

RacerPro User's Guide

Version 2.0

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

October 24, 2012

Contents

1	Introduction	1
1.1	System Overview	2
1.1.1	Design Principles	2
1.1.2	Description and Query Languages	4
1.1.3	Interfaces	5
1.2	Different views on RacerPro	5
1.2.1	RacerPro as a Semantic Web Reasoning System and Information Repository	5
1.2.2	RacerPro as a Knowledge Representation and Reasoning System	6
1.3	Application Areas	8
2	Using RacerPro and its Interfaces	11
2.1	Sample Session	11
2.1.1	Using RacerPro with RacerPorter	13
2.1.2	Using RacerPro from the Command Line	16
2.2	The RacerPro Server	18
2.2.1	The File Interface	18
2.2.2	TCP Interfaces	20
2.2.3	Options for RacerPro Server	35
2.2.4	How to Send Bug Reports	37
2.2.5	Online Reference Manual	38
2.3	RacerPorter in Details	41
2.3.1	Distribution of GUI elements	41
2.3.2	Preferences	44
2.3.3	RacerEditor	45
2.3.4	Tabs in RacerPorter	47
2.3.5	Known Problems	55
2.4	Other Graphical Client Interfaces	55
2.4.1	Protégé	55
2.4.2	Using Protégé and RacerPorter in Combination	58
3	Modeling Languages and Basic Reasoning Services	61
3.1	Naming Conventions	61

3.2	Concept Language	62
3.3	Concept Axioms	64
3.4	Role Axioms	65
3.5	Concrete Domains	66
3.6	Individual Assertions and A-boxes	69
3.7	Specifying Rules	71
3.8	Hints on Modeling and Reasoning with RacerPro	74
3.8.1	Representing Data with Description Logics (?)	74
3.8.2	Nominals or Concrete Domains?	75
3.8.3	Open-World Assumption	77
3.8.4	Closed-World Assumption	77
3.8.5	Unique Name Assumption	78
3.9	Evaluating RacerPro and Configuring Reasoning Modes	78
4	The New RacerPro Query Language - nRQL	81
4.1	The nRQL Language	85
4.1.1	Query Atoms, Objects, Individuals, and Variables	85
4.1.2	Query Head Projection Operators to Retrieve Told Values	101
4.1.3	Complex Queries	108
4.1.4	Defined Queries	120
4.1.5	ABox Augmentation with Simple Rules	125
4.1.6	Complex TBox Queries	129
4.1.7	Hybrid Representations with the Substrate Representation Layer	133
4.1.8	Functional Lambda Expressions	146
4.1.9	Formal Syntax of nRQL	150
4.2	The nRQL Query Processing Engine	157
4.2.1	The Query Processing Modes of nRQL	157
4.2.2	The Life Cycle of a Query	160
4.2.3	The Life Cycle of a Rule	161
4.2.4	How to Implement Your Own Rule Application Strategy	161
4.2.5	Configuring the Degree of nRQL Completeness	164
4.2.6	Automatic Deadlock Prevention	168
4.2.7	Reasoning with Queries	171
4.2.8	The Query Repository - The QBox	177
4.2.9	The Query Realizer	180
4.2.10	The nRQL Persistency Facility	181
5	Additional Facilities	185
5.1	Retraction and Incremental Additions	185
5.2	Rule Formalism Extended to Support Abduction	185
5.3	Reasoning with Triples in Secondary Memory	188

5.4 The Publish-Subscribe Mechanism	193
5.4.1 An Application Example	193
5.4.2 Using JRacer for Publish and Subscribe	198
5.5 OWL Interface	199
5.6 Explanation	201
5.7 Non-standard Inferences	202
5.8 Event Recognition	203
A Editions and Licenses	205
A.1 Editions	205
A.2 Licenses	206
B Acknowledgments	209
Index	217

Chapter 1

Introduction

RacerPro (**R**enamed **A**Box and **C**oncept **E**xpression **R**easoner **P) is a server system for building knowledge-based applications based on ontologies. The backbone of RacerPro is a description logic reasoning engine. It provides inference services for descriptions of terminological knowledge as well as for knowledge about individuals. Based on a plethora of optimization techniques developed in the research field of description logics throughout the years, a mature architecture optimized for typical-case application scenarios involving expressive description languages is provided. The system has been used in hundreds of research projects and industrial contexts throughout the last twelve years. W3C standards as well as detailed feedback reports from numerous users have influenced the design of the system architecture in general, and have also shaped the RacerPro knowledge representation and interface languages in particular.**

Description logics provide the foundation for international approaches to standardize conceptual modeling languages in the context of the so-called semantic web, RacerPro can be used as a system for managing semantic web ontologies based on syntaxes provided by the web ontology language OWL. It can be used as a reasoning engine for ontology editors such as Protégé. In addition, RacerPro can also be seen as a semantic web information repository with an optimized retrieval engine. It can handle large sets of data descriptions (e.g., defined using the standardized resource description format RDF). Last but not least, the system can also be used as a satisfiability tester for modal logics such as K_m (with axioms).

The RacerPro User's Guide is intended for different audiences, i.e., students of computer science who participate in research efforts about semantic technologies, or for developers of semantic web applications with interests in description logic reasoning, interoperability of the RacerPro reasoner with current semantic web standards as well as APIs for programming semantic web applications. Given the diversity of the semantic web research field, this document is not intended as an introductory first-level source of information. Some chapters might require additional background knowledge. We suggest to follow the provided references in case some terms are new to the reader, and encourage the reader to also use the virtually endless number of online resources provided by the semantic web community. As a beginner you should probably start with the examples provided in Chapter 2. As a Protégé user there is also excellent material available from the project's web site. You might also want to follow the Protégé examples explained in Section 2.4.

Before you try to run any of the examples explained in this document, please make sure that your RacerPro system is properly installed and the server is running. For more information about the available software editions and various licenses please read the Appendix A of this document and check the ReadMe-file that comes with the software, or check our web site for further information.

You should also have the interactive GUI for RacerPro, RacerPorter, installed properly. While generic knowledge editing environments such as Protégé have their merits, a reasoner-specific GUI gives you more control. RacerPorter is in a sense unique as it is the only reasoner-specific graphical user interface for ontology-based software development that supports debugging in the context of

reasoning applied to large knowledge bases. When developing your knowledge bases you should use RacerPorter to inspect them in order to better understand the structure and implications of your data descriptions. RacerPorter's visualization, browsing and edition tools are quite useful not only for editing and displaying concepts, taxonomies, axioms and so forth. RacerPorter is also helpful for managing multiple RacerPro servers.

Software developers who want to use RacerPro as a reasoning back-end will want to have a closer look into Section 2.2 for more information about possible application scenarios and for further information about how to integrate RacerPro into software system design.

This document contains various hyperlinks useful to navigate through the user guide, they are distinguished by their colors. Green is used for citations, pink for URLs and red for all other references within the user guide, e.g., contents, images, tables, etc.

In case you encounter difficulties you can contact support@racer-systems.com and submit [bug reports](#). We are highly interested into your feedback, and your comments are highly appreciated! However, Racer Systems cannot provide free assistance about how to solve a specific problem or about how to configure a specialized system. Professional support can be provided on-line or on-site on a contractual basis. Please refer to <http://www.racer-systems.com/services> or contact us for more information how to use our knowledge and experience for your projects.

1.1 System Overview

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.

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

<http://www.racer-systems.com/products/download/index.shtml>

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.

1.1.1 Design Principles

RacerPro is available as a server version (RacerPro Server) with APIs e.g. for Java or Common Lisp. RacerPro communicates with client programs via various interfaces, either RacerPro-specific ones (maximum expressivity) or standardized ones (maximum portability). A graphical interface, called RacerPorter, is provided for interacting with the server as well as for submitting server extensions and queries. See Figure 1.1 for an overview on the system architecture. It should be noted that RacerPro can be extended using a simple plugin mechanism. For the users' convenience, parts of the RacerPro code are open source, and can be used to extend the RacerPro reasoning server.

In an ontology-based application, usually multiple representation languages are used for different purposes. The backbone is a description logic language for defining the terminological or intensional part of a knowledge base, which is often extended with other logical languages for the extensional part, such as, for instance, logic programming rules, the region connection calculus for aspects of spatial reasoning, or Allen's interval algebra for aspects of temporal reasoning, just to name a few [64]. While terminological reasoning (T-box reasoning) is supported with many optimizations, the RacerPro system is tailored to applications which build on the exploitation of assertional reasoning (A-box reasoning). The main idea is that A-boxes are not static parts of the ontology, but are efficiently generated on the fly (referring to a shared T-box which is "processed" only once). T-boxes (ontologies) and A-boxes are maintained using the RacerPro server system,

which communicates with remote application programs using well-defined axiom manipulation languages as well as query languages that refer to the semantics of the axioms. In addition, a rule language (based on SWRL syntax) is used to conveniently extend the A-boxes stored by the reasoning server, i.e., rules that are transferred to the server can be used to extend the expressivity w.r.t. assertional reasoning and/or make implicit information explicit. Server-side A-boxes can be remotely cloned and easily extended such that variants of assertional knowledge can be conveniently managed as lightweight objects while the T-box part they refer to is shared.¹ Besides this approach of “lightweight A-boxes”, the RacerPro architecture nevertheless also supports large A-boxes stored in a triple store database (see Section 5.3). One of the main design principles of RacerPro is to automatically select applicable optimizations based on an analysis of the language of the input knowledge bases and the queries being processed.

There are various usage scenarios of DL systems for which RacerPro is optimized. In one scenario, T-boxes are usually rather large, and A-boxes are small (< 100 individuals), but many (variants of) A-boxes need to be handled. In the other scenario, T-boxes are rather small, and one large A-box (100.000 individuals) is referred to in queries. RacerPorter provides default configurations to best support these scenarios from a GUI point of view.

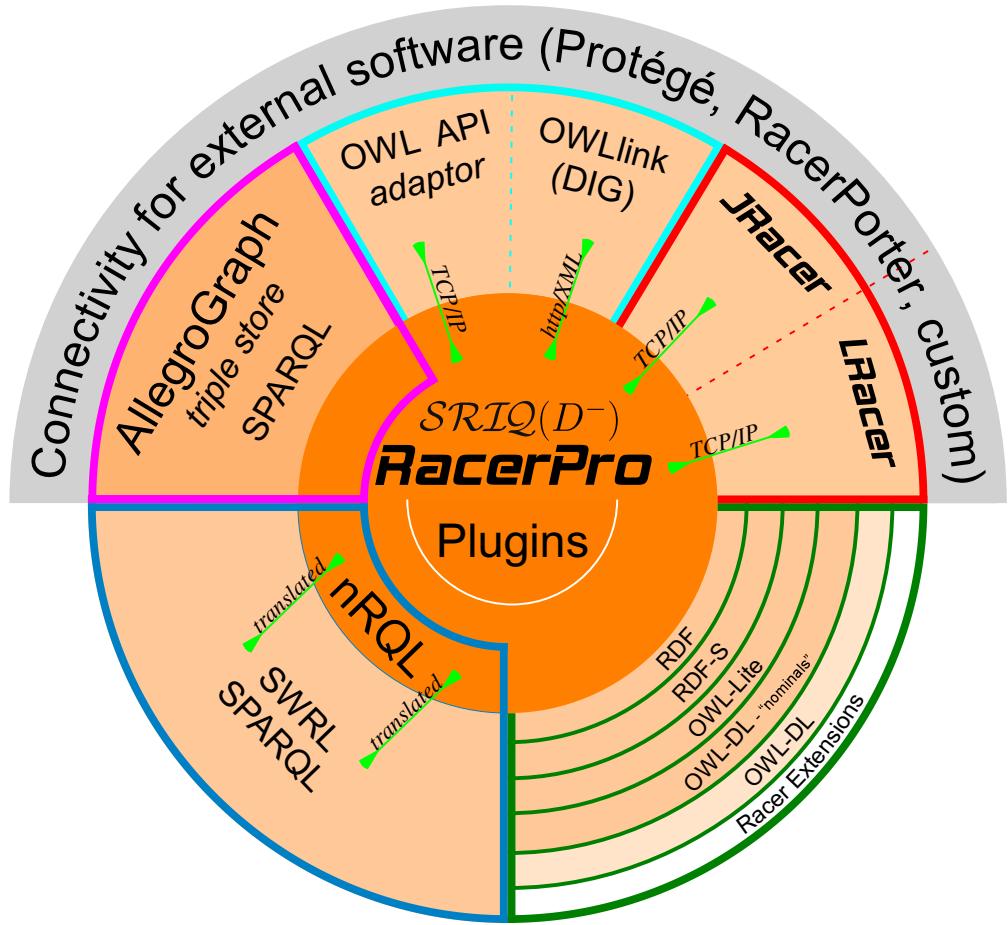


Figure 1.1: RacerPro System Architecture

¹The T-box, an A-box refers to, can also be changed, but obviously, this requires complete reprocessing of the assertions in the A-box.

1.1.2 Description and Query Languages

Ontologies are based on fragments of first-order logic for describing a shared conceptualization of a domain using concept and role descriptions (called classes and properties in OWL, respectively). The initial representation language of RacerPro was $\mathcal{ALC}\mathcal{NH}_{R^+}(\mathcal{D})^-$ [23], and RacerPro was the first system which efficiently supported concrete domains for T-box and A-box reasoning [31, 32]. RacerPro was then extended to also support inverse roles and qualitative number restrictions [36] as part of the description logic (DL) \mathcal{SHIQ} [44], a practically relevant subset of OWL. Role axioms (\mathcal{SRIZQ} [40]) are another language construct that has been integrated into RacerPro.

Since RacerPro supports concrete domains effectively, it was found that nominals (\mathcal{SHOIQ} [41]) were not of utmost importance for RacerPro users [35]. Since multiple A-boxes should refer to a single T-box (preprocessed and indexed offline before A-box query answering), A-box assertions should not introduce implicit subsumption relationships, a design principle that is, in general, broken if nominals are supported. RacerPro does support nominals from a syntactical point of view in terms of the OWL language, and it maps nominals to atomic concepts (the standard approximation of nominals [12]). While interesting optimization techniques for “true” nominals have been developed [58], reasoning with nominals is known as hard not only from a theoretical [41] but also from a practical point of view (i.e., hard also for typical-case input). RacerPro might be extended with nominals, once optimization techniques for reasoning with nominals get mature enough such that RacerPro applications can effectively exploit this feature (see, e.g., [18, 19]).

Inference services for concept subsumption and the taxonomy of a T-box have been part of description logic reasoning systems right from the beginning in the eighties (T-box classification). Classification is supported in RacerPro with specific optimization techniques [24, 25, 28, 71, 70] that are based on or are integrated with results obtained in other projects [20, 21] as well as techniques implemented in mature predecessor DL systems such as \mathcal{KRIS} [6, 5] and FaCT [38, 39, 61]. Still, T-box classification is a very fruitful research area, and new techniques are being integrated into RacerPro. Interestingly, for dealing with a non-cyclic \mathcal{ELH} [3] version of the Snomed/CT knowledge base, a very old structural subsumption technique being integrated into RacerPro provided for classification in the range of minutes for this very large T-box [30], with the additional advantage that (small) parts of the T-box can indeed use more expressive language fragments.

Inference services for A-boxes are influenced by many earlier DL systems, for instance, CLASSIC [11, 12]. In these systems, the query language for finding individuals is based on concept descriptions, and, thus, rather limited (see [9, 10]). In addition, in order to effective answer queries, in CLASSIC the most-specific concept names of which individuals are instances are computed in advance (A-box realization). In contrast, RacerPro was designed in such a way that the user can decide whether to compute index structures in advance or on the fly [29]. Research on RacerPro has focused on concept-based instance retrieval [26] as well as on a more expressive form of queries, namely grounded conjunctive queries [34, 35].

The new Racer Query Language (nRQL, pronounced “miracle” and to be heard as “miracle”) was one of the first expressive query languages for DL systems (see Section 4 for a description of nRQL) providing conjunctive queries [16] with variables ranging over named domain objects, negation as failure, a projection operator, as well as group-by and aggregation operators. The formal semantics of nRQL was described in [67, 68]. Interestingly, much later, a query language semantics based on a different viewpoint was described in [14]. However, nRQL is nowadays seen as an implementation of the approach proposed in [14]. It should be emphasized that RacerPro supports query answering with the pull mode (clients send queries and retrieve result sets incrementally from the server) or with the push mode (clients subscribe conjunctive queries and receive notifications about elements in the result set incrementally) [27].

1.1.3 Interfaces

RacerPro can be used as a server application in a network-wide context (see Section 2.2). In addition to raw TCP communication interfaces with APIs for Java (JRacer, OWLAPI), Common Lisp (LRacer), and also C, RacerPro supports the OWLlink communication interface [53, 56] (a successor of DIG [8]). RacerPro supports various OWL 2 syntaxes, namely RDF/XML as well as OWL functional syntax. See again Figure 1.1 for an overview of the RacerPro system architecture. A standardized query language for RacerPro is SPARQL. RacerPro also supports many extensions in a KRSS-like syntax (e.g., the nRQL query language, publish/subscribe interface etc.). RacerPro implements the OWLAPI [37] for Java, and, additionally, a plugin for Protégé 4 is available [55].

Ontologies can be read from files, or can be retrieved from the web as well as from an RDF triple store managed by the built-in AllegroGraph system (version 3) from Franz Inc. [1]. AllegroGraph can be used to store materialized inferences and also provides for a powerful query language based on SPARQL syntax [1]. AllegroGraph can store very large A-boxes for which users need nRQL query answering.

The RacerPro inference server can be programmed in a functional language called miniLisp. For instance, query results can be postprocessed by the miniLisp interpreter running functional programs being sent to the server such that query results can be sent in application-specific XML formats to client programs via the built-in RacerPro web server. The language miniLisp is designed in such a way that termination of miniLisp programs is guaranteed, and miniLisp can be used to specify rather complex queries and server extensions while the reasoning server is running. The functional language miniLisp can be extended by application programs (e.g., in the same sense that application programs can generate Javascript program on the fly and send them to the browser). In combination with a miniLisp for manipulating query results on the server, nRQL queries ensure that it is not necessary to transfer large sets of A-box assertions from the inference server to application programs.

For more complex extensions, RacerPro supports a plugin interface. Thus, compiled programs can encode arbitrary algorithms on the RacerPro server. Plug-ins have been used, for instance, for developing non-standard inference algorithms [62]. Using the open source library OntoLisp², large parts of the RacerPro code for syntactically processing ontologies are publicly available. Ontolisp can be used for developing new reasoners as well (various Common Lisp systems are supported).

The graphical interface RacerPorter [69] can be used to explore axioms and terminologies. It provides a read-eval-print loop (shell) for interactively extending and querying one or more RacerPro servers, or for inspecting, for instance, explanation results for certain inferences. For the user's convenience, a complete Emacs-style text editor is provided to edit nRQL queries or miniLisp function definitions. Query results (bindings for variables) can be interactively inspected and A-boxes can interactively explored in the A-box inspector showing told and inferred assertions.

1.2 Different views on RacerPro

1.2.1 RacerPro as a Semantic Web Reasoning System and Information Repository

The semantic web is aimed at providing new “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 documents (OWL ontologies). Ontologies may be taken off the shelf or may be extended for domain-specific purposes (domain-specific ontologies extending core ontologies). For doing this, a reasoning system is required as part of the ontology editing system. RacerPro can process OWL Lite, OWL DL as well as OWL

²See <http://sourceforge.net/projects/ontolisp/>.

2 ontologies (knowledge bases). OWL ontologies are processed with approximations for individuals mentioned in class expressions. User-defined XML datatypes are not yet supported.

An implementation of the semantic web rule language (SWRL) is provided with RacerPro 2.0 such that rules may be specified as part of an OWL document in SWRL syntax (syntax restrictions apply see below, see also below for a description of the semantics of rules).

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

- Load ontologies specified in OWL 2 format. An HTTP client for retrieving ontology resources from the web is integrated. Multiple resources can be imported into one ontology.
- Check the consistency of an OWL ontology and a set of data descriptions.
- Find implicit subclass relationships induced by the axioms of the ontology.
- Find synonyms for properties (roles), classes (concepts) or instances (individuals).
- Query extensional information collected from OWL/RDF documents (OWL instances and their interrelationships) using a SPARQL query processing system or, for more expressive queries, using the Racer-specific query language nRQL.
- Incremental query answering for information retrieval tasks (retrieve the next n results of a query). With this feature, 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.
- Support for specific locales and external formats (e.g., for Asian character sets)

1.2.2 RacerPro as a Knowledge Representation and Reasoning System

RacerPro is a knowledge representation and reasoning 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{SRIQ} (see [40]). This is the basic logic \mathcal{ALC} augmented with qualifying number restrictions, inverse roles, and acyclic role axioms (for specifying, e.g., role hierarchies, symmetric, reflexive, and transitive roles, etc.). In addition to these basic features, RacerPro also provides facilities for algebraic reasoning using so-called 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.

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.
- Determine the least common subsumer (LCS) of a set of concepts w.r.t. a T-box: The LCS is the most specific concept that subsumes a set of concepts, in other words, the concept description that represents the properties that all the elements of the set have in common.

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.
- Computation of the least concept description of an individual defined in an A-box. A service which is also called most specific concept (MSC-k), where k defines a nesting level of existential restrictions.

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. RacerPro supports loading triples from secondary memory (e.g., as provided by the AllegroGraph triple-store <http://www.franz.com>).

1.3 Application Areas

For all software systems, and in particular for a knowledge representation and reasoning engine, it holds that the system architecture is influenced by typical application areas for which it should be most-effective. This is true also for the RacerPro system, a practical software system for building knowledge-based systems for demanding application scenarios ranging from autonomous agents on the semantic web to knowledge-based software engineering.

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 when 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”). Thus, rather dealing with data, description logics systems deal with *data descriptions*, which might possibly involve indefinite knowledge about particular domain objects.

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 queried 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. 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)
- Event recognition (e.g., given a set of temporal propositions about basic events coming from low-level image sequence analysis processes, and exploiting relational algebras based on so-called event-rules, RacerPro can detect whether a temporal high-level event can be instantiated)

Various features of RacerPro have been used in the research community to support these application areas, for example, the explanation facility of RacerPro has been used to support the development of the OWL translator for the SUMO ontology [57]. Explanation features for inconsistent concepts are available for knowledge bases as large as Snomed CT (the RacerPro explanation facility uses built-in data structures of the tableau reasoner). It should be noted that A-box reasoning services can be used for problem solving. For instance, in [63], an application of A-box realization for

computing solutions to Sudoku problems is presented (note that nominals are not required for this purpose [60]).

In [59] nRQL has been used to represent integrity constraints as queries which must return an empty result set. This early use case also has shaped the functionalities provided by nRQL. Rob Lemmens has used Racer for investigating the semantic interoperability of distributed geo-services [52].

A-box reasoning (consistency, direct types, etc.) has been used to formalize scene interpretation problems [51]. In particular, it was shown that complete reasoning is necessary for efficiently integrating different “clues” obtained from sensors into a coherent whole. Furthermore, in [51], deduction proved useful to really find interesting object classifications as well as interesting events for making decisions. In the industrial context, RacerPro was used for driver assistance [50], e.g., knowledge-based intersection assistance [49], and visual information fusion [22].

Event recognition was also explored in the BOEMIE project [54, 15, 17] as well as in the ContextWatcher project [66, 65]. In the former approach, RacerPro was extended with CLP(R)-like techniques for dealing with quantitative information where in the latter approach the expressivity of nRQL is explored for qualitative event recognition. Another very interesting approach in this context is the use of nRQL as a target language for compiling linear temporal logic (LTL) event descriptions, and using assertional reasoning provided by RacerPro for solving the actual event recognition problem [2].

Stream-based processing of data descriptions based on incremental interpretation using abduction operators based on decription logics and logic programming was investigated with RacerPro in the CASAM project for multimedia annotation (see <http://www.casam-project.eu>).

Chapter 2

Using RacerPro and its Interfaces

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 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 XML-based interfaces provided by RacerPro.

2.1 Sample Session

There are various ways to interact with RacerPro, through GUIs such as RacerPorter and Protégé, through the command line interface(shell), or by storing a set of commands in a batch file. In this section, we will show first how to use RacerPro in combination with the RacerPorter interface and afterwards with how to interact with the command line.

Before starting the sample session, localize the files containing the knowledge base that we will be using. These files are the "`family.racer`" which contains the T-box and A-box parts and the "`family-queries.racer`" which contains queries. Both files can be found in the folder named "`.../examples/native`" of the RacerPro distribution.

The following code is found in the file "`family.racer`":

```
;;; initialize the knowledge base
;;; with a T-box name "family" and an A-box name "smith-family"
(in-knowledge-base family smith-family)

;;; Supply the signature for this knowledge base
(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)

```

The following code is found in the file "family-queries.racer":

```

(concept-ancestors mother)

(concept-descendants man)

(all-transitive-roles)

(individual-instance? doris woman)

(individual-types eve)

(individual-filters alice has-descendant)

(individual-direct-types eve)

(concept-instances sister)

```

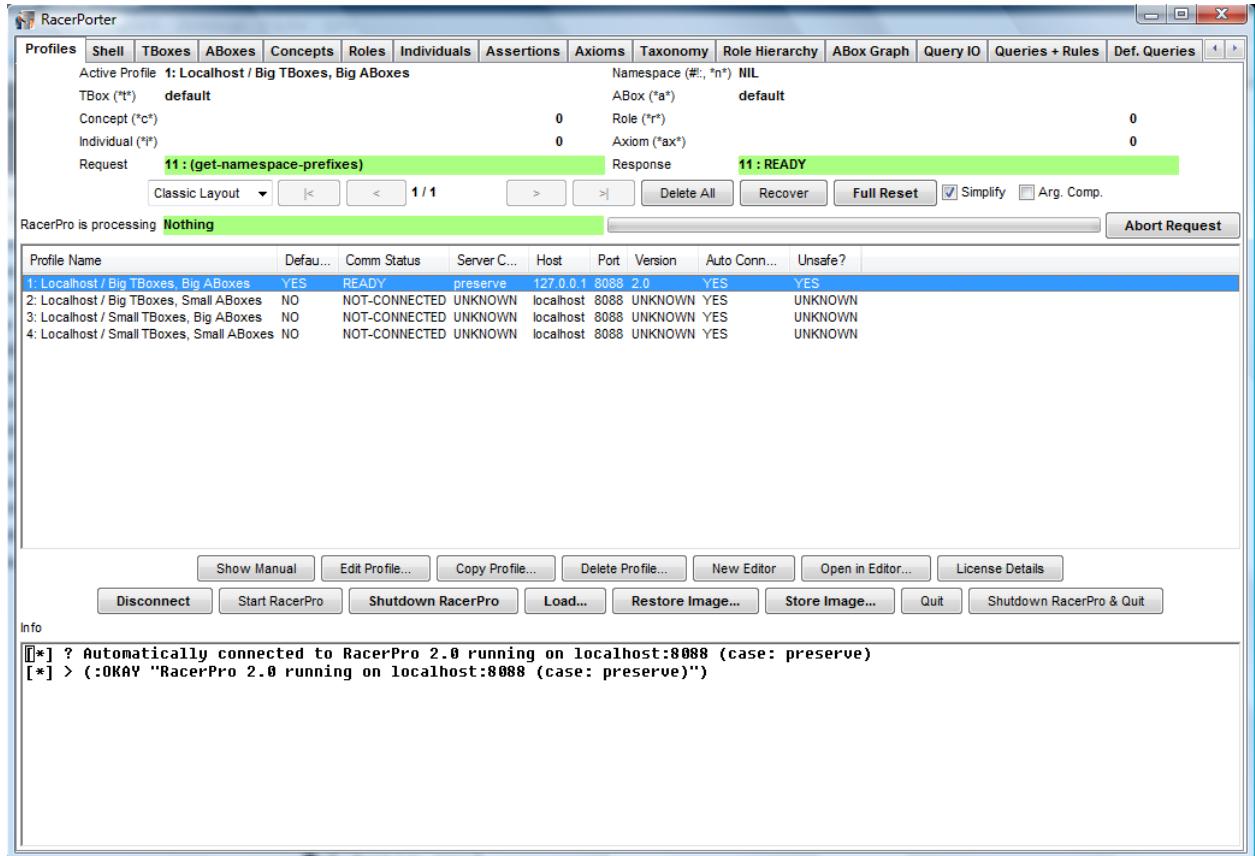


Figure 2.1: The RacerPorter interface for RacerPro.

2.1.1 Using RacerPro with RacerPorter

Start RacerPorter by double-clicking the program icon. When RacerPorter is opened, it will locate the RacerPro executable and automatically start it and connect to it. To locate the RacerPro executable, the RacerPorter looks within the installation path of RacerPorter or looks if there has been an environment variable set to RacerPro. However if you are interested in executing a specific RacerPro executable, you can configure this within RacerPorter as follows.

Click the “Edit Profile...” button in order to configure RacerPorter. Once the “Profile Editor” dialog is opened, edit the “RacerPro Executable” field by providing the required path, e.g., “C:/Program Files/RacerPro-20-Preview/RacerPro.exe”, then press “OK & Save Profile”. By saving the profile, RacerPorter will be able to localize the RacerPro executable you specified every time RacerPorter is started. The configuration is saved in a file called “racereditor-config.racereditor” found in the same path of the RacerPro executable. Profiles are stored in the User Home directory under the file called “profiles.racerporter”. To know more about the profiles see Section 2.3.4.

If the RacerPorter was able to localized the RacerPro server then the connection to RacerPro is set automatically as shown in Figure 2.1. After a successful connection the RacerPorter will become “alive” and the functionality provided by the different tabs can be accessed. The button “Connect” can be used to connect or disconnect to the RacerPro server. For example, to connect to a freshly started server. Note that it is possible to connect to a local, as well as to a remote RacerPro server, even if it is used by another GUI such as Protégé.

