining-sets-of-rule-consequences :rule-1)
> (((instance eve mother) (related eve child-of-eve has-child))
    ((instance doris mother)
     (related doris child-of-doris has-child)))

? (get-next-n-remaining-sets-of-rule-consequences :rule-1)
> (:warning-kb-has-changed)

? (get-next-n-remaining-sets-of-rule-consequences :rule-1)
> NIL

? (all-role-assertions)
> (((betty child-of-betty) has-child)
    ((alice child-of-alice) has-child)
    ((eve doris) has-sister)
    ((doris eve) has-sister)
    ((charles betty) has-sibling)
    ((betty eve) has-child))
```

```
((betty doris) has-child)
((alice charles) has-child)
((alice betty) has-child))
```

6.2.5 Configuring the Degree of nRQL Completeness

So far we have only used the complete modi of nRQL. In these modi, nRQL uses the basic ABox querying functions of RacerPro in order to achieve completeness. However, on very big or pathological KBs, this mode might fail to scale. In such cases you might consider trying the *incomplete modi* of nRQL. For ABoxes / KBs which have a simple structure, for example, ABoxes created from RDF(S) documents, the incomplete modi will be complete.

Many ABoxes simply contain *data*; those ABoxes are basically sets of *ground terms* and thus contain no disjunctions, no role qualifications, etc. Bulk data from relational databases would qualify for the incomplete modi. Note that in RDMS, usually the Closed World Assumption (CWA) is made, in contrast to RacerPro, which uses the Open World Assumption (OWA). However, note that nRQL offers negation as failure, and thus, some form of CWA negation is available.

In order to query such a potentially huge but simply structured ABox, nRQL can be configured in such a way that it uses only the *syntactic, told information from the ABox* for query answering, and exploits also information from the taxonomy to achieve better completeness. Thus, query answering can be implemented using plain relational lookup techniques, and only shallow reasoning is required. As mentioned, is also possible to take TBox information into account; however, in this case, the TBox should be *classified* first in order to get as many answers as possible.

6.2.5.1 The nRQL Modi in Detail

The nRQL engine offers the following degrees of completeness for query answering, which are now discussed in order of increasing completeness:

- *Told information querying (Mode 0)*. In this mode, only the *told syntactic information* from an ABox (the ABox assertions) will be used for query answering

For example, if an ABox contains the assertion `(instance betty mother)`, then `(retrieve (?x) (?x mother))` will correctly return `((?x betty))`, but the query `(retrieve (?x) (?x woman))` will fail, since this information is not explicitly given in the ABox.

If the ABox contains 2 assertions `(instance i a)` and `(instance i b)`, then both `(retrieve (?x) (?x a))` and `(retrieve (?x) (?x b))` return `((?x i))`, but the query `(retrieve (?x) (?x (and a b)))` fails as well. However, `(retrieve (?x) (and (?x a) (?x b)))` will succeed. The mode is therefore severely incomplete, but sufficient for data. You could also write a nRQL rule to augment the ABox syntactically: `(firerule (and (?x a) (?x b)) ((instance ?x (and a b))))`.

In order to achieve better completeness, conjunctions in the concept instance assertions are broken up and made syntactically explicit: If an ABox concept assertion such

as `(instance i (and a b))` is found, then also `(instance i a)` and `(instance i b)` will be added. Please note that these concept assertions are only “virtually” added; the original ABox is not modified. Thus, `(retrieve (?x) (?x a))` and `(retrieve (?x) (?x b))` will succeed.

Regarding the *relational structure* of the ABox which is spawned by the role assertions, nRQL is only complete in this mode if there are no **at-most** number restrictions present in the ABox, and if there are no features used.

Thus, nRQL will be complete up to the description logic \mathcal{ALCHI}_{R+} concerning the relational structure of the ABox, even in this incomplete mode. This means, the effects of role hierarchies, transitively closed roles and inverse roles are completely captured.

If you consider to query a plain relational structure (and you don’t need a TBox) with labeled nodes (for example, a big graph representing a public transportation network with a few 10.000 nodes or the like), then this is the nRQL setting you will need and which is “complete enough” for your application. You could also use the data substrate.

In order to enable this mode, use `(set-nrql-mode 0)`.

- Told information querying *plus taxonomic information for atomic concepts / concept names (Mode 1)*. In addition to the “virtual” concept membership assertions already added in Mode 0, this mode considers the concept assertions of the form `(instance i C)` for *atomic concepts* C, and adds implied concept assertions of the form `(instance i D)`, if D is implied by C w.r.t. the TBox. In order to find appropriate D to add, nRQL classifies the TBox (computes the taxonomy), and uses the `atomic-concept-synonyms` and `atomic-concept-ancestors` of C for D.

For example, if the ABox contains the concept assertion `(instance betty mother)`, then also `(instance betty woman)` will be added, since `woman` is a member of `(atomic-concept-ancestors mother)`. Moreover, conjunctions are broken up, like in Mode 0, and the same process is applied recursively until no more assertions can be added. The resulting “upward saturated ABox” is then queried like in the previous setting.

Note that, in contrast to the previous setting, `(retrieve (?x) (?x woman))` now correctly returns `((?x betty))`. However, this mode will still fail for the query `(retrieve (?x) (?x (and woman human)))`. This is what the next mode achieves.

In order to enable this mode, use `(set-nrql-mode 1)`.

- Told information querying *plus taxonomic information for all concepts (Mode 2)*. This mode is like Mode 1, but now, the atomic concept synonyms and ancestors will also be computed for *arbitrary* (and not only atomic) concepts C in concept membership assertions `(instance i C)`.

Thus, if the ABox contains, `(instance betty (or woman mother))`, then RacerPro will compute the equivalent concepts and concept ancestors of the compound, non-atomic concept `(or woman mother)`. This will result in assertions such as `(instance betty woman)` and `(instance betty human)`, etc.

But carefully: For big ABoxes containing many different concept expressions, this process might take a long time, since each of this concept expressions must be classified into the TBox.

In order to enable this mode, use `(set-nrql-mode 2)`.

- Complete RacerPro ABox reasoning (Mode 3). We don't need to discuss this mode, since it is the default mode and has been discussed all the time in this manual.

In order to enable this mode, use `(set-nrql-mode 3)`.

- See below for an explanation of `(set-nrql-mode 4)` and `(set-nrql-mode 5)`.
- If you use `(set-nrql-mode 6)`, then this mode behaves like `(set-nrql-mode 3)`. Thus, mode 6 is a complete mode. However, mode 6 might be faster than mode 3. See below for an explanation.

Please note that the incomplete modi will only achieve a certain degree of completeness if you restrict yourself to concept query atoms which use only concept names instead of arbitrary concept expressions. If you insist on using complex concepts in the concept query atoms of your nRQL queries, then the complete nRQL modi will be needed.

6.2.5.2 An Example Demonstrating the Different nRQL Modi

The following log demonstrates the different nRQL modi and the degrees of completeness they achieve:

```
? (full-reset)
> :okay-full-reset

? (instance alice grandmother)
> :OKAY

? (instance betty (and woman (some has-child top)))
> :OKAY

? (implies grandmother mother)
> :OKAY

? (define-concept mother (and woman (some has-child top)))
> :OKAY

? (set-nrql-mode 0)
> :okay-mode-0

? (retrieve (?x) (?x woman))
> (((?x betty)))
```

```

? (set-nrql-mode 1)
> :okay-mode-1

? (retrieve (?x) (?x woman))
> (((?x alice)) ((?x betty)))

? (retrieve (?x) (?x mother))
> (((?x alice)))