Once the RacerPro server is executed you should see a shell showing the following lines:

```
;; Welcome to RacerPro Version 2.0 2010-10-02!

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

;;; Copyright (C) 2004-2009 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 8.2 [Windows] (Oct 2, 2010 12:27)
;;; Copyright (C) 1985-2010, 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
;;; C:\Program Files\RacerPro-20-Preview\license.racerlicense
;;; This copy of RacerPro is licensed to:
;;;
;;; Juan Perez
;;; Technische Universitaet Hamburg-Harburg
;;; Harburger Schlossstrasse 20
;;; 21073, Hamburg
;;; DE
;;;
;;; Initial license generated on November 03 2010, 10:22 for 1.9.0.
;;; Desktop, Educational, for X86 Win32.
;;; This license is valid up to version 2.0.999.
;;; This license is valid until April 30 2011, 23:59.
;;;
;;; =====

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.

To edit knowledge bases (or ontologies and data descriptions), you can either use the “Load...” button in the “Profiles” tab (or by selecting “Load...” in the menu “File” on Mac OS X), or you can also work with the built-in editor. The following file types / syntaxes for KBs are recognized and supported: OWL RDF (*.owl, *.rdf), KRSS / native RacerPro syntax (*.racer, *.lisp), and OWL 2 functional (*.funct). For this sample session we will use the built-in editor.

Press the button “Open in Editor” (or select “Open in Editor” in the “File” menu on Mac OS

X) and double-click the file "family.racer". 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 Expression" in the "Buffer" menu (make sure the window RacerEditor is the active window).

RacerPro supports the standard Emacs commands and key bindings, e.g., change buffer is bound to **c-x b**. Press tab to see a list of possible completions.

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. 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, for this, you can use the "Shell" tab to execute queries as we have done in Figure 2.3. You can use the queries of the "family-queries.racer" file specified above. 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. By using the **tab** key you can complete a partially typed command. A command is executed once it is syntactically complete.

RacerPorter allows you to inspect data descriptions in an A-box, for this select the "A-box Graph"

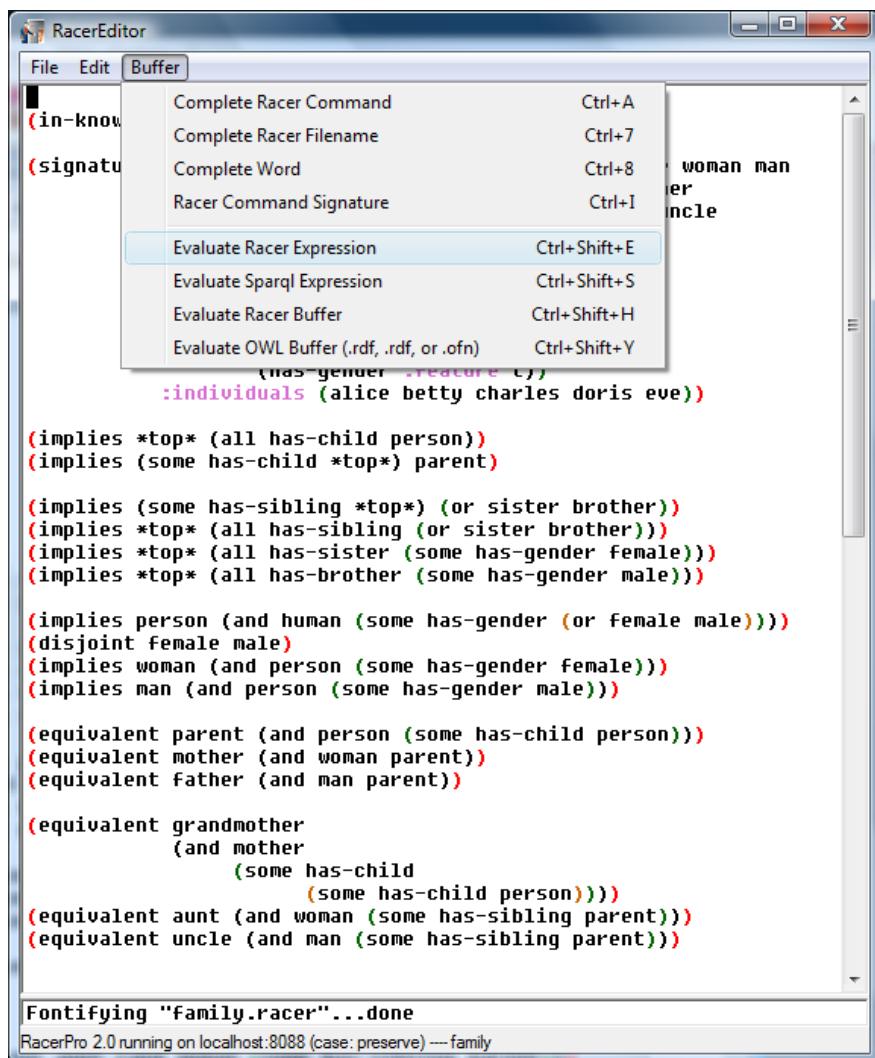


Figure 2.2: The RacerEditor interface for RacerPro.

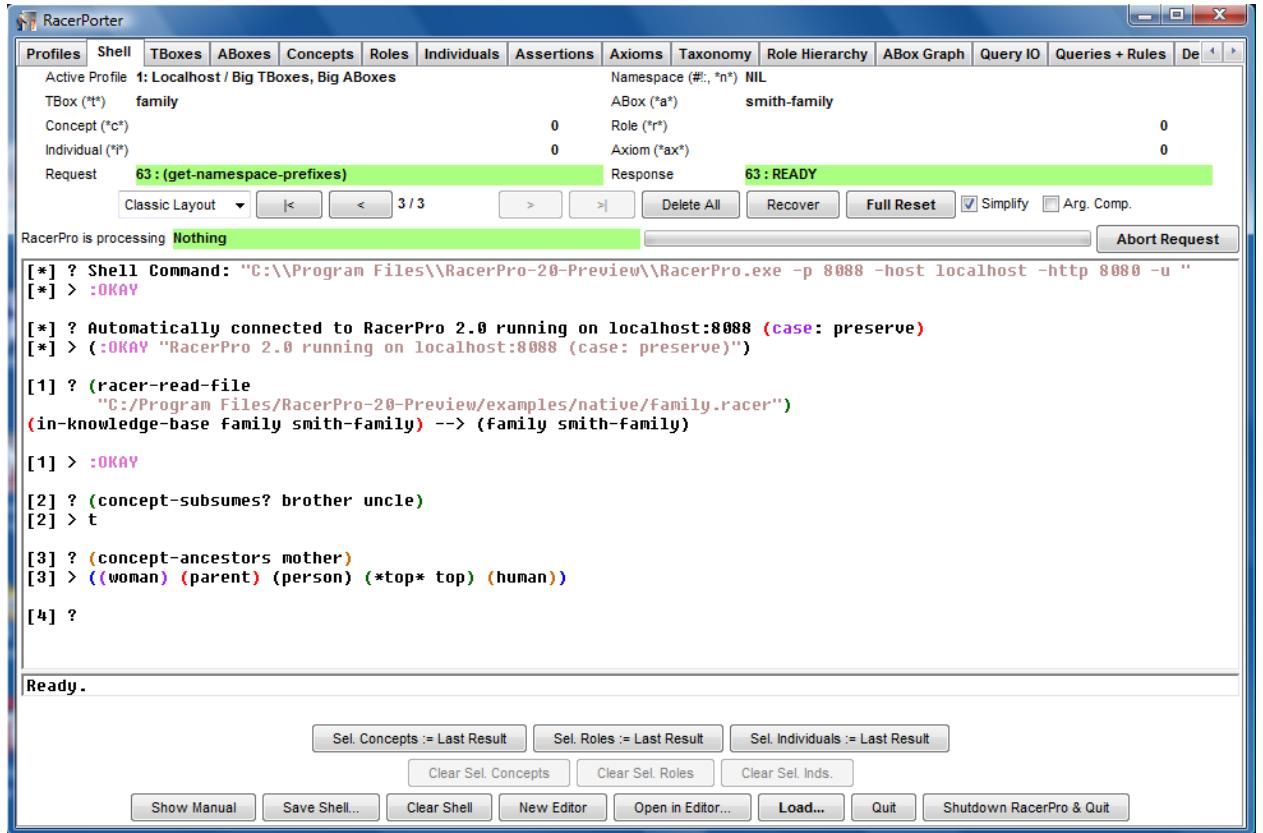


Figure 2.3: The Shell tab of RacerPorter used for executing queries.

tab. In the “A-box Graph” tab (see Figure 2.4) adjust the display options to all individuals (All Inds.) and select a depth, for example depth 3. The graph can show told and implicit assertions.

2.1.2 Using RacerPro from the Command Line

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

Type RacerPro -f examples/native/family.racer -q examples/native/family-queries.racer into a shell window in order to see the results (under Windows type RacerPro -- -f examples/native/family.racer -q examples/native/family-queries.racer), see Figure 2.5. The following section describes more details on how to use RacerPro from the command line.

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

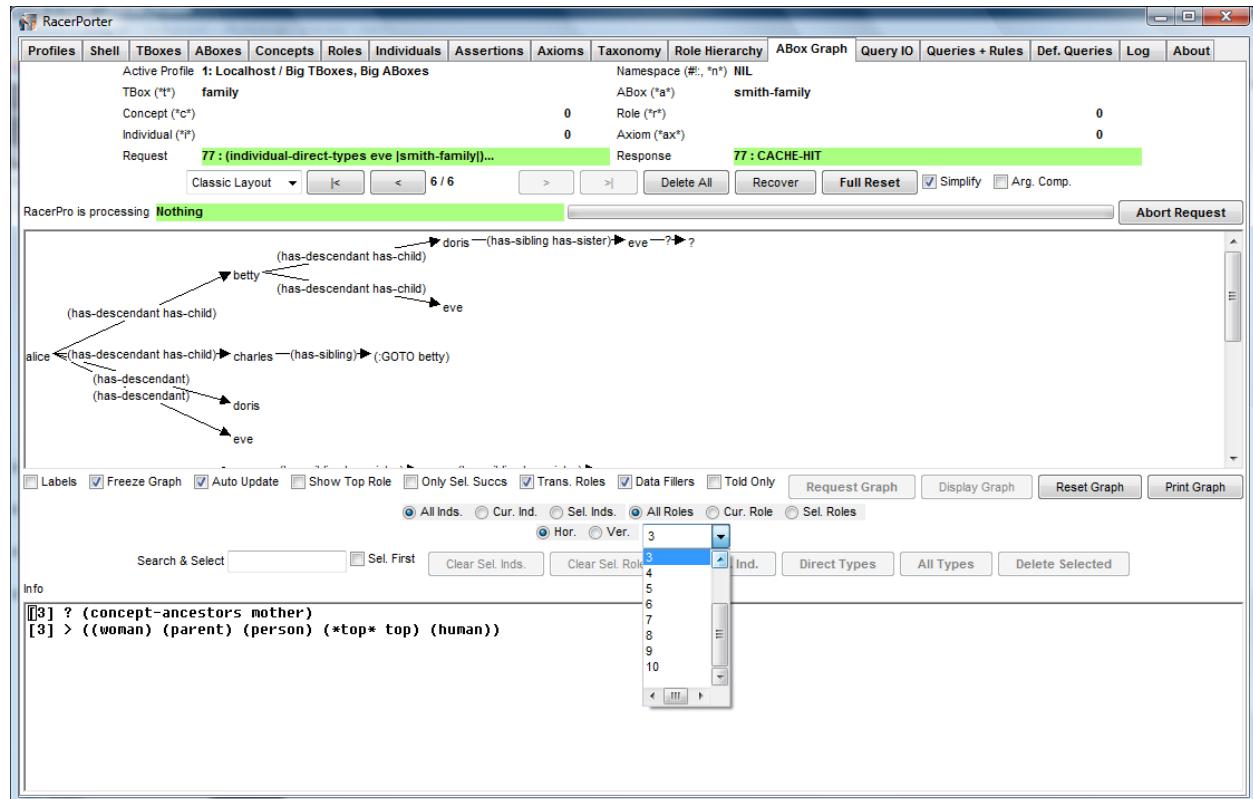


Figure 2.4: The A-box Graph tab shows the explicit relations for A-box individuals.

```

RacerPro

(in-knowledge-base family smith-family) --> (family smith-family)

Concept (parent) causes a cycle in TBox family.(concept-subsumes?
brother
uncle) --> t

(concept-ancestors mother) --> ((woman)
(parent)
(person)
(*top* top)
(human))

(concept-descendants man) --> ((*bottom* bottom) (uncle) (brother)
(father))

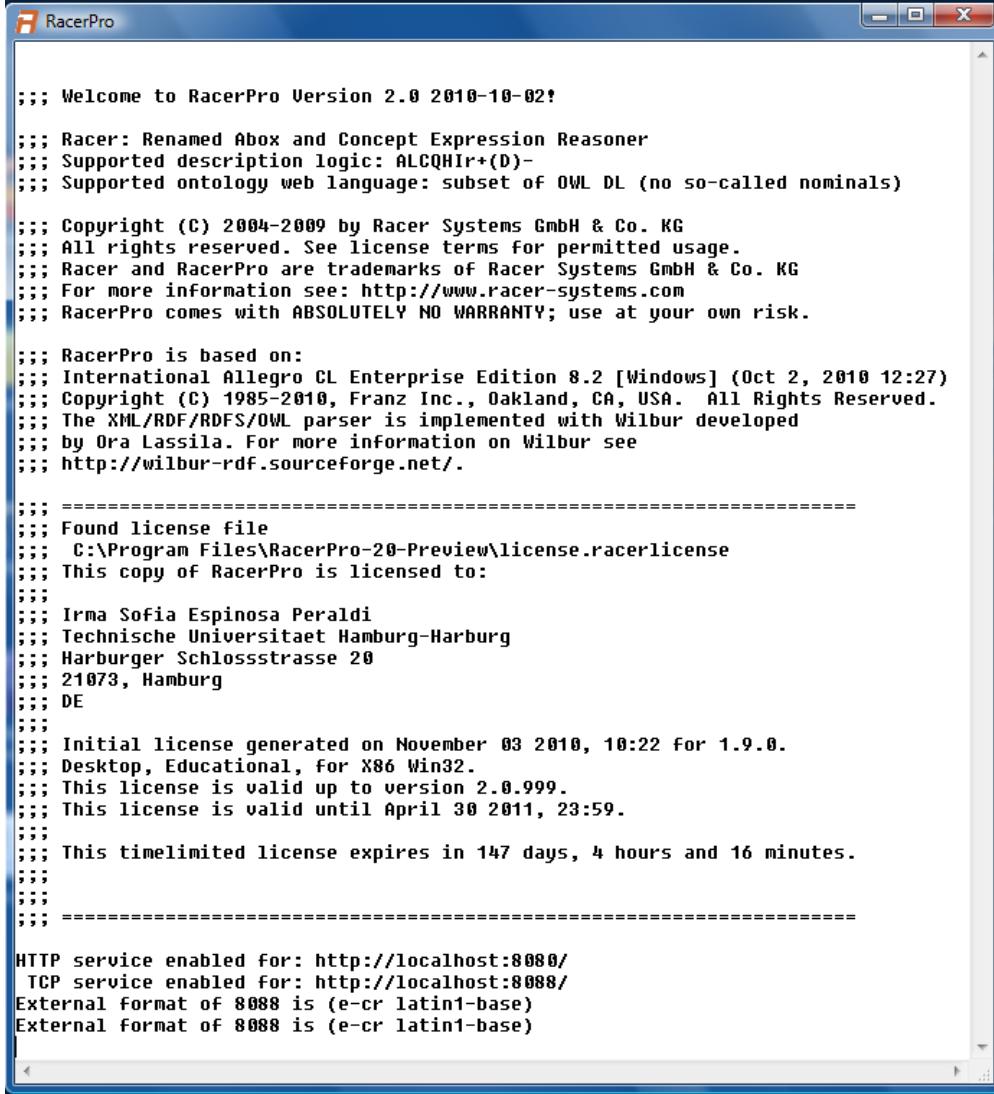
(all-transitive-roles) --> ((inv has-descendant) has-descendant)
(individual-instance? doris woman) --> t
(individual-types eve) --> ((sister) (woman) (person) (human)
(*top* top))
(individual-fillers alice has-descendant) --> (charles eve doris betty)
(individual-direct-types eve) --> ((sister))
(concept-instances sister) --> (betty eve doris)

```

Figure 2.5: Console of RacerPro.

2.2 The RacerPro Server

The RacerPro server is an executable file available for Linux, Mac OS X, Solaris 2, and Windows. It can be started implicitly using RacerPorter or explicitly 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 2.6.



```

RacerPro

;;; Welcome to RacerPro Version 2.0 2010-10-02!
;;;
;;; 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) 2004-2009 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 8.2 [Windows] (Oct 2, 2010 12:27)
;;; Copyright (C) 1985-2010, 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
;;; C:\Program Files\RacerPro-20-Preview\license.racerlicense
;;; This copy of RacerPro is licensed to:
;;;
;;; Irma Sofia Espinosa Peraldi
;;; Technische Universitaet Hamburg-Harburg
;;; Harburger Schlossstrasse 20
;;; 21073, Hamburg
;;; DE
;;;
;;; Initial license generated on November 03 2010, 10:22 for 1.9.0.
;;; Desktop, Educational, for X86 Win32.
;;; This license is valid up to version 2.0.999.
;;; This license is valid until April 30 2011, 23:59.
;;;
;;; This timelimited license expires in 147 days, 4 hours and 16 minutes.
;;;
;;; =====
HTTP service enabled for: http://localhost:8080/
TCP service enabled for: http://localhost:8088/
External format of 8088 is (e-cr latin1-base)
External format of 8088 is (e-cr latin1-base)

```

Figure 2.6: 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.

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 and that you are providing the correct search path for the `.racer` files. According to the distributions the files used in the following examples are found under `examples/native/`. In your favorite shell on Unix-based systems type:

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

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.racer
```

The option `--` separates window management options from RacerPro options. A window option we use here is `+c` which suppresses 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.racer
```

If an error occurs you can read the error message in the shell window. The option `-h` prints some information about the usage of the program RacerPro (see Figure 2.7).

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

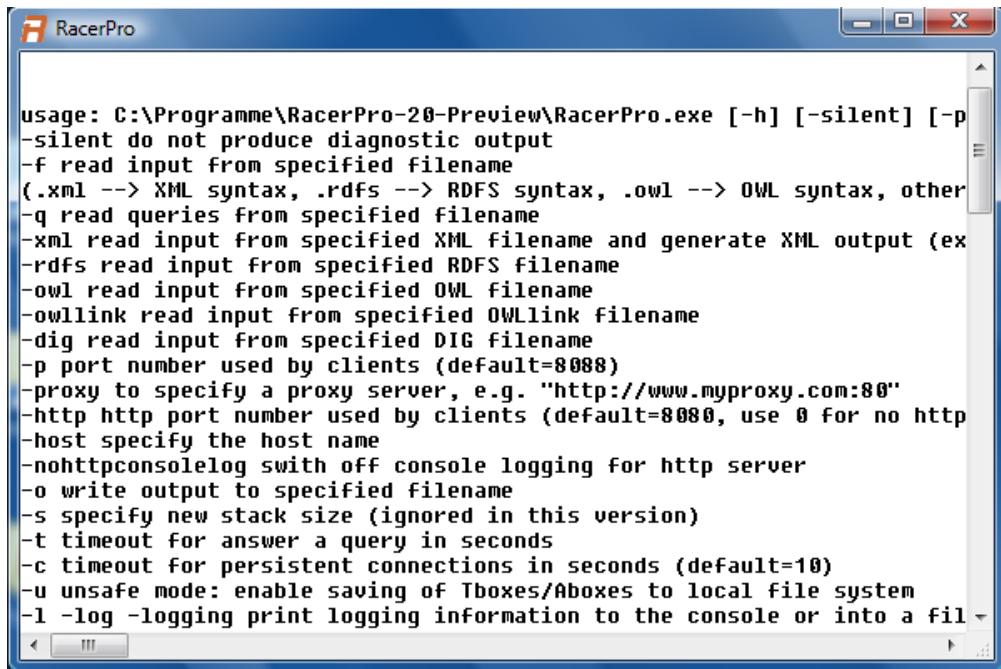


Figure 2.7: The shell window of RacerPro showed when using the option `-h`.

The `-f` RacerPro option has the following meaning. The input file is `family.racer` and the queries file is `family-queries.racer`. 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.racer -o ouput.text
```

Or use the following under Windows as explained above:

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

Let us assume an OWLlink knowledge base is given in the file `kb.owllink`. In order to submit a query to RacerPro just create a file `q.racer` with queries in KRSS syntax and start RacerPro as follows:

```
$ RacerPro -owllink kb.owllink -q q.racer
```

Note the use of the option `-owllink` for specifying that the input knowledge base is in OWLlink syntax. If you would like to use OWLlink syntax for queries as well, just create `q.racer` with contents (`owllink-read-file "queries.owllink"`)

Under Windows, you might want to suppress the shell window:

```
$ RacerPro +c -- -owllink kb.owllink -q q.racer
```

For any performance evaluation please refer to Section 3.9 in order to properly measure system runtimes for reasoning subtasks.

2.2.2 TCP Interfaces

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.

JRacer

JRacer is the client library to access the services of a RacerPro server from Java client programs. In addition to the jar archive for the class files, the JRacer interface is provided with source code in your RacerPro distribution directory. See <http://www.racer-systems.com/products/download> for the latest JRacer version. JRacer uses the socket interface to communicate with the RacerPro server. 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 such that the results can be used in the Java program. The following very simple code fragment explains how to send messages to a RacerPro server running on 127.0.0.1 (localhost) under port 8088. The file used in the code is provided in the RacerPro distribution directory. You might need to adapt the string according to your installation directory for RacerPro.

```
package test;

import com.racersystems.jracer.*;

public class Test {

    public static void main(String[] argv) {
        String ip = "127.0.0.1";
        int port = 8088;
        String filename="\\""/Applications/RacerPro 2.0 preview/examples/owl/people-pets.owl"";

        RacerClient racer = new RacerClient(ip,port);
        try {
            racer.openConnection();
            System.out.println(racer.sendRaw("(owl-read-file " + filename + ")"));
            System.out.println(racer.sendRaw("(all-atomic-concepts)"));
        }
    }
}
```

```

        racer.closeConnection();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

}

```

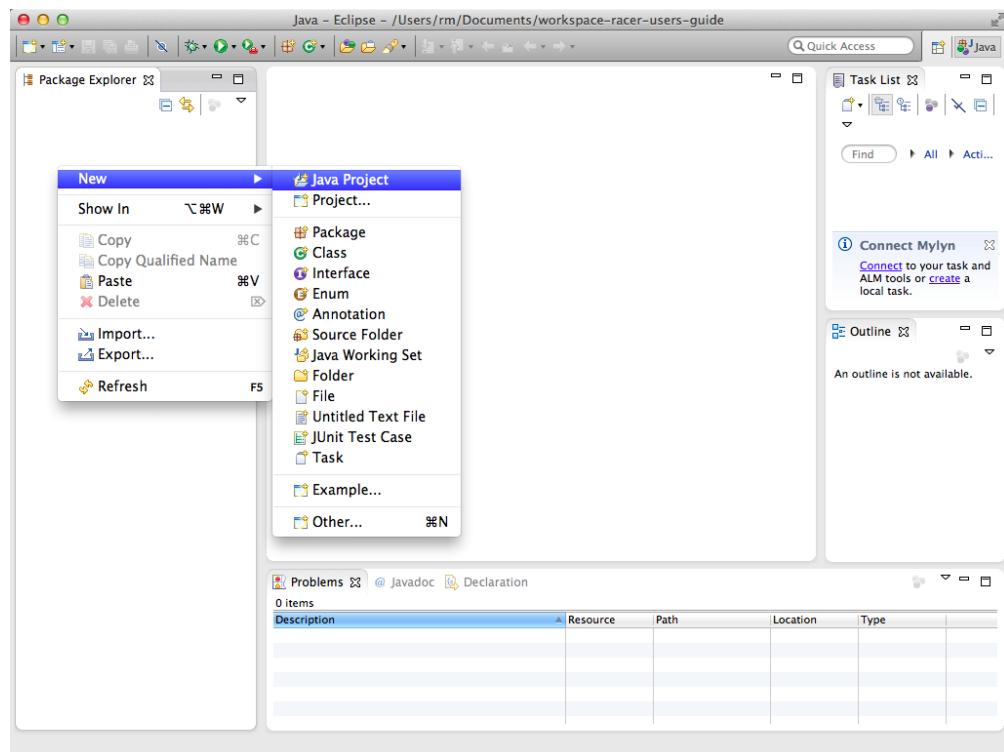


Figure 2.8: Create a project with Eclipse.

We demonstrate how to run the test program from within Eclipse (here the Juno release of Eclipse is used). We use `workspace-racer-users-guide` as a workspace name, and a corresponding directory is assumed to exist in the standard workspace path for Eclipse. For running the test program, create a new project in Eclipse (see Figures 2.8 and 2.9). For demonstration purposes, we have used `test1`. In the workspace directory, a project directory will be created automatically. In the project directory `test1` you should create a directory `lib` into which you copy the file “`jtracer.jar`” (in `test1` you should already find the directories `src` and `bin`). After returning to Eclipse you refresh the display to show the `lib` directory (see Figure 2.10). Then, select “`jtracer.jar`” in the `lib` subdirectory, do a right-click, and add the file to the project’s build path (menu item Add to Build Path, see Figure 2.11). With a right-click on the `src` directory, create a new package `test` (see Figures 2.12 and 2.13). Then, right-clicking on the package `test` you create a new file with name “`Test.java`” (Figure 2.14). Double-click this file and copy the code of the example into the window (see Figure 2.15). If necessary, change the value of the variable `filename` according to your installation.

Then, start RacerPro, for instance using RacerPorter, such that RacerPro runs on the local host and provides its services using port 8088 (the default, see Figures 2.16 and 2.17). If RacerPorter is connected to RacerPro, press the button “Disconnect” (see Figure 2.16) in order to allow you

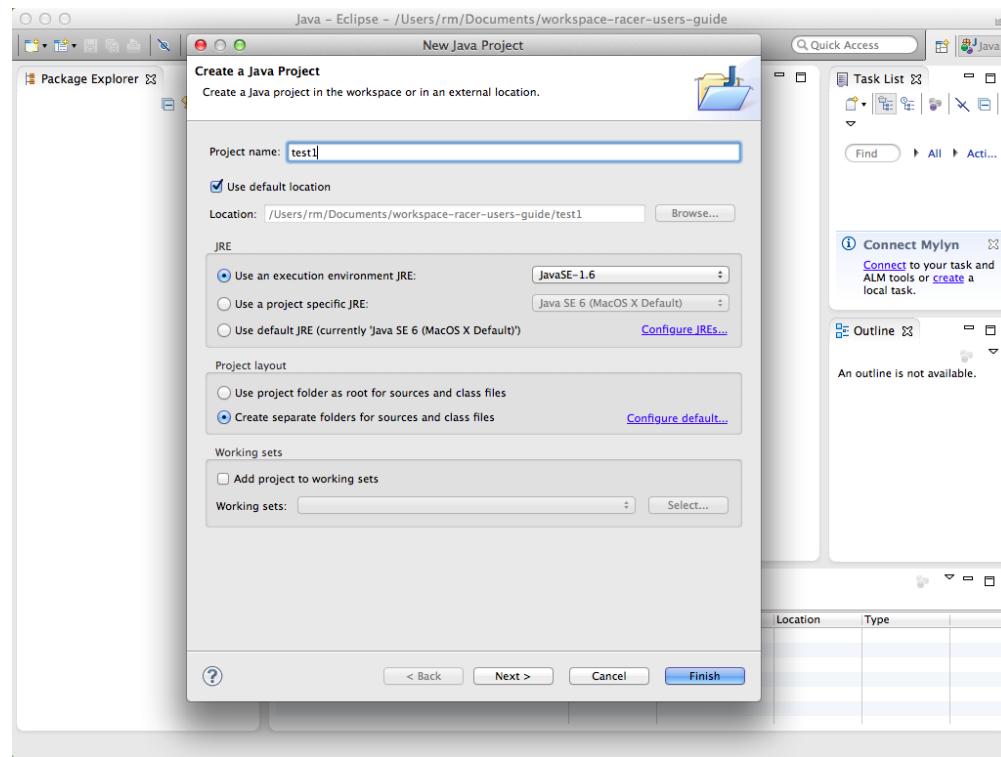


Figure 2.9: Select a name for the project.

client application to connect to the RacerPro server. You can then run the project `test1` (see Figure 2.18). The output is printed to the Eclipse Console window (Figure 2.19).

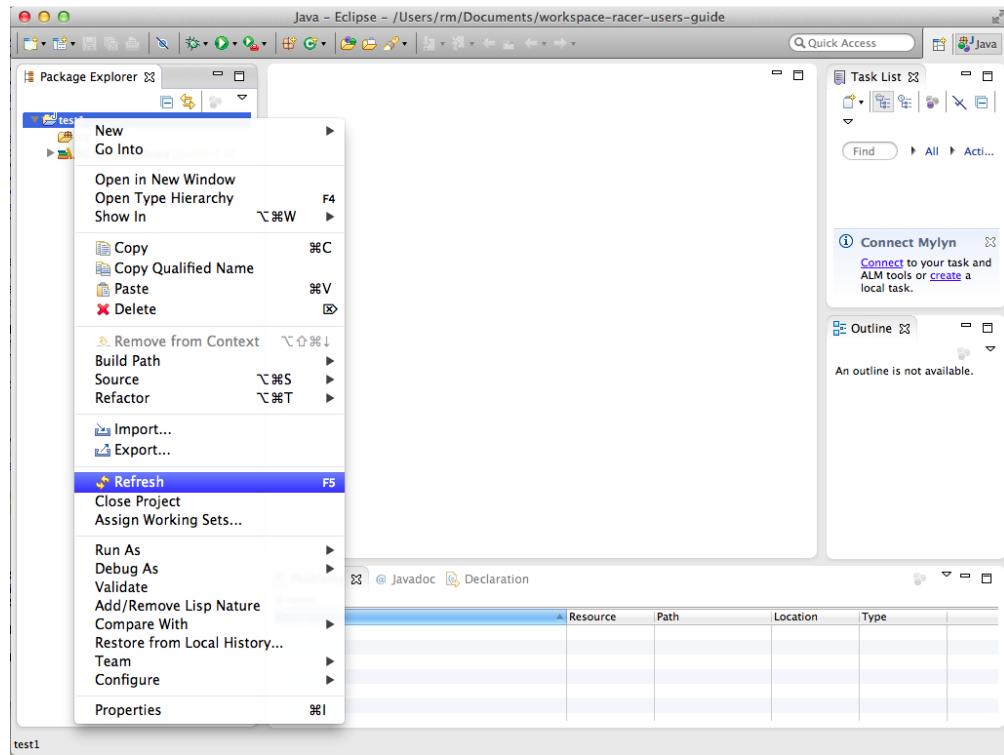


Figure 2.10: Refresh the project display after having created a directory `lib` into which you copied “jracer.jar”.

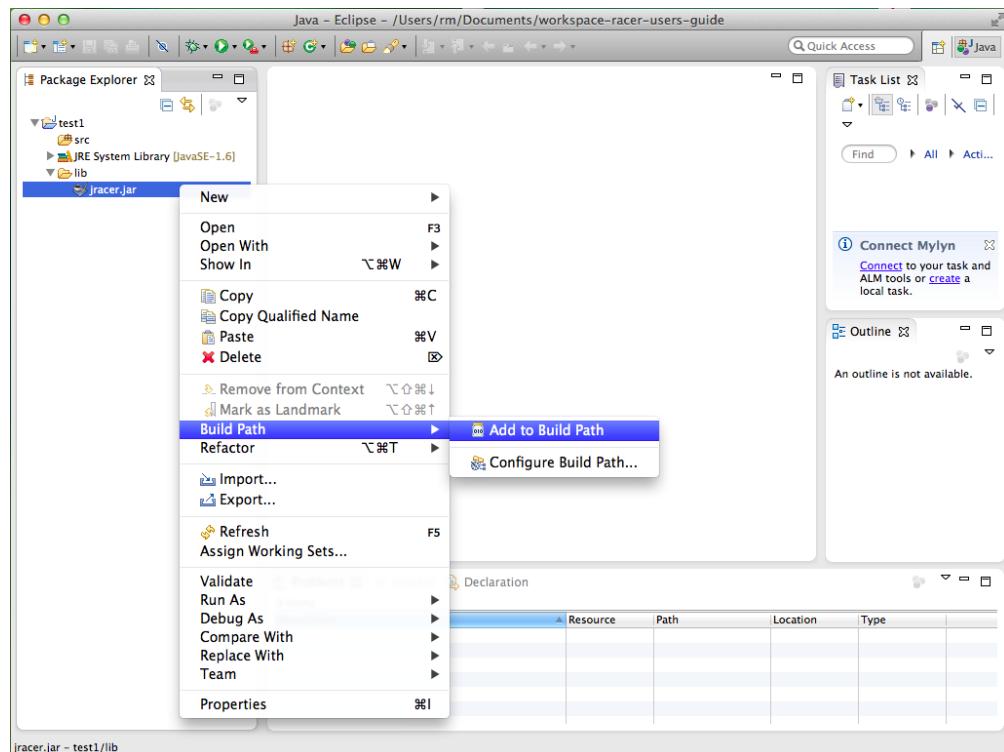


Figure 2.11: Select “jracer.jar” in the lib directory, do a right-click, and add the file to the project’s build path.

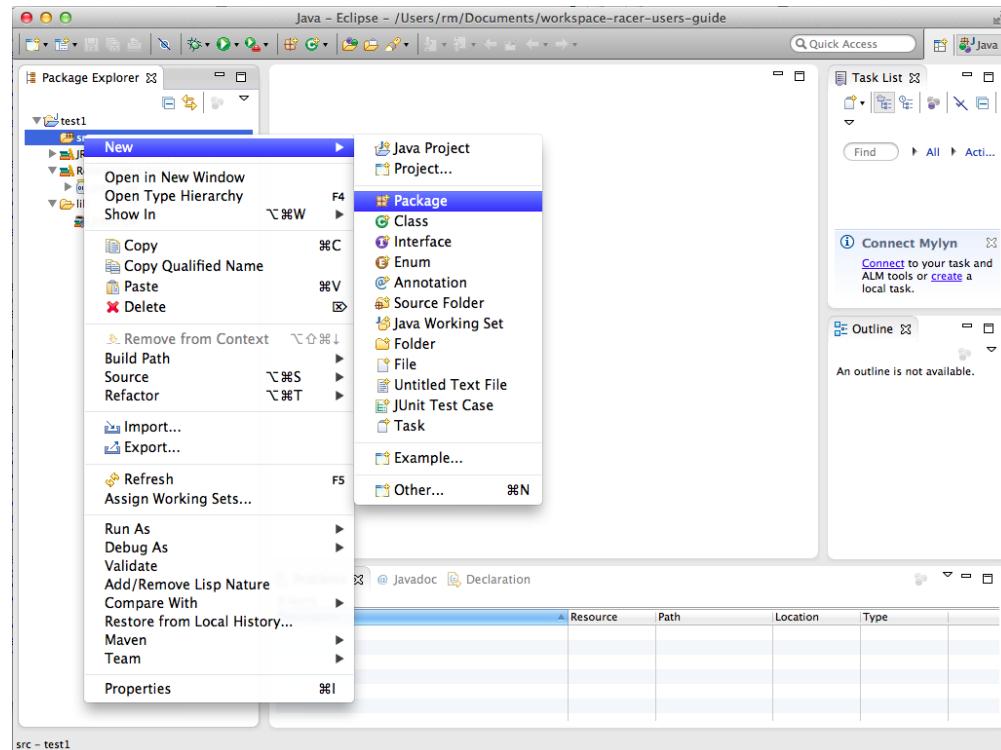


Figure 2.12: Create a new package.

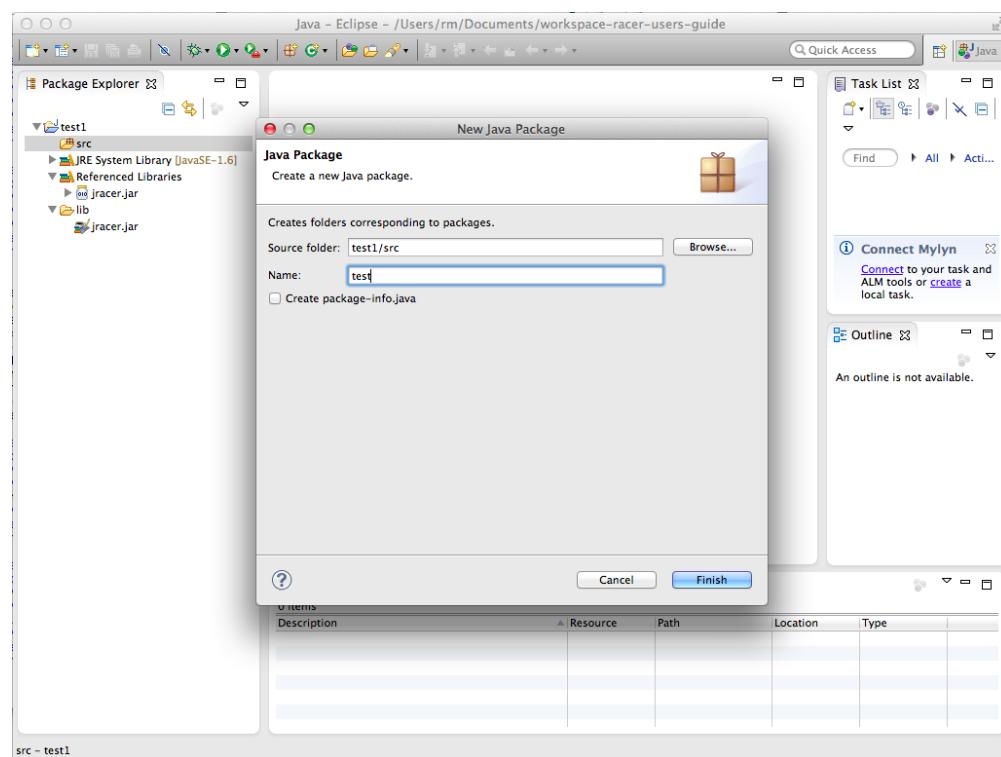


Figure 2.13: Call the package test.

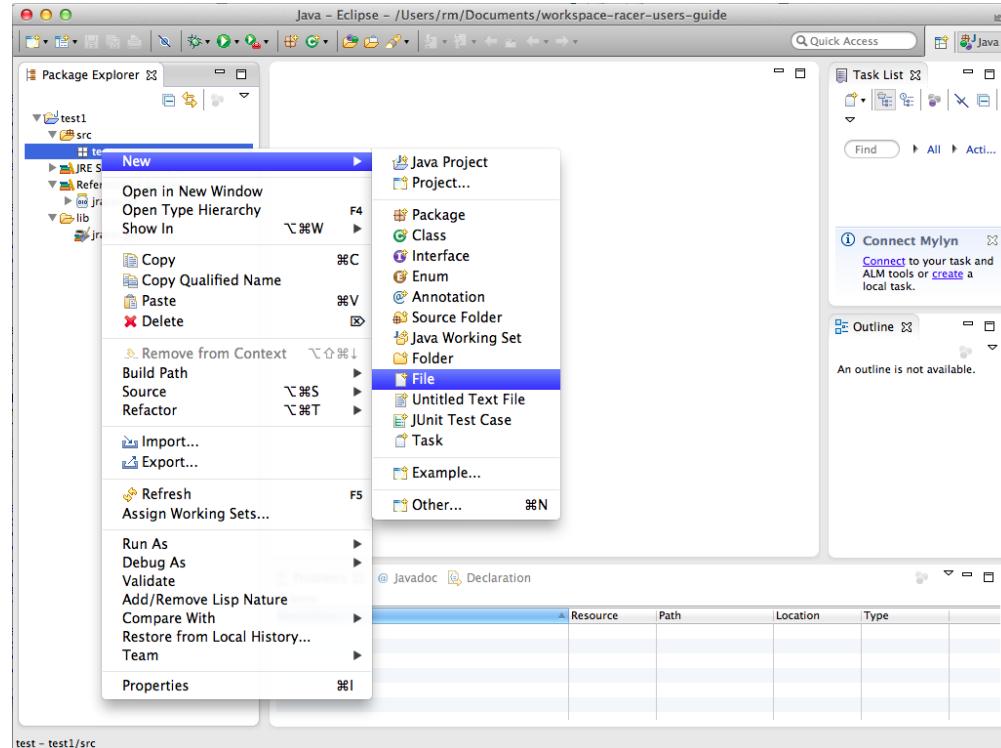


Figure 2.14: Create a file “Test.java”.

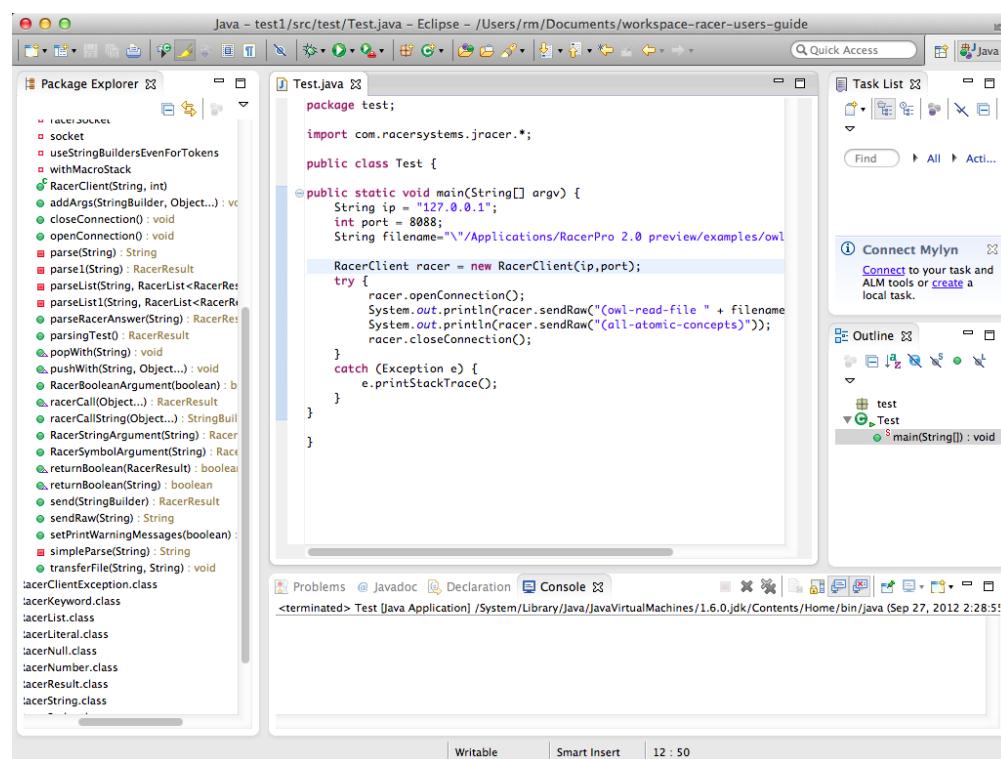


Figure 2.15: Source code for the example inserted

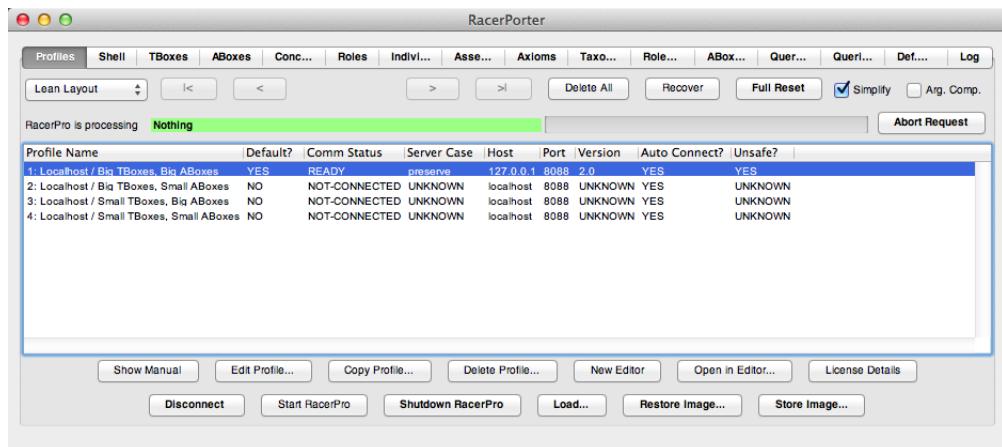


Figure 2.16: RacerPorter.

```
rm — RacerPro — 132x40

;; RacerPro is based on:
;; International Allegro CL Enterprise Edition 8.2 [Mac OS X (Intel)] (Apr 30, 2012 5:56)
;; Copyright (C) 1985-2011, 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-master.racerlicense
;; 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 November 01 2006, 18:06 for 1.9.1.
;; Site, Commercial, for MASTER.
;; This license is valid up to version 9.9.99.
;; This license is valid forever.
;;
;; This is a MASTER LICENSE for Ralf Moeller
;;
;;
=====

HTTP service enabled for: http://localhost:8080/
TCP service enabled for: http://localhost:8088/
TCP control enabled for: http://localhost:8089/
External format of 8088 is utf8-base
```

Figure 2.17: RacerPro with output.

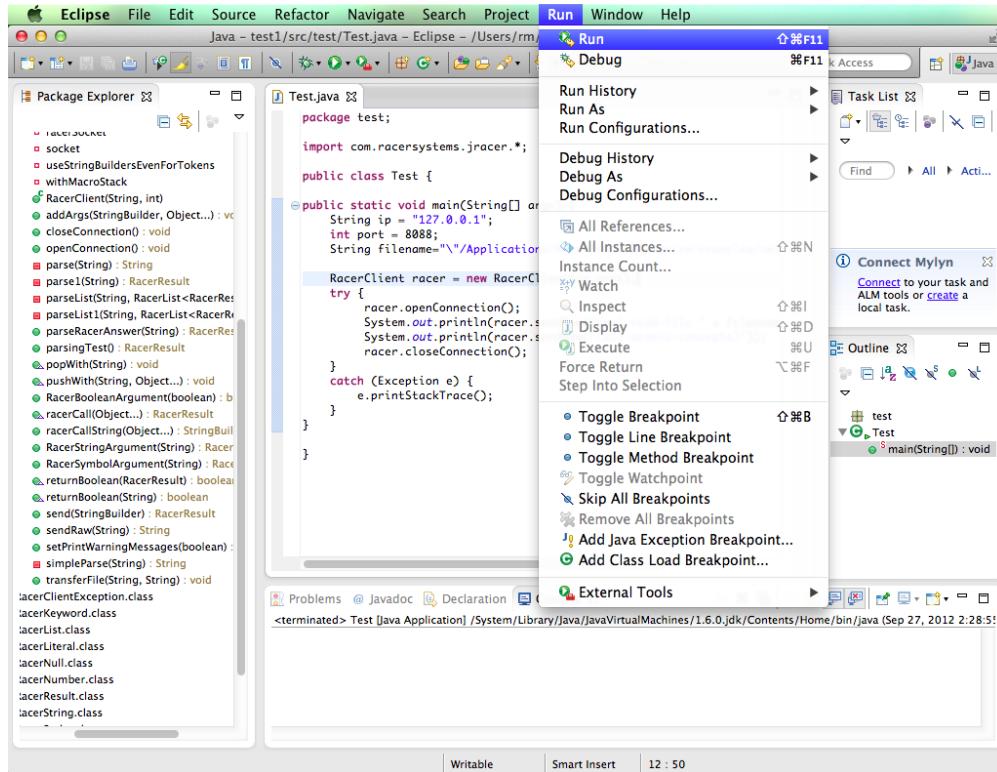


Figure 2.18: Run the project.

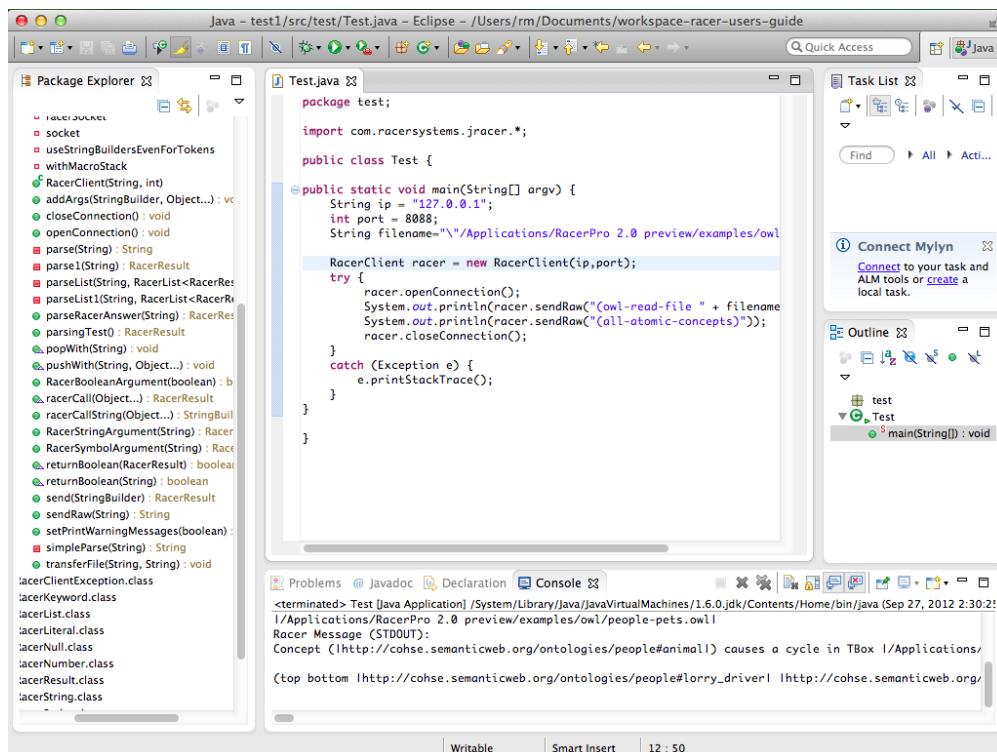


Figure 2.19: The output is printed to the Console window.

Sending raw strings to the RacerPro server is simple, but rather cumbersome from a software engineering point of view. Therefore, as part of JRacer, RacerPro provides a simple API with Java classes and methods for creating and manipulating axioms in ontologies, and for invoking inference services. An example is given below.

```

package test;

import com.racersystems.jracer.*;

public class Test2 {

    public static void main(String[] argv) {
        String ip = "127.0.0.1";
        int port = 8088;
        RacerClient racer = new RacerClient(ip, port);

        try {
            racer.openConnection();

            racer.fullReset();
            racer.owlapiNewReasoner("test");
            racer.owlapiNewOntology("test");
            racer.owlapiAutoAddAxiomsTo("test");

            racer.owlapiGetOWLClassAssertionAxiom("d", "j");
            racer.owlapiGetOWLObjectPropertyAssertionAxiom("j", "r", "k");

            racer.owlapiLoadOntology("test");

            System.out.println(racer.tboxCoherentP("test"));
            System.out.println(racer.aboxConsistentP("test"));

            racer.closeConnection();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Javadoc is provided for all JRacer methods (Figure 2.20).

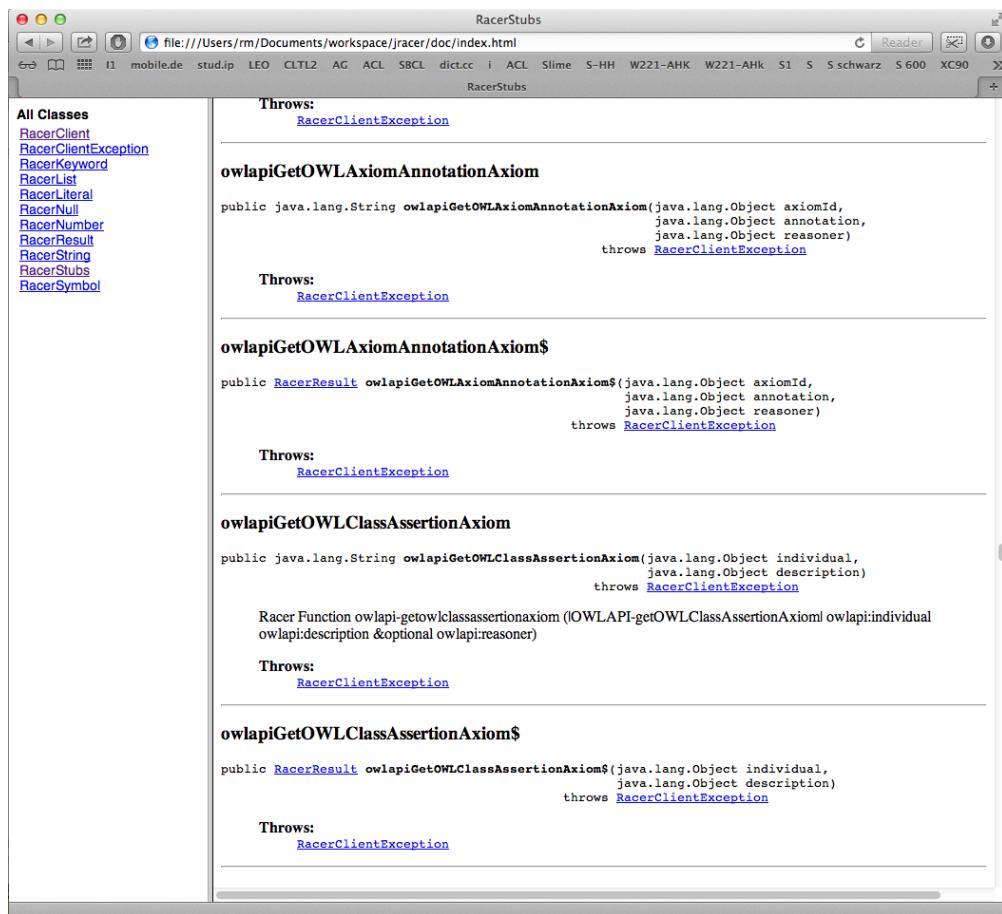


Figure 2.20: JRacer documentation using Javadoc.

In Eclipse the example can be run in a similar way as explained above. In Figure 2.21 the output of the above program is shown in the Console window.

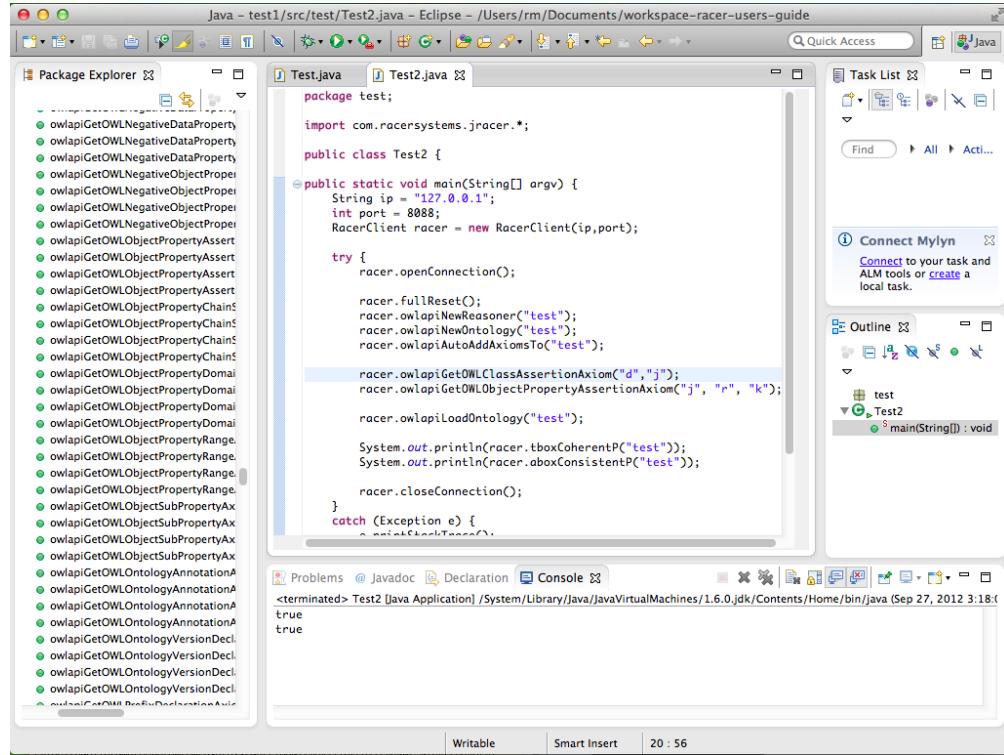


Figure 2.21: Second example with output printed to the Console window.

In the following example we demonstrate how to access the result(s) returned from RacerPro. For instance, a list of binding lists is returned as an answer to a conjunctive query. A binding is a list consisting of a variable name and an individual name (both of which are symbols). The example shows how to iterate over RacerPro result data structures from within Java programs. As you can see in the code, the method `racerAnswerQuery` has a \$ as a suffix in the second call. In case the suffix \$ is used, a data structure is returned rather than only a string (which can be easily printed, as shown in the first call, but otherwise have to be parsed by the user to access the results for subsequent computations). The Figure 2.22 for the output.

```

package test;

import com.racersystems.jracer.*;

public class Test3 {

    public static void main(String[] args) {

        String filename=
            "/Applications/RacerPro 2.0 preview/examples/owl/people-pets.owl";
        String ip = "localhost";
        int port = 8088;

        RacerClient racer = new RacerClient(ip, port);

        try {
            racer.openConnection();

```

```
racer.fullReset();

racer.owlReadFile("\\""+filename+"\\");

System.out.println(racer.racerAnswerQuery
  ("(?x ?y)",
   "(and (?x #!:person) (?x ?y #!:has_pet))"));

@SuppressWarnings("unchecked")
RacerList<RacerList<RacerList<RacerSymbol>>>
  res2 = (RacerList<RacerList<RacerList<RacerSymbol>>>
    racer.racerAnswerQuery$("(?x ?y)",
      "(and (?x #!:person) (?x ?y #!:has_pet))");

for (RacerList<RacerList<RacerSymbol>> bindings : res2) {
  for (RacerList<RacerSymbol> binding : bindings) {
    for (RacerSymbol varval : binding) {
      System.out.println(varval);
    }
  }
}

} catch (Exception e) {
  e.printStackTrace();
}

}

}
```

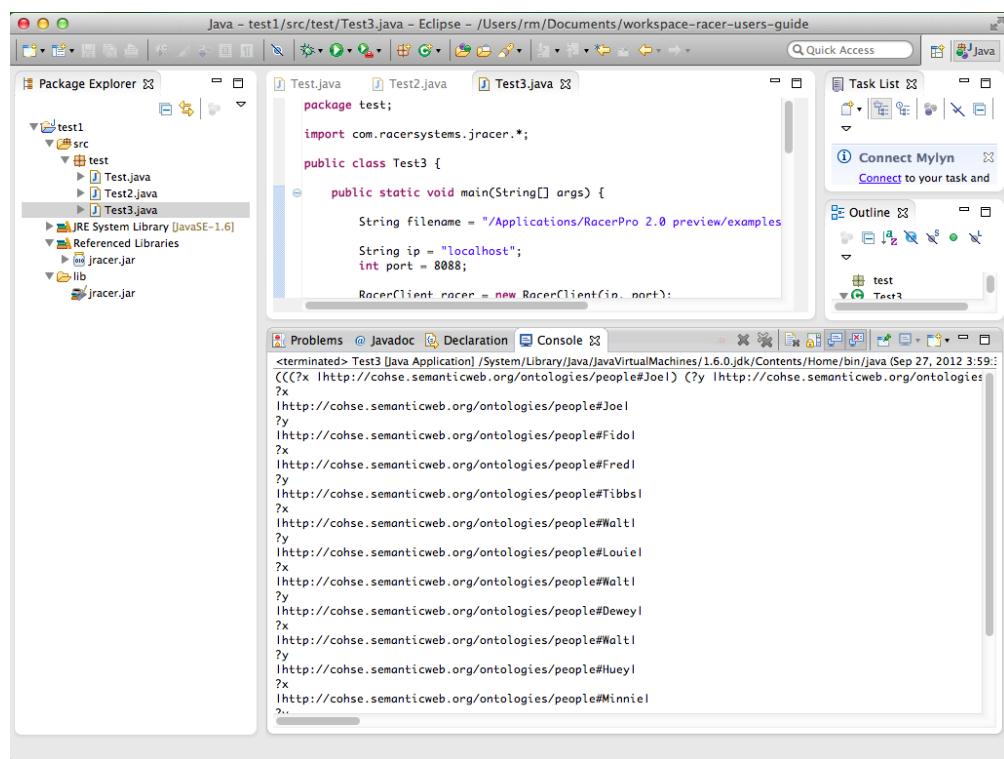


Figure 2.22: Third example with output printed to the Console window.

Java OWLAPI Interface

The Manchester OWLAPI is also supported by RacerPro. You can download the respecitve jar files from <http://www.racer-systems.com/products/download>. As shown in Figure 2.23 you can set up the project as described for JRacer, and you need to put “owlapi-bin.jar” and “racer.jar” (NB: not “jracer.jar” but “racer.jar”) onto the build path of the project.²

```

package test;

import com.racersystems.racer.*;
import org.semanticweb.owlapi.apibinding.*;
import org.semanticweb.owlapi.model.*;
import org.semanticweb.owlapi.reasoner.*;
import java.util.Set;

public class example {

    public static void main(String[] args) {

        String DOCUMENT_IRI =
            "file:///Applications/RacerPro%202.0/examples/owl/people-pets.owl";

        try {
            OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
            IRI docIRI = IRI.create(DOCUMENT_IRI);
            OWLOntology ont = manager.loadOntologyFromOntologyDocument(docIRI);
            OWLReasonerFactory reasonerFactory = new ReasonerFactory();
            ConsoleProgressMonitor progressMonitor = new ConsoleProgressMonitor();
            OWLReasonerConfiguration config =
                new SimpleConfiguration(progressMonitor);
            OWLReasoner reasoner = reasonerFactory.createReasoner(ont, config);

            boolean consistent = reasoner.isConsistent();
            System.out.println("Consistent: " + consistent);
            System.out.println("\n");

            Node<OWLClass> bottomNode = reasoner.getUnsatisfiableClasses();
            Set<OWLClass> unsatisfiable = bottomNode.getEntitiesMinusBottom();
            if (!unsatisfiable.isEmpty()) {
                System.out.println("The following classes are unsatisfiable: ");
                for (OWLClass cls : unsatisfiable) {
                    System.out.println("    " + cls);
                }
            } else {
                System.out.println("There are no unsatisfiable classes");
            }
            System.out.println("\n");

            reasoner.dispose();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

²The packages in the code are not hierarchical although the names suggest this. You need three imports for the OWLAPI.

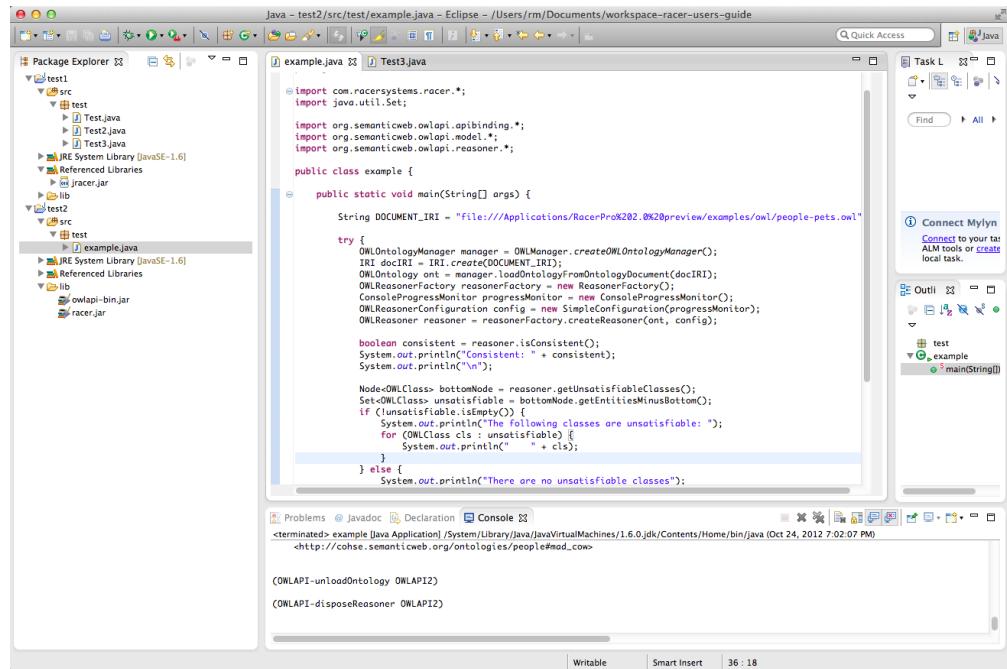


Figure 2.23: An Eclipse project for using RacerPro with the Manchester OWL API.

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 function 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-sysdcl.lisp")`. Then, evaluate `(compile-load-racer "~/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.

OWLlink Interface

In a similar way as the direct TCP socket interfaces, the RacerPro HTTP interfaces can be used from application programs (and graphical interfaces) in order to communicate with the RacerPro reasoner. OWLlink is a W3C member submission (<http://www.w3.org/Submission/owllink-HTTPxml-binding/>) for an HTTP-based communication protocol. RacerPro implements OWLlink directly and does not need an adapter as others reasoners (see <http://owllink-owlapi.sourceforge.net> for a more detailed introduction).

```
$ RacerPro -- -protocol owllink
```

Instructs RacerPro to use the OWLlink protocol using the default HTTP port (8088). The default OWLlink syntax is XML. In order to use the functional syntax start RacerPro as follows.

```
$ RacerPro -- -protocol owllink -owllink-input-syntax functional
```

You can also use

```
$ RacerPro -- -protocol owllink -owllink-input-syntax sexpr
```

if you prefer the sexpr syntax. The option `-owllink-output-syntax` with the obvious semantics is also available.

2.2.3 Options for 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 `--` before specifying options 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`, `.owllink`, `.rdfls`, `.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`, `.owllink`, `-rdfls` instead of `-f`.
- With `-o <filename>` RacerPro can be instructed to write output to `filename` rather than to the console.
- 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).
- `-socket-retries <n>` specifies the number of retries that RacerPro uses for starting communication sockets if they are still in use.
- `-httplogdir <directory>` specifies the logging directory (see above).
- `-nohttpconsolelog` disables console logging for HTTP connections (see above).
- `-ef <external-format>` defines the external format the RacerPro server uses for socket communication (e.g., `@utf8` is a possible value).

- `-fef <external-format>` defines the external format the RacerPro server uses for file input (e.g., `@utf8` is a possible value).
- With the option `-profile <profile-name>` the protocol for HTTP communication is specified (use, e.g., `owllink` for using the OWLlink protocol).
- You can use `-owllink-input-syntax sexpr` or `-owllink-input-syntax functional` to specify the OWLlink syntax. Ditto for `-owllink-output-syntax`.
- 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 5.5 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 2.2.4 about how to send a bug report.
- Specify the option `-m` (or `-x`) to save some memory by instructing Racer that the KRSS interface is not used for manipulating axioms. This is particularly useful if the OWLAPI is used since axioms are maintained as part of the OWLAPI itself.
- 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 `-dig11` 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>]`. If no filename is supplied output is directed to stdout. In case only the license string is required, use `dump-license-string` instead (same arguments).

- With `-t <s>` a timeout for queries can be specified (in seconds).
- With `-c <s>` a timeout for persistent connections can be specified (in seconds).
- With the option `-alisp` RacerPro is instructed not to be case sensitive (used for backward compatibility only).
- `-no-patches` disables automatic loading of default patches from default patch directory
- `-no-plugins` disables automatic loading of default plugins from default plugin directory
- `-patch-directory <directory>` specifies the default patch directory (these patches are automatically loaded)
- `-plugin-directory <directory>` specifies the default plugin directory (these plugins are automatically loaded)
- `-patches <directory>` loads all patches from the specified directory after the default patches have been loaded (if any)
- `-plugins <directory>` loads all plugins from the specified directory after the default plugins have been loaded (if any)
- `-patch <filename>` loads a single patch after the default patches and other patches have been loaded (if any)
- `-plugin <filename>` loads a single plugin after the default plugins and other plugins have been loaded (if any)
- `-check4updates` checks for online updates
- `-update` download and install online updates
- `-update-from <URL>` (default: <http://www.racer-systems.com>)

Note again that under Windows, RacerPro options have to be separated with `--` from window manager options (see the next section for an example). The following options are important to control the behavior of the console window:

- The option `+p` makes 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).

2.2.4 How to Send Bug Reports

Although RacerPro has been used in some application projects, 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.

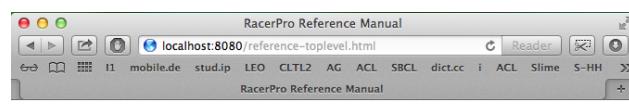
In case you have problems with the license file `license.racerlicense` you can start RacerPro with the option `-license <license-file>`. This prints information about the file `<license-file>`. More detailed information is printed with `-dump-license-info [<filename>]`. Please send also this information.

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.

2.2.5 Online Reference Manual

The reference manual is now available online (see Figures 2.24 and 2.25). Make sure a RacerPro server is started. Use your favorite web browser to view <http://localhost:8080/reference-toplevel.html>. A collection of all functions provided by RacerPro is available under <http://localhost:8080/all-functions.html>.

Help on function arguments is also provided by RacerPorter (see the example shown in Figure 2.26).



The screenshot shows a web browser window titled "RacerPro Reference Manual". The address bar displays "localhost:8080/reference-toplevel.html". The toolbar includes standard icons for back, forward, search, and refresh, along with links to "mobile.de", "stud.ip", "LEO", "CLTL2", "AG", "ACL", "SBCL", "dict.cc", "i", "ACL", "Slime", "S-HH", and "Reader". The main content area is titled "Section RacerPro Reference Manual" and contains the following text:

Please choose:

Sub Sections

[Knowledge Bases](#)
[General Functions](#)
[The OWL Interface](#)
[The new RacerPro Query Language \(nRQL\)](#)
[Input/Output Functions](#)
[The RacerPro OWLAPI \(NOSA\)](#)
[The OWLlink Interface](#)
[The DIG Interface](#)
[Server-Scripting with MiniLisp](#)
[The Explanation Facility](#)
[The Publish-Subscribe Facility](#)
[The Online Documentation Facility](#)
[Functions for Benchmarking and Testing](#)
[The Timenet Event Recognition Facility](#)
[RacerPro Plugins](#)
[RacerPro Patches](#)
[Updating RacerPro](#)

Figure 2.24: RacerPro Reference Manual (specific sections)

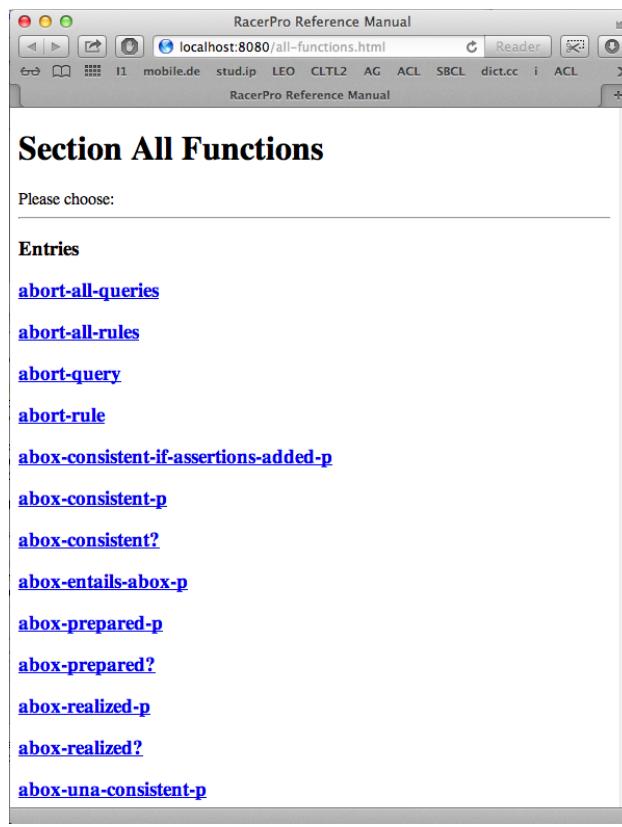


Figure 2.25: RacerPro Reference Manual (alphabetical list)

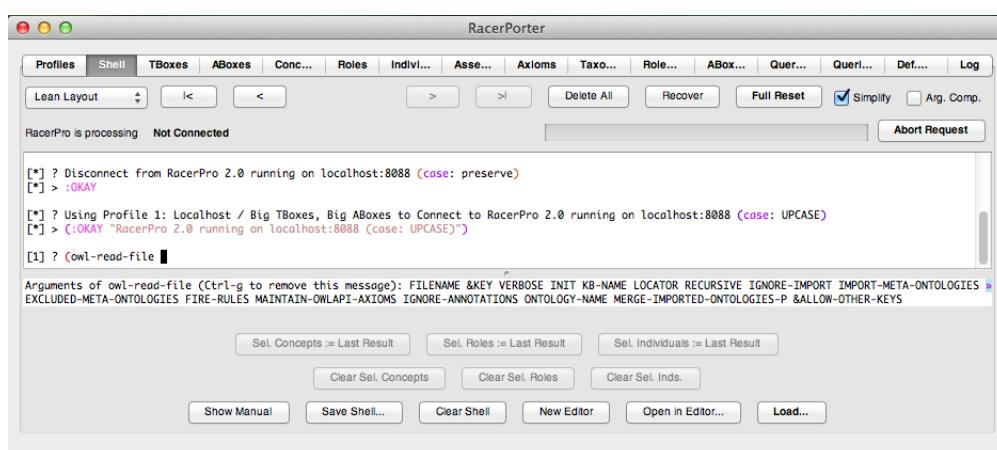


Figure 2.26: Documentation of function arguments in RacerPorter.

2.3 RacerPorter in Details

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.

The following items show a general overview of the features offered by RacerPorter:

- A powerful and convenient shell with shell history, completion facility, pretty-printing and syntax coloring.
- An Emacs-compatible editor with buffer evaluation mechanism. Supports KRSS, OWL(2) RDF, OWL2 Functional, SPARQL.
- The transmission of KBs or editor buffers to remote RacerPro servers.
- A-box, T-box and role hierarchy visualization facilities.
- Advanced focus and update control mechanisms.
- A clipboard mechanism for information interchange and focus control.
- Multiple session / multi-connection awareness (e.g., work with a different server while a taxonomy graph is computed in the background).
- An asynchronous, multi-threaded, non-blocking GUI.

2.3.1 Distribution of GUI elements

Each tab has a uniform organization, which, we believe, makes the GUI consistent and comprehensible. With the exceptions of the “Log” tab and the “About” tab, each tab has **six areas**. Figure 2.27 shows the “Taxonomy” tab, based on this tab, let us describe the six areas.