? (set-nrql-mode 3)
> :okay-mode-3

? (retrieve (?x) (?x mother))
> (((?x betty)) ((?x alice)))

```

6.2.5.3 The Two-Phase Query Processing Modi

The so-called *two-phase query processing modi* are special lazy incremental (tuple at a time) modi. The modi 4, 5 and 6 are two-phase processing modi. All modi are complete.

Mode 4 and 5 are lazy incremental modi, whereas mode 6 is (like mode 3) a set-a-t-time mode.

Let us describe the modi 4 and 5 first. If nRQL is used in these modi, then delivery of tuples will be arranged in *two phases*:

- In *phase one*, the so-called *cheap tuples* will be delivered,
- followed by the *expensive tuples* in *phase two*.

The idea is simple: The tuples from phase one (the cheap tuples) will be provided by the incomplete modi just discussed. In mode 4, mode 1 is used, and in mode 5, mode 2. The remaining (expensive tuples) delivered in phase 2 will be provided by the complete mode 3. Thus, all two-phase processing modi are complete. Note that RacerPro's basic ABox retrieval functions are only called in phase two.

6.2.5.3.1 The Warning Token nRQL can be advised to deliver a so-called *phase-two-starts warning token* before phase two starts, informing the application or user that the next call to `get-next-tuple` will invoke RacerPro's ABox retrieval functions. However, delivery of this warning token is optional. Please refer to the Reference Manual for more details.

6.2.5.3.2 Mode 4 and 5 Cannot be Used for Queries with NEG Please note that only queries that do not make use of the NAF-negation `neg` can take advantage of the two-phase query processing modi, because the complement set constructed by `neg` will be too big if `neg` is applied to a set which contains *less* tuples (than the complete answer). Thus, the complement will contain *wrong* (incorrect) tuples.

If nRQL is in mode 4, 5, or 6, and a containing `neg` is posed, then mode 3 is used. Please note that mode 3 can be used in set-at-a-time as well as tuple-at-a-time mode.

6.2.5.3.3 Two-Phase Modi Example Session The following log demonstrates the nRQL modi 4 and 6. Let us first set up the small example KB:

```
? (full-reset)
> :okay-full-reset

? (implies mother woman)
> :OKAY

? (implies grandmother mother)
> :OKAY

? (define-concept mother (and woman (some has-child top)))
> :OKAY

? (instance alice grandmother)
> :OKAY

? (instance betty (or mother grandmother))
> :OKAY

? (related betty eve has-child)
> :OKAY

? (instance betty (all has-child woman))
> :OKAY
```

Let us switch to mode 4 and retrieve the instances of **woman**:

```
? (set-nrql-mode 4)
> :okay-mode-4

? (retrieve (?x) (?x woman))
> (:query-3 :running)

? (get-next-tuple :last)
> ((?x alice))

? (get-next-tuple :last)
> :warning-expensive-phase-two-starts

? (get-next-tuple :last)
> ((?x betty))

? (get-next-tuple :last)
> ((?x eve))
```

Now let us switch to mode 5. Please observe that phase one is no able to produce one more tuple (before `:warning-expensive-phase-two-starts` is delivered):

```
? (set-nrql-mode 5)
> :okay-mode-5

? (retrieve (?x) (?x woman))
> (:query-4 :running)

? (get-next-tuple :last)
> ((?x alice))

? (get-next-tuple :last)
> ((?x betty))

? (get-next-tuple :last)
> :warning-expensive-phase-two-starts

? (get-next-tuple :last)
> ((?x eve))
```

6.2.5.4 Mode 6 vs. Mode 3

Mode 6 is like mode 3 in set-at-a-time mode. However, unlike mode 3, in mode 6 nRQL will exploit the two-phase query processing schema. Thus, the more expensive ABox retrieval functions are avoided whenever possible. However, since phase one is incomplete, it may be the case that no tuples can be computed in phase one at all, thus leaving all the work for phase two. Nevertheless it is worth trying to use mode 6 whenever you encounter performance problems in mode 3.

6.2.6 Automatic Deadlock Prevention

Certain types of queries (or rules) must temporary change the queried ABox in order to be answered, i.e., ABox assertions must be added. This is, for example, the case if classical negated roles are used in role query atoms. If more than one query (or rule) is currently active which reference the same ABox, then these different queries must be isolated from one another. nRQL uses locking techniques to achieve this isolation: The queried ABox is thus locked, so other queries cannot access this ABox as long as the query is active. This means that nRQL may not permit the execution of other queries on that ABox. The query has to wait until all active queries querying that ABox have been terminated.

However, there is a problem: In lazy mode, these queries may not terminate automatically. If such a situation is encountered, nRQL will deny starting yet another query and print out a warning message:

```
? (full-reset)
```

```
> :okay-full-reset

? (define-primitive-role r :range (not c))
> :OKAY

? (instance i top)
> :OKAY

? (instance j c)
> :OKAY

? (instance k c)
> :OKAY

? (process-tuple-at-a-time)
> :okay-processing-tuple-at-a-time

? (enable-lazy-tuple-computation)
> :okay-lazy-mode-enabled

? (retrieve (?x ?y) (?x ?y (not r)))
> (:query-1 :running)

? (get-next-tuple :last)
> ((?x k) (?y k))

? (get-next-tuple :last)
> :warning-kb-has-changed

? (active-queries)
> (:query-1)

? (waiting-queries)
> (:query-1)

? (running-queries)
> NIL

? (retrieve (?x) (?x c))
*** NRQL WARNING: DENIED DUE TO DEADLOCK PREVENTION!
    THE FOLLOWING QUERIES WILL NOT TERMINATE AUTOMATICALLY,
    SINCE THEY HAVE BEEN STARTED IN LAZY INCREMENTAL MODE:
    (query-1).
> :denied-due-to-deadlock-prevention

? (get-answer :last)
```

```
> (((?x k) (?y k))
    ((?x k) (?y j))
    ((?x j) (?y k))
    ((?x j) (?y j))
    ((?x i) (?y k))
    ((?x i) (?y j)))

? (retrieve (?x) (?x c))
> (:query-2 :running)

? (get-answer :last)
> (((?x j)) ((?x k)))
```

6.2.7 Reasoning with Queries

The services described in this Section should be considered as non-essential add-ons and are still experimental.

6.2.7.1 Reporting Inconsistent Queries

If you are wondering why a certain query never returns any tuples and you think it should, you should consider using the consistency checking service for queries. An inconsistent query must necessarily produce an empty answer on all ABoxes; thus, CPU cycles can be saved if such queries are recognized in advance, before executing them.

Even though the reasoning mechanism offered by nRQL for queries are incomplete (and yet experimental), they are still useful. The consistency checking service is sound and complete for queries which do contain *at most one occurrence of* **neg**.

The idea of the algorithm is simple: The query is first transformed into DNF. Then, for each disjunct, a canonical ABox is constructed, witnessing the satisfiability of the query. Note that each disjunct is a grounded conjunctive query.

Please note that the ABox is not taken into account; e.g., even if **betty** is an instance of **woman**, and **woman** and **man** are disjoint, then the query (**retrieve** () (**betty man**)) is *not* recognized as inconsistent. The reason is that (in principle) there *could be* an ABox in which **betty** is indeed a **man**.

Here is an example session, demonstrating the utility of nRQL's reasoning facilities:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file
  "/home/mi.wessel/nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (report-inconsistent-queries-and-rules)
> :okay-reporting-inconsistent-queries-and-rules

? (retrieve (?x) (and (?x woman) (?x man)))
> :inconsistent

? (describe-query :last)
> (:query-1
   (:accurate :processed)
   (retrieve (?x) (and (?x woman) (?x man)) :abox smith-family))

? (reexecute-query :last)
> :inconsistent

? (prepare-abox-query
```



```

    (?x)
    (and
      (?x woman)
      (?x ?y has-child)
      (?y (all has-parent (all has-descendant woman)))
      (?y ?z has-child)
      (?z ?u has-child)
      (?u uncle)))
*** NRQL WARNING: query-2 IS INCONSISTENT
> (:query-2 :ready-to-run)

? (prepare-abox-query
  (?x)
  (and
    (?x woman)
    (?x ?y has-child)
    (?y (all descendant-of (all has-descendant woman)))
    (?y ?z has-child)
    (?z ?u has-child)
    (?u uncle)))
> (:query-3 :ready-to-run)

? (define-primitive-role descendant-of :inverse has-descendant)
> :OKAY

? (prepare-abox-query
  (?x)
  (and
    (?x woman)
    (?x ?y has-child)
    (?y (all descendant-of (all has-descendant woman)))
    (?y ?z has-child)
    (?z ?u has-child)
    (?u uncle)))
> (:query-4 :ready-to-run)

? (execute-query :last)
> :inconsistent

? (retrieve
  (?x)
  (union
    (and (?x man) (?x woman))
    (and (?x grandmother) (?x uncle))))
*** NRQL WARNING: query-5 IS INCONSISTENT
> :inconsistent

```

```
? (retrieve nil (and (?x grandmother) (neg (?x woman))))
*** NRQL WARNING: query-6 IS INCONSISTENT
> :inconsistent

? (retrieve nil (and (?x grandmother) (?x (not woman))))
*** NRQL WARNING: query-7 IS INCONSISTENT
> :inconsistent

? (concept-satisfiable? (and woman man))
> NIL

? (prepare-abox-query nil (betty man))
> (:query-8 :ready-to-run)

? (query-consistent-p :last)
> t
```

6.2.7.2 Reporting Inconsistent Rules

The described reasoning services are also provided for rule. In this case, also the consequence of the rule is taken into account:

```
? (full-reset)
> :okay-full-reset

? (report-inconsistent-queries-and-rules)
> :okay-reporting-inconsistent-queries-and-rules

? (prepare-abox-rule
  (and (?x c) (?y d))
  ((instance ?x d)
   (related ?x ?y r)
   (instance ?y (all (inv r) (not d)))))
*** NRQL WARNING: rule-1 IS INCONSISTENT
> (:rule-1 :ready-to-run)

? (prepare-abox-rule (and (?x c) (?x (not c))) ((instance ?x d)))
*** NRQL WARNING: rule-2 IS INCONSISTENT
> (:rule-2 :ready-to-run)

? (prepare-abox-rule
  (and (?x c) (?x d))
  ((instance ?x (not (and c d)))))
*** NRQL WARNING: rule-3 IS INCONSISTENT
> (:rule-3 :ready-to-run)
```

6.2.7.3 Checking for Query Entailment

The function `query-entails-p` checks whether one query is *more specific* than another one. This is also known as *query subsumption*. The function `query-equivalent-p` checks whether both queries *mutually entail (subsume) each other*. The query entailment check is complete for queries which do not contain `neg`.

The query entailment check is reduced to a series of query consistency checks. The query `(and a11 ... a1n)` entails the query `(and a21 ... a2m)` iff all queries `(and a11 ... a1n (neg a21))` are inconsistent; moreover, if the queries contain `union`, then they are brought into DNF, and each combination of disjuncts is checked accordingly. Note that we have just stated that the query inconsistency checker is complete for queries containing at most one `neg`. Thus, we claim that the entailment check is complete for queries which contain *no negs*.

Moreover, it should be noted that the entailment check implemented here only considers the bodies of the queries; the head is irrelevant. What matters for query subsumption is thus not a subsumption / set inclusion relationship between the query results, but between the tuple sets denoted by the *query bodies*. Thus, the query `(retrieve (?x) (?x woman))` is entailed by `(retrieve () (?x grandmother))`.

Obviously, there is also *variable renaming issue*: Should the queries `(?x woman)` and `(?y woman)` be considered as equivalent? And what about `(and (?x man) (?y woman))` vs. `(and (?y man) (?x woman))`? Are they equivalent?

In nRQL, the answer to the first question is yes, and the answer to the second question is no. The rationale is that we want to avoid having to consider an exponential (factorial) number of possible ways to rename variables. Each nRQL query body is associated with a so-called *object vector* which is a sequence / list of lexicographic ordered objects (variables and individuals) mentioned in the query body. So, the object vector of `(?x woman)` is `<?x>`, and `<?y>` for `(?y woman)`. Moreover, `<?x,?y>` for `(and (?x man) (?y woman))` and `(and (?x woman) (?y man))`.

In order to check whether the query *a* is more specific than query *b*, the variables in *b* are renamed to match the names of the variables in *a*, *according to their positions in the object vectors*. So we are *not* considering all possible ways to rename the variables, but only consider one way to rename them: We match them according to their positions in the object vectors which are lexicographically ordered. The names of the variables thus matter.

Thus, for the queries *a* = `(and (?x man) (?y woman))` and *b* = `(?a man) (?b woman)` we would rename `<?a,?b>` into `<?x,?y>` and thus see that `(and (?x man) (?y woman) (neg (?x man)))` as well as `(and (?x man) (?y woman) (neg (?y woman)))` are inconsistent. This shows that *b* entails *a* (and vice versa). However, for *b* = `(?b man) (?a woman)`, neither `(and (?x man) (?y woman) (neg (?x woman)))` nor `(and (?x man) (?y woman) (neg (?y man)))` are inconsistent. This shows that there is no entailment in this case.

Note that it is quite reasonable not to consider all possible mappings if one accepted the “typed tuple positions” perspective: Under this perspective, the object vectors denote the structure of the tuple sets denoted by the query bodies. The element positions of the tuples in these sets can be considered as “typed”, e.g., the type of the tuples in `(and (?x woman) (?y man))` is `<woman,man>`. In contrast, the types of the tuples in `(and (?b woman) (?a`

man)) is <man,woman>. So it is obvious that these two sets can never be in a subsumption / subset relationship to one another. Note that this is very similar to the *covariance paradigm* used in programming languages.

Let us give some practical examples:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file
  "/home/mi.wessel/nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (prepare-abox-query nil (?x woman))
> (:query-1 :ready-to-run)

? (prepare-abox-query nil (?x grandmother))
> (:query-2 :ready-to-run)

? (query-entails-p :query-2 :query-1)
> t

? (query-entails-p :query-1 :query-2)
> NIL

? (prepare-abox-query nil (and (?x woman) (?x ?y has-child)))
> (:query-5 :ready-to-run)

? (prepare-abox-query nil (?z mother))
> (:query-6 :ready-to-run)

? (query-entails-p :query-5 :query-6)
> t

? (query-entails-p :query-6 :query-5)
> NIL

;;; Please note that :query-6 and :query-5 are *not*
;;; equivalent. :Query-5 is stronger (more specific),
;;; since it requires the existence of some has-child filler
;;; in the ABox, whereas for :query-6 this child most
;;; not be explicitly present in the ABox

;;; However, an alternative definition of "mother" will be
;;; recognized as equivalent to the :query-6
;;; (not that
```

```
? (prepare-abox-query
  nil
  (and (?x (and woman (some has-child top)))))
> (:query-9 :ready-to-run)

? (query-entails-p :query-9 :query-6)
> t

? (query-entails-p :query-6 :query-9)
> t

;;;
;;;
;;;

? (prepare-abox-query (?x) (?x mother))
> (:query-12 :ready-to-run)

? (prepare-abox-query nil (?x grandmother))
> (:query-13 :ready-to-run)

? (query-entails-p :query-13 :query-12)
> t

? (prepare-abox-query nil (and (?x woman) (?y man)))
> (:query-16 :ready-to-run)

? (prepare-abox-query nil (and (?a woman) (?b man)))
> (:query-17 :ready-to-run)

? (prepare-abox-query nil (and (?b woman) (?a man)))
> (:query-18 :ready-to-run)

? (query-equivalent-p :query-16 :query-17)
> t

? (query-equivalent-p :query-16 :query-18)
> NIL

? (prepare-abox-query
  nil
  (and (?a mother) (?b uncle) (?c grandmother)))
> (:query-22 :ready-to-run)