1. The tabs area. It shows the available tabs: Profiles, Shell, T-boxes, A-boxes, Concepts, Roles, Individuals, Assertions, Taxonomy, Role Hierarchy, A-box Graph, Query IO, Queries + Rules, Def. Queries, Log, and About tab.

2. The status display. There is a general notion of current / selected objects employed in RacerPorter, e.g., current T-box, current A-box, current concept, current individual, and so on. These current “objects” are shown in the corresponding fields in the status display. The current objects can be referred to in the shell via the indicated *...* variables as arguments to shell commands (e.g., enter “(describe-tbox *t*)” in the shell to get a description of the current T-box). The current profile represents the current server connection as well as the current communication status. The clipboard content is not shown, only the cardinality of the sets of selected objects (in the small number fields). The selected objects are highlighted once an appropriate tab is selected.

In general, the current / selected objects are employed for information flow and for focus control (= “what to display / visualize”) in RacerPorter. For example, each T-box “contains” a set of concept names. Thus, the content of the “Concepts” tab depends on the current T-box. RacerPro API functions often require certain arguments. These arguments are supplied by the current and selected objects (of the appropriate type).

A “postfix command mode pattern” is thus used as the main interaction pattern - first select the arguments, then select and invoke the operation on the selected arguments. If an operation requires only one argument, then the current object (of the required type) is used, e.g., the current concept. If an operator requires more than one argument, also the selected objects are passed as

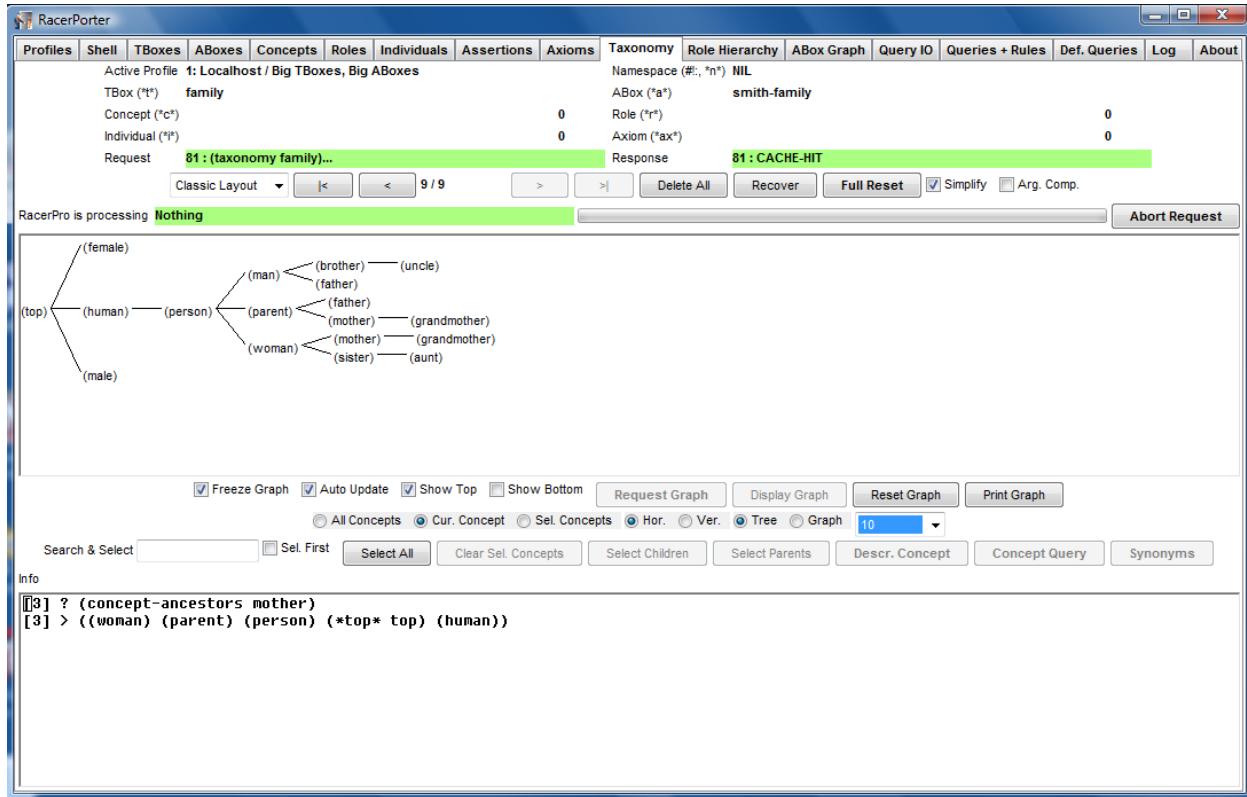


Figure 2.27: RacerPorter – The Taxonomy Tab

arguments to the operation. In general, an operator can be invoked in various ways (e.g., via a push button or via the shell). Multiple arguments (and thus, multi-selection of objects) are currently only supported for objects of type: concept, individual, role, and axiom. The number of selected objects (of the corresponding type) is shown in the little numeric box after the current object (of the corresponding type). Moreover, the selected objects are displayed in a highlighted style in the corresponding item lists, e.g., the selected concepts are highlighted in the “Concepts” tab which simply lists the concept names which are available in the current T-box.

3. The history navigator. For each interaction, a so-called history entry is created and maintained in a list, the so-called history. Previous views / states of the RacerPorter GUI can be reestablished easily with the help of the navigator buttons. Thus, it is possible to go back to / reawake a previous session state. Similar to a web browser, this navigation in interaction history is supported with the help of navigator buttons (back and forth). The current history position and length of the history is shown as well. The history entries can be deleted with the help of the “Delete All” button. Note that this does not affect the RacerPro server.

Sometimes, in case of errors, the widgets of RacerPorter are disabled, even though RacerPorter is still connected to a “living” RacerPro server which accepts requests. In order to recover RacerPorter from such errors and “reawake” its GUI, please use the “Recover” button. This button also invalidates the cache of RacerPorter. So, if you suspect that RacerPorter displays incorrect information, please press this button as well (the cache may have become inaccurate if another application has modified the state of the connected RacerPro server).

Next to the history navigator, two checkboxes are included, the “Simplify” and the “Arg. Comp.” checkbox.

If the “Simplify” checkbox is enabled, then namespace prefixes in OWL URIs (and long filename di-

rectory prefixes) are truncated and abbreviated with the “#!:”-prefix syntax. Please note that “#!:” denotes the current namespace. You can define additional prefixes with the help of the shell and the “define-prefix” command (e.g., “(define-prefix “racer” “http://www.racer-systems.com””)”). By default, the items to be displayed in this area (e.g., in the “Concepts” tab) are sorted alphabetically. If the “Arg. Comp.” (arguments completion) checkbox is enabled, then RacerPorter will collect the results of executed RacerPro commands in order to present them as input arguments in shell interactions. In general, if the tab key is pressed in the shell, a pop-up menu is presented, offering possible completions. This includes operator names as well as arguments (provided argument completion was enabled).

4. The tab-specific area. This area changes according to the selected tab. We describe the different tabs and their functionalities in Section 2.3.4. In general, there is a set of tab-specific buttons below the tab-specific main area.

The “Taxonomy” and the “A-box Graph” tabs use graph panes for this area. With the exception of the “Shell”, “Log” and “About” tab, the other tabs use list panes. List panes allow single or multiple selections of items; selected items represent the selected objects (clipboard). The last selected item specifies the current object. Some list panes display additional information on their items in multiple columns; e.g., in case of the T-box pane, not only the T-box name is shown, but also the number of concepts in that T-box, profile and DLS server information is shown in the profiles list, etc.

5. The tab-specific buttons area. This area shows options and commands that are useful for the related tab. Appropriate radio buttons are provided which control whether only told role assertions, or also inferred role assertions shall be shown.

Many tabs offer the “Search & Select” field which is used to select objects (individuals, concepts, roles, axioms) based on their names (or structured content in the case of axioms). Simply enter a text string into the “Search & Select” field and the objects which contain the search string somehow in their names (or in their structure, in the case of axioms) as a substring will become selected (note that you need to press enter). Please note that “Search & Select” always adds additional selected objects, but never deselects already selected objects (and thus works in an accumulating way). If required, the set of selected objects should be cleared manually by pushing the appropriate “Clear Sel. ...” button before a fresh “Search & Select” is performed.

The graph panes are more complicated to handle since they allow to specify focus, layout as well as update options. In the case of the “A-box Graph” pane, one can determine which individuals and which edges are displayed. Thus, for both individuals and roles, the focus can be set to the current objects, to the selected objects, or to all objects. Selected objects are highlighted. Selected items will appear on the top of the list if the “Sel. First” checkbox is enabled (e.g., selected concepts are listed before the other concepts in the “Concepts” tab). Sometimes, one wants to focus only on the selected objects. This is the purpose of the “Sel. Concepts”, “Sel. Roles” or “Sel. Inds.” checkbox.

Additional buttons allow to specify whether the graph display shall be updated automatically if the focus or layout options changes, or whether the user determines when an update is performed. In the latter case, the user first uses the button “Request Graph” to acquire the information from RacerPro (phase 1). Once the graph is available, the “Display Graph” button becomes enabled; if pushed, the graph layout is computed and displayed (phase 2). Both phases can be canceled (and different focus and layout options selected subsequently) if they should take too long³.

Some of the push buttons send commands to a connected RacerPro server (e.g., the “Concept Query” button). These buttons are called command buttons. The titles of these command but-

³Although RacerPorter does not block in phase 1, unfortunately we have to block the GUI in phase 2 due to a restriction of the GUI framework we are currently using.

tons are displayed somewhat highlighted (italic font on Windows and Linux and bold font on the Macintosh).

Some of the RacerPro requests invoked by command buttons may take some time to compute. These requests are executed in the same way as if they were invoked from the shell. So, commands which take more than 3 seconds to finish also produce “:busy-backgrounding” messages.

In the profile editor, you can specify whether the commands produced by commands buttons will be put into the shell command history or not.

Please note that many command buttons automatically select certain objects (i.e., concepts, individuals, roles). For example, the “Concept Query” button returns a (possibly empty) set of individuals. The returned individuals become selected automatically.

6. The info display area. This area is similar to the shell; however, it only “echos” the shell interaction or all user-invoked commands, thus it accepts no input (e.g., the result produced by pushing the “Concept Query” button in the “Concepts” tab on a selected concept is shown there, as well as in the shell, so it can be reused). This helps to avoid opaqueness, and as a side effect, the user learns the correct KRSS syntax.

Please note that all this displayed information is session and thus profile specific (depends on the active profile and thus changes if another profile / session is activated).

2.3.2 Preferences

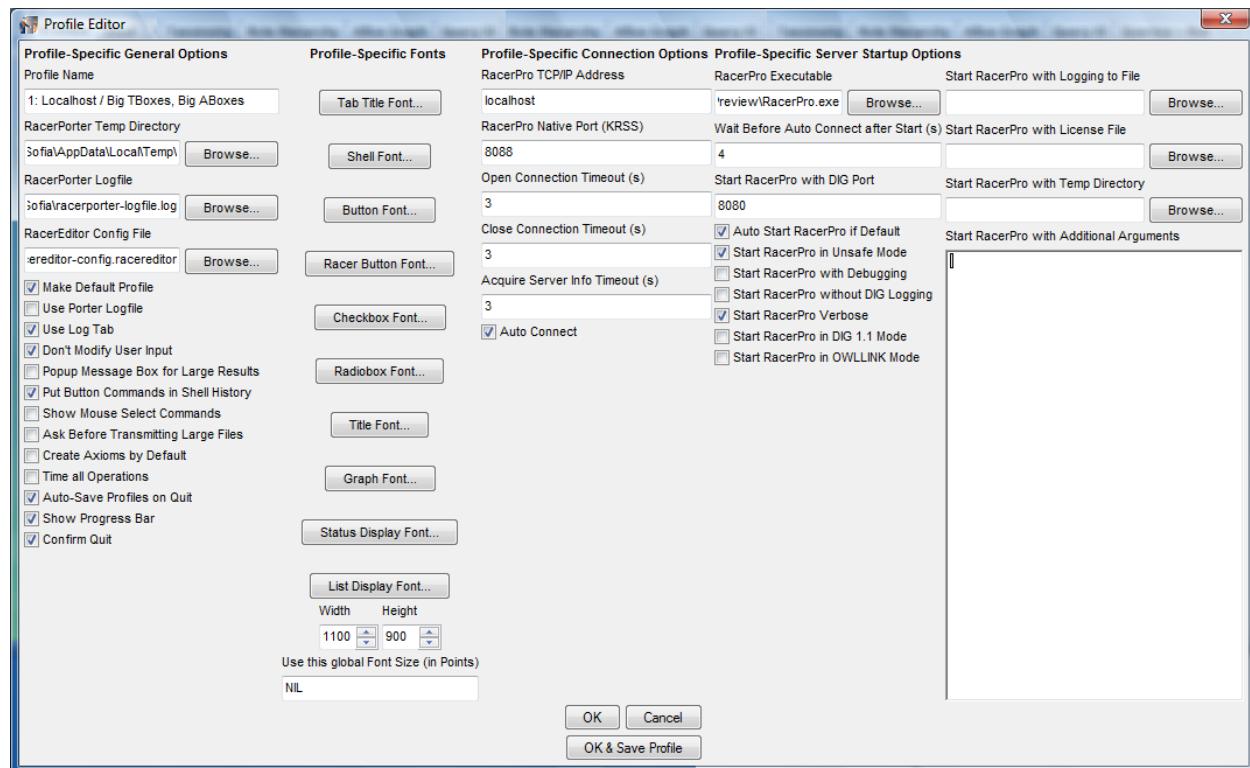


Figure 2.28: The editor of preferences for RacerPro with the use of RacerPorter.

In order to specify preferences simply press the “Edit Profile...” button in the “Profiles” pane (Mac: choose “Preferences...” item from the menu), specify the RacerPro host and port to connect to, press “OK” and then “Connect” to that RacerPro server. See also Figure 2.28. Note that you can manage different server settings with profiles. The file “profiles.racerporter” 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 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 T-box from the T-boxes list panel, updates the T-box 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 A-box in the state display, the “Taxonomy” tab will always show the taxonomy of T-box in the state display etc.

Note that selecting an A-box (T-box) from the A-boxes (T-boxes) pane does NOT change the (*current-abox*) (resp. (*current-tbox*)) of RacerPro. Use the buttons “Set Server T-box” and “Set Server A-box” 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 T-boxes resp. A-boxes tab by marking the T-boxes resp. A-boxes name like this: >>> DEFAULT <<<.

2.3.3 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 press the button “Open in Editor...” in the “Shell” or in the “Profiles” 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. The editor uses standard Emacs-compatible key bindings (e.g., Ctrl-x Ctrl-w saves a file, Ctrl-x Ctrl-f loads a file, etc.). The “GNU Emacs Reference Card” is a good reference to get to know the usual key bindings.

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

The editor has three modes:

1. Racer mode. This mode is useful for editing RacerPro KBs in RacerPro native syntax. You can incrementally evaluate expressions using Ctrl-Shift-e, or send a whole buffer to a RacerPro server using Ctrl-Shift-h (see also the “Buffer” Menu). Note that Ctrl-Shift-e also puts the evaluated command into the shell and in the shell command history, unlike Ctrl-Shift-h.
2. OWL RDF (XML) mode. In this mode Ctrl-Shift-y can be used to evaluate an OWL buffer. The OWL buffer evaluation will only work if the connected RacerPro server is running in unsafe mode (consult the “Profiles” tab for this information) since buffer evaluation requires creation of a temporary file on the machine on which RacerPro is running.
3. OWL Function mode. In this mode Ctrl-Shift-f can be used.

If you want to evaluate SPARQL queries in native syntax, then it is important to note that SPARQL queries are enclosed between blank lines (one before and one after the query), since otherwise start and end of the SPARQL query cannot be detected. To evaluate a single SPARQL query, put the cursor on the query and press Ctrl-Shift-s.

In case you are working with OWL, you often want to work with an axiomatic, auxiliary representation of your OWL KB. RacerPro can create so-called OWL axiom objects from the content of an OWL file. The created axioms can then be inspected and edited conveniently with the axioms editor provided in the “Axioms” tab.

OWL axiom objects can also be created automatically if an OWL buffer is evaluated (Ctrl-Shift-y, Ctrl-Shift-f) from the editor. It is determined by the active profile whether an OWL buffer evaluation request creates auxiliary OWL axioms (see checkbox “Create Axioms by Default”). Note that reasoning will also work without these auxiliary OWL axioms. However, the presence of the OWL axioms is required if the OWL KB (or OWL file) shall be edited in the “Axioms” tab on an axiomatic level. Please note that axiom objects may require a lot of memory if the OWL KB is big (and thus contains a lot of axioms).

2.3.4 Tabs in RacerPorter

The Profiles tab. Communication with (possibly different) RacerPro servers is managed using so-called profiles. A profile has a name and specifies at least the parameters of the connection (TCP/IP address and port) to a RacerPro server (see “Profile Specific Connection Options” in the “Edit Profile...” dialog) so that RacerPorter knows how to connect to that server.

Profiles are managed with the “Profiles” tab. Simply select a profile from the list of profiles in the “Profiles” tab to activate it. There is always one active profile at a time. The name of the active profile is always shown in the status display (field “Active Profile”).

There is one default profile. The default profile is the one which is automatically activated during startup of RacerPorter. If auto connect is enabled for this profile, the connection to the specified RacerPro server is automatically established.

Moreover, a profile can also specify parameters required for starting up a RacerPro server on the local machine (see “Profile Specific Server Startup Options” in the “Edit Profile...” dialog). For the default profile, it is possible to startup a fresh RacerPro server during startup of RacerPorter, with parameters specified in the default profile. For non-default profiles, the “Start RacerPro” button can be used to startup a fresh RacerPro server. In any case, in order to enable RacerPorter to start RacerPro servers, you’ll have to specify the “RacerPro Executable” in the “Edit Profile...” dialog. Additional server command line arguments can be specified. Note that these additional arguments will apply to freshly started RacerPro servers only and will not affect already running RacerPro servers.

A profile contains more information than meets the eye - information which cannot be edited with the profile editor. A profile also contains the session context (e.g., the current shell content, the shell history and the navigator history), as well as the state of the RacerPorter GUI (e.g., the status of the check boxes and other GUI-gadgets controlling visualization options). This session-specific information is used to revert the state of the RacerPorter GUI to a previous state if a previously used profile is reactivated again. Note that, all though the session-specific information is not made persistent (when the profile is saved from the profiles editor), the visualization options specified in a profile are in fact made persistent. This is how the four default profiles differ - they provide different default settings for various visualization options, depending on the size of the A-boxes / T-boxes under consideration.

The “Connect” button is used to connect / disconnect to / from the RacerPro server specified in the (currently active) profile. The whole set of profiles shown in the “Profiles” tab. The profiles can be saved into a profiles file (made persistent) from the profile editor with the “OK & Save Profile” button in the “Edit Profile...” dialog. On startup, RacerPorter automatically looks for the profiles file “C:/Users/Juan/profiles.racerporter” and loads it if it exists.

Once you have established a connection to a RacerPro server, the functionality provided in the other tabs becomes available. RacerPorter can manage multiple connections to possibly different RacerPro servers at a time. For example, you may send a time consuming request to one server. While that server is busy, simply switch to a different profile to work with a different RacerPro server. Once the time consuming request has been processed, you will be notified and RacerPorter can automatically switch back to the original session (and profile). You can switch between different profiles (and thus, servers and corresponding sessions) at any time.

The “Profiles” tab offers some auxiliary functionality via command buttons. It is possible to store / restore an image of a RacerPro server. This image contains the complete state (data structures) of the server in binary form. Since the server state contains the results of inferences (which might have took a long time to be reached, e.g., a taxonomy computation can easily last an hour or more on complex OWL KBs), a server state may be very valuable and it should thus be possible to store and restore it quickly. This is the purpose of the “Store Image...” and “Restore Image...” buttons. The “Full Reset” not only sends the “(full-reset)” command to RacerPro (which then disposes all

its A-boxes, T-boxes, Queries, Rules, OWL Ontologies, etc.), but also resets the RacerPorter GUI (clears the histories and session context, etc.). However, “Full Reset” only affects the active profile and its corresponding server.

A very important button is the “Load...” button. The load function can even transmit a file to a remote RacerPro server (which does not have access to the file system on which RacerPorter is running). However, this will only work if this remote server is running in the so-called “unsafe mode”. You can see from the profiles list whether the connected server is running in unsafe mode. If the server has been started by RacerPorter, then you can specify in the profile editor whether this server should be started in unsafe mode (see checkbox “Start RacerPro in Unsafe Mode”). Load accepts files with the following extensions: *.owl, *.rdf, *.funct, *.racer, *.krss, *.lisp. For OWL files, you will be asked whether a set of auxiliary OWL axioms shall be created from the OWL file. Answer with “Yes” if you wish to edit or browser the axioms with the help of the “Axioms” tab. However, axiom objects need some additional memory.

The Shell tab. This tab offers textual, command-based interaction with RacerPro for advanced users. The syntax of the native RacerPro commands to be entered into the shell is documented in the RacerPro Reference Manual. The shell tab offers the following facilities:

- Completion. This facility provides automatic completion for commands, arguments and file names, thus it can be used to learn about the available commands and required arguments for each command. For command completion simply press the tab key and all available commands are presented in a pop-up list. If you have written a partial command name (e.g., start typing “(describe-con” into the shell and press tab) you can complete the command with the use of the tab key, in case there is more than one possible completion, a popup menu appears with the possible choices (use enter to confirm and escape to exit the menu). Argument completion can be executed by using also the tab key once a command has been written or chosen from the pop-up list. The list of acceptable arguments - the so-called function lambda list is shown in the line below the shell. For file name completion - try “(owl-read-file” and press tab key. Finally, you can get rid of these popup menus by using Esc.
- Command history. The shell maintains a command history in order to make it easy to re-execute commands (not to be mistaken with the navigation history). To use this, simply use Alt-p or Ctrl-p to get the previous command, and use Alt-n or Ctrl-n to get the next command in the command history. It is also possible to position the cursor in a previously entered command expression and simply press enter. The command is then reexecuted. Moreover, commands in the command history are numbered. To reexecute command with number <n>, simply enter <n> and press enter.

Syntax requirements

Now consider the following syntax requirements when writing on the shell tab.

The shell is tailored for KRSS commands in Lisp-syntax, therefore commands always start with “(”. When entering a complex command spanning multiple lines, you can simply press enter to start a fresh line. RacerPorter will not send the command to RacerPro until the last closing “)” parenthesis has been entered. We provide parenthesis matching, convenient multi-line input as well as pretty printing.

The shell also recognizes SPARQL queries which have a different syntax. For entering SPARQL queries spanning multiple lines, use Ctrl-Enter. When the SPARQL query is complete, press Enter. The “~” (needed for the specification of pathnames on Linux or Mac) is on Alt-z (Apple-z).

The current RacerPorter T-box, A-box, concept, etc., as indicated in the status display, are available as bindings of special variables: use *t*, *a*, *c*, etc., as documented in the status display. For

example, use “(describe-tbox *t*)” to supply the current RacerPorter T-box as an argument to the “describe-tbox” function.

Furthermore, the results of the last three RacerPro commands executed via the shell are available as bindings of the three special variables *, ** and ***. For example, if you have executed “(current-tbox)” and RacerPro has returned “default”, then * is bound to “default”. Thus, you can use “(describe-tbox *)” instead of “(describe-tbox default)”.

If you are working with OWL, then concepts, roles, individuals will have so-called namespace prefixes, e.g. “<http://www.owl-ontologies.com/unnamed.owl#>” is the namespace prefix of the concept “<http://www.owl-ontologies.com/unnamed.owl#cat>”.

It is often awkward of having to type these rather long names. An abbreviated syntax is thus offered. The same abbreviation syntax which is used for display if the “Simplify” checkbox is enabled is also accepted for input, e.g., “#!:cat” refers to the default namespace. The default namespace is shown in the status display. You can define additional prefixes with the help of the shell and the “define-prefix” command (e.g., “(define-prefix “racer” “<http://www.racer-systems.com>”)”). There is also the #!: prefix which is used for nRQL (mirror) data substrate queries (see Section 4 for more details). Thus, “#!:cat” expands into “*<http://www.owl-ontologies.com/unnamed.owl#cat>” (please note the * at the beginning).

As explained, certain special variables are bound to the current objects, e.g., *t* is bound to the current T-box. The reverse direction of data flow regarding the current objects is also supported, since it is possible to set the current or selected objects to a result or result set returned by some shell command. This is the purpose of the “Selected ...” buttons, i.e., “Sel. Concepts := Last Result”, “Sel. Roles := Last Result”, and “Sel. Individuals := Last Result”. For example, one possibility to select the concept instances of the concept “C” is to enter “(concept-instances C)” and push the “Sel. Individuals := Last Result” button. The returned individuals will become the selected individuals. The selection can be cleared using the appropriate “Clear Sel.” button.

In some cases, the result of a RacerPro command can be very big (e.g., the result of “(all-individuals)” with a KB containing thousands of individuals). In such cases there is the option to put an abbreviated result into the shell instead of the whole result (a popup confirmmer will ask you).

RacerPro may need some time to execute a command. Whenever a command is send, RacerPorter waits 3 seconds for its result from RacerPro. If the result is still not available after 3 seconds, the execution of the command is performed as a background request (in a background task). Thus, the shell returns with a notification “:busy-backgrounding”, and the RacerPorter GUI becomes available again. Thus, it is not blocking further user requests and “appears alive”. However, it will accept no more commands for this session (RacerPro server), and thus, RacerPorter disables its widgets (they will be reactivated as soon as the request has been satisfied and the answer is available). However, RacerPorter is still alive and you can indeed continue working - you can either connect to another RacerPro server using a different profile, continue editing with RacerEditor, or switch to the Log tab to see what’s going on, etc. If you haven’t changed the active profile, RacerPorter automatically switches back to the original tab when the result is available. If you have continued working with a different profile, RacerPorter notifies you that the background request issued by the other sessions has now returned a result and asks whether it should switch back to the original session (profile) and tab to inspect the result.

The shell offers automatic pretty printing and reindentation of user input. However, if the “Don’t modify user input” is enabled in the active profile (see profile editor), then user input is not pretty printed, since users might feel as being patronized. By default, pretty printing of user input is disabled.

The current communication with RacerPro is also visualized in the request and response status fields; appropriate colors are used to visualize the different stages of such a communication (first

the request is send, then RacerPro is busy, then the result is received over the socket, finally the result is parsed, etc.; note that errors can occur at any time in such a processing chain).

The T-boxes tab. This tab lists the available T-boxes on the RacerPro server (specified by the active profile) and provides some general information about each T-box: the name of the T-box, whether the T-box is classified, the number of role and concept names present in the T-box, whether the T-box has cyclic definitions, whether it has a so-called “Meta Constraint” (indicating that reasoning with the T-box may be hard), the employed description logic, the number of roles, and the names of the A-boxes which reference this T-box.

The current T-box is highlighted in the list (returned by “(current-tbox)”) and is shown between “>>> ... <<<” markers. In general, the content of the T-box-specific tabs (i.e., the concepts tab, the roles tab, the taxonomy tab, and the role hierarchy tab) depend on the RacerPorter T-box, and NOT on the current RacerPro T-box. Thus, it is NOT necessary to change the current T-box (and thus the state) of the RacerPro server in order to inspect another T-box. Just select (click on) another T-box in the “T-boxes” tab. Use the main (left) mouse button to select the current RacerPorter T-box.

T-box-specific commands invoked by RacerPorter (e.g., via command buttons such as “Describe”) are always applied to the current RacerPorter T-box, not to the current RacerPro T-box. If you really want to change the current RacerPro T-box, use the “Set Server T-box” button. But in general, this will not be necessary if you work with RacerPorter since the current RacerPorter T-box (as shown in the status display) will always be supplied as an argument to all T-box-specific commands invoked by RacerPorter.

The A-boxes tab. This tab lists the available A-boxes on the connected RacerPro server and provides some general information about each A-box, such as the name of the A-box, whether it is consistent and/or realized, which associated T-box it has, the number of individuals, the utilized description logic, the number of concept and role assertions, etc.

The “A-boxes” tab works analogous to the “T-boxes” tab. Note that RacerPro also has a current A-box (result of “(current-abox)”). The situation regarding the differences between the current RacerPorter A-box (shown in the status display) and the current RacerPro A-box is analogous to the “T-boxes” tab (see above).

Some A-box-specific commands can be invoked with command buttons.

The Concepts tab. This tab lists the available concept names (also called OWL classes, atomic concepts) in the current RacerPorter T-box. The concepts are sorted alphabetically. The tab can show either all, show only the defined, show only the primitive, or show only the unsatisfiable concepts in the T-box. Selected concepts are shown highlighted (the concepts in the “clipboard”). If the “Sel. First” checkbox is turned on, then the selected concepts are at the top of the list. If “Sel. Only” is on, then only the selected concepts are shown (note that this might result in an empty list if no concepts are selected).

Concepts can be selected and deselected as usual (use the main / left mouse button). Multi-selection is supported. The last selected concept always becomes the current concept - note how the “Concept” field in the status display changes when you (de)select concepts. The small number behind the current concept in the status display shows the total number of selected concepts (the cardinality of the set of selected concepts). Concepts can be selected with the help of the “Search & Select” field as well. The selection can be cleared with the “Clear Sel. Concepts” button.

Some concept-specific commands can be invoked by command buttons and either apply to the current RacerPorter concept (e.g., “Concept Query”), or to the selected concepts (e.g., “Select Parents”). Note that the “Concept Query” button allows you to retrieve the instances of the current

concept. The retrieved instances will automatically become selected individuals (see “Individuals” tab).

The Roles tab. This tab lists the roles in the current RacerPorter T-box (and their corresponding inverses). It works analogous to the “Concepts” tab. The list either shows all roles, shows only the transitive roles, shows only the functional roles, etc., depending on the selection of the provided radio buttons. Multi-selection of roles is supported. The last selected role becomes the current role, and selected roles are highlighted.

The Individuals tab. This tab lists the individuals in the current RacerPorter A-box and works analogous to the “Concepts” tab. Multi-selection is supported. The last selected individual becomes the current individual, and selected individuals are highlighted as usual.

The Assertions tab. This tab presents the A-box assertions contained in the current RacerPorter A-box. Depending on the selection of the radio buttons, different kinds of assertions are shown (concept assertions, role assertions, attribute assertions, constraint assertions, annotation concept assertions and annotation role assertion in case of OWL A-boxes).

Assertions can be selected and deselected as usual. Multi-selection is supported. However, there is no current assertion, but only a set of selected assertions. The selected assertions can be deleted with the “Delete Selected” button. However, in general this kind of editing should be performed with the “Axioms” tab (in case an OWL KB is edited).

The Axioms tab. This tab offers a convenient and powerful interactive graphical OWL2 axiom browsing, axiom inspection, and axiom editing.

OWL2 axioms for OWL KBs can be either created by hand with the “New Axiom...” button, or automatically during load of an OWL file (see “Load...” button). The RacerPro-internal OWL parser is invoked by “Load...”. The function “owl-read-file” (e.g., “(owl-read-file <file.owl> :maintain-owlapi-axioms t)”), can create OWL axioms automatically during the parsing of an OWL document or created during the evaluation of an OWL buffer with the RacerEditor.

An OWL axiom object is simply a record-like data structure (an object) which has an ID, a type, and various axiom type-specific attributes. With the axiom type selector (on the left side) the types of axioms to be displayed (in the list on the right side) is determined. Note that axiom types are organized in a class hierarchy. Some axiom types are abstract. Axioms of abstract axiom classes cannot be created.

Axioms are contained in an ontology container. Each ontology container contains a set of axioms. Moreover, an ontology container is contained in a so-called reasoner container. Each reasoner container corresponds to a RacerPro KB - a T-box/A-box pair (with same name). An axiom in an ontology container can either be loaded or unloaded. A loaded axiom is considered for reasoning with the T-box/A-box which corresponds to the reasoner container. There is one default reasoner container which is called the OWLAPI-KB reasoner. Please note that there is also a corresponding T-box/A-box with that name. The reasoner and ontology containers can be selected from the combo boxes showing the current reasoner and current ontology container. The content of the ontology combo box depends on the current reasoner container.

The axioms tab always shows the axioms which are contained in the current reasoner container. Single axioms can be selected and deselected. The last selected axiom becomes the current axiom. All other selected axioms are shown highlighted. Command buttons are provided for loading and unloading of axioms, etc. They all apply to the current ontology (in the current reasoner). Please note that such a reasoner container is NOT identical with a RacerPro server. Each single RacerPro

server instance can host an arbitrary number of these reasoner containers. Reasoner Containers are simply OWLAPI-conformant notions of KBs.

Attributes of axioms can be edited. In order to edit an attribute you'll have to select the appropriate row of that axiom. The selected attribute of that axiom can then be edited by pressing the “Edit Axiom...” button.

In case you want to create a new axiom, make sure to select a non-abstract axiom type from the axiom-type selector on the left. Select the axiom type from the type selector until the word “NEW AXIOM” appears behind it, and the “New Axiom...” button will get enabled. If you push it, an interactive graphical axiom editor will pop up. Basically, you'll have to supply an attribute value for each attribute of an axiom. You can either enter expressions by hand using the supplied editor fields, or select “building blocks” for interactive expression composition from the presented combo boxes. An item which is selected from such a combo box list will be inserted in the editor at the current cursor position. In case PLURAL is used for the attribute description (e.g., DESCRIPTIONS instead of DESCRIPTION), more than one valid (concept / description) expression may be entered into the corresponding editor field. Simply separate the different expression by some whitespace (e.g., space or newline).

Sometimes, only the axioms which mention a certain concept (role, individual) shall be selected. For example, suppose you are looking for all individual-specific axioms which mention a certain individual. Or suppose you are looking for all class axioms referring to a certain class. The current / selected objects (concepts, roles, individual, ...) can be exploited for this purpose. This is what the button “Select Axioms using *c*, *i*, *r*” does. E.g., suppose you are interested in the SubClass axioms which mention the concept #!:cat. First, ensure that #!:cat is a current / selected concept. This can be achieved, for example, with the help of the “Concepts” (or “Taxonomy”) tab. First press the button “Clear Sel. Concepts” (and “Clear Sel. Roles” as well). Then, select the #!:cat concept (it becomes the current concept). Switch back to the “Axioms” tab. Deselect all axiom types in the axiom type selector and select only the “SubClassAxiom” type. Next, push the button “Select Axioms using *c*, *i*, *r*”. If you turn on the “Sel. First” checkbox, then only the SubClass axioms which mention the #!:cat class are displayed.

Some other axiom-, ontology container-, and reasoner container-specific commands are provided by command buttons. Note that “Dispose Axioms” disposes (unloads and then deletes) all selected axioms.

The Taxonomy tab. This tab presents a graph display of the so-called concept subsumption hierarchy (the so-called taxonomy).

What is described in the following also applies to the other graph displays in RacerPorter, e.g., “Role Hierarchy” and the “A-box Graph” tabs.

All graph displays can be configured to either update their content automatically or manually. Automatic updates are performed if the “Auto Update” checkbox is enabled. In case the checkbox is disabled, updates must be performed by hand. For this purpose use the “Request Graph”, “Display Graph” and “Reset Graph” buttons.