? (query-entails-p :query-22 :query-16)
> t
```

```
? (query-entails-p :query-16 :query-22)
> NIL
```

6.2.8 The Query Repository - The QBox

The query entailment check (see previous Section) is used for the maintenance of a *query repository*. This repository is also called the *QBox*. If enabled, a QBox is maintained for each queried ABox. The QBox is created on demand (if needed). The QBox serves as a *hierarchical cache*. The service is still experimental and should be considered an non-essential “add on”.

For each new query to be answered, its set of *most specific subsumers* as well as its *most general subsumes* are computed. This process is called *query classification*. The QBox can be seen as a “taxonomy” for queries.

If a classified query is about to be executed, the cached answer sets of its parent queries (direct subsuming queries) can be utilized as *superset caches*, and the cached answer sets of its children queries (direct subsumed queries) serve as *subset caches*. In case the QBox already contains an equivalent query with a cached answer set, this set will be returned immediately.

The query entailment check which is used for query classification is slightly stricter than the query entailment check just discussed. Here we require in addition that the *object vectors* must have the same length. The rationale is that we want to exploit and directly reuse the cached tuples; the tuples from the caches must therefore have the same arity as the requested tuples.

Please consider the following example session to get familiar with the QBox facility:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file
  "/home/mi.wessel/nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (enable-query-repository)
> :okay-query-repository-enabled

? (retrieve (?x) (?x top))
> (((?x alice)) ((?x betty)) ((?x charles)) ((?x doris)) ((?x eve)))

? (retrieve (?a) (?a human))
> (((?a eve)) ((?a doris)) ((?a charles)) ((?a betty)) ((?a alice)))

? (retrieve (?b) (?b woman))
> (((?b alice)) ((?b betty)) ((?b doris)) ((?b eve)))

? (retrieve (?y) (and (?x woman) (?x ?y has-child) (?y human)))
> (((?y doris)) ((?y eve)) ((?y betty)) ((?y charles)))

? (show-qbox-for-abox)
```

```

;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; 0:master-top-query
;;; |---4:query-9
;;; | \___0:master-bottom-query
;;; \___1:query-1
;;; \___2:query-2 = (SUBQUERY-3-OF-query-9)
;;; \___3:query-5 = (SUBQUERY-1-OF-query-9)
;;; \___0:master-bottom-query
> :see-output-on-stdout

? (show-qbox-for-abox smith-family t)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; master-top-query
;;; |---(and (?x-ano1 woman) (?x-ano1 ?y has-child) (?y human))
;;; | \___master-bottom-query
;;; \___(?x top)
;;; \___(?a human)
;;; \___(?b woman)
;;; \___master-bottom-query
> :see-output-on-stdout

? (query-equivalents :last)
> NIL

? (retrieve (?y) (and (?x mother) (?y ?x has-parent) (?y man)))
> (((?y charles)))

? (show-qbox-for-abox smith-family t)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; master-top-query
;;; |---(and (?x-ano1 woman) (?x-ano1 ?y has-child) (?y human))
;;; | \___(and (?x-ano1 mother) (?y ?x-ano1 has-parent) (?y man))
;;; | \___master-bottom-query
;;; \___(?x top)
;;; \___(?a human)
;;; \___(?b woman)
;;; \___master-bottom-query
> :see-output-on-stdout

? (query-children :last)

```



```

> (master-bottom-query)

? (show-qbox-for-abox smith-family nil)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; 0:master-top-query
;;; |---4:query-9
;;; | \___5:query-43
;;; | \___0:master-bottom-query
;;; \___1:query-1
;;; \___2:query-2 = (SUBQUERY-3-OF-query-9)
;;; \___3:query-5 = (SUBQUERY-1-OF-query-9)
;;; \___0:master-bottom-query
17 > :see-output-on-stdout

? (query-children :query-9)
> (:query-43)

? (query-parents :query-43)
> (:query-9)

? (retrieve nil (and (?a mother) (?a ?b has-child) (?b man)))
> t

? (show-qbox-for-abox smith-family t)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; master-top-query
;;; |---(and (?x-ano1 woman) (?x-ano1 ?y has-child) (?y human))
;;; | \___(and (?x-ano1 mother) (?y ?x-ano1 has-parent) (?y man))
;;; | \___master-bottom-query
;;; \___(?x top)
;;; \___(?a human)
;;; \___(?b woman)
;;; \___master-bottom-query
> :see-output-on-stdout

```

```

? (show-qbox-for-abox smith-family nil)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; 0:master-top-query
;;; |---4:query-9
;;; |   \___5:query-43 = (query-56)
;;; |   \___0:master-bottom-query
;;; |   \___1:query-1
;;; |   \___2:query-2 = (SUBQUERY-3-OF-query-9)
;;; |   \___3:query-5 = (SUBQUERY-1-OF-query-9)
;;; |   \___0:master-bottom-query
> :see-output-on-stdout

? (query-equivalents :last)
> (:query-43)

? (query-equivalents :query-56)
> (:query-43)

? (query-equivalents :query-43)
> (:query-56)

? (retrieve (?a) (?a mother))
> (((?a betty)) ((?a alice)))

? (show-qbox-for-abox smith-family nil)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; 0:master-top-query
;;; |---4:query-9
;;; |   \___5:query-43 = (query-56)
;;; |   \___0:master-bottom-query
;;; |   \___1:query-1
;;; |   \___2:query-2 = (SUBQUERY-3-OF-query-9)
;;; |   \___3:query-5 = (SUBQUERY-1-OF-query-9)
;;; |   \___6:query-88
;;; |   \___0:master-bottom-query
> :see-output-on-stdout

```

```
? (show-qbox-for-abox smith-family t)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; master-top-query
;;; |---(and (?x-ano1 woman) (?x-ano1 ?y has-child) (?y human))
;;; |   \___(and (?x-ano1 mother) (?y ?x-ano1 has-parent) (?y man))
;;; |       \___master-bottom-query
;;; \___(?x top)
;;;       \___(?a human)
;;;           \___(?b woman)
;;;               \___(?a mother)
;;;                   \___master-bottom-query
> :see-output-on-stdout
```

6.2.9 The Query Realizer

Query realization can be called a *semantic query optimization technique* which will enhance the amount of information that is available for the heuristic, cost-based query optimizer. Moreover, the added conjuncts can also enhance the overall query answering speed, since the search process computing the answer tuples is somewhat “better informed” if more information regarding the query is made explicit by means of inference. However, the service is still experimental.

The query realization process is similar to an ABox realization process: *logically implied conjuncts are added*. The *realized query* is semantically equivalent to the original query.

Let us consider the following example session:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file
> :OKAY

? (enable-query-realization)
> :okay-query-realization-enabled

? (retrieve (?x) (and (?x woman) (?x ?y has-child) (?y human)))
> (((?x betty)) ((?x alice)))

? (describe-query :last)
> (:query-1
  (:accurate :processed)
  (retrieve
   (?x)
   (and
    (?y-ano1 (and human person))
    (?x ?y-ano1 has-child)
    (?x (and woman mother)))
   :abox
   smith-family))

;;; DESCRIBE-QUERY outputs, by default,
;;; always the internally rewritten query
;;; Note that nRQL has deduced that
;;; ?x is in fact a mother, not only a woman!

;;; You can also get the original query by
;;; providing the optional nil argument:

? (describe-query :last nil)
> (:query-1
```

```

    (:accurate :processed)
    (retrieve
      (?x)
      (and (?x woman) (?x ?y has-child) (?y human))
      :abox
      smith-family))

? (retrieve
  (?x)
  (and
    (?x woman)
    (?x (all has-descendant man))
    (?x ?y has-child)
    (?y human)))
> NIL

? (describe-query :last)
> (:query-2
  (:accurate :processed)
  (retrieve
    (?x)
    (and
      (?x (and woman (all has-descendant man) mother))
      (?x ?y-ano1 has-child)
      (?y-ano1 (and human man)))
    :abox
    smith-family))