The display of a graph is a two-step process. First, the information to be displayed has to be acquired from RacerPro. If the “Request Graph” button is pushed, a background job is started which acquires the graph information. This job can be canceled with the “Abort Request” button. Once the graph is available, the “Display Graph” button can be used - this is the second step. A layout has to be computed for the graph. Please note that this can even take more time than the first phase. Unfortunately, the RacerPorter GUI is not available during layout computation (this job can't be processed in the background); thus, the GUI blocks and disables its widgets. However, in the Linux and Windows version of RacerPorter, the layout computation can be canceled if it takes too long. Unfortunately, this is not the case for the Macintosh version of RacerPorter.

In general, “Auto Update” can be problematic for big graphs. In case graph computation or graph layout takes too long, you may cancel the operation, and retry with some other layout and/or focus options, e.g., by focusing only on the current or selected concepts and by specifying a max graph depth cut-off limit. This depth limit can be specified in the little combo box; default is “0”, for this reason select always a depth in order to see the graph. In some cases, also the tree display visualization option results in unacceptable performance, due to a combinatorial explosion.

A node in the taxonomy graph represents a set of synonym concept (OWL class) names, and an edge between two nodes represents a (direct) subsumption relationship between the concepts (OWL classes) represented by these nodes. Selected concepts will be shown in blue, and the current concept will be highlighted.

Either the top concept is taken as the root of the taxonomy graph, or the current concept, or each selected concept is taken as a root. This is controlled by the radio buttons labeled “All Concepts”, “Cur. Concept” and “Sel. Concepts”. The layout of the graph can be horizontal or vertical. Moreover, the graph can be drawn as a tree or a DAG (directed acyclic graph). The Top and Bottom concepts can be omitted. In some cases (if the taxonomy is very big) it is helpful to limit the maximum depth of the graph display. Use the depth-limit combo box for this. In the taxonomy, select a “?” node in order to further expand this node. Obviously, these “?” nodes only appear if max-depth is not set to “unbounded”.

As with the “Concepts” tab, nodes can be selected and deselected with the mouse. Multi-selection is supported.

If “Freeze Graph” is turned on, then selection of a node and thus, changing the current concept, does not automatically change the graph layout. Note that, if “Freeze Graph” is turned off, and “Cur. Concept” display mode is used, then changing the current concept with a click of the left mouse button sets the graph root to the current concept. This behavior is sometimes undesirable.

Some concept-specific commands can be applied with the present command buttons. Note that, if “Auto Update” and “Selected Concepts” is turned on, and “Freeze Graph” is turned off, then using the “Search & Select” field results in a very dynamic display.

Please note that the default profiles for “Big T-boxes” differ from the default profiles for “Small T-boxes” only w.r.t. the default display options, e.g., the big T-boxes profile specifies a max depth, whereas the small T-boxes profile specifies unbounded depth, etc. The same distinction applies to the “Big A-boxes” vs. “Small A-boxes” profiles (w.r.t. A-box visualization options).

The Role Hierarchy tab. This tab works analogous to the “Taxonomy” tab and should not require any further explanation. Nodes in the graph represent synonym role names, and edges direct role subsumption relationships. Please note that, different to the “Taxonomy” tab, there is no “Top” and “Bottom” role. Roles can be selected and deselected by clicking on the nodes. Multi-selection is supported and works as expected.

The A-box Graph tab. This tab is slightly more complicated than the “Role Hierarchy” tab and the “Taxonomy” tab, since it allows to focus on roles as well as on individuals. Thus, focus options are present for roles as well as for individuals.

In the A-box graph, nodes represent individuals, and edges represent told and/or inferred (logically entailed) role assertions. So, an R-labeled edge between node “i” and “j” represents a role assertion “R(i,j)” (or “(related i j R)” assertion in RacerPro syntax). Individuals can be selected and deselected by clicking on the nodes. Multi-selection is supported and works as expected. Edges cannot be selected (although it would make sense in order to select roles - this will be supported in a future version of RacerPorter).

The display of inferred role assertions and the display of role assertions for transitive roles can be enabled or disabled (checkboxes “Told Only” and “Trans. Roles”).

Selected roles can be used to determine which role assertions to display. In this mode, a role assertion is visualized if the corresponding role is selected. The set of selected individuals is exploited in the obvious way.

Each individual supplies a graph root. Since the exploited graph layout algorithm currently cannot display cyclic graphs, a tree unraveling of the graph, starting from the root individual, is used. Cycles are indicated with the help of the “:back-to” annotation. Please note that the “Search & Select” field can select individuals as well as roles by name here.

As expected, some individual-specific commands are present as command buttons (as in the “Individuals” tab).

The Query IO tab. This tab offers a tabular display for query results from which query results can be selected. Whenever a nRQL or SPARQL query is entered into the shell (or editor using Ctrl-Shift-e), its result set is shown in the list below the shell (which is duplicated here from the “Shell” tab for convenient query input) in this tab. From this result list, items in answer tuples (individuals or concept names) can be selected. Note that, in principle, a nRQL query can return individuals or concept names (in case of *tbox-retrieve*). Multi-selection of result items (individuals or concept names) is supported.

The Queries + Rules tab. This is a simple list display. Queries and rules are maintained as objects (similar to T-boxes and A-boxes) by RacerPro. As such, queries and rules have a complex life cycle and a state. This tab is used to manage queries and rules as well as their states. Queries and rules are A-box-specific. The list shows basic information about the loaded queries for the current (RacerPorter) A-box. Each query has an ID, a state, etc. Moreover, the list can be advised to display only queries / rules in certain states. It is thus possible to focus exclusively on queries / rules which have terminated, which are currently running, which are prepared and thus ready to run, etc.

The list supports only single selection. Thus, only one current query / rule can be selected (see status field “Query / Rule” and variable *qor*).

Some query- / rule-specific commands are available via buttons. Note that the commands are applied to the current query / rule, as indicated by the “Query / Rule” field in the status display.

The Defined Queries tab. A defined query is a simple macro for a complex nRQL query body. Defined queries must be acyclic. This tab simply lists all query definitions. Defined queries are T-box-specific. Some commands to maintain the definitions are provided as buttons as well.

The list supports only single selection. Thus, only one current definition can be selected (see status display field “Request”).

The Log tab. This tab keeps all communication with RacerPro servers logged. It can be inspected at any time in order to learn what the DLS is currently doing. Each request gets a unique ID and can either be synchronous (S) or asynchronous (A). Moreover, it is indicated whether a request is answered with a local cache lookup, whether a request is performed in the background, etc. This log is cleared automatically every 10000 lines. Please note that RacerPorter can also maintain a log file. This depends on the active profile. Please consult the profile editor for RacerPorter log file specific settings.

Information-flow In the following a sequence of steps are described for you to follow as a manner of exercise, such that you can get to know how the principle of information-flow allows to link different interactions between the different tabs of 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 depth of the tree as necessary. If you push “Sel. Concepts” while in the Tree display mode, then only the subconcepts of the current concept will be shown.

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

Select the “A-box Graph” tab afterwards in order to show the A-box structure focusing on the selected individual. Adapt the tree depth accordingly and push “Sel. Inds.” 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 A-box structure, simply push the “Clear Sel. Inds.” button. You can also focus on a subset of the A-box edges - simply select the roles you are interested in from the “Roles” tab and check out “Selected Roles” in the “A-box Graph” 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 + 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.

2.3.5 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`).

2.4 Other Graphical Client Interfaces

In this section we present open-source Java-based graphical client interfaces for the RacerPro Server. First, we will give a short presentation of the Protégé system with RacerPro as a backend reasoner and then the coordinated use of Protégé and RacerPorter is sketched.

2.4.1 Protégé

Protégé may be used as another graphical interface for RacerPro. Protégé is available from <http://protege.stanford.edu> and can be used to graphically construct T-boxes (or ontologies) and A-boxes. This section describes the use of RacerPro with version 4.2 of Protégé. Thus, from here on, when referring to Protégé we refer specifically to Protégé 4.2.

The first time you enter Protégé, you have to install the “RacerProTG Reasoner”, in order to use RacerPro as the reasoner for Protégé. In order to install the plugin, use the “Check for plugins...” option in the “File” menu, then select “RacerProTG Reasoner” (see Figure 2.29) and install it. The next time you start Protégé, you are allowed to choose RacerPro as a reasoner, to do this, use the “Reasoner” menu to select “RacerPro” and click on “Start Reasoner”.

The plug-in offers way to configure the connection to RacerPro, to do so, go to the “Preferences” dialog window from the “File” menu and choose the tab of “RacerPro”. As can be seen in Figure 2.30, the plug-in allows to set the preference to connect to an external RacerPro executable or (by default) to connect to the RacerPro that is integrated to Protégé through the plug-in. Usually, the

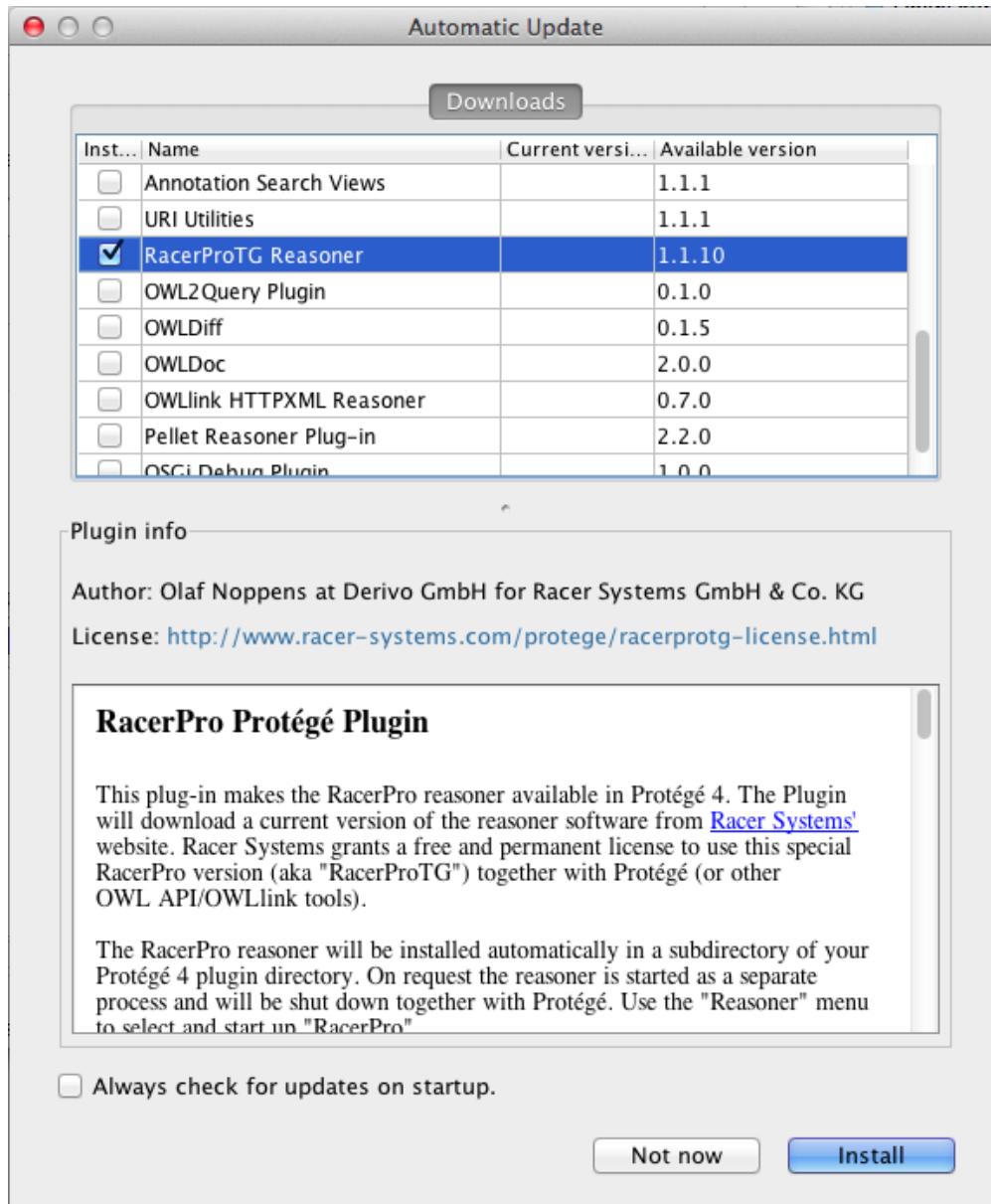


Figure 2.29: Installing RacerPro as the reasoner used by Protégé.

defaults (Local Port:8088) are ok. If you want to have a logging file provide the required path. Once making the required changes press the “OK” button.

The example presented in Figure 2.31 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 2.31 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 both obvious subclass relationships, as well as implicit class subsumption relationships (see “Class hierarchy (inferred)” in Figure 2.31). The later are provided due to the reasoning services from RacerPro. By using the reasoning services from RacerPro unsatisfiable concept names are marked (if there are any) and implicit subsumption relationships between concept names are found. Figure 2.31 shows implicit subsumption relationships, for instance, an `uncle` is also a `brother`, while from the tab

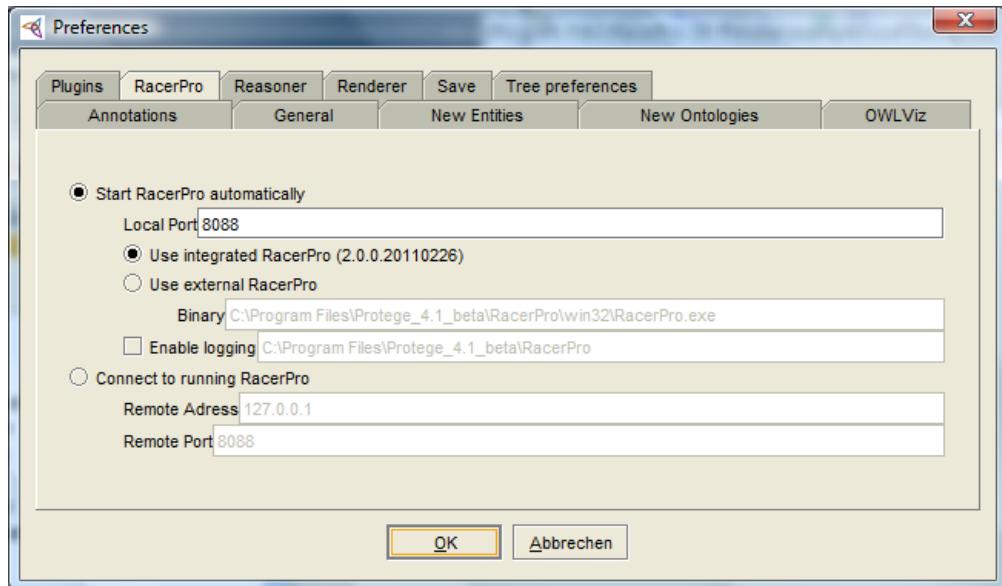


Figure 2.30: Setting the preferences to Protégé to connect to RacerPro.

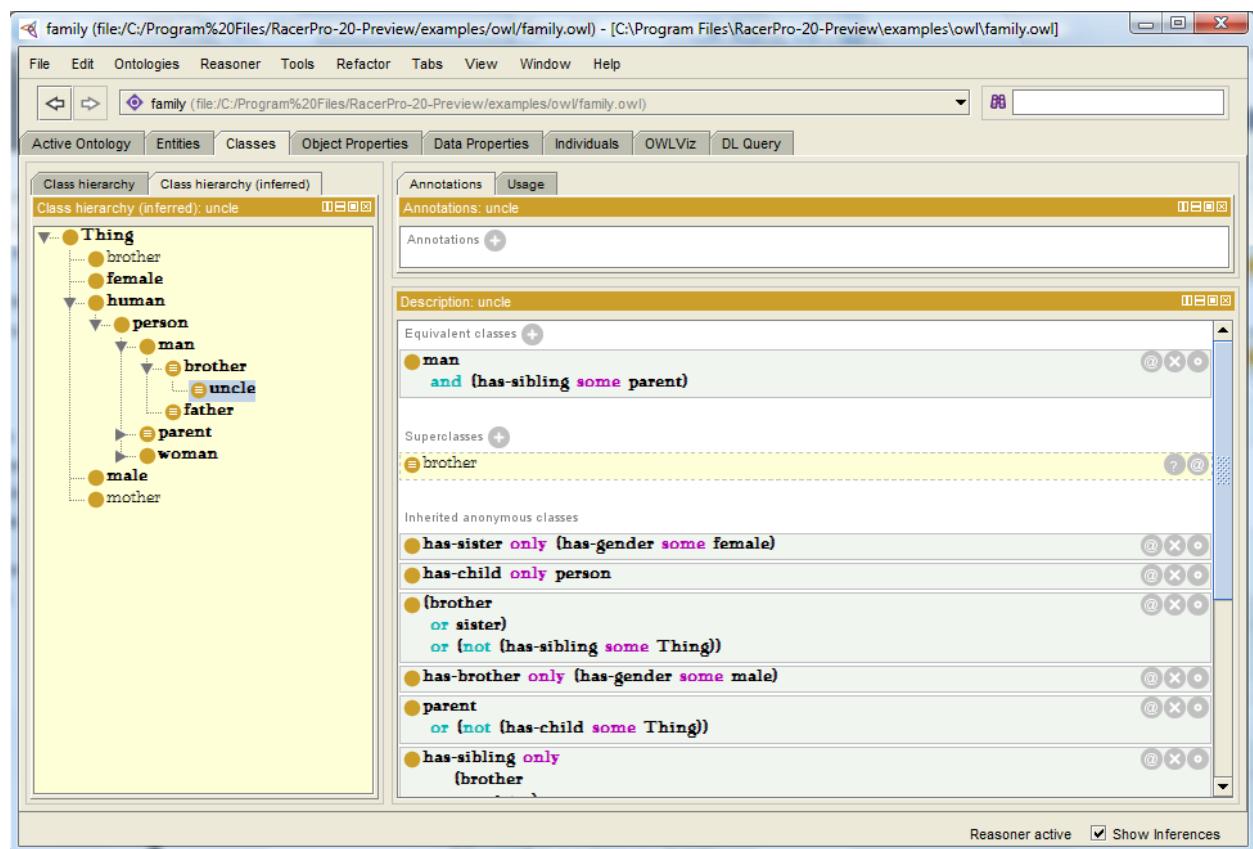


Figure 2.31: Protégé displaying the Family knowledge base. The class `uncle` is selected, and in the subwindow for inferred conditions are shown graphically.

“Class hierarchy” `uncle` is a subclass of `man`.

When selecting the different tabs in Protégé, it is possible to view the told and the inferred axioms of the knowledge base by changing to the tab with the note “(inferred)”.

2.4.2 Using Protégé and RacerPorter in Combination

Protégé uses the DIG interface to connect to RacerPro. Therefore, one can use RacerPorter to inspect details about knowledge bases that have been edited and verified with Protégé. In this section we will show that it is possible for RacerPorter to manage and execute queries on knowledge bases which have been previously edited and verified with Protégé.

Start by opening the “pizza.owl”⁴ file in Protégé, found in the RacerPro distribution in the following path: /examples/owl/. Once the “pizza.owl” is uploaded to Protégé, activate RacerPro as the reasoner to be used by Protégé. From the “Reasoner” menu select “RacerPro” and click on “Start Reasoner”. By activating RacerPro the ontology that is opened in Protégé will be automatically classified. Classifying an ontology means that RacerPro will compute the inferred class hierarchy and check the logical consistency of the ontology. As Figure 2.32 shows, during consistency checking RacerPro localized inconsistencies which are friendly highlighted by Protégé under the class “Nothing” within the pane “Class hierarchy (inferred)”.

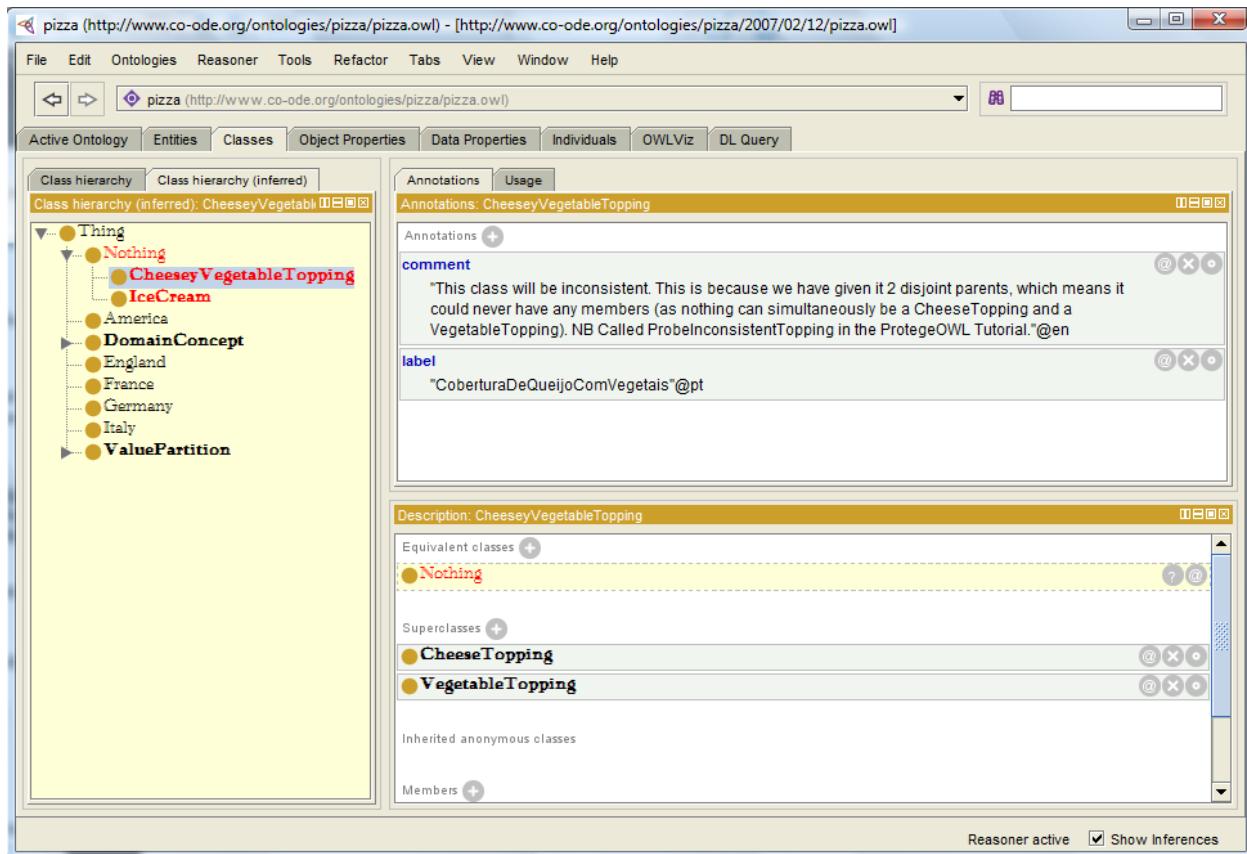


Figure 2.32: The pizza ontology loaded and classified by RacerPro within Protégé.

For demonstration purposes, the editors of the pizza ontology nicely documented the ontology to understand the inconsistencies. As the annotations of the class `CheesyVegetableTopping` describes (see Figure 2.32), the class is inconsistent due to a disjointness axiom between two of the parent classes of the `CheesyVegetableTopping` class. On the lower pane the description of the selected class is shown and the parent classes are specified, which are `CheeseTopping` and `VegetableTopping`. Next, we will the disjointness axiom causing this inconsistency.

⁴The pizza ontology was taken from: <http://www.co-ode.org/ontologies/pizza/2007/02/12/pizza.owl>

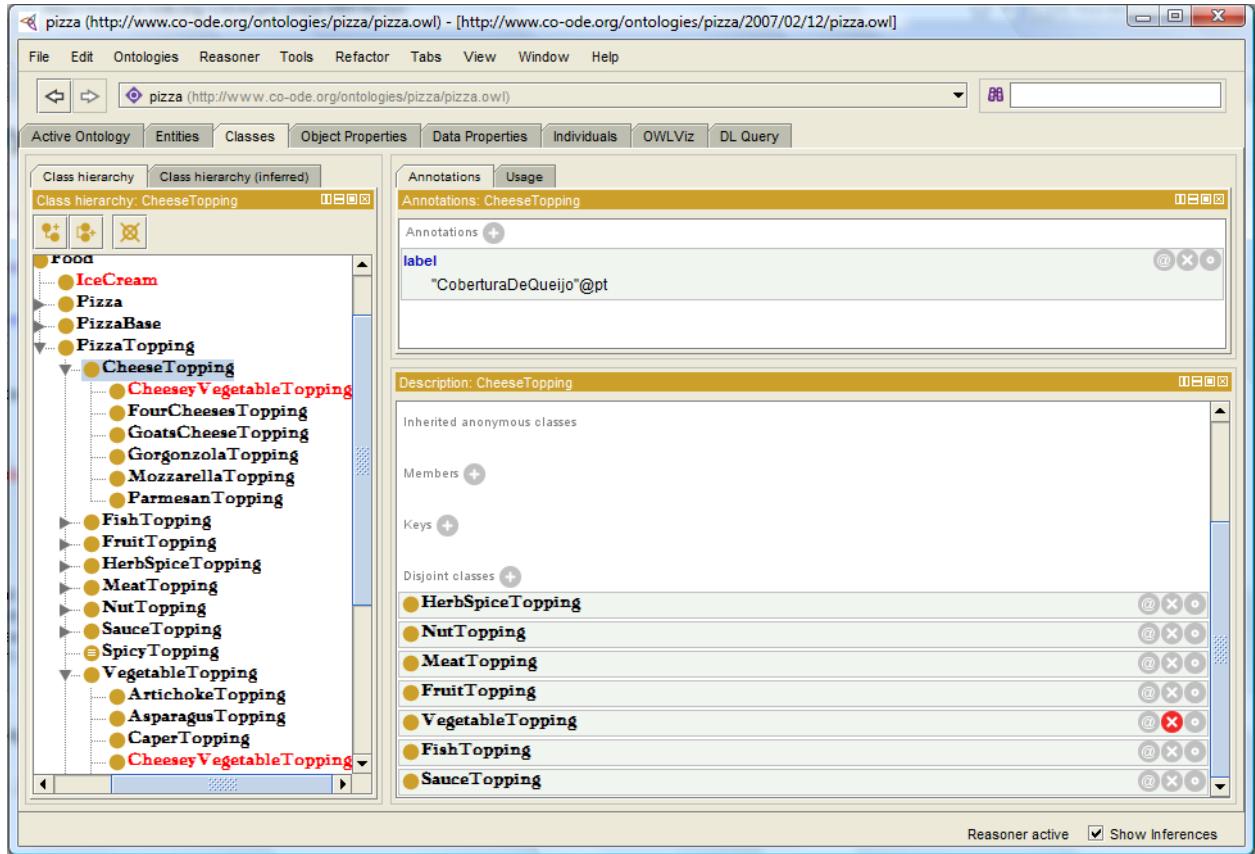


Figure 2.33: Removing the disjointness axiom that causes inconsistencies to the class `CheeseVegetableTopping`.

By selecting the class `CheeseTopping` (see Figure 2.33) view the field “Disjoint classes” in the “Description” pane, then localize the `VegetableTopping` class and click on the symbol “x” to remove the axiom. Finally, click on the option “Synchronize Reasoner” of the “Reasoner” menu. A new inferred class hierarchy is computed and only the class `IceCream` is inconsistent. To remove the inconsistency from the class `IceCream`. Remove the domain restriction to the property `hasTopping`. Given that `hasTopping` has an inverse functional role called `isToppingOf`, the restrictions set on its domain and range have to be deleted as well. Finally, synchronize the reasoner and all the inconsistencies should have been removed from the ontology. Save the ontology and close Protégé.

Now we will open the “pizza.owl” ontology with RacerPorter. In Figure 2.34 a graph of the taxonomy of the pizza ontology is presented.

With RacerPorter, for instance, queries can be answered. Figure 2.35 shows the result of the following queries executed in `pizza.owl`: `(all-individuals)` and `(retrieve (?x) (?x #! :Country))`. The symbol `#! :` is used to avoid writing the namespace. The field “Namespace” shown on the top right part of Figure 2.35 provides two options to simplify the namespace, e.g., `#! :` and `*n*`.

Queries can be inspected also using the tab “Queries + Rules” in RacerPorter. To try this, click on the “Queries + Rules” tab and reexecute the query by selecting the `:query-1` and click on the “Re-Execute” button. In this way, using RacerPorter, queries can be inspected as objects.

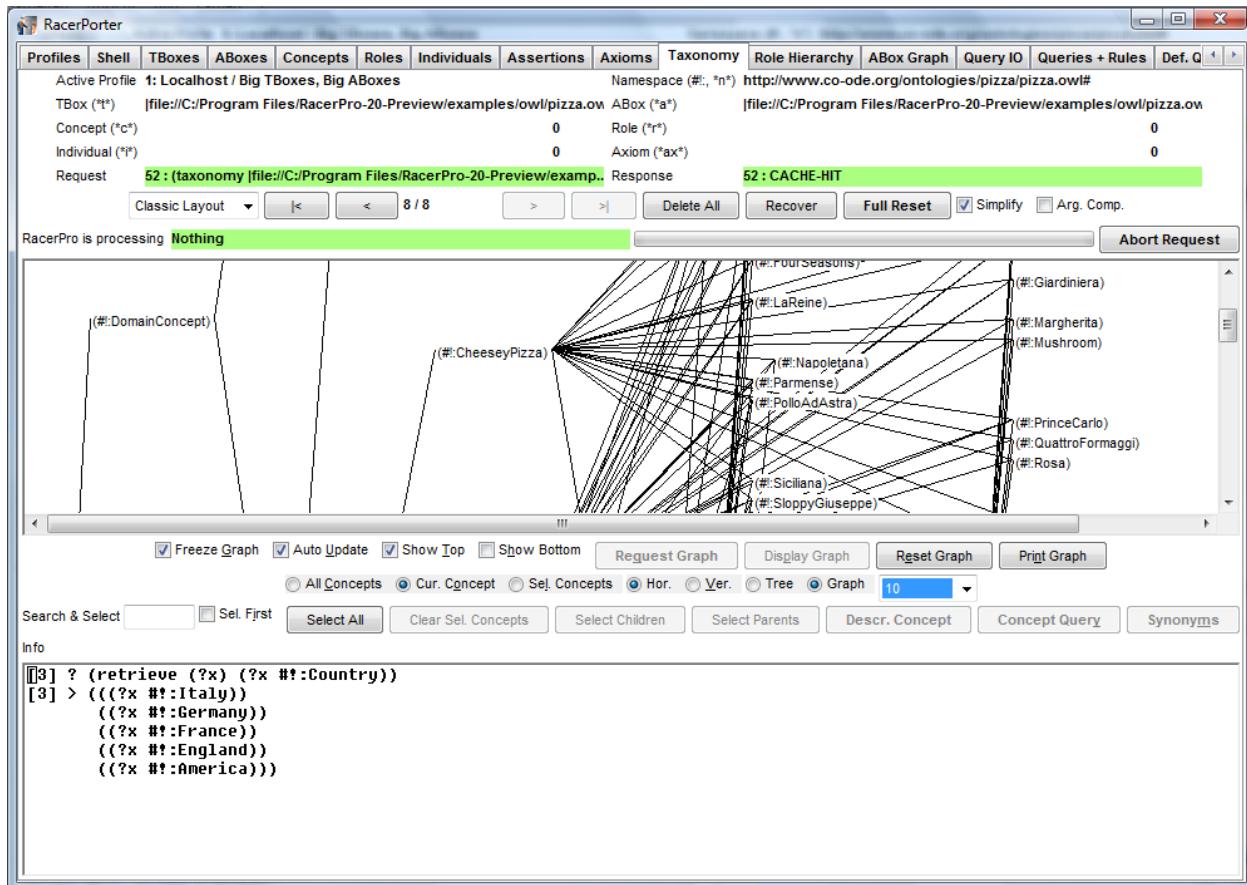


Figure 2.34: RacerPorter used to inspect an ontology edited with Protégé.

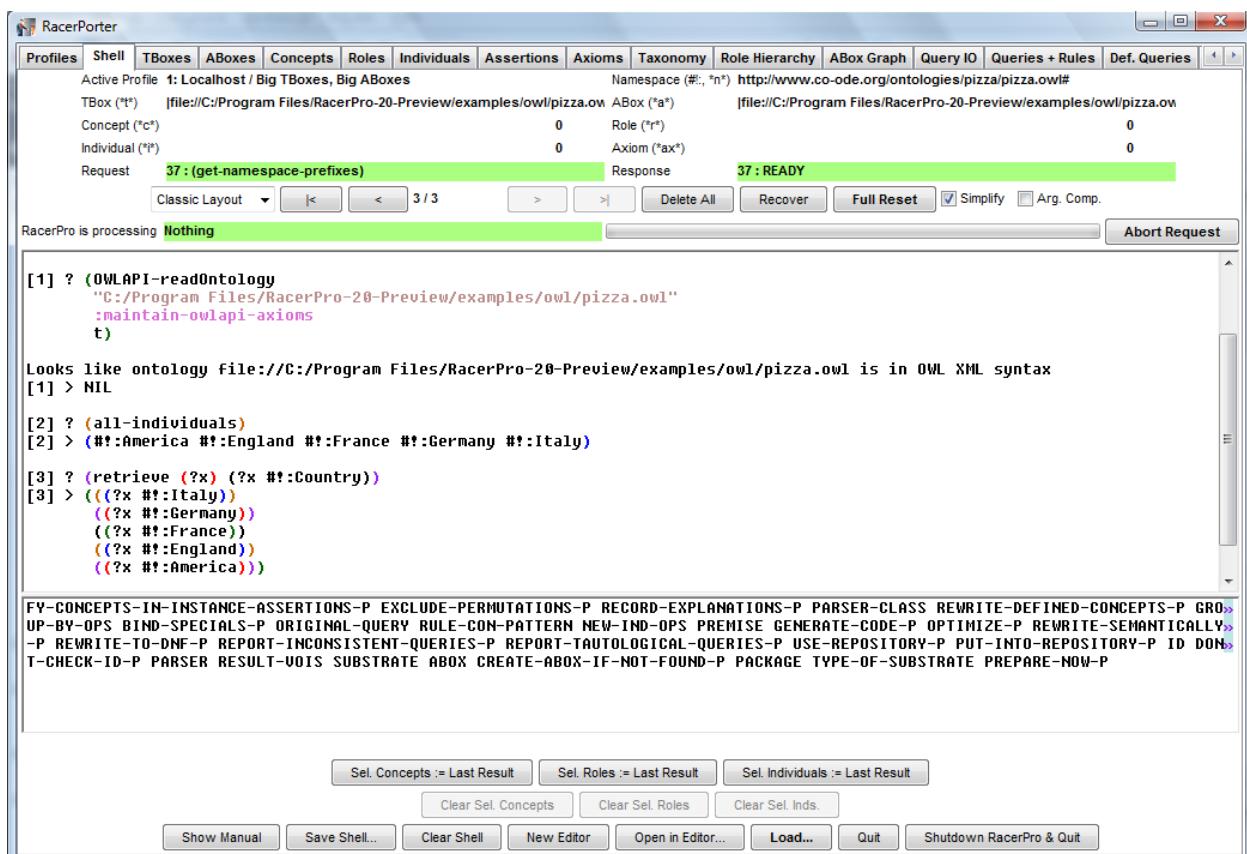


Figure 2.35: RacerPorter used to query an ontology edited with Protégé.

Chapter 3

Modeling Languages and Basic Reasoning Services

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 \mathcal{SHIQ} supported by RacerPro see [43]. The extension of expressive description logics with concrete domains is discussed in [33]. In this section, the constructs that RacerPro supports for building knowledge bases are described, as well as the inference modes that are supported. But before, the naming conventions used in this document are provided.

3.1 Naming Conventions

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

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

The API is designed to the following conventions.¹ For most of the services offered by Racer-Pro, 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. Furthermore, if an argument $T - box$ or $A - box$ 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.

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

Figure 3.1: RacerPro concept and role terms.

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.

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 $R C$)

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!

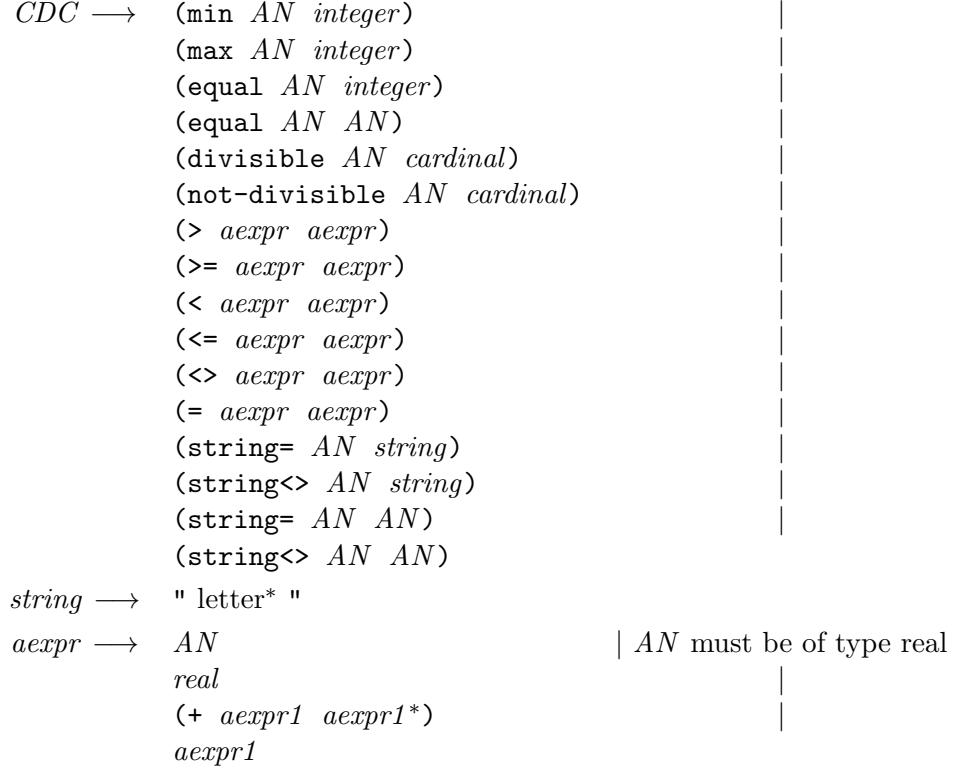


Figure 3.2: RacerPro concrete domain concepts and attribute expressions.

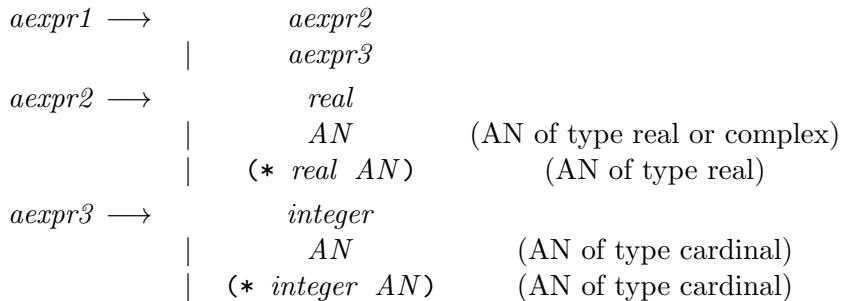


Figure 3.3: Specific expressions for predicates.

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 [43].

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

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.

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 ($>$, \geq , $<$, \leq , $=$, \neq), 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 ($=$, \neq), but no order constraints ($>$, \geq , $<$, \leq) 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 \neq** . The use of concepts with concrete domain expressions is illustrated with examples in Section 3.5. For the declaration of types for attributes, see also Section 3.5.

	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 $z \in \mathbb{Z}$	$\exists A.\min_z$ $\exists A.\max_z$ $\exists A.=_z$	(min <i>A</i> <i>z</i>) (max <i>A</i> <i>z</i>) (equal <i>A</i> <i>z</i>)
Real predicate exists restriction with $P \in \{>, \geq, <, \leq, =\}$	$\exists A_1, \dots, A_n.P$	(<i>P</i> aexpr <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

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** *C₁* *C₂*)

Concept equations state the equivalence between two concept terms.

DL notation: $C_1 \doteq C_2$
RacerPro syntax: (**equivalent** *C₁* *C₂*)

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

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

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

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

RacerPro syntax: (`disjoint` $C_1 \dots C_n$)

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 Axioms

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.

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.

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.

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

Figure 3.4: Conflicting declared and inherited role properties.

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. [13]). The KRSS-like declaration of the role (`define-primitive-role RN (and RN1 RN2)`) can be approximated in RacerPro by: (`define-primitive-role RN :parents (RN1 RN2)`).

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	
(<code>define-primitive-role RN :domain C</code>)	$(\exists RN.\top) \sqsubseteq C$
(<code>define-primitive-role RN :range D</code>)	$\top \sqsubseteq (\forall RN.D)$
(<code>implies-role RN₁ RN₂</code>)	$RN_1 \sqsubseteq RN_2$
(<code>implies-role (RN₁ ... RN_n) RN_{n+1}</code>)	$RN_1 \circ \dots \circ RN_n \sqsubseteq RN_{n+1}$

Figure 3.5: Domain or range restrictions and role axioms 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. [13]) or declared via the keywords `:domain` and `:range`. For role axioms the standard acyclicity condition applies [40].

Other role axiom declaration forms are:

- (`symmetric RN`)
- (`asymmetric RN`)
- (`transitive RN`)
- (`reflexive RN`)
- (`irreflexive RN`)
- (`functional RN`)
- (`inverse RN1 RN2`)

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 $\frac{3}{4}$). 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.

Concrete domain 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 polynomials. In a similar way, currently, an attribute of type `integer` may not be used in a term for a linear polynomials, either. If the coefficients are integers, then `cardinal` (natural number, including 0) for the type of attributes may be used in a linear polynom. Furthermore, attributes of type `string` may not be used on polynomials, and non-strings may not be used in constraints for strings. Last but not least, please note that concrete domain attributes should not be confused with datatype properties as defined in OWL.

3.6 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.

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.

3.7 Specifying Rules

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 in order to make implicit information explicit.

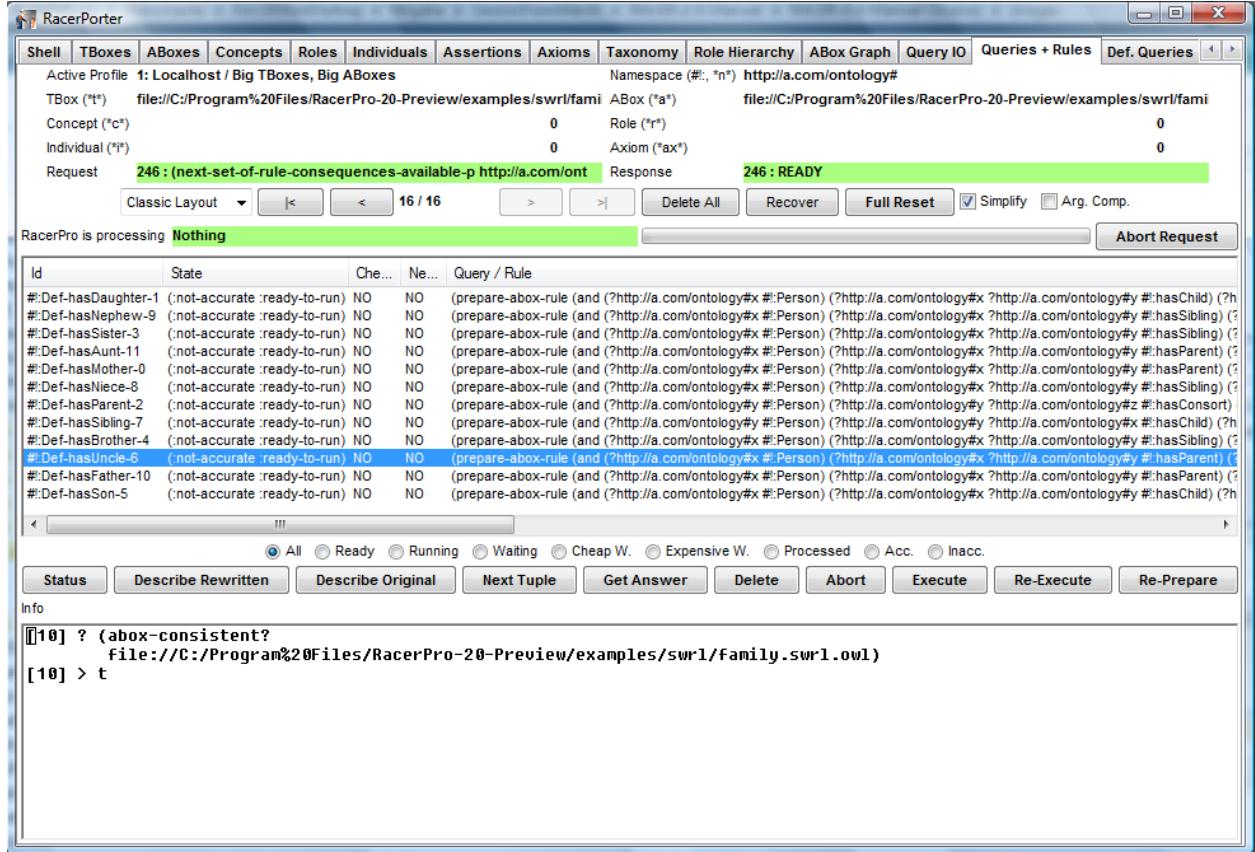


Figure 3.6: SWRL rules shown in the “Queries + Rules” tab of RacerPorter.

To show how the OWL/SWRL specification is interpreted by RacerPro we will use an example ontology containing SWRL rules. The ontology is called “family.swrl.owl”⁶, which is found in the RacerPro distribution under the path “examples/swrl/”. In Figure 3.6 the ids of the various SWRL rules are shown. We will execute the rule with Id “#Def-hasUncle-6”, which in a simplified representation looks as follows:

$$\text{Person}(\text{?}x) \wedge \text{hasParent}(\text{?}x \text{?}y) \wedge \text{hasBrother}(\text{?}y \text{?}z) \rightarrow \text{hasUncle}(\text{?}x \text{?}z)$$

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 (see the left-hand side of the arrow in the rule above) are also called the precondition and the head (see the right-hand side of the arrow in the rule above) is the consequence. Variables are bound to a finite set of individuals, namely those, explicitly mentioned in the ontology

⁶Downloaded from <http://protege.cim3.net/file/pub/ontologies/family.swrl.owl/family.swrl.owl> and edited for the purposes of this example.

(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. RacerPro applies the rule semantics. This can be seen as an advantage or as a disadvantage.

To continue with the example, now we will add the necessary individuals to the A-box part of the ontology that satisfy the precondition of the rule above. For this, we will add the individuals MARY, SUE and JOHN as instances of Person, then relate MARY with SUE under the role hasParent, and SUE with JOHN under the role hasBrother. For this we use the following commands:

```
(instance #!:MARY #!:Person)
(instance #!:SUE #!:Person)
(instance #!:JOHN #!:Person)
(related #!:MARY #!:SUE #!:hasParent)
(related #!:SUE #!:JOHN #!:hasBrother)
```

Figure 3.7 shows the resulting A-box graph. To execute the famous “has uncle” rule one can use the “Queries + Rules” tab of RacerPorter. With the “Queries + Rules” tab, rules can be inspected (see buttons “Describe Original”), executed, re-executed, etc. To execute the rule, select the id of the corresponding rule and click on the button “Execute”. As Figure 3.8 shows a new role is added between the individuals MARY and JOHN called hasUncle. This new assertion can be seen in the A-box graph of RacerPorter which allows us to analyze the effect graphically, the shell window also updates the information accordingly. Thus, the rule is responsible for asserting a new role.

RacerPro offers a forward chainer with the command (**swrl-forward-chaining**) that applies SWRL rules until no new information is added.

In RacerPro, for a fixed variable binding to a individual in an A-box, the predicates in the body must be satisfied in all models. This has important consequences as we will see in the following example.

Let us assume, we have class names French, Italian, Wine-drinker, Not-wine-drinker, and French-or-italian. For the latter two classes the following necessary and sufficient conditions are specified:

$$\begin{aligned} \text{French-or-italian} &\equiv \text{French} \sqcup \text{Italian} \\ \text{Not-wine-drinker} &\equiv \neg \text{Wine-drinker} \end{aligned}$$

The individual john is an instance of Not-wine-drinker and French-or-Italian. Thus, for john we have indefinite information. More over, if the following rules are specified:

$$\begin{aligned} \text{Rule-1: French(?x)} &\rightarrow \text{Wine-drinker(?x)} \\ \text{Rule-2: Italian(?x)} &\rightarrow \text{Wine-drinker(?x)} \end{aligned}$$

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. 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. 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.

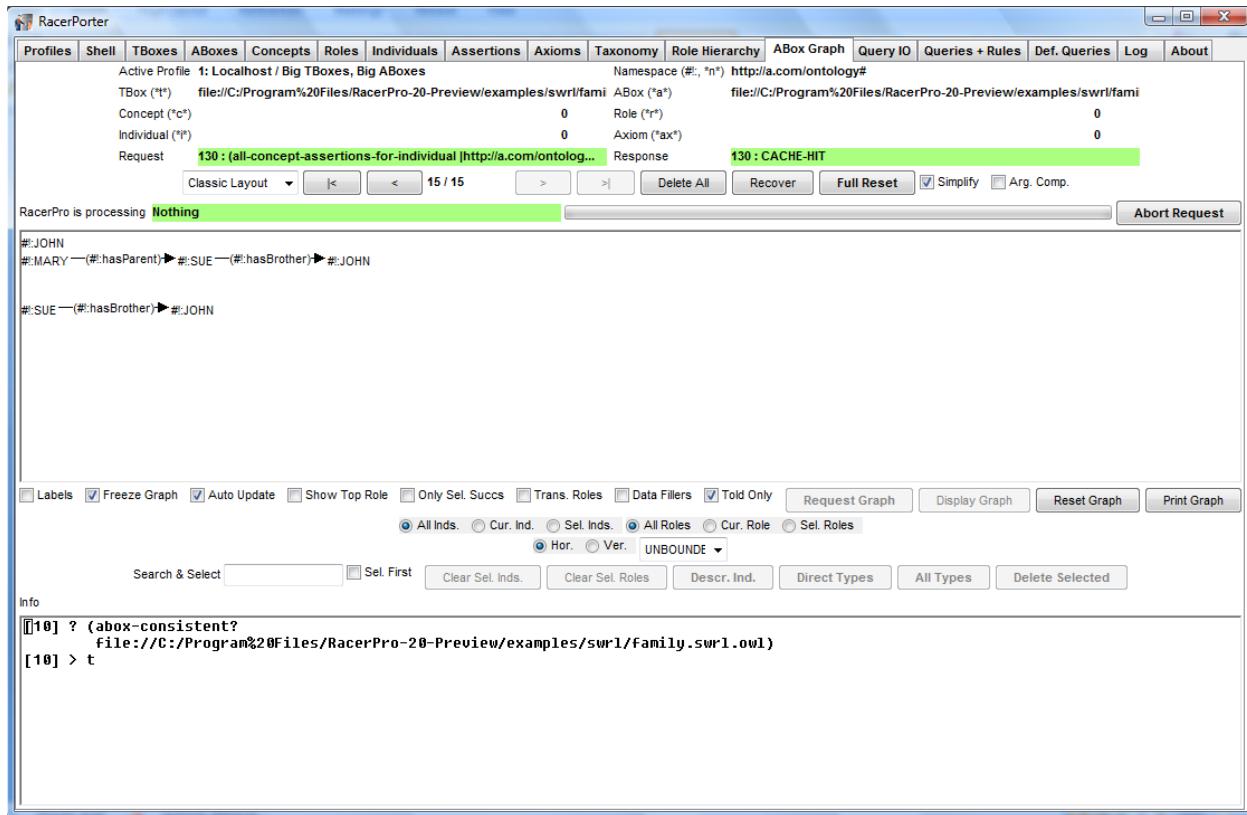


Figure 3.7: A-box Graph of RacerPorter before applying the “uncle” rule.

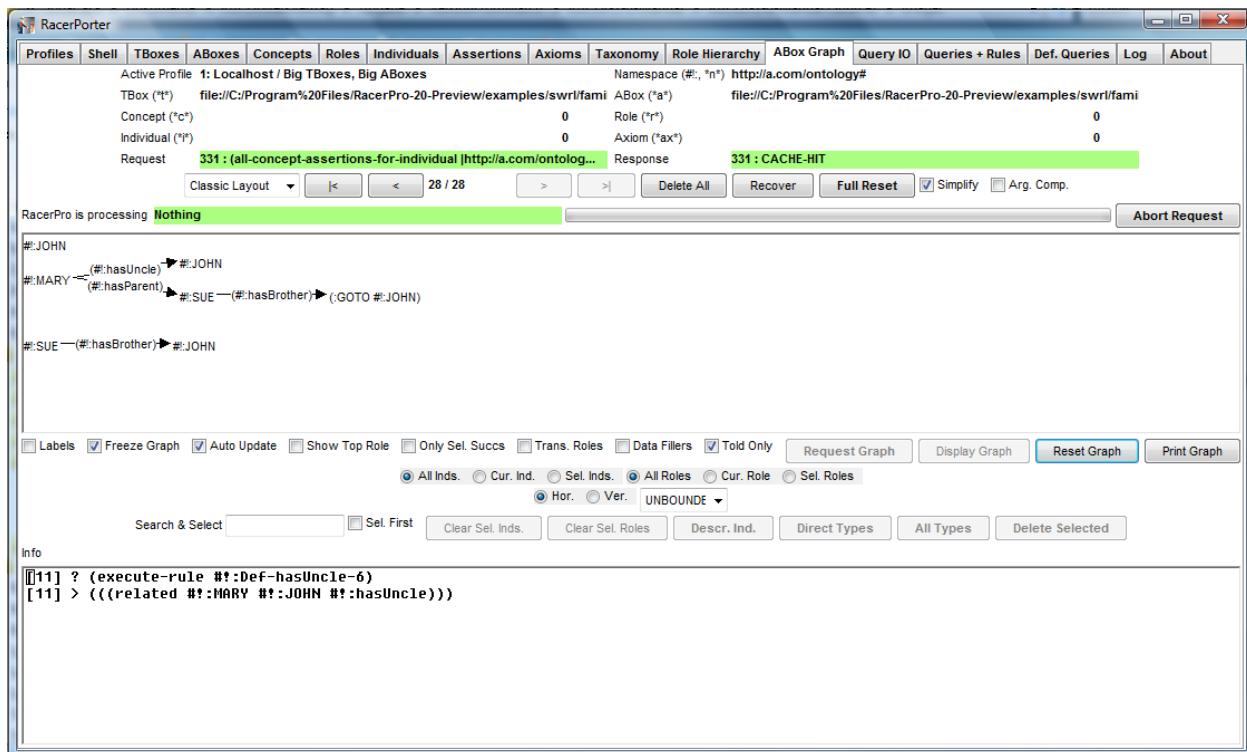


Figure 3.8: A-box Graph of RacerPorter after applying the “uncle” rule.

3.8 Hints on Modeling and Reasoning with RacerPro

There are several excellent books and introductory papers on description logics (e.g., [4, 7]). In this Chapter we discuss additional issues that are important when RacerPro is used in practical applications.

3.8.1 Representing Data with Description Logics (?)

Almost nothing is required to use a description logic inference system to store data. In particular, there is not 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 declaration for “data”, query functions can be used to retrieve data.

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

returns **i**. You could also use the nRQL query language (see Chapter 4).

```
(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 4. 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.

3.8.2 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 achieve 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 redish-object (some has-color (one-of orange, red)))
(instance car-1 (and redish-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.

3.8.3 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 close 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.

3.8.4 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 5.4). The nRQL query language allows you to query A-boxes using negation as failure semantics (see Chapter 4).

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 other 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.

3.8.5 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
```

3.9 Evaluating RacerPro and Configuring Reasoning 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.

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 bbox-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 4). 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{ALC} 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 4).

Chapter 4

The New RacerPro Query Language - nRQL

In this chapter of the user guide we will describe an expressive ABox query language for RacerPro, called *nRQL*. nRQL is an acronym for *new Racer Query Language*, pronounce: *Nercle*. 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 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 query language,
- an OWL query language.

nRQL allows for the formulation of *conjunctive queries*. In a nRQL query, *query variables* are used which will be bound to those ABox individuals that *satisfy* the query. Queries will make use of arbitrary concept and role terms. TBoxes (or OWL class definitions) supply the domain specific vocabulary to be exploited in the queries. However, nRQL offers much more than plain old conjunctive queries.

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

- Complex queries are built from query atoms.

nRQL offers concept query atoms, role query atoms, constraint query atoms, and **SAME-AS** query atoms (as well as some auxiliary query atoms). Query atoms are combined to form complex queries with the query constructors **and**, **union**, **neg** and **project-to**.

- nRQL has a well-defined syntax and clean compositional semantics.

We claim that the language is easy to understand, since users of RacerPro are already familiar with the notion of *logical entailment*.

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

Purely *existentially quantified statements* (“there exists a filler of the `has-child` role of `?x` such that …”) can be made by exploiting the expressive power of RacerPro’s concept expressions (e.g., use statements such as (`?x (some has-child ...)`)).

- *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 (e.g., we can ask for the individuals from the ABox which are *known to be woman but not known to be mothers*). nRQL is currently the only practically available OWL query language which allows for the execution of such “autoepistemic” queries.

- Special support for querying the *concrete domain* part of an ABox. nRQL allows for the specification of complex retrieval conditions on concrete domain attribute fillers of ABox individuals by means of complex *concrete domain predicates*. Moreover, in case satisfying concrete domain values are known (“told”) in the ABox, these can be retrieved. For example, we can query for the individuals which are adults, i.e., the filler of the `has-age` concrete domain attribute of these individuals of type `real` must satisfy the complex concrete domain predicate ≥ 18 .

Constraint checking on the concrete domain is also supported by nRQL, again by means of complex predicate expressions. For example, we can use nRQL to query for all pairs of individuals such that one individual is (at least) 8 years older than the other individual.

- Adequate handling of *synonym individuals* and proper treatment of `same-as` assertions.
- A projection operator `project-to` for query bodies.

nRQL is more expressive than relational algebra (or non recursive Datalog); however, the semantics of nRQL is specified in an algebraic way which is easy to understand for users which are familiar with relational algebra. Please refer to the list of research publications at www.racer-systems.com for the formal specification of the semantics of nRQL.

- Various projection operators (e.g., `types`, `direct-types`, `describe`, `individual-synonyms`), as well as a more complete `told-value-if-exists` head projection operator to retrieve told values of concrete domain attributes.
- nRQL is also a powerful OWL and RDF query language which is far more expressive than typical RDF query languages, and even more expressive in certain aspects than the semi-official OWL query language OWL QL.

nRQL allows you to exploit the expressive power of (arbitrary, anonymous) concept expressions in concept query atoms, use negated roles in role query atoms, have queries with NAF semantics as well as with classical negation in one query, etc. None of these expressive means is supported by typical RDF query languages.

nRQL allows for the specification of complex retrieval conditions on the (XML Schema) datatype fillers (values) of OWL datatype properties. Like in the case of the concrete domain,

we might be interested in retrieving those individuals from an OWL document which are adults; thus, the filler of the `has-age` *OWL Datatype Property* should satisfy the predicate ≥ 18 .

For this purpose, nRQL offers an *extended RacerPro concept syntax* which can be used in concept query atoms. The extended syntax allows for the user friendly specification of such retrieval conditions. Moreover, the known (“told”) concrete fillers of such OWL datatype (or annotation) properties can be retrieved, like in the concrete domain case.

Note that OWL itself does not support the specification of complex predicates. Thus, nRQL is currently the only OWL query language which offers this expressivity.

Also the *constraint checking* facilities of nRQL apply to the OWL case. The constraint checking facility is unique to nRQL.

- Complex TBox queries are also available. These enable you to search for certain patterns of sub/superclass relationships in a taxonomy (of a RacerPro TBox or OWL KB).
- nRQL offers now termination-safe functional lambda expressions. These lambda expressions can be used in query heads and rule antecedences. nRQL allows a restricted kind of server-sided programming and can thus be used to implement, for example
 1. user-defined output formats for query results (e.g., the query results can also be written into a file),
 2. certain kinds of combined ABox/TBox queries,
 3. efficient aggregation operators (e.g., sum, avg, like in SQL).
- nRQL also supports so-called hybrid queries, see below.

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 engine are:

- Cost-based heuristic *optimizer*.
- A facility to *define queries*.
- A simple *rule mechanism*. In the following, we only speak of queries - most described functionality also applies to rules.
- Queries (and rules) are maintained as *objects* within the engine. Thus, the engine offers life-cycle management for queries, etc.
- *Multi-processing* of queries: More than one query can be answered “simultaneously” (concurrently).
- Support for *different querying modes*:
 - “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 in this mode which means that the API is blocked (not available) until the current answer has been computed and is returned to the client.
 - “Tuple at a time mode”: In this mode, the answer to a query is computed and retrieved incrementally, *tuple by tuple*. The API then works in a *asynchronous* fashion. A client can request additional tuples of a query answer one by one. True multiprocessing of queries is available in this mode, since the API is not blocked if a query is submitted to the nRQL engine.

This mode works either *lazy*, computing the next tuple only on demand, or *eager*, pre-computing next tuple(s) in the background even though they have not been requested by the client yet. The lazy mode maximizes the availability of nRQL, whereas the eager mode has the advantage that requesting a next tuple of a certain query will eventually be faster if it has already been computed.

- Configurable *degree of completeness*: nRQL offers a *complete mode* as well as *various incomplete modes* which differ w.r.t. the amount of completeness they achieve. An incomplete mode will only deliver a subset of the answer to a query (compared with the answer which would be computed in the complete mode). However, the incomplete modes can be much more performant and will also be complete for KBs which do not require the full expressivity of $\mathcal{ALCQH}\mathcal{I}_{\mathcal{R}^+}(\mathcal{D}^-)$ or OWL. Moreover, the incomplete modes scale better than the complete mode.

The so-called *two-phase processing modes* take care that query processing works in two phases: In the *first phase*, the so-called *cheap tuples* are computed and delivered to the client. ABox reasoning is not required for the computation of these cheap tuples. Thus, RacerPro is not involved for their computation. The client (application) can now choose to request more tuples. Then, the *second phase* is used to compute the missing tuples, exploiting full RacerPro ABox reasoning. These tuples are therefore called *expensive tuples*. nRQL can be advised to deliver a so-called *warning token* which will be delivered before phase two starts. Clients can then choose to retrieve these additional expensive tuples or not.

- Configurable *runtime resources*. The number of delivered answer tuples can be bounded, a timeout is settable, and the incomplete modes are available. Also permutations of answer tuples can be excluded.
- Support for maintaining and querying so-called *hybrid representations*. A so-called “substrate representation layer” can be associated with an ABox to create a layered hybrid representation. The *data substrate* as well as the *RCC substrate* are currently available and work for OWL.

Once created, the hybrid representation can be queried with nRQL - nRQL is a *hybrid query language*. Thus, some atoms in a hybrid nRQL query address the ABox, whereas the other query atoms address the substrate representation which is associated with the ABox to be queried. Expressive means (e.g., certain predicates for constraint checking) which cannot be allowed in the ABox part of nRQL due to reasons of decidability are offered in the substrate part of a hybrid nRQL query. The substrate query atoms can be used for constraint and model checking purposes. For example, if an RCC substrate (RCC network) is associated with an ABox, then RCC constraint checking is supported through nRQL. Note that the specific characteristics of qualitative spatial RCC relationships cannot be adequately represented and enforced in the ABox; thus, specific facilities representation and querying facilities such as the RCC substrate are offered.

- *Reasoning on queries*. This functionality is currently still experimental (and incomplete). nRQL offers (incomplete) query satisfiability and query subsumption (entailment) checks.

nRQL can be advised to maintain a so-called *query repository (QBox)*, which is a DAG-structured query cache. nRQL uses the query subsumption check to maintain and compute this DAG. This can be called an semantic optimization. A further optimization technique which is also still experimental is *query realization*, see below.

We will first introduce the nRQL language in a tutorial-like style, and then describe the nRQL engine. The complete nRQL API is discussed in the nRQL Reference Manual.

In the next Section we describe the nRQL language by following a bottom up way of description. First, the available atoms are discussed, then we show how to combine the described atoms into complex queries. We then turn to more complex (and non-essential) issues, such as defined queries, nRQL rules, complex TBox queries, and the substrate representation facility. Finally, we give an EBNF syntax description of the nRQL language.

In the subsequent section we describe the nRQL engine by simply presenting and discussing the features of this engine.

4.1 The nRQL Language

We will introduce the nRQL language in a tutorial-like style. For the description of nRQL (from the ABox query language perspective) we will use the knowledge base `family.racer` as a running example. You will find this example KB in the file containing the example KBs which you can find on your personal RacerPro download page. However, for the description of nRQL from the OWL / RFD query language perspective, we will sometimes use OWL snippets and/or ad hoc OWL examples which you will not find among the example KBs (however, you can use an OWL ontology editor such as Protégé or simply type in the listed OWL examples into the RacerEditor).

In order to get the same answers for the queries as printed in this manual, it is assumed that you evaluate (`full-reset`) before you load a new KB or OWL document. You can either use the “Load” button of RacerPorter, or use RacerPro API commands such as (`owl-read-file ...`), (`racer-read-file ...`). We assume that you use RacerPorter as an interactive shell for the following example queries in the remainder of this manual.

4.1.1 Query Atoms, Objects, Individuals, and Variables

The basic expressions of the nRQL language are so called *query atoms*, or simply atoms.

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

An *object* is either an *ABox individual*, or a *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 *different kinds of variables*:

- *Injective variables* are prefixed with “`?`”, e.g. `?x`, `?y` are injective variables. An injective variable can only be bound to an ABox individual which is not already bound by another injective variable - the mapping from variables to ABox individuals is an *injective mapping*. Thus, if `?x` is bound to `betty`, then `?y` *cannot* be bound to `betty` as well.
- *Ordinary (non-injective) variables* are prefixed with “`$?`”, e.g. `$?x`, `$?y` are *(ordinary) variables*. The mapping need not be injective; an arbitrary mapping is fine. Thus, if `$?x` is bound to `betty`, then `$?y` *can* be bound to `betty` as well.

Moreover, if only *certain pairs of variables* shall be bound to different individuals, then `SAME-AS` query atoms such as (`neg (same-as $?x $?y)`) can be used, see below. This will enforce that `$?x`, `$?y` are bound to different ABox individuals.

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.

There are some auxiliary atoms (“syntactic sugar”), such as **bind-individual** atoms, etc.

We will now introduce each type of atom. Each atom can also be *negated*, in two ways: nRQL offers negation as failure (NAF) as well as classical, true negation.

Concept Query Atoms

Concept query atoms are unary atoms. A concept query atom is used to retrieve the individuals (members) of a concept, or an OWL (or RDFS) class. Please enter (**full-reset**) and load **family.racer**. Our first nRQL query uses a concept query atom (**?x woman**) to retrieve the instances of the concept **woman**. Thus,

```
(retrieve (?x) (?x woman))
```

asks RacerPro for all instances of the concept (name) **woman** from the (**current-abox**), which is **smith-family** (note that the TBox of this ABox is **family**). These instances are successively bound to the variable **?x**. Thus, the answer is:

```
((?X EVE) ((?X DORIS) ((?X ALICE) ((?X BETTY))))
```

Terminology - Query Head and Query Body The *query body* is satisfied if the variable **?x** is bound to **Eve**, to **Doris**, to **Alice**, or to **Betty**. Racer has returned a *list of binding lists*. Each binding list lists a number of variable-value-pairs. The *format* of these returned binding lists is specified by the *query head*. In this case, the head of the query is **(?x)**, and the *query body* or *query expression* is given by the single concept query atom **(?x woman)**. The set of objects mentioned in the query head must be a subset of the set of objects referenced in the query body. An empty query head is permitted (see below).

The Active Domain Semantics A so-called *active domain semantics* is employed for the variables. Variables can only be bound to *explicitly modeled ABox individuals* in the current ABox. **Please note that there is no guarantee on the order in which the possible bindings are delivered. Currently, there are no ordering functions available.**

What is the semantics of the atom **(?x woman)**? It is equivalent to **(concept-instances woman)** which returns

```
(BETTY ALICE EVE DORIS).
```

A concept query atom **(?x C)** for an arbitrary concept **C** has the same semantics as **(concept-instances C)**. Instead of a simple concept name such as **woman** we can of course also use a complex concept term for **C**, e.g. **(and human (some has-gender female))**.

Queries with individuals Suppose we just want to know if there are *any* woman at all modeled in the current ABox. We could simply query with

```
(retrieve () (?x woman))
```

and RacerPro replies: T,

which means *yes*. This query has an empty head – such a query never returns any bindings, but only a boolean answer T or NIL. T is returned if *any* satisfying binding can be found, and NIL otherwise.

We already mentioned that query atoms use *objects*. Since ABox individuals are objects, it is also possible to use an ABox individual instead of a variable. Suppose we just want to know if betty is a woman:

```
(retrieve () (bette woman))
```

RacerPro replies: T,

and consequently we get NIL for

```
(retrieve () (bette man))
```

Other nRQL Peculiarities Certain names are reserved in nRQL and thus cannot be used to identify ABox individuals having the same name - please consult <symbol> in the EBNF specification, Section 4.1.9.

If ABox individuals are listed in the query head, then they will be included in the returned binding lists as well, since the head always specifies the format of the returned binding lists:

```
(retrieve (bette) (bette woman))
```

RacerPro replies:

```
((($?BETTY BETTY)))
```

The answer (((\$?BETTY BETTY))) is returned instead of (((BETTY BETTY))) (as probably expected) since the query is internally rewritten into

```
(retrieve ($?bette) (and (same-as $?bette betty) (bette woman)))
```

See below for an explanation of the SAME-AS query atom.