;;; Note that nRQL has deduced that ?y-ano1 / ?y must be a man,
;;; and that ?x is a mother

```

Since a realized query contains more syntactically explicit information, the search for answer tuples can eventually be more constrained and thus eliminate candidate bindings for variables which otherwise would be considered. So-called *thrashing* (a term borrowed from constraint programming) is thus minimized. However, candidate generation for variable bindings will also be more expensive, since the concepts used in concept query atoms will be more complex. Thus, it depends on the KB and the query if the realized query will perform better than the non-realized one.

6.2.10 The nRQL Persistency Facility

nRQL maintains a so-called *substrate* data structure for each RacerPro ABox which was queried. This ABox-associated substrate data structure contains index structures and other auxiliaries needed for query answering. Computation of these index structures is triggered on demand (if needed). The first time a query is posed to a certain ABox which has not seen a nRQL query before, nRQL creates a corresponding substrate for the ABox as well as all its index structures. Thus, answering the first query to an ABox takes considerably longer than subsequent queries to the same ABox.

In order to avoid recomputation of these index structures, it is possible to dump (store) the complete substrate data structure into a file. Moreover, also the *data substrate* or *RCC substrate* may contain nodes which have been created by hand and which must be saved somehow. The API functions `store-substrate-for-abox` and `store-substrate-for-current-abox` are provided for this purpose. You can also dump all substrates: `store-all-substrates`. Restoring substrates is easy as well: `restore-substrate`, `restore-all-substrates`. Please consult the Reference Manual for more details on these API functions.

Dumping a substrate will always automatically dump the associated ABox and TBox as well. Moreover, there can be *defined queries* (see Section 6.1.5) associated with a TBox, the substrate can also have an associated QBox, etc. All these data structures are also saved.

However, it is not possible to resurrect the queries from the QBox - they merely serve as caches. Thus, you cannot call `reexecute-query` on a query Id which you see in a restored QBox.

This list of queries, processed queries, active queries etc. are NOT saved. If you really want to save queries into a file, we ask you to *define* these queries (see Section 6.1.5). Restored definitions can be reused. See the example below.

We have also discussed in Section 6.1.8 that there are *specialized types of substrates* available, tailored for special representation tasks. For example, the data substrate, or the RCC substrate. These substrates can be saved as well, and thus, the created hybrid representation can be preserved.

The following session demonstrates the utility of the nRQL persistency facility:

```
? (full-reset)
> :okay-full-reset

? (racer-read-file
  "/home/mi.wessel/nrql-user-guide-examples/family-no-signature.racer")
> :OKAY

? (enable-query-repository)
> :okay-query-repository-enabled

? (retrieve (?x) (?x woman))
> (((?x alice)) ((?x betty)) ((?x eve)) ((?x doris)))
```

```

? (defquery
  mother-with-son
  (?x ?y)
  (and (?x woman) (?x ?y has-child) (?y man)))
> mother-with-son

? (show-qbox-for-abox)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; 0:master-top-query
;;; \___1:query-1
;;; \___0:master-bottom-query
> :see-output-on-stdout

? (describe-all-definitions)
> ((defquery
  mother-with-son
  (?x ?y)
  (and (?x woman) (?x ?y has-child) (?y man))))

? (store-substrate-for-abox "test-substrate-image")
> :done

```

The substrate can now, for example, be restored on a different RacerPro server:

```

? (restore-substrate "test-substrate-image")
> smith-family

? (all-aboxes)
> (default smith-family)

? (all-individuals)
> (eve doris charles betty alice)

? (all-atomic-concepts)
> (top
  bottom
  father
  uncle
  woman
  human
  male
  sister
  brother

```

```
grandmother
man
aunt
female
person
mother
parent)

? (describe-all-definitions)
> ((defquery
    mother-with-son
    (?x ?y)
    (and (?x woman) (?x ?y has-child) (?y man))))

? (show-qbox-for-abox)
;;;
;;; QBOX FOR racer-dummy-substrate FOR ABOX smith-family
;;;
;;; nil:master-top-query
;;; \__nil:query-1
;;; \__nil:master-bottom-query
> :see-output-on-stdout
```


Chapter 7

Outlook

Future releases of RacerPro will provide:

- Support for complete reasoning on $\mathcal{SHOIQ}(\mathcal{D}_n)^-$ knowledge bases (\mathcal{SHIQ} + nominals in concept terms, arithmetic concrete domains with n-ary predicates and without features chains) [13]. New optimization techniques for nominals have to be developed.
- Instead of role hierarchies, more expressive role axioms (so-called acyclic role axioms) can be supported and are useful in practice. The description logic is called \mathcal{SRIQ} [13]. Future version of RacerPro will support acyclic role axioms. New optimization techniques have to be developed for this language feature.
- Feature chains for ω -admissible concrete domains (such as \mathcal{RCC} or pointizable \mathcal{ALLEN}) [16].
- Feature chain equality for $\mathcal{ALCF}(\mathcal{D})$.
- SWRL rules with first-order semantics.
- Persistency for ontologies, persistency for A-boxes, database access.
- Proper algorithmic support for incremental A-box changes.

The order in this list says nothing about priority, and this list is probably not complete.

Appendix A

Another Family Knowledge Base

In this section we present another family knowledge base (see the file `family-2.racer` in the examples folder).

```
(in-knowledge-base family)

(define-primitive-role descendants :transitive t)

(define-primitive-role children :parents (descendants))

(implies (and male female) *bottom*)
(equivalent man (and male human))
(equivalent woman (and female human))
(equivalent parent (at-least 1 children))
(equivalent grandparent (some children parent))
(equivalent mother (and woman parent))
(equivalent father (and man parent))
(implies (some descendants human) human)
(implies human (all descendants human))
(equivalent father-having-only-male-children (and father human (all children male)))
(equivalent father-having-only-sons (and man
                                     (at-least 1 children)
                                     (all children man)))
(equivalent grandpa (and male (some children (and parent human))))
(equivalent great-grandpa (and male
                                   (some children (some children (and parent human)))))

(instance john male)
(instance mary female)
(related john james children)
(related mary james children)
(instance james (and human male))
```

```
(instance john (at-most 1 children))
```

```
(individual-direct-types john)
```

```
(individual-direct-types mary)
```

```
(individual-direct-types james)
```

Appendix B

A Knowledge Base with Concrete Domains

In this section we present another family knowledge base with concrete domains (see the file `family-3.racer` in the examples folder).

```
(in-knowledge-base family smith-family)
```

```
(signature :atomic-concepts (human female male woman man
                             parent mother father
                             mother-having-only-female-children
                             mother-having-only-daughters
                             mother-with-children
                             mother-with-siblings
                             mother-having-only-sisters
                             grandpa great-grandpa
                             grandma great-grandma
                             aunt uncle
                             sister brother sibling
                             young-parent normal-parent old-parent
                             child teenager teenage-mother
                             young-human adult-human
                             old-human young-child
                             human-with-fever
                             seriously-ill-human
                             human-with-high-fever)
:roles ((has-descendant :domain human :range human
                        :transitive t)
        (has-child :domain parent
                    :range child
                    :parent has-descendant)
        (has-sibling :domain sibling :range sibling)
        (has-sister :range sister)
```

```
                :parent has-sibling)
      (has-brother :range brother
                  :parent has-sibling))
:features ((has-gender :range (or female male)))
:attributes ((integer has-age)
             (real temperature-fahrenheit)
             (real temperature-celsius))
:individuals (alice betty charles doris eve)
:objects (age-of-alice age-of-betty age-of-charles
          age-of-doris age-of-eve
          temperature-of-doris
          temperature-of-charles))
```

```

;; the concepts
(disjoint female male human)
(implies human (and (at-least 1 has-gender) (a has-age)))
(implies human (= temperature-fahrenheit
                  (+ (* 1.8 temperature-celsius) 32)))

(equivalent young-human (and human (max has-age 20)))
(equivalent teenager (and young-human (min has-age 10)))
(equivalent adult-human (and human (min has-age 21)))
(equivalent old-human (and human (min has-age 60)))
(equivalent woman (and human (all has-gender female)))
(equivalent man (and human (all has-gender male)))
(implies child human)
(equivalent young-child (and child (max has-age 9)))

(equivalent human-with-fever
  (and human (>= temperature-celsius 38.5)))
(equivalent seriously-ill-human
  (and human (>= temperature-celsius 42.0)))
(equivalent human-with-high-fever
  (and human (>= temperature-fahrenheit 107.5)))

(equivalent parent (at-least 1 has-child))
(equivalent young-parent (and parent (max has-age 21)))
(equivalent normal-parent (and parent (min has-age 22) (max has-age 40)))
(equivalent old-parent (and parent (min has-age 41)))
(equivalent mother (and woman parent))
(equivalent father (and man parent))
(equivalent teenage-mother (and mother (max has-age 20)))

(equivalent mother-having-only-female-children
  (and mother
    (all has-child (all has-gender (not male)))))
(equivalent mother-having-only-daughters
  (and woman
    (at-least 1 has-child)
    (all has-child woman)))
(equivalent mother-with-children
  (and mother (at-least 2 has-child)))
(equivalent grandpa (and man (some has-child parent)))
(equivalent great-grandpa
  (and man (some has-child (some has-child parent))))

```

```

(equivalent grandma (and woman (some has-child parent)))
(equivalent great-grandma
  (and woman (some has-child (some has-child parent))))
(equivalent aunt (and woman (some has-sibling parent)))
(equivalent uncle (and man (some has-sibling parent)))

(equivalent sibling (or sister brother))
(equivalent mother-with-siblings (and mother (all has-child sibling)))
(equivalent brother (and man (at-least 1 has-sibling)))
(equivalent sister (and woman (at-least 1 has-sibling)))

(implies (at-least 2 has-child) (all has-child sibling))
;(implies (some has-child sibling) (at-least 2 has-child))

(implies sibling (all (inv has-child) (and (all has-child sibling)
                                           (at-least 2 has-child))))
(equivalent mother-having-only-sisters
  (and mother
    (all has-child (and sister
                     (all has-sibling sister)))))

;; Alice is the mother of Betty and Charles
(instance alice (and woman (at-most 2 has-child)))
;; Alice's age is 45
(constrained alice age-of-alice has-age)
(constraints (equal age-of-alice 45))
(related alice betty has-child)
(related alice charles has-child)

;; Betty is mother of Doris and Eve
(instance betty (and woman (at-most 2 has-child)))
;; Betty's age is 20
(constrained betty age-of-betty has-age)
(constraints (equal age-of-betty 20))
(related betty doris has-child)
(related betty eve has-child)
(related betty charles has-sibling)
;; closing the role has-sibling for charles
(instance betty (at-most 1 has-sibling))

```



```
; Charles is the brother of Betty (and only Betty)
(instance charles brother)
;; Charles's age is 39
(constrained charles age-of-charles has-age)
(constrained charles temperature-of-charles temperature-fahrenheit)
(constraints (equal age-of-charles 39) (= temperature-of-charles 107.6))
(related charles betty has-sibling)
;; closing the role has-sibling for charles
(instance charles (at-most 1 has-sibling))

;; Doris has the sister Eve
(related doris eve has-sister)
(instance doris (at-most 1 has-sibling))
;; Doris's age is 2
(constrained doris age-of-doris has-age)
(constrained doris temperature-of-doris temperature-celsius)
(constraints (equal age-of-doris 2) (= temperature-of-doris 38.6))

;; Eve has the sister Doris
(related eve doris has-sister)
(instance eve (at-most 1 has-sibling))
;; Eve's age is 1
(constrained eve age-of-eve has-age)
(constraints (equal age-of-eve 1))
```

```
;;; some T-box queries
;; are all uncles brothers?
(concept-subsumes? brother uncle)

;; get all super-concepts of the concept mother
(concept-ancestors mother)

;; get all sub-concepts of the concept man
(concept-descendants man)

;; get all transitive roles in the T-box family
(all-transitive-roles)

;;; some A-box queries
;; Is Doris a woman?
(individual-instance? doris woman)

;; Of which concepts is Eve an instance?
(individual-types eve)

;; get all descendants of Alice
(individual-fillers alice has-descendant)

(individual-direct-types eve)

(concept-instances sister)

(describe-individual doris)

(describe-individual charles)
```

Appendix C

SWRL Example Ontology

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.daml.org/rules/proposal/swrlb.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/rules/proposal/swrl.owl"/>
  </owl:Ontology>
  <owl:Class rdf:ID="Person"/>
  <owl:ObjectProperty rdf:ID="hasChild"/>
  <owl:ObjectProperty rdf:ID="hasSibling"/>
  <swrl:Imp rdf:ID="Rule-1">
    <swrl:body>
      <swrl:AtomList>
        <rdf:first>
          <swrl:IndividualPropertyAtom>
            <swrl:argument2>
              <swrl:Variable rdf:ID="y"/>
            </swrl:argument2>
            <swrl:propertyPredicate rdf:resource="#hasChild"/>
            <swrl:argument1>
              <swrl:Variable rdf:ID="x"/>
            </swrl:argument1>
          </swrl:IndividualPropertyAtom>
```

```

</rdf:first>
<rdf:rest>
  <swrl:AtomList>
    <rdf:rest>
      <swrl:AtomList>
        <rdf:first>
          <swrl:DifferentIndividualsAtom>
            <swrl:argument2>
              <swrl:Variable rdf:ID="z"/>
            </swrl:argument2>
            <swrl:argument1 rdf:resource="#y"/>
          </swrl:DifferentIndividualsAtom>
        </rdf:first>
        <rdf:rest rdf:resource=
          "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      </swrl:AtomList>
    </rdf:rest>
    <rdf:first>
      <swrl:IndividualPropertyAtom>
        <swrl:propertyPredicate rdf:resource="#hasChild"/>
        <swrl:argument1 rdf:resource="#x"/>
        <swrl:argument2 rdf:resource="#z"/>
      </swrl:IndividualPropertyAtom>
    </rdf:first>
  </swrl:AtomList>
</rdf:rest>
</swrl:AtomList>
</swrl:body>
<swrl:head>
  <swrl:AtomList>
    <rdf:first>
      <swrl:IndividualPropertyAtom>
        <swrl:propertyPredicate rdf:resource="#hasSibling"/>
        <swrl:argument2 rdf:resource="#z"/>
        <swrl:argument1 rdf:resource="#y"/>
      </swrl:IndividualPropertyAtom>
    </rdf:first>
    <rdf:rest rdf:resource=
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
  </swrl:AtomList>
</swrl:head>
</swrl:Imp>
<Person rdf:ID="alice">
  <hasChild>
    <Person rdf:ID="betty"/>
  </hasChild>

```

```
<hasChild>
  <Person rdf:ID="charles"/>
</hasChild>
</Person>
</rdf:RDF>
```


Appendix D

LUBM benchmark

```
;;; -*- Mode: Lisp; Syntax: Ansi-Common-Lisp; Base: 10 -*-

(in-package :CL-USER)

(setf (logical-pathname-translations "racer")
      '(("lubm;**/*.*)" "c:/Ralf/LUBM/**/*.*"))) ; edit this line

(defconstant +no-of-runs-per-query+ 1)

(defmacro benchmark (universities departments no query vars)
  '(let ((t1 (get-internal-real-time))
        (number-of-answers nil)
        (id ',(intern (format nil "LUBM-QUERY-~A" no))))
    (dotimes (i +no-of-runs-per-query+)
      (racer-prepare-query ',vars ',query :id id)
      (setf number-of-answers (get-answer-size id t)))
    (let ((time (/ (- (get-internal-real-time) t1)
                    internal-time-units-per-second
                    +no-of-runs-per-query+)))
      (format t "*** Universities: ~2,D Max. Deps: ~2,D Query: ~2,A Answers: ~7,D Time: ~A~%"
              (1+ ,universities)
              (if (null ,departments)
                  :all
                  (1+ ,departments))
              ,no
              (if (null number-of-answers)
                  0
                  number-of-answers)
              (float time))))))

;;; =====

(defun load-kbs (n-universities max-n-departments)
  (let ((kb-name (owl-read-file (namestring
                                (translate-logical-pathname
                                 "racer:lubm;university;univ-bench.owl"))
                                :verbose t :kb-name 'lubm)))
```

```

(loop for i from 0 to n-universities
  as n-departments
  = (1- (length
    (directory
      (namestring
        (translate-logical-pathname
          (format nil "racer:lubm;university;university~A-*.owl" i))))))
  do
    (loop for j from 0 to (if (null max-n-departments)
      n-departments
      (min max-n-departments n-departments))
      as filename =
      (namestring
        (translate-logical-pathname
          (format nil "racer:lubm;university;university~A-~A.owl" i j)))
      do
        (owl-read-file filename
          :verbose t
          :init nil
          :kb-name kb-name
          :ignore-import t))))))

(defun prepare-lubm-data-n-universities (check-abox-consistency
  n-universities max-n-departments)
  (let ((t1a (get-internal-real-time)))

    (time (load-kbs n-universities max-n-departments))

    (let ((t1b (get-internal-real-time)))

      (let ((t2a (get-internal-real-time)))

        (format t "~%~%ABox preparation ")
        (time (prepare-abox))
        (format t "done.~%")

        (let ((t2b (get-internal-real-time)))

          (when check-abox-consistency
            (format t "~%~%ABox consistency checking... ")
            (time (abox-consistent?))
            (format t "done.~%"))

          (let ((t2c (get-internal-real-time)))
            (format t "~%Compute index structures... ")
            (time (prepare-racer-engine))
            (format t "done.~%"))

          (let ((t2d (get-internal-real-time)))
            #+Allegro (gc t)
            (format t
              "~%Load: ~,20T~,4F ~%Preparation: ~,20T~,4F ~%~%
              Consistency: ~,20T~,4F ~%Index: ~,20T~,4F~%"
              (/ (- t1b t1a) internal-time-units-per-second)
              (/ (- t2b t2a) internal-time-units-per-second)
              (if check-abox-consistency