Querying OWL KBs (RDF Data) with Concept Query Atoms OWL KBs (and RDF data) can be queried with concept query atoms as well. Consider the following OWL document, 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 `http://www.owl-ontologies.com/unnamed.owl#person`:

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

RacerPro replies:

```
(((?X |http://www.owl-ontologies.com/unnamed.owl#michael|)))
```

It is important to use the “pipes” (`| ... |`):

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

does not work. Moreover, the correct namespace etc. must be used. You can use RacerPorter to find out the correct syntactical names for the concepts, roles, and individuals that RacerPro has created for an OWL document.

Note that you can also query RDF documents in this way. nRQL is more powerful than other RDF query languages. For example, complex concept expressions can be used, which is not possible in other RDF query languages:

```
(retrieve (?x) (?x (or |http://www.owl-ontologies.com/unnamed.owl#person|
|http://www.owl-ontologies.com/unnamed.owl#book|)))
```

Answer:

```
(((?X |http://www.owl-ontologies.com/unnamed.owl#michael|))
((?X |http://www.owl-ontologies.com/unnamed.owl#book123|)))
```

Role Query Atoms

The second type of nRQL atoms are the role query atoms. These are binary atoms, in contrast to the previously discussed unary concept query atoms. Role query atoms are used to retrieve pairs of role fillers from an ABox, or pairs of OWL (RDF) individuals which stand in a certain OWL object property relationship to one another,

Suppose we are looking for all explicitly modeled “mother child” pairs in the `smith-family` ABox. The role `has-child` is declared in the associated TBox `family`. We can therefore make the following query:

```
(retrieve (?mother ?child) (?mother ?child has-child))
```

RacerPro replies:

```
((?MOTHER BETTY) (?CHILD DORIS))
((?MOTHER BETTY) (?CHILD EVE))
((?MOTHER ALICE) (?CHILD BETTY))
((?MOTHER ALICE) (?CHILD CHARLES)))
```

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

If we are just interested in the children of Betty, we could ask Racer like this:

```
(retrieve (?child-of-betty) (betty ?child-of-betty has-child))
```

and RacerPro replies:

```
((?CHILD-OF-BETTY DORIS)) ((?CHILD-OF-BETTY EVE)))
```

Role Terms in Role Query Atoms We mentioned that arbitrary concept expressions can be used in concept query atoms, not only concept names. The same applies to role query atoms – *role terms* can be used, not only role names, as the following example demonstrates:

```
(retrieve (?child-of-betty) (?child-of-betty betty (inv has-child)))
```

Again, RacerPro replies:

```
((?CHILD-OF-BETTY DORIS)) ((?CHILD-OF-BETTY EVE))).
```

Please note that the set of role terms is rather limited in RacerPro (only `inv` is available as a term constructor). However, nRQL adds one more constructor which is only available in role query atoms: *classical negated roles*, which are provided with the `not` constructor.

Classical Negated Roles in nRQL It is obvious that the instances of the concept `(not mother)` can be retrieved with `(retrieve (?x) (?x (not mother)))`: RacerPro replies `((?X CHARLES))`, since `charles` is a man, and `man` and `woman` are disjoint. This follows from the definitions of these concepts in the `family` TBox. Thus, RacerPro *can prove* that `charles` is an instance of the concept `(not mother)`. In the $\mathcal{ALCQH}\mathcal{I}_{\mathcal{R}^+}(\mathcal{D}^-)$ description logic (the DL implemented by RacerPro), *negated roles* are not offered, in contrast to negated concepts, due to reasons of decidability. However, negated roles can be used in nRQL role query atoms for querying purposes!

Analogously as for `(?x (not mother))` and `charles` where RacerPro can prove that `charles` is not an instance of `mother`, sometimes it can also be proven that a certain pair of ABox individuals *cannot* be in a certain role relationship. To demonstrate this, let us add the features (functional roles) `has-father` and `has-mother` to `family.racer`. Please modify the KBs signature as follows (*don't* forget to modify the role `has-child` as well):

```
(signature ...
  :roles ( ...
    (has-child :parent has-descendant
      :inverse has-parent
      :domain parent
      :range person)
    ...
  )
  ...
  :features ((has-father :parent has-parent :range father)
    (has-mother :parent has-parent :range mother)))
)
```

At the end of the file, add the following ABox role membership assertion to the file:

```
(related eve charles has-father)
```

Please enter (**full-reset**) and reload the modified file **family.racer** into RacerPro. Now, the query

```
(retrieve (?x ?y) (?x ?y has-father))
```

should return

```
(((?X EVE) (?Y CHARLES)))
```

Let us now use a *negated role* in a role query atom. Since *female persons cannot be fathers*, RacerPro is actually able to *prove* that certain individuals *cannot* be in a **has-father** role relationship to one another. Thus,

```
(retrieve (?x ?y) (?x ?y (NOT has-father)))
```

gives us

```
(((?X CHARLES) (?Y BETTY))
((?X CHARLES) (?Y ALICE))
((?X CHARLES) (?Y EVE))
((?X CHARLES) (?Y DORIS))
((?X BETTY) (?Y ALICE))
((?X BETTY) (?Y EVE))
((?X BETTY) (?Y DORIS))
((?X ALICE) (?Y BETTY))
((?X ALICE) (?Y EVE))
((?X ALICE) (?Y DORIS))
((?X EVE) (?Y BETTY))
((?X EVE) (?Y ALICE))
((?X EVE) (?Y DORIS))
((?X DORIS) (?Y BETTY))
((?X DORIS) (?Y ALICE))
((?X DORIS) (?Y EVE)))
```

Please note that only female persons appear as bindings for **?y**. The role expression (**not has-father**) is an example for a negated role.

However, please note that the computational complexity of this feature is rather high. Thus, the extensive use of negated roles might be unadvantageous.

Features and Attributes in Roles Query Atoms Please note that you may *only* use roles (or role terms) in role query atoms. However, also *features* classify as roles (these are functional roles). However, *concrete domain attributes* are *not* permitted. If

```
(define-concrete-domain-attribute age :type cardinal)
```

had been defined as a concrete domain attribute meant to represent the ages of the individual family members, then you *cannot* use the following query to retrieve the ages of the family members:

```
(retrieve (?person ?age) (?person ?age age))
```

RacerPro will raise an error:

```
*** NRQL ERROR: PARSER ERROR: UNRECOGNIZED EXPRESSION (?PERSON ?AGE AGE)
```

As explained, *variables in nRQL can only be bound to ABox individuals*. However, conceptually `?age` would either be an object of type `cardinal`, or a so-called *concrete domain object*. Since these are not ABox individuals it is impossible to bind a nRQL variable to such an object. Nevertheless it *is* possible to retrieve the individual ages of the family members - please consult Section 4.1.2. A special set of so-called *head projection operators* is offered for this purpose.

Inverted Roles Role query atoms can also be *inverted*:

```
(retrieve (?mother ?child) (inv (?mother ?child has-child)))
```

is equivalent to

```
(retrieve (?mother ?child) (?child ?mother (inv has-child)))
```

Of course, the answer will be the same as for

```
(retrieve (?mother ?child) (?mother ?child has-child)).
```

Thus, inverted roles are only provided for the sake of completeness.

Querying for Semantically Equal Individuals – The NRQL-EQUAL-ROLE Please consider the ABox constructed as the result of executing the following sequence of statements (we denote the prompt of RacerPorter or another RacerPro client with “>” and the answer of RacerPro in the following line):

```
> (full-reset)
:OKAY-FULL-RESET

> (define-primitive-role f :feature t)
F

> (related i j f)
> (related i k f)
```

We have constructed an ABox in which `j` and `k` denote the same individual in the domain of discourse - they are *semantically equal*, although they have different names. Since `k` and `j` are fillers of a feature `f` (and features are functional roles), `k` and `j` are in fact different names for the (semantically) same domain individual (in the interpretation domain).

nRQL offers a special so-called *equal role* which can be used to query for those pairs of individuals by means of a role query atom which are semantically equal. The special role `NRQL-EQUAL-ROLE` is reserved for use within role query atoms and has a special meaning there. The special meaning of `NRQL-EQUAL-ROLE` will only be recognized in role query atoms (not in concept expressions):

```
> (retrieve (?x ?y) (?x ?y nrql-equal-role))
((?X K) (?Y J)) ((?X J) (?Y K))

> (retrieve ($?x $?y) ($?x $?y nrql-equal-role))
((?$X K) ($?Y J)) ((?$X J) ($?Y K))
((?$X I) ($?Y I)) ((?$X J) ($?Y J)) ((?$X K) ($?Y K))
```

Please note that the second query delivers more answer tuples; namely those which are excluded from the answer of the first query due to the inventiveness requirement. We can add some more atoms to the last query to make it equivalent to the first query:

```
> (retrieve ($?x $?y) (and ($?x $?y nrql-equal-role) (neg (same-as $?x $?y))))
((?$X K) ($?Y J)) ((?$X J) ($?Y K))
```

Here we query for individuals which are semantically equal, but have different names. The `SAME-AS` atom is explained in more detail below. Basically, `SAME-AS` only checks whether *ABox individual names* are syntactically equal, whereas `NRQL-EQUAL-ROLE` check whether these ABox individual names semantically denote the same domain individual in all models of the KB.

Sometimes one also wants to know which pairs of ABox individuals must not necessarily be mapped to the same domain individual:

```
> (retrieve (?x ?y) (neg (?x ?y nrql-equal-role)))
((?X I) (?Y K)) ((?X I) (?Y J)) ((?X K) (?Y I)) ((?X J) (?Y I)))
```

Note that `((?X K) (?Y J)) ((?X J) (?Y K))` is missing, and that

```
> (retrieve ($?x $?y) (neg ($?x $?y nrql-equal-role)))
((?$X I) ($?Y K)) ((?$X I) ($?Y J)) ((?$X K) ($?Y I)) ((?$X J) ($?Y I)))
```

necessarily produces the same answer.

Currently, it is not possible to use the *negated* `NRQL-EQUAL-ROLE`: `(?x ?y (not nrql-equal-role))` will raise an error.

Querying OWL KBs (RDF Data) with Role Query Atoms In the OWL realm, so-called *object properties* are the equivalent of roles. Consider the following OWL document:

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

<person rdf:ID="margrit">
  <hasChild>
    <person rdf:ID="michael"/>
  </hasChild>
</person>
</rdf:RDF>

```

In this KB, the individual `http://www.owl-ontologies.com/unnamed.owl#michael` is the filler of the `http://www.owl-ontologies.com/unnamed.owl#hasChild` object property of the individual `http://www.owl-ontologies.com/unnamed.owl#margrit`. Object properties are mapped to roles; thus, simple role query atoms such as

```
(retrieve (?x ?y)
  (?x ?y |http://www.owl-ontologies.com/unnamed.owl#haschild|))
```

will work for object properties:

```
(((?X |http://www.owl-ontologies.com/unnamed.owl#margrit|)
  (?Y |http://www.owl-ontologies.com/unnamed.owl#michael|)))
```

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

Explicit and Implicit Role Fillers Since nRQL uses the active domain semantics, you can only retrieve those role fillers which are explicitly modeled as individuals in the ABox. Suppose that the concept `mother` is defined as follows

```
(define-concept mother (and woman (some has-child person))),
```

and that there is the following ABox:

```
(instance jenny mother)

(instance susan mother)
(related susan peter has-child)
```

Thus, both `susan` and `jenny` are known to be mothers. As such, this implies that both must have, in every model of the KB, an individual as a filler of the `has-child` role, since `mother` implies `(some has-child person)`. However, if you use

```
(retrieve (?x ?y) (and (?x woman) (?x ?y has-child)))
```

you will only get

```
(((?X SUSAN) (?Y PETER))),
```

since there are no known children of `jenny`. However, we can still *check* (if not `retrieve`) which individuals necessarily must have children in all models of the KB. The concept query atom

```
(retrieve (?x) (?x (some has-child top)))
```

yields

```
(((?X SUSAN)) ((?X JENNY))).
```

However, note that we have used a concept query atom, not a role query atom.

Constraint Query Atoms

Another important type of query atom is the *constraint query atom*. Constraint query atoms are binary atoms, like the role query atoms. They are meant to address the concrete domain part of a KB. A constraint query atom can be used to retrieve those pairs of ABox (or OWL) individuals whose fillers of certain specified concrete domain attributes (or OWL datatype properties) satisfy a specified concrete domain predicate, also called *constraint*. For example, we can retrieve those pairs of individuals that have the same age, if we assume that `age` is a concrete domain attribute.

Getting Ready for Constraint Query Atoms To demonstrate a constraint query atom, we need to make some changes to the `family.racer` KB. Please load `family.racer` into your favorite text editor and add the following concept membership assertions representing the ages of the individual family members at the end of the file:

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

Please enter `(full-reset)` and reload the modified file `family.racer` into RacerPro.

Every person should now have an age. This we can verify with the concept query atom `(?x (an age))`:

```
(retrieve (?x) (?x (an age)))
```

RacerPro replies:

```
(((?X CHARLES)) ((?X DORIS)) ((?X BETTY)) ((?X EVE)) ((?X ALICE)))
```

Fine. Now we can ask RacerPro who is at least 75 years old:

```
(retrieve (?x) (?x (>= age 75)))
```

RacerPro replies:

```
(((?X ALICE)))
```

We can now proceed with the constraint query atoms.

Our First Constraint Query Atom Concept query atoms such as `(?x (>= age 75))` are insufficient if we want, for example, find out who is older than whom, or who is older than `alice`. However, using constraint query atoms we can make the following queries:

```
(retrieve (?x ?y) (?x ?y (constraint age age >)))
```

RacerPro replies:

```
((?X CHARLES) (?Y EVE))
((?X CHARLES) (?Y DORIS))
((?X CHARLES) (?Y BETTY))
((?X ALICE) (?Y CHARLES))
((?X ALICE) (?Y EVE))
((?X ALICE) (?Y DORIS))
((?X ALICE) (?Y BETTY))
((?X DORIS) (?Y EVE))
((?X BETTY) (?Y EVE))
((?X BETTY) (?Y DORIS)))
```

Thus, `charles` is older than `eve`, `doris`, and `betty`, etc. Note that `constraint` is a keyword, `age` is a concrete domain attribute, and `>` is one of the concrete domain predicates offered by RacerPro.

As usual, individuals can be used at variable (object) positions. Let us verify that nobody is older than `alice`:

```
(retrieve (?y) (?y alice (constraint age age >)))
```

RacerPro replies: NIL.

Role and Feature Chains in Constraint Query Atoms A constraint query atom such as `(?x ?y (constraint age age >))` retrieves the set of all tuples `(?x, ?y)` such that the value (filler) of the `age` attribute of the first argument in the tuples which is bound to `?x` is greater than (`>`) the value (filler) of the `age` attribute of the second argument of the tuple which is bound to `?y`.

Instead of using two single concrete domain attribute names (like `age`, `age`), it is also possible to use *role chains of arbitrary length such that the last argument in each chain is a concrete domain attribute*.

Let us consider an example using *feature chains*. Note that a feature is just a functional role. Again we need to modify the signature of the `family.racer` KB. It should now read like this:

```
(signature ...
  :roles ( ...
    (has-child :parent has-descendant
      :inverse has-parent
      :domain parent
      :range person)
    ...
  )
  ...
  :features ((has-father :parent has-parent :range father)
    (has-mother :parent has-parent :range mother)))
```

That means, add the **has-father** and **has-mother** features, and modify the **has-child** role such that its inverse is named **has-parent**. Then, add the following ABox assertions at the end of the file:

```
(related betty alice has-mother)
(related charles alice has-mother)

(related doris betty has-mother)
(related eve betty has-mother)
(related eve charles has-father) ; for the sake of KB brevity ;-)
```

Enter **(full-reset)** and reload the file **family.racer** into RacerPro.

Now that we have the features **has-mother** and **has-father**, we can query for persons whose father is older than their mother:

```
(retrieve (?x) (?x ?x (constraint (has-father age)
                                         (has-mother age) >)))
```

RacerPro replies:

```
(((?X EVE)))
```

Note that **(has-father age)** and **(has-mother age)** are role (feature) chains ended by concrete domain attributes, as required. The chains can be of arbitrary length:

```
(retrieve (?x) (?x ?x (constraint (has-father has-mother age)
                                         (has-mother has-mother age) =)))
```

RacerPro replies:

```
(((?X EVE)))
```

The query succeeds since **betty** and **charles** are actually siblings but also have a common child **eve** (for the sake of KB brevity).

We can use “true” *role chains* instead of features:

```
(retrieve (?x ?y) (?x ?y (constraint (has-child has-child age)
                                         (has-child age) =)))
```

RacerPro replies:

```
(((?X ALICE) (?Y BETTY))
 ((?X ALICE) (?Y CHARLES))).
```

Please note that you can even use *role terms* instead of single roles in these chains! Thus, even negated and inverted roles are allowed.

Complex Predicate Expressions So far we have only used simple predicate names (like “ $>$ ”) in constraint query atoms. You can also use *complex* predicate expressions. Suppose you want to find out who is *at least 40 years older than whom*. Thus, there must be a difference of at least 40 between the ages of ?x and ?y . The nRQL query

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

retrieves

```
(((?X ALICE) (?Y EVE)) ((?X ALICE) (?Y DORIS))).
```

So what is **age-1** and **age-2** in the expression $(> \text{age-1} (+ \text{age-2} 40))$? The answer is simple: In order to differentiate the **age** attribute of the individual bound to ?x from the **age** attribute of the individual bound to ?y , nRQL has internally renamed the attributes by adding suffixes. Thus, **age-1** represents the filler of the **age** attribute of the individual reached by following the first role chain, whereas **age-2** references the filler of the **age** attribute of the individual reached by following the second role chain. If the two attributes fillers to compare are different, then the suffixes will not be added, e.g. as in

```
(retrieve (?x ?y) (?x ?y (constraint (age) (works-for-years-in-company)
                                         (< age (+ works-for-years-in-company 10))))).
```

Please consult the EBNF in Section 4.1.9 to learn more about the syntax for complex predicate expressions.

Querying OWL KBs with Constraint Query Atoms In the OWL realm, the equivalent of concrete domain attributes are called OWL *datatype properties*. Thus, it would be good if constraint query atoms could be used to query OWL documents with constraint query atoms referencing datatype properties as if they were concrete domain attributes.

In principle this is possible with nRQL! However, the price is that, at the time of this writing, additional assertions must be added to the original ABox (extensional information in an OWL document is represented as an ABox in RacerPro) which is therefore “polluted”.

Consider the following OWL KB in which we have two datatype properties **age** and **name** (namespace prefixes skipped), as well as 3 instances **a**, **b**, **c**:

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

<person rdf:ID="b">
  <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">45</age>
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">betty</name>
</person>

<person rdf:ID="a">
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">betty</name>
  <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">35</age>
</person>

<person rdf:ID="c">
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">michael</name>
  <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">35</age>
</person>

</rdf:RDF>

```

Let us assume that this OWL document is stored as a file under `~/dtp.owl`. We can then use a constraint query atoms to retrieve those individuals which have same `age`, as in the concrete domain case:

```

> (full-reset)
:OKAY-FULL-RESET

> (owl-read-file "~/dtp.owl")
Reading ~/dtp.owl...
done.

> (add-role-assertions-for-datatype-properties)
:OKAY-ADDING-ROLE-ASSERTIONS-FOR-DATATYPE-PROPERTIES

> (retrieve (?x ?y)
  (?x ?y
    (constraint |http://www.owl-ontologies.com/unnamed.owl#age|
      |http://www.owl-ontologies.com/unnamed.owl#age|
      =)))
(((?X |http://www.owl-ontologies.com/unnamed.owl#c| )
  (?Y |http://www.owl-ontologies.com/unnamed.owl#a| ))
((?X |http://www.owl-ontologies.com/unnamed.owl#a| )
  (?Y |http://www.owl-ontologies.com/unnamed.owl#c| )))

```

Again, you can also use complex predicates (as discussed above). nRQL is the only OWL query language which allows for this kind of constraint checking on OWL documents. Note (`add-role-assertions-for-datatype-properties`) is crucial here, otherwise the query will not succeed and return NIL.

SAME-AS Query Atoms

These atoms are binary. We already mentioned that queries such as

```
(retrieve (betty) (betty woman))
```

are internally rewritten into

```
(retrieve ($?betty) (and (same-as $?betty betty) ($?betty woman)))
```

A **SAME-AS** query atom can be used to enforce a binding of a variable (e.g., `$?betty`) to an ABox individual (e.g., `betty`), or to enforce that two non-injective variables are bound to the same (or to a different!) ABox individual.

Please note that **SAME-AS** works purely syntactically. If you want to know which pair of ABox individuals must always be interpreted as the same domain individual (of the interpretation domain) in the models of a KB, then you must use the **NRQL-EQUAL-ROLE** in a role query atom, see the Section on role query atoms.

The following examples require no further explanations:

```
> (full-reset)
:OKAY-FULL-RESET

> (instance i top)
> (instance j top)

> (retrieve (?x) (same-as ?x i))
(((?X I)))

> (retrieve (?x ?y) (same-as ?x ?y))
NIL

> (retrieve ($?x $?y) (same-as $?x $?y))
(((?X J) (?Y J)) ((?X I) (?Y I)))

> (retrieve ($?x $?y) (and (same-as $?x $?y) (same-as j $?y)))
(((?X J) (?Y J)))

> (retrieve () (same-as i j))
NIL

> (retrieve () (same-as i i))
T
```

In older versions of RacerPro (< 1.9) it was required that the first argument of **SAME-AS** must be a variable, and the second argument had to be an individual. This is no longer required, as the examples demonstrate.

Suppose we have one more individual `k`:

```
> (instance k top)
```

The query using three injective variables `?x`, `?y`, `?z`

```
> (retrieve (?x ?y ?z) (and (?x top) (?y top) (?z top)))
(((?X I) (?Y J) (?Z K)) ((?X I) (?Y K) (?Z J)) ((?X J) (?Y I) (?Z K))
((?X J) (?Y K) (?Z I)) ((?X K) (?Y I) (?Z J)) ((?X K) (?Y J) (?Z I)))
```

is in fact equivalent to

```
> (retrieve ($?x $?y $?z)
           (and ($?x top) ($?y top) ($?z top)
                 (neg (same-as $?x $?y))
                 (neg (same-as $?x $?z))
                 (neg (same-as $?y $?z))))
(((?X I) (?Y K) (?Z J)) ((?X I) (?Y J) (?Z K)) ((?X J) (?Y K) (?Z I))
((?X J) (?Y I) (?Z K)) ((?X K) (?Y J) (?Z I)) ((?X K) (?Y I) (?Z J)))
```

See below for an explanation of the `and`, `neg` constructors.

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.

HAS-KNOWN-SUCCESSOR Query Atoms Sometimes one just wants to retrieve individuals which have a certain explicitly modeled role successor in an ABox, but one is not interested in actually retrieving this successor. For example, suppose we want to know for which individuals we have explicitly modeled children in the ABox. The query

```
(retrieve (?x) (?x (has-known-successor has-child)))
```

gives us

```
(((?X CHARLES) ((?X BETTY) ((?X ALICE))),
```

since these are the individuals for which we have `has-child` role successors in the ABox. Alternatively we could have used

```
(retrieve (?x) (?x ?y has-child)).
```

However, there is a *subtle difference* between the two alternative formulation which will come into play if we consider NAF negated atoms, see below.

Also note that, as a consequence of the *active domain semantics for variables*, the query

```
(retrieve (?x) (?x (has-known-successor has-child)))
```

is *not* equivalent to

```
(retrieve (?x) (?x (some has-child top))).
```

Suppose we add the axiom

```
(instance doris mother)
```

to the ABox and query with

```
(retrieve (?x) (?x (some has-child top))).
```

We then get

```
(((?X DORIS)) ((?X CHARLES)) ((?X BETTY)) ((?X ALICE))),
```

but

```
(retrieve (?x) (?x (has-known-successor has-child)))
```

still only gives us

```
((?X CHARLES)) ((?X BETTY)) ((?X ALICE)),
```

since the child of Doris is not explicitly present in the ABox. However, its existence is *logically implied* due to Doris' motherhood. But there is no such *known successor*.

The `has-known-successor` is just “syntactic sugar”. The atom `(?x (has-known-successor has-child))` is rewritten into `(project-to (?x) (?x ?y has-child))`, see below for an explanation of `project-to` and `and`.

4.1.2 Query Head Projection Operators to Retrieve Told Values

Before we continue with the description of the syntax of the query expressions (how to combine the introduced query atoms to form *complex query bodies*), we want to discuss another more basic feature – the *head projection operators*. These projection operators are needed in order

- to retrieve the fillers of concrete domain attributes of ABox individuals (so-called concrete domain objects), and
- to retrieve the told values of such concrete domain attributes (so-called concrete domain values),
- to retrieve the XML Schema datatype values which are told fillers of OWL datatype and OWL annotation properties of OWL individuals.

It is not only possible to retrieve these told values, but also to specify complex retrieval conditions (using concrete domain predicates) on the told values to be retrieved.

Motivation

We have mentioned that the *query head* is simply a *list of objects*. But this is only half of the story. Moreover, special *head projection operators* are permitted in the head of a query as well.

Such an operator is denoted in a *functional style*, e.g. $\langle \text{op1} \rangle ?x$, $\langle \text{op2} \rangle (\langle \text{op1} \rangle \text{alice})$, meaning the projection operator $\langle \text{op1} \rangle$ is applied to the current binding of $?x$, the projection operator $\langle \text{op2} \rangle$ is applied to the result of applying the projection operator $\langle \text{op1} \rangle$ to `alice`, etc.

So what is the purpose of these projection operators? We already mentioned that variables can only be bound to ABox individuals, but never to concrete domain objects or even values from the concrete domain. The same applies to fillers of OWL annotation and OWL datatype properties, which are elements in the domain of a primitive XML Schema datatype, e.g. `xsd:int`.

Thus, to retrieve concrete domain objects or these told values, the head projection operators can be used. With them we can retrieve the told value of the `age` concrete domain attribute of the individual `alice`.

Some Terminology from the Concrete Domain

Let us explain some terminology first. First we want to fix the `age` (which is a concrete domain attribute of type cardinal) of `alice` in `family.racer`. This can be achieved in a number of ways:

1. The `age` of `alice` can be specified with a concept membership axiom such as `(instance alice (= age 80))` in the ABox. Note that `(= age 80)` is an ordinary concept expression.
2. Alternatively, a *concrete domain object* can be added to the ABox: `(constrained alice alice-age age)`. The object `alice-age` is a so-called *concrete domain object*, which is the *filler* of the `age` *concrete domain attribute* of `alice`. The admissible *concrete domain values* of this concrete domain object (`alice-age`) can then be *constrained* by adding `(constraints (= alice-age 80))`.

In this case, 80 is called a *told (concrete domain) value*. This told value can be retrieved with a nRQL head projection operator. If we just have `(instance alice (= age 80))` in the ABox, then this value *cannot* be retrieved.

Note that RacerPro also permits the specification of told values using `(attribute-filler alice 80 age)`. An *anonymous* concrete domain object will be created, which then plays the same role as `alice-age`. Thus, also told values specified by means of `attribute-filler` can be retrieved.

Retrieving Told Values from the Concrete Domain

So how can we now actually *retrieve* the age of `alice`? We already mentioned that the following query *does not* work:

```
(retrieve (alice ?age) (alice ?age age)),
```

since `age` is not a role, but a concrete domain attribute. To solve this retrieval problem, we can proceed as follows. First, according to our discussion above, add

```
(constrained alice alice-age age)
(constraints (= alice-age 80))
```

to the file `family.racer`. As always, enter `(full-reset)` and reload the file `family.racer` into RacerPro. Now the head projection operators come into play:

```
(retrieve (alice (age alice) (told-value (age alice)))
         (alice (an age)))
```

will give us

```
((($?ALICE ALICE)
  ((AGE $?ALICE) (ALICE-AGE))
  (:TOLD-VALUE (AGE $?ALICE)) (80))).
```

Note that RacerPro has returned a much more complex binding list: **ALICE** is the binding of the variable **\$?ALICE**. By applying the **age** attribute projection operator to the binding of **\$?ALICE** we get a list of concrete domain objects which are fillers of this concrete domain attribute, in this case **(ALICE-AGE)**. Finally, the told values of these concrete domain objects are retrieved using **(:TOLD-VALUE (AGE \$?ALICE))**. This yields a list of concrete domain values: **(80)**.

Of course, also variables can be used:

```
(retrieve (?x (age ?x) (told-value (age ?x)))
         (?x (and human (an age) (> age 30))))
```

RacerPro replies:

```
(((?X ALICE)
  ((AGE ALICE) (HUHU ALICE-AGE))
  (:TOLD-VALUE (AGE ?X)) (:NO-TOLD-VALUE 80)))

((?X BETTY)
 ((AGE ?X) :NO-KNOWN-CD-OBJECTS)
 (:TOLD-VALUE (AGE ?X)) :NO-KNOWN-CD-OBJECTS))

((?X CHARLES)
 ((AGE ?X) :NO-KNOWN-CD-OBJECTS)
 (:TOLD-VALUE (AGE ?X)) :NO-KNOWN-CD-OBJECTS)).
```

Please note that we can use ordinary concept query atoms such as **(?x (and human (an age) (> age 30)))** to specify *rather complex retrieval conditions* on the fillers. The same applies to OWL datatype properties, see below.

The Attribute Projection Operator in More Detail

The attribute projection operator **(age alice)** retrieves the concrete domain objects from the ABox which are known to be fillers of the **age** concrete domain attribute of **alice**. This operator is available for each concrete domain attribute defined in the referenced TBox.

Thus,

```
(retrieve (alice (age alice))
         (bind-individual alice))
```

returns

```
((($?ALICE ALICE) ((AGE $?ALICE) (ALICE-AGE)))).
```

In case there is *more than one concrete domain object filler* for the age attribute of `alice`, e.g. if we also add `(constrained alice huhu age)` to the KB, then we will get `((($?ALICE ALICE) ((AGE $?ALICE) (HUHU ALICE-AGE))))` for the previous query. Semantically, `huhu` and `alice-age` denote the *same* concrete domain object resp. value. If there is *no* concrete domain object which is known to be a filler of an attribute, e.g. as for

```
(retrieve (bettey (age bettey))
         (bind-individual bettey))
```

we will get

```
((($?BETTY BETTY) ((AGE $?BETTY) :NO-KNOWN-CD-OBJECTS))).
```

Thus, the token `:no-known-cd-objects` indicates that there is no concrete domain object which is a filler of this attribute.

The Told Value Projection Operator in More Detail

As already seen, the projection operator `(told-value (age alice))` can be used on concrete domain objects to retrieve their actual told values.

This is how we can actually *retrieve* the age of Alice:

```
(retrieve (alice (age alice) (told-value (age alice)))
         (bind-individual alice))
```

and get

```
((($?ALICE ALICE)
  ((AGE $?ALICE) (HUHU ALICE-AGE))
  (:TOLD-VALUE (AGE $?ALICE)) (:NO-TOLD-VALUE 80))))
```

Note that we have no told value for the concrete domain object `huhu`. RacerPro indicates this using the token `:no-told-value`.

Also note that, even though `huhu` and `alice-age` *semantically denote the same concrete domain object*, only for `alice-age` the correct told value of 80 is returned, whereas for `huhu`, the binding is `no-told-value`. This behavior is inherited from RacerPro's ABox querying function `told-value`.

Remarks on Completeness

Please note that `told-value` is somehow incomplete. Consider the concept `(and (> age 18) (< age 20))`. If `age` is a concrete domain attribute of type `cardinal` or `integer`, then `age` must be 19. Thus, 19 could somehow be regarded as a “told value” as well. However, there is no way for RacerPro or nRQL to return this value as a `told-value`, since RacerPro can only check the satisfiability of a concrete domain constraint system, but not compute its solutions (even if it could it would not be possible to return these solutions as single values in most cases).

However, `told-value` in nRQL is slightly more complete than the API function `told-value`, since the effect of equality statements is analyzed and taken into account. Consider this example:

```
(full-reset)
(define-concrete-domain-attribute age :type cardinal)

(constrained a a-age age)
(constrained b b-age age)
(constrained c c-age age)

(constraints (= a-age b-age)
             (= b-age c-age)
             (= c-age 10))
```

If we use the `told-value` API function, we will only get `b-age`, `c-age`, 10 as answers to `(told-value 'a-age)`, `(told-value 'b-age)`, `(told-value 'c-age)`, even though 10 would be a “more complete” answer in all three cases. With nRQL we can get 10 for all three told values:

```
(retrieve (?x (age ?x) (told-value (age ?x))) (?x (an age)))
```

RacerPro replies:

```
((?X C) ((AGE ?X) (C-AGE)) ((:TOLD-VALUE (AGE ?X)) (10)))
((?X B) ((AGE ?X) (B-AGE)) ((:TOLD-VALUE (AGE ?X)) (10)))
((?X A) ((AGE ?X) (A-AGE)) ((:TOLD-VALUE (AGE ?X)) (10))).
```

Retrieving Told Values Fillers of OWL Datatype Properties

As already mentioned, OWL offers so-called *datatype properties*. Let us consider an example KB:

```
<?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="test" />

  <owl:DatatypeProperty rdf:ID="p1">
    <rdfs:domain rdf:resource="#test" />
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int" />
  </owl:DatatypeProperty>

  <test rdf:ID="i">
    <p1 rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</p1>
    <p1 rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</p1>
    <p1 rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</p1>
  </test>
</rdf:RDF>
```

Let us assume this OWL document is stored as a file under `~/dtp2.owl`. Please load it into RacerPro:

```
> (FULL-RESET)
:OKAY-FULL-RESET

> (OWL-READ-FILE "~/dtp2.owl")

Reading ~/dtp2.owl...
done.
```

In this document, `http://a.com/ontology#p1` is a *datatype property* of type `int`. For the individual `i`, the three concrete integer values `1`, `2`, `3` are defined as fillers of the `http://a.com/ontology#p1` datatype property.

Let us now demonstrate how to retrieve these told values as well as how to *specify retrieval conditions* on the datatype values to be retrieved. The attribute projection operators can be used for datatype properties as follows:

```
(retrieve
  (?x
    (told-value (|http://www.owl-ontologies.com/unnamed.owl#p1| ?x))
    (?x |http://www.owl-ontologies.com/unnamed.owl#test|))
```

yields

```
(((?X |http://www.owl-ontologies.com/unnamed.owl#i|)
  (:TOLD-VALUE
    (|http://www.owl-ontologies.com/unnamed.owl#p1| ?X))
  (1 2 3)))
```

as if `http://www.owl-ontologies.com/unnamed.owl#p1` were a concrete domain attribute. Instead of `told-value`, also `fillers`, `datatype-filters`, `told-values`, or `told-filters` can be used. Please consult the EBNF specification in Section 4.1.9.