```

```

        (/ (- t2c t2b) internal-time-units-per-second)
        0)
        (/ (- t2d t2c) internal-time-units-per-second)))))))))

;;; =====

(defun run-lubm-benchmark (benchmark-function
                          check-abox-consistency
                          &optional (univs 0) (min-deps 0) (max-deps min-deps))

  (loop for deps1 from (if (null min-deps)
                          0
                          min-deps)
        to (if (or (null max-deps) (not (zerop univs)))
              0
              max-deps)
        do

      (format t "~%Initializing...~%"

        (prepare-lubm-data-n-universities check-abox-consistency
                                           univs
                                           (if (or (null max-deps) (not (zerop univs)))
                                               nil
                                               deps1))

        (format t "~%Querying...~%"

        (funcall benchmark-function
                  univs (if (or (null max-deps) (not (zerop univs)))
                          nil
                          deps1)))

      (values))

(defun original-lubm (universities departments)

  ;;; Query 1

  (benchmark universities
              departments 1
              (and
               (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#GraduateStudent|)
               (?x |http://www.Department0.University0.edu/GraduateCourse0|
                  |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#takesCourse|))

              (?x))

  ;;; Query 2

  (benchmark universities
              departments 2
              (and

```

```

    (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#GraduateStudent|)
    (?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#University|)
    (?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Department|)

    (?x ?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#memberOf|)
    (?z ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#subOrganizationOf|)

    (?x ?y
|http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#undergraduateDegreeFrom|))

    (?x ?y ?z))

;;; Query 3

(benchmark universities
  departments 3
  (and

    (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Publication|)
    (?x |http://www.Department0.University0.edu/AssistantProfessor0|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#publicationAuthor|))

    (?x |http://www.Department0.University0.edu/AssistantProfessor0|))

;;; Query 4

(benchmark universities
  departments 4
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Professor|)
    (?x |http://www.Department0.University0.edu|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#worksFor|)
    (?x (a |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#name|))
    (?x (a |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#emailAddress|))
    (?x (a |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#telephone|))
  )

  (?x
    (:datatype-fillers
      (|http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#name| ?x))
    (:datatype-fillers
      (|http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#emailAddress| ?x))
    (:datatype-fillers
      (|http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#telephone| ?x))))

;;; Query 5

(benchmark universities
  departments 5
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Person|)
    (?x |http://www.Department0.University0.edu|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#memberOf|))

    (?x))

;;; Query 6

```

```

(benchmark universities
  departments 6
  (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Student|)

  (?x))

;;; Query 7

(benchmark universities
  departments 7
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Student|)
    (?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Course|)
    (|http://www.Department0.University0.edu/AssociateProfessor0|
     ?y
     |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#teacherOf|)
    (?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#takesCourse|))

  (?x ?y))

;;; Query 8

(benchmark universities
  departments 8
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Student|)
    (?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Department|)
    (?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#memberOf|)
    (?y |http://www.University0.edu|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#subOrganizationOf|)
    (?x (a |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#emailAddress|)))

  (?x ?y (:datatype-fillers
    (|http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#emailAddress| ?x))))

;;; Query 9

(benchmark universities
  departments 9
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Student|)
    (?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Faculty|)
    (?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Course|)
    (?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#advisor|)
    (?x ?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#takesCourse|)
    (?y ?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#teacherOf|))

  (?x ?y ?z))

;;; Query 10

(benchmark universities
  departments 10
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Student|)
    (?x |http://www.Department0.University0.edu/GraduateCourse0|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#takesCourse|))

  (?x))

```

;;; Query 11

```
(benchmark universities
  departments 11
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#ResearchGroup|)
    (?x |http://www.University0.edu|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#subOrganizationOf|))

  (?x))
```

;;; Query 12

```
(benchmark universities
  departments 12
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Chair|)
    (?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Department|)
    (?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#memberOf|)
    (?y |http://www.University0.edu|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#subOrganizationOf|))

  (?x ?y))
```

;;; Query 13

```
(benchmark universities
  departments 13
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Person|)
    (|http://www.University0.edu|
      ?x
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#hasAlumnus|))

  (?x))
```

;;; Query 14

```
(benchmark universities
  departments 14
  (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#UndergraduateStudent|)

  (?x)))
```

(defun simple-lubm (universities departments)

```
(benchmark universities
  departments 1
  (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Chair|)
  (?x))

(benchmark universities
  departments 2
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Chair|)
    (?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Department|)
    (?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#memberOf|)
    (?y |http://www.University0.edu|
      |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#subOrganizationOf|))
```

```

      (?x ?y))

(benchmark universities
  departments 3
  (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Student|)
    (?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Faculty|)
    (?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Course|)
    (?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#advisor|)
    (?x ?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#takesCourse|)
    (?y ?z |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#teacherOf|))
  (?x ?y ?z)))

;;;
;;;
;;;

(defun run-lubm-tests (benchmark-function univs
  &key (max-deps-univ-1 nil)
  (mode 1)
  (check-abox-consistency (> mode 2)))

  (full-reset)
  (set-nrql-mode mode)
  (enable-optimized-query-processing (= mode 1))
  (set-unique-name-assumption t)
  (when (and (> univs 1) max-deps-univ-1)
    (error "Maximum number of departments may only be specified ~
      if only one university is processed."))
  (run-lubm-benchmark benchmark-function
    check-abox-consistency
    (1- univs) (and max-deps-univ-1 (1- max-deps-univ-1))))

(defun test1 (n &optional (mode 1) (check-abox-consistency (> mode 2)))
  (run-lubm-tests 'simple-lubm n
    :check-abox-consistency check-abox-consistency
    :mode mode))

(defun test2 (n &optional (mode 1) (check-abox-consistency (> mode 2)))
  (run-lubm-tests 'original-lubm n
    :check-abox-consistency check-abox-consistency
    :mode mode))

```

Bibliography

- [1] F. Baader, D. Calvanese, D. MacGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, Cambridge, UK, 2003.
- [2] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. In *Festschrift in honor of Jörg Siekmann, Lecture Notes in Artificial Intelligence. Springer, 2003.*, 2003.
- [3] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, and Enrico Franconi. An empirical analysis of optimization techniques for terminological representation systems or “making KRIS get a move on”. In B. Nebel, W. Swartout, and C. Rich, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference*, pages 270–281, San Mateo, 1992. Morgan Kaufmann.
- [4] S. Bechhofer, P. Crowther, and R. Möller. The description logic interface. In D. Calvanese, G. De Giacomo, and E. Franconi, editors, *International Workshop on Description Logics*, pages 196–203, September 2003.
- [5] M. Buchheit, F.M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.
- [6] V. Haarslev, R. Möller, and M. Wessel. The description logic \mathcal{ALCNH}_{R+} extended with concrete domains: A practically motivated approach. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning. First International Joint Conference (IJCAR’01), Siena, Italy, June 18–23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 29–44, Berlin, 2001. Springer-Verlag.
- [7] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.
- [8] I. Horrocks, U. Sattler, and S. Tobies. A PSPACE-algorithm for deciding \mathcal{ALCI}_{R+} -satisfiability. LTCS-Report 98-08, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1998.
- [9] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic shiq. In David MacAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, number 1831 in *Lecture Notes in Computer Science*, pages 482–496, Germany, 2000. Springer Verlag.

- [10] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic SHIQ. In David MacAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, number 1831 in Lecture Notes in Computer Science, Germany, 2000. Springer Verlag.
- [11] I. Horrocks and S. Tobies. Reasoning with axioms: Theory and practice. In *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 285–296, 2000.
- [12] Ian Horrocks and Ulrike Sattler. Optimised reasoning for *SHIQ*. In *Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002)*, pages 277–281, July 2002.
- [13] Ian Horrocks and Ulrike Sattler. Decidability of *SHIQ* with complex role inclusion axioms. *Artificial Intelligence*, 160(1–2):79–104, December 2004.
- [14] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. A description logic with transitive and converse roles, role hierarchies and qualifying number restrictions. LTCS-Report 99-08, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1999.
- [15] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Computer Science, pages 161–180. Springer-Verlag, 1999.
- [16] C. Lutz and M. Milicic. A tableau algorithm for dls with concrete domains and gcis. In *Proceedings of the 2005 International Workshop on Description Logics (DL2005)*, number 147 in CEUR-WS, 2005.
- [17] P.F. Patel-Schneider and B. Swartout. Description logic knowledge representation system specification from the krss group of the arpa knowledge sharing effort. Technical report, Bell Labs, 1993. Available as <http://www-db.research.bell-labs.com/user/pfps/papers/krss-spec.ps>.

Index

Protégé, [31](#)
RacerEditor, [27](#)
RacerMaster, [6](#)
RacerPlus, [5](#)
RacerPorter, [5](#), [9](#), [25](#)
RacerPro, [5](#)
AND query, [119](#)
HAS-KNOWN-SUCCESSOR query atom, [109](#)
NEG query, [124](#)
PROJECT-TO operator, [130](#)
SAME-AS query atom, [107](#)
UNION query, [120](#)

ABox augmentation, [139](#)
ABox modification, [139](#)
active domain semantics, [93](#)
APIs, [20](#)
assertion, [65](#)
associated ABox individual, [150](#)
associated Abox individual, [150](#)
associated substrate node, [150](#)

binary query atom, [92](#)
body projection operator, [130](#)
boolean query, [130](#)
bug reports, [24](#)

cheap query, [182](#)
cheap rule, [182](#)
cheap tuple, [182](#)
closed-world assumption, [73](#)
complex query, [118](#)
complex TBox query, [144](#)
concept axioms, [59](#)
concept definition, [60](#)
concept equation, [59](#)
concept query atom, [93](#)

concept term, 56
concrete domain restriction, 59
concrete domain value query, 110
concrete domains, 61
concurrent query processing, 173
conjunction of roles, 61
conjunctive query, 119
consistency of qualitative spatial networks, 161
constraint query atom, 102
creating individuals with a rule, 140

data representation, 149
data substrate, 149
data substrate edge, 149
data substrate edge query expression, 152
data substrate label, 149
data substrate node, 149
data substrate node query expression, 152
data substrate predicate, 153
data substrate query expression, 152
deadlock prevention, 185
defined query, 133
defined query and NAF, 135
defined query and PROJECT-TO, 135
DIG, 22
disjoint concepts, 59
disjunctive query, 120
domain restriction, 61

eager mode, 174
editions, 5
equal role, 99
exists restriction, 57
expensive query, 182
expensive rule, 182
expensive tuple, 182
explicit role filler query, 101
extended OWL query, 154, 158

feature, 60
feature chains in queries, 104

GCI, 59
graphical client interfaces, 29

head projection operators, 110
hybrid ABox individual, 150

hybrid OWL query, 154, 158
hybrid query, 151, 154
hybrid representation, 149

implicit role filler query, 101
incomplete mode, 179
incremental mode, 172
incremental query processing, 173
individual, 92
inference modes, 66
injective variable, 92
installation, 6

JRacer, 20

lazy mode, 174
local closed-world assumption, 85
LRacer, 21

macro query, 133
marker, 157
mirror data substrate, 154
mirror of an ABox, 154
mirror substrate, 154
mirror substrate marker, 157
mirror substrate query, 154
mirroring an ABox, 154
mirroring an OWL file, 154
mirroring OWL documents, 158

NAF, 124
NAF in constraint query atom, 128
NAF in query with individuals, 128
NAF in SAME-AS query atom, 128
NAF query, 124
negation as failure, 124
nominals, 70, 142
nominals in queries, 142
nRQL API, 172
nRQL EBNF, 164
nRQL engine, 172
nRQL grammar, 164
nRQL mode, 179
nRQL mode 0, 179
nRQL mode 1, 180
nRQL mode 2, 181
nRQL mode 3, 181

nRQL mode 4, 181
nRQL mode 5, 181
nRQL mode 6, 181
nRQL modes, 172, 181
nRQL persistency, 201
nRQL syntax, 164
number restriction, 57

object, 92
open-world assumption, 72
optimization strategies, 77
options, 22
OWL, 74
OWL annotation property filler query, 115
OWL annotation property value query, 115
OWL constraint checking, 106
OWL constraint query atom, 106
OWL datatype property filler query, 115
OWL datatype property value query, 115
OWL instances query, 95
OWL mirror, 154, 158
OWL object property filler query, 100
OWL query, 95, 100, 106, 115, 158

persistency, 78
predicate, 153
predicate query, 153
preferences, 25
primitive concept, 60
projection, 130
projection operator, 130
pseudo-nominal, 142
publish-subscribe mechanism, 79

QBox, 195
qualitative spatial reasoning, 161
query API, 172
query atom, 92
query body, 93
query cache, 195
query conjunct, 119
query consistency, 188
query disjunct, 120
query engine, 172
query entailment, 190
query head, 93, 110
query inconsistency, 188

query inference, 188
query life cycle, 175
query predicate, 153
query realization, 199
query realizer, 199
query reasoning, 188
query repository, 195
query subsumption, 190
querying qualitative spatial networks, 161

range restriction, 61
RCC consistency, 161
RCC constraint checking, 162
RCC query, 162
RCC substrate, 161
RDF edge query, 100
RDF instance query, 95
RDF node query, 95
referencing a data substrate node, 153
region connection calculus, 161
relational algebra, 4
retraction, 67
retrieving concrete domain values, 110
role chains in queries, 104
role hierarchy, 61
role query atom, 96
role query atom with features, 98
role query atom with negated role, 97
rule, 139
rule antecedent, 139
rule application, 139, 176
rule application strategy, 176
rule consequence, 139
rule engine, 139, 172
rule firing, 176
rule life cycle, 176
rule postcondition, 139
rule precondition, 139
rule strategy, 176
rules and pseudo-nominals, 142
rules and the concrete domain, 141

scalable query answering, 179
semantic cache, 195
semantic web, 1
semantically identical individuals, 99

semi-structured data, 149
services, 1, 3
set at a time mode, 172
signature, 56
socket interface, 20
spatial query, 162
spatial reasoning, 161
spatial reasoning substrate, 161
spatio-thematic query, 162
stored defined query, 201
stored QBox, 201
stored substrate, 201
substrate, 149
substrate layer, 149
substrate persistency, 201
substrate query, 151
SWRL, 44
system requirements, 6

TBox query, 144
told value query, 110
transitive role, 60
tuple at a time mode, 172
two-phase query processing, 182

unary query atom, 92
unique name assumption, 65, 73

value restriction, 57
variable, 92

warning token, 182