To enable the specification of complex retrieval conditions on the fillers of such datatype properties, we have extended the RacerPro concept syntax. The basic Idea is that datatype properties can be used in a similar way like concrete domain attributes in concept expressions. For example,

```
(retrieve (?x (datatype-filters
  (|http://www.owl-ontologies.com/unnamed.owl#p1| ?x)))
  (?x (at-least 3 |http://www.owl-ontologies.com/unnamed.owl#p1|
    (and (min 0) (max 5) (not (equal 4))))))
```

will be recognized as a valid query. The answer is (again)

```
(((?X |http://www.owl-ontologies.com/unnamed.owl#i|)
  (:TOLD-VALUE
    (|http://www.owl-ontologies.com/unnamed.owl#p1| ?X))
  (1 2 3)))
```

Note that the extended RacerPro concept expression

```
(at-least 3 |http://www.owl-ontologies.com/unnamed.owl#p1|
  (and (min 0) (max 5) (not (equal 4))))
```

can *only* be used in concept query atoms and thus only for retrieval purposes. It is not possible to use such concepts in other places. nRQL rewrites this concept expression into a native RacerPro concept syntax:

```
(at-least 3 |http://www.owl-ontologies.com/unnamed.owl#p1|)
  (and (min racer::racer-internal%has-integer-value 0)
    (max racer::racer-internal%has-integer-value 5)
    (not (equal racer::racer-internal%has-integer-value 4))))
```

More generally, a concept which references a role that represents an OWL datatype property is “rewritten” in such a way that the qualification used in the concept will make sense for the declared datatype property, as in the examples. Datatype properties can also be used instead of concrete domain attributes in concept expressions in concept query atoms. Thus, also expressions such as (an |http://a.com/ontology#p1|) are valid, even though |http://a.com/ontology#p1| is not a concrete domain attribute.

Retrieving Told Values of OWL Annotation Properties

In OWL, the so-called *annotation properties* are used to annotate resources with meta information (e.g., comments on authorship of an ontology, etc.). Annotation properties are not used for reasoning. Typically, the fillers of these annotation properties are plain strings - these told values can be retrieved with nRQL.

nRQL can be used to retrieve the fillers of annotation properties. OWL annotation (datatype or object) properties are handled in a similar way as told values of concrete domain objects (see above).

Consider the following OWL KB snippet in which two annotation properties `annot1` and `annot2` are declared (header information etc. excluded) as well as some instances:

```
<owl:DatatypeProperty rdf:ID="annot1">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="annot2">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
</owl:ObjectProperty>

<c rdf:ID="i">
  <annot2 rdf:resource="#j"/>
  <r rdf:resource="#j"/>
  <r rdf:resource="#k"/>
  <annot1 rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Annotation
  </annot1>
</c>
```

Thus, the three instances `i,j,k` of the class `C` are defined. The annotation datatype property `annot2` is defined for `i`, with the value ‘‘Annotation’’. Moreover, `i` and `j` are set into relation using annotation object property `annot1`.

We can now retrieve the told values of these annotation properties as follows:

```
(retrieve ((annotations
            (|http://www.owl-ontologies.com/unnamed.owl#annot1| ?x)))
          (?x (an |http://www.owl-ontologies.com/unnamed.owl#annot1|)))
```

RacerPro returns:

```
(((:TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#annot1| ?X))
  ("Annotation"))),
```

and

```
(retrieve (?x ?y)
          (?x ?y |http://www.owl-ontologies.com/unnamed.owl#annot2|))
```

returns

```
(((?X |http://www.owl-ontologies.com/unnamed.owl#i|)
  (?Y |http://www.owl-ontologies.com/unnamed.owl#j|))).
```

Again, `annotations` is just syntactic sugar for `told-value`.

4.1.3 Complex Queries

After having discussed the available query atoms and the structure of a nRQL head (which may contain head projection operators) we can continue with the specification of the valid nRQL query bodies. A nRQL body is inductively defined as being either a single query atom, or as a complex query body which is constructed by supplying component query bodies as arguments to the following query constructors, which are denoted in prefix syntax:

- **AND** is an *n*-ary constructor which is used for the formulation of conjunctive queries. The arguments of the **AND** are called *conjuncts*.
- **UNION** can be used to compute the union of the answer sets of the argument query bodies. The arguments of the **UNION** are called *disjuncts*.
- **NEG** implements a *negation as failure semantics*.
- **INV** can be used to “reverse” all role query atoms eventually present in the argument query body (this constructor is rarely needed).
- **PROJECT-TO** is the projection operator for query bodies. The first argument to this constructor is a *projection list*; this must be a list of objects. Head projection operators are not permitted here. The second argument is an ordinary query body. However, as for a nRQL query, the objects mentioned in the projection list must form a subset of the objects mentioned in this query body. This operator must not be confused with the head projection operators which have been discussed before.

Please look up the EBNF specification to learn more about the compositional syntax of nRQL (see Section 4.1.9).

The AND Constructor – Conjunctive Queries
--

Suppose we want to retrieve all mothers of male persons in the `family.racer` KB. This is a classic conjunctive query:

```
(retrieve (?x ?y) (and (?x mother) (?y man) (?x ?y has-child)))
```

RacerPro replies:

```
((?X ALICE) (?Y CHARLES))
```

About Variables in Conjunctive (AND) Queries We have already mentioned that nRQL offers different kinds of variables. If we query `family.racer` with

```
(retrieve (?x ?y) (and (?x man) (?y man)))
```

then we will get NIL. Again, the rationale is that there is only one known man in the ABox. Due to the constraint that injective variables must be mapped injectively to ABox individuals, `?x` and `?y` must be bound to *different* man in the ABox, but there is only `charles`. The alternative would be to use *non-injective variables* – if we use

```
(retrieve ($?x $?y) (and ($?x man) ($?y man)))
```

instead, then we get

```
((($?X CHARLES) ($?Y CHARLES)).
```

If ABox individuals are used within a query, then these individuals are NOT automatically excluded as bindings for other injective variables in the query. Thus, injective variables “only see other injective variables”. A query referencing an individual such as

```
(retrieve (?x charles) (and (?x man) (charles man)))
```

is rewritten into

```
(retrieve (?x $?charles) (and (?x man) (charles man)
                               (same-as $?charles charles))),
```

which means that `$?charles` is a non-injective variable (see below). We will thus get the desired result `((?X CHARLES) ($?CHARLES CHARLES))`.

A More Complex Example

nRQL queries are especially useful when searching for complex role filler graph structures in an ABox. Consider the following query, which searches for children having a common mother:

```
(retrieve (?mother ?child1 ?child2)
        (and (?child1 human)
              (?child2 human)
              (?mother ?child1 has-child)
              (?mother ?child2 has-child)))
```

RacerPro replies:

```
(((?MOTHER BETTY) (?CHILD1 DORIS) (?CHILD2 EVE))
 ((?MOTHER BETTY) (?CHILD1 EVE) (?CHILD2 DORIS))
 ((?MOTHER ALICE) (?CHILD1 BETTY) (?CHILD2 CHARLES))
 ((?MOTHER ALICE) (?CHILD1 CHARLES) (?CHILD2 BETTY)))
```

An even more complex query is required if we want to search for odd family interrelationships (it is assumed that you have added the additional ABox axioms from the previous chapters to the original `family.racer` KB, otherwise the answer will be NIL):

```
(retrieve (?x ?y ?z ?u)
         (and (?x ?y has-descendant) (?x ?z has-descendant)
              (?y ?u has-descendant) (?z ?u has-descendant)))
```

RacerPro replies:

```
(((?X ALICE) (?Y BETTY) (?Z CHARLES) (?U EVE))
 ((?X ALICE) (?Y CHARLES) (?Z BETTY) (?U EVE)))
```

The properties of the roles are of course correctly preserved in nRQL queries; for example, `has-descendant` is a transitive role. Other properties of roles are correctly handled as well.

The UNION Constructor

nRQL also offers a UNION constructor:

```
(retrieve (?x) (union (?x woman) (?x man)))
```

RacerPro replies:

```
(((?X CHARLES)) ((?X EVE)) ((?X DORIS)) ((?X BETTY)) ((?X ALICE)))
```

Understanding the UNION Constructor As the name suggests, the UNION just constructs the union of the answer sets of the argument queries (disjuncts). However, the union is only well-defined and meaningful if the argument sets have the same arity. Thus, nRQL must ensure that the answer sets will have the same arity. The arguments query bodies of a `union` query are thus transformed in such a way that they will all produce answer sets of same arity. Consider the query

```
(retrieve (?x ?y) (union (?x ?y has-child) (?x man))).
```

nRQL will rewrite this query into the following:

```
(retrieve (?x ?y) (union (?x ?y has-child) (and (?x man) (?y top))))).
```

Now, both component queries `(?x ?y has-child)` and `(and (?x man) (?y top))` return lists of `(?x,?y)` pairs. Thus, the union is meaningful.

There is another issue. If argument query bodies of a UNION reference *different* variables, then nRQL will ensure that each argument references the same variables, even if the arguments have identical arities. For example, the query

```
(retrieve (?x ?y) (union (?x woman) (?y man)))
```

will be rewritten into

```
(retrieve (?x ?y) (union (and (?x woman) (?y top))
                           (and (?x top) (?y man)))).
```

The variable names matter, since each (differently) named variable creates a new axis in an n -dimensional tuple space. The result of the previous query will be

```
(((?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 BETTY) (?Y DORIS))
 ((?X BETTY) (?Y EVE))
 ((?X BETTY) (?Y CHARLES))
 ((?X BETTY) (?Y ALICE))
 ((?X ALICE) (?Y DORIS))
 ((?X ALICE) (?Y EVE))
 ((?X ALICE) (?Y CHARLES))
 ((?X ALICE) (?Y BETTY)))
```

As expected, this is simply the union of the two component queries

```
> (retrieve (?x ?y) (and (?x woman) (?y top)))
```

```
(((?X EVE) (?Y BETTY))
 ((?X DORIS) (?Y BETTY))
 ((?X ALICE) (?Y BETTY))
 ((?X EVE) (?Y DORIS))
 ((?X BETTY) (?Y DORIS))
 ((?X ALICE) (?Y DORIS))
 ((?X EVE) (?Y CHARLES))
 ((?X DORIS) (?Y CHARLES))
 ((?X BETTY) (?Y CHARLES))
 ((?X ALICE) (?Y CHARLES))
 ((?X DORIS) (?Y EVE))
 ((?X BETTY) (?Y EVE))
 ((?X ALICE) (?Y EVE))
 ((?X EVE) (?Y ALICE))
 ((?X DORIS) (?Y ALICE))
 ((?X BETTY) (?Y ALICE)))
```

and

```
> (retrieve (?x ?y) (and (?x top) (?y man)))
```

```
((?X BETTY) (?Y CHARLES))
((?X DORIS) (?Y CHARLES))
((?X EVE) (?Y CHARLES))
((?X ALICE) (?Y CHARLES)))
```

However, the second disjunct does not produce any additional tuples.

Now consider the same query, but with a modified head:

```
(retrieve (?y) (union (?x woman) (?y man)))
```

Thus, we are only interested in the bindings of `?y`. A common pitfall is to think that this query is equivalent to the query

```
(retrieve (?y) (?y man)).
```

However, it is not. As already described, RacerPro will rewrite this query into

```
(retrieve (?y) (union (and (?x woman) (?y top))
                      (and (?x top) (?y man)))).
```

Thus, the possible bindings for `?y` are from the union of `top` and `man` (and thus `top`). RacerPro therefore replies:

```
((?Y ALICE) ((?Y DORIS) ((?Y EVE) ((?Y CHARLES) ((?Y BETTY))))),
```

whereas

```
(retrieve (?y) (?y man)).
```

just returns

```
((?Y CHARLES)).
```

UNION vs. OR

Consider the query

```
(retrieve (?x) (union (?x C) (?x (NOT C)))),
```

for some concept `C`.

A common pitfall is to consider this as equivalent to

```
(retrieve (?x) (?x (or C (NOT C)))),
```

which in turn is equivalent to

```
(retrieve (?x) (?x TOP))
```

and should thus return all individuals. However, this is *not* the case. The answer of this query is given by the union of the answers of the argument queries

```
(retrieve (?x) (?x C))
```

and

```
(retrieve (?x) (?x (not C))).
```

Note that, *due to the Open World Semantics*, there can be individuals which can neither be proven to be instances of the concept *C*, nor be proven to be instances of the concept (*NOT C*). For example, consider the ABox (*instance i top*). On this ABox, (*retrieve (?x) (union (?x c) (?x (not c)))*) returns NIL, but (*retrieve (?x) (?x (or c (not c)))*) returns of course (((?x i))).

NEG – The Negation As Failure Constructor

A NEG constructor is provided which implements a *Negation as Failure (NAF) Semantics*.

NAF is especially useful for measuring the completeness of the modeling in an ABox, which is important for many applications. Users will probably be familiar with the PROLOG logic programming language, which also offers NAF.

NAF is quite different from classic true negation. The following discussions will make this clear.

Negation as Failure for Concept Query Atoms Consider this query to *family.racer*:

```
(retrieve (?x) (?x grandmother))
```

RacerPro replies:

```
((?X ALICE)).
```

Thus, RacerPro *can prove* that *alice* is a *grandmother*. Fine. If we now use a negated concept in the concept query atom in this query

```
(retrieve (?x) (?x (NOT grandmother))),
```

then we get

```
((?X CHARLES)),
```

since *charles* is a *man*, and thus, he cannot be a *grandmother*. RacerPro is able to prove that *charles* is an instance of (*not grandmother*), given the definitions of *man* and *grandmother*. However, due to the open world semantics, *charles* is the only individual for which this can be proven: If we consider *betty*, then someday *betty* may become a *grandmother*, or she already is, and we just do not have complete knowledge about her. Currently, it is just *not known* that *betty* is a *grandmother*.

So suppose we want to know which individuals are currently *not known* to be instances of *grandmother*. This is where NAF comes into play. To retrieve the individuals for which RacerPro *cannot prove* that they are instances of *grandmother*, we will use a NAF-negated concept query atom as follows:

```
(retrieve (?x) (NEG (?x grandmother)))
```

RacerPro replies:

```
(((?X DORIS)) ((?X EVE)) ((?X CHARLES)) ((?X BETTY))).
```

Note that the `NEG` is placed “around” the entire atom, and that the answer is complementary to the answer set returned by `(retrieve (?x) (?x grandmother))`. In general, for any concept `C`, the query `(retrieve (?x) (union (?x C) (neg (?x C))))` will always return the set of all ABox individuals. Moreover, `(retrieve (?x) (?x (not C)))` will always return a subset of `(retrieve (?x) (neg (?x C)))`.

Things get tricky if NAF is used in combination with classic true negation. Consider

```
(retrieve (?x) (neg (?x (not grandmother)))),
```

which returns

```
(((?X DORIS)) ((?X EVE)) ((?X BETTY)) ((?X ALICE))).
```

We have been asking for all individuals for which RacerPro cannot prove that they are instances of `(not grandmother)`. Since `charles` is the only individual for which membership in `(not grandmother)` can be proven (see above), `((?X CHARLES))` is not in the answer set, since a NAF-negated atom and its non-NAF-negated variant are always complementary.

Negation as Failure for Role Query Atoms The `neg` operator can also be applied to role query atoms. Again, a role query atom and its NAF-negated variant are complementary to one another. Thus, since

```
(retrieve (?x ?y) (?x ?y has-child))
```

yields the 5 tuples

```
(((?X BETTY) (?Y DORIS))
 ((?X BETTY) (?Y EVE))
 ((?X CHARLES) (?Y EVE))
 ((?X ALICE) (?Y BETTY))
 ((?X ALICE) (?Y CHARLES))),
```

on `family.racer`, its NAF-negated variant

```
(retrieve (?x ?y) (neg (?x ?y has-child)))
```

returns the “remaining” 15 tuples (note that we have 5 individuals in the ABox; thus, there must be $5 \times 4 = 20$ pairs of individuals if injective variables are used):

```
(((?X EVE) (?Y BETTY))
 ((?X EVE) (?Y DORIS))
 ((?X EVE) (?Y CHARLES))
 ((?X EVE) (?Y ALICE))
 ((?X BETTY) (?Y CHARLES))
```

```
((?X BETTY) (?Y ALICE))
((?X DORIS) (?Y EVE))
((?X DORIS) (?Y BETTY))
((?X DORIS) (?Y CHARLES))
((?X DORIS) (?Y ALICE))
((?X CHARLES) (?Y BETTY))
((?X CHARLES) (?Y DORIS))
((?X CHARLES) (?Y ALICE))
((?X ALICE) (?Y EVE))
((?X ALICE) (?Y DORIS))).
```

The query

```
(retrieve (?x ?y) (union (?x ?y R) (neg (?x ?y R))))
```

will always return the set of *all pairs* of individuals from an ABox (minus the ones which are excluded due to injective variables), for all role terms R.

Suppose we are now looking for people *without known (explicitly modeled) children*. Unfortunately, the query

```
(retrieve (?x) (neg (?x ?y has-child)))
```

will not solve the task, since the answer is

```
(((?X EVE)) ((?X BETTY)) ((?X DORIS)) ((?X CHARLES)) ((?X ALICE)))
```

However, we already know that only **betty** has children (**doris** and **eve**). What has gone wrong? nRQL has first computed the answer of `(?x ?y has-child)`. Then, the set difference w.r.t. the set of all pairs of individuals was constructed. Finally, a projection to the first component of these pairs was carried out. Since, as we have already seen, the answer to `(neg (?x ?y has-child))` also includes the pairs `((?X BETTY) (?Y ALICE))`, the projection to `?x` also includes **betty**.

In a next attempt we try the query

```
(retrieve (?x) (?x (not (some has-child top)))),
```

which yields NIL. What has gone wrong? Again, due to the open world semantics, if an individual in the ABox does not have a known **has-child** successor, then this does *not* mean that there cannot be any children at all for this individual. There is just no *known* such child. Again, we will need the NEG operator instead of NOT:

```
(retrieve (?x) (neg (?x (some has-child top))))
```

which then yields

```
(((?X DORIS)) ((?X EVE))).
```

However, there is a problem with this query as well! Suppose that `(instance doris mother)` is added to the Abox. Thus, if we query again with

```
(retrieve (?x) (neg (?x (some has-child top)))),
```

then **doris** will be excluded from the answer, even if **doris** does not have a known child:

```
((?X EVE)).
```

The reason is that RacerPro can now prove that **doris** has a child, since **doris** is an instance of the concept **mother**.

Thus, if we want to get a *positive answer* that **doris** does not have any explicitly modeled children in the ABox, we must use the query

```
(retrieve (?x) (neg (?x (has-known-successor has-child))))
```

and finally get

```
((?X DORIS) (?X EVE)),
```

even if **(instance doris mother)** has been added to the ABox.

Please note that this query is equivalent to

```
(retrieve (?x) (neg (project-to (?x) (?x ?y has-child))))).
```

Thus, before computing the complement set of **(?x ?y has-child)** with **neg** we must make a projection to **?x**, and then compute the unary complement of the result of this projection. This will solve the retrieval problem. We will describe the **project-to** constructor in more detail later on.

We also borrowed the following syntax from the query language of the LOOM system:

```
(retrieve (?x) (?x NIL has-child)),
```

which is just syntactic sugar for the previous query.

Please note that, since **has-parent** is the inverse role of **has-child**, the following query achieves the same:

```
(retrieve (?x) (NIL ?x has-parent)).
```

As for the concept query atoms, it is the case that **(?x ?y (not R))** always returns a subset of **(neg (?x ?y R))**. This holds for an arbitrary role term **R**.

Negation as Failure for Constraint Query Atoms

As for the role query atoms, a negated *constraint query atom* returns the complement of its non-NAF-negated variant. Since

```
(retrieve (?x ?y) (?x ?y (constraint (has-father age) age =)))
```

returns

```
((?X EVE) (?Y CHARLES))
```

(note that **charles** is the father of **eve**), we will get the remaining 19 tuples from

```
(retrieve (?x ?y) (neg (?x ?y (constraint (has-father age) age =)))).
```

Negated SAME-AS Query Atoms (NAF)

A negated SAME-AS query atom simply enumerates the complemented of its positive variant. Since

```
(retrieve (?x) (same-as ?x eve))
```

returns

```
(((?X EVE))),
```

we can get all other individuals but **eve** with

```
(retrieve (?x) (neg (same-as ?x eve)))
```

which consequently returns

```
(((?X CHARLES)) ((?X BETTY)) ((?X ALICE)) ((?X DORIS))).
```

NAF-Negated Query Atoms Referencing Individuals

What is the semantics of a NAF atom which references ABox individuals? We already mentioned that a query atom and its NAF-negated counterpart always behave dually. Consider the query

```
(retrieve (betty) (betty top)).
```

which is rewritten into

```
(retrieve ($?betty) (and (same-as $?betty betty) ($?betty top))),
```

and thus gives use the answer `((($?BETTY BETTY)))`.

As a consequence, since `(betty top)` returns a singleton answer set `((($?BETTY BETTY)))`, we should expect that `(neg (betty top))` behaves in fact like a variable, enumerating all individuals that are not **betty**:

```
(retrieve (betty) (neg (betty top))).
```

In fact, RacerPro replies:

```
((($?BETTY DORIS)) ((($?BETTY EVE)) ((($?BETTY CHARLES)) ((($?BETTY ALICE))))
```

Consequently,

```
(retrieve (betty) (neg (betty top)))
```

is an abbreviation for

```
(retrieve ($?betty) (neg (and (same-as $?betty betty) ($?betty top)))).
```

Due to DeMorgan's Law, this query is equivalent to the query

```
(retrieve ($?betty) (union (neg (same-as $?betty betty))
    (and (neg (?$betty top))
        (betty top)))).
```

Thus, whenever an ABox individual appears within a negated query atom it is important to remember that it behaves in fact like a non-injective variable.

However, sometimes this behavior is unwanted. Suppose we want to verify that it cannot be proven that `eve` is a mother, i.e. we want to get a *positive answer* iff it *cannot* be proven that `eve` is a `mother`. However, the query

```
(retrieve (eve) (neg (eve mother))),
```

yields

```
((($?EVE EVE))
((?EVE DORIS))
((?EVE CHARLES))
((?EVE BETTY))
((?EVE ALICE)),
```

as described above, since `eve` turned into a variable. We can “focus on `eve`” if we add an additional conjunct to the query; either `(same-as $?eve eve)`, or `(bind-individual eve)` (which is just syntactic sugar for `(same-as $?eve eve)`):

```
(retrieve (eve) (and (bind-individual eve) (neg (eve mother))))
```

yields

```
((($?EVE EVE)).
```

Thus,

```
(retrieve () (and (bind-individual eve) (neg (eve mother))))
```

returns T.

A Note on Boolean Complex Queries

So far we have introduced the `and`, `union`, `neg` and `inv` query constructors for complex query bodies. Thus, nRQL allows for the orthogonal composition of arbitrarily “boolean query bodies”.

For processing purposes, nRQL will bring these boolean query bodies into *Negation Normal form (NNF)*. In an NNF query, the `NEG` operator appears only in front of query *atoms*. The semantics of the original query is preserved by this transformation.

Moreover, the queries are then brought into *Disjunctive Normal Form (DNF)*, for optimization purposes. This transformation might result in an exponential blowup in query size. We would like to inform the user of this potential performance pitfall.

PROJECT-T0 – The Projection Operator for Query Bodies
--

Consider the query

```
(retrieve (?x) (and (?x c) (?x ?y r) (?y d)))
```

on the ABox

```
(instance a c)
(instance b d)
(instance c top)
(related a b r)
```

which gives us the expected answer

```
(((?X A))).
```

During the evaluation of the body `(and (?x c) (?x ?y r) (?y d))`, bindings for `?x` and `?y` are computed. Internally, nRQL computes an *answer set* for this conjunctive query body, which is simply a list (set) of pairs listing bindings for `?x` and `?y` – in this example, the internal answer set is `((A B))`. The final answer `(((?X A)))` is then computed by *projecting* the tuples to the variables mentioned in the head; in this case to `?x`. Thus, the pairs are projected to their first components which specify the bindings for `?x`, since the head of the example query is `(?x)`; moreover, head projection operators etc. are applied. The projection to the variables mentioned in the head is usually the last step in the query processing chain.

However, for some queries it is required that a projection operation is applied somewhere in between in the processing chain. Suppose you want to retrieve *the C instances which do not have known R successors which are instances of D*. Thus, you want to get the complement of previous query answer; we want to find a query which returns `(((?X B)) ((?X C)))`. In a first attempt we try

```
(retrieve (?x) (neg (and (?x c) (?x ?y r) (?y d)))),
```

which is equivalent to

```
(retrieve (?x) (union (and (neg (?x c)) (top ?y))
                      (neg (?x ?y r)))
                      (and (neg (?y d)) (top ?x))))
```

(please refer to the section on UNION).

However, in this query a projection to `?x` will be applied to the complement of the set `((A B))`, which is the set `((B A) (B C) (C A) (C B) (A C))` – note that the pairs `(B B)`, `(A A)` and `(C C)` are missing, since `?x`, `?y` are injective variables. The projection to `?x` then gives us

```
(((?X B)) ((?X C)) ((?X A))),
```

which is not what we want.

The problem here is that the complement operator `NEG` has been applied to a two-dimensional set, which again (naturally) yields a two-dimensional set, and then the projection is applied.

The solution to the retrieval problem is to apply the complement operator `NEG` *after* the projection to `?x` has been carried out. However, then we must use an explicit projection operator in the query body, since the projection is no longer the last required processing step for this query. We can use the `project-to` operator for query bodies as follows:

```
(retrieve (?x) (neg (project-to (?x) (and (?x c) (?x ?y r) (?y d))))),
```

which returns the desired result

```
((?X B) ((?X C))).
```

Instead of **project-to**, also **project** and **pi** can be used as keywords.

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))).

4.1.4 Defined Queries

nRQL offers a simple macro mechanism for the specification of *defined queries*. For example, we can associate the symbol **mother-of** with the head (?x ?y) of arity 2 and the body (**and** (?x **woman**) (?x ?y **has-child**)) by using the **defquery** facility:

```
(defquery mother-of (?x ?y) (and (?x woman) (?x ?y has-child)))
```

RacerPro replies:

MOTHER-OF

to indicate that the definition has been stored.

Formal and Actual Parameters

The list (?x ?y) in the **mother-of** definition is called the list of *formal parameters* of the definition. Such a list formal parameter list differs from a query head, since only objects (variables and individuals) are allowed, and no head projection operators are permitted.

A defined query can be reused resp. referenced and its body inserted with the **substitute** keyword:

```
(retrieve (?a ?b) (substitute (mother-of ?a ?b))),
```

or, using an alternative syntax,

```
(retrieve (?a ?b) (?a ?b mother-of)).
```

Given the definition of the **mother-of** query, both queries simply expand into

```
(retrieve (?a ?b) (and (?a woman) (?a ?b has-child))).
```

In the expression (**substitute** (**mother-of** ?a ?b)), the parameters ?a ?b are called the *actual parameters*. The number of actual parameters must always match the number of *formal parameters* in the referenced definition, i.e. in the definition (**defquery** **mother-of** (?x ?y) ...). Note that the formal parameters ?x, ?y in the definition of **mother-of** have been renamed to match the actual parameters ?a, ?b).

Thus, the result is:

```
((?A ALICE) (?B CHARLES)) ((?A ALICE) (?B BETTY))
((?A BETTY) (?B EVE)) ((?A BETTY) (?B DORIS)))
```

If one is not interested in the bindings of a certain formal parameter, then one can simply use `NIL` as an actual parameter as well. Thus, the bindings of the corresponding variable in the definition are ignored:

```
(retrieve (?mother) (substitute (mother-of ?mother NIL))),
```

or, using the alternative syntax,

```
(retrieve (?mother) (?mother NIL mother-of)).
```

RacerPro replies:

```
(((?MOTHER ALICE)) ((?MOTHER BETTY))).
```

A defined query can be used at any position where a query body is accepted. For example, you can NAF-negate a defined query:

```
(retrieve (?a ?b) (neg (substitute (mother-of ?a ?b)))),
```

or use

```
(retrieve (?a ?b) (neg (?a ?b mother-of))),
```

which naturally expands into

```
(retrieve (?a ?b) (neg (and (?a woman) (?a ?b has-child)))).
```

an this query is equivalent to

```
(retrieve (?a ?b) (union (and (neg (?a woman)) (?b top))
                           (neg (?a ?b has-child))))).
```

Sometimes it might be necessary to put a `project-to` operator around the referenced defined query, as already discussed:

```
(retrieve (?a) (neg (project-to (?a) (?a ?b mother-of))))
```

Syntactically Ambiguous Queries

Please note that the query

```
(retrieve (?a ?b) (?a ?b mother-of))
```

is *syntactically ambiguous*, given the definition of `mother-of`. The expression is syntactically indistinguishable from a role query atom, since `mother-of` might be a role name as well. The same problem occurs for concept query atoms, where the second argument of the atom can either refer to a concept name or to a defined unary query. In these cases, nRQL will output a warning such as

*** NRQL WARNING: MOTHER-OF EXISTS IN TBOX DEFAULT.
ASSUMING YOU ARE REFERRING TO THE ROLE MOTHER-OF!

informing you that the query is ambiguous, but then assume that you are referring to the role (or the concept) with that name, and not the defined query. Otherwise you must use the `substitute` operator to disambiguate. These ambiguities obviously cannot appear for definitions having more than 2 formal parameters. Moreover, `substitute` is therefore a keyword and thus cannot be used to refer to an individual name. Thus, you should not have an individual named `substitute` in an ABox which you intend to query with nRQL.

Using Defined Queries in Query Definitions

Of course it is possible to use defined queries in query definitions. However, cyclic definitions are prohibited. Consider

```
(defquery mother-of-male-child (?m)
  (and (substitute (mother-of ?m ?c)) (?c man)))

(retrieve (?x) (substitute (mother-of-male-child ?x)))
```

RacerPro replies:

```
(((?X ALICE))).
```

Again, the alternative syntax can be used:

```
(defquery mother-of-male-child (?m) (and (?m ?c mother-of) (?c man)))

(retrieve (?x) (?x mother-of-male-child)).
```

Problems with NAF-negated Defined Queries

Let us discuss a problem which is related to NAF-negated defined queries.

Suppose we want to know who is *not a mother of a male child*. Since we have just defined a query (`mother-of-male-child ?x`) we will first come up with the query

```
(retrieve (?x) (neg (substitute (mother-of-male-child ?x)))).
```

But this query returns the unintended answer

```
(((?X CHARLES)) ((?X JAMES)) ((?X DORIS)) ((?X EVE)) ((?X BETTY)) ((?X ALICE))).
```

Since `((?X ALICE))` is included in the answer of `(retrieve (?x) (?x mother-of-male-child))` we have expected that `((?X ALICE))` would be missing from the answer of `(retrieve (?x) (neg (?x mother-of-male-child)))`. This problem can again be resolved by applying one more projection operator before the complement with NEG is constructed. In fact, we do not want to have the complement of

```
(and (substitute (mother-of ?m ?c)) (?c man)),
```

which is the expression

```
(union (and (neg (?x woman)) (top ?c))
      (neg (?x ?c has-child))
      (and (neg (?c man)) (top ?x))),
```

but the complement of

```
(project-to (?m) (and (substitute (mother-of ?m ?c)) (?c man))),
```

which is the expression

```
(neg (project-to (?x)
      (union (and (neg (?x woman)) (top ?c))
            (neg (?x ?c has-child))
            (and (neg (?c man)) (top ?x))))).
```

See also Section 4.1.3 for the discussion of `project-to`.

Thus, we must use

```
(retrieve (?x) (neg (project-to (?x) (substitute (mother-of-male-child ?x))))).
```

to get the desired result

```
(((?X CHARLES)) ((?X DORIS)) ((?X JAMES)) ((?X EVE)) ((?X BETTY))).
```

The API for Defined Queries

There are various nRQL API functions for accessing and manipulating the defined queries. Please refer to the Reference Manual. It is not yet possible to define TBox queries. However, defined queries can be used for rules as well.

Defined Queries Belong to the TBox

Please note that the definitions are not global, but local. By default, the defined queries are put into a data structure according 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. The following terminal log demonstrates these possibilities:

```
> (full-reset)
:OKAY-FULL-RESET

> (in-tbox a)
A

> (in-abox a)
A
```

```
> (defquery test (?x) (?x a))
TEST

> (instance a a)

> (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)

> (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 TEST (?X) (?X A)) (DEFQUERY TEST2 (?X) (?X TOP)))

> (retrieve (?x) (substitute (test2 ?x)))

Error:
*** NRQL ERROR: CAN'T FIND DEFINITION OF QUERY TEST2 IN DBOX FOR TBOX B!

> (in-tbox a)
A

> (in-abox a)
A
```

```
> (describe-all-definitions)
((DEFQUERY TEST (?X) (?X A)) (DEFQUERY TEST2 (?X) (?X TOP)))

> (retrieve (?x) (substitute (test2 ?x)))
(((?X A)))
```

4.1.5 ABox Augmentation with Simple Rules

nRQL offers a simple ABox augmentation mechanism: rules. This rule engine is also used as a basis engine for the experimental SWRL facilities of RacerPro.

Terminology - Rule Antecedent and Consequence

nRQL rules have an *antecedent* and a *consequence*. 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 Simple nRQL Rule

The following simple nRQL rule “promotes” `woman` which are not yet known to be `mother` to `mothers`:

```
(firerule (and (?x woman) (neg (?x mother))) ((instance ?x mother)))
```

The antecedence is the simple conjunctive query body `(and (?x woman) (neg (?x mother)))`, and the consequence is a single generalize instance assertion. If this rule is applied, then the bindings for `?x` from the answer set of `(and (?x woman) (neg (?x mother)))` are used to “instantiate” the generalized concept assertion, thus producing a set (list) of ordinary instance assertions by substituting `?x` with the current binding. RacerPro returns

```
((INSTANCE EVE MOTHER) ((INSTANCE DORIS MOTHER)))
```

to indicate which ABox assertions have been added by the rule. The query

```
(retrieve (?x) (?x mother))
```

now returns

```
(((?X ALICE) ((?X BETTY) ((?X DORIS) ((?X EVE)))).
```

Note that nRQL rules can behave *non-monotonically* - this rule cannot be applied again, since now all `woman` are known to be `mothers`; thus, `(neg (?x mother))` must fail.

Automatically Adding Rule Consequences 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.

Creating New ABox Individuals and Establishing ABox Relationships with Rules

A nRQL rule consequence can contain generalized concept assertions, generalized role assertions, generalized **constrained** as well as generalized **constraint** assertions.

Suppose that, for each mother which does not have a known child yet in the ABox, we want to create a new child:

```
(retrieve (?x) (and (?x mother)
                      (neg (?x (has-known-successor has-child)))))
```

Answer:

```
((?X DORIS) ((?X EVE))).
```

Thus, **doris** and **eve** are known to be **mothers**, but they have no known children. 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 (new-ind first-child-of ?x) ?x has-mother)))
```

which adds the role assertions

```
((INSTANCE FIRST-CHILD-OF-EVE HUMAN)
 (RELATED FIRST-CHILD-OF-EVE EVE HAS-MOTHER))
 ((INSTANCE FIRST-CHILD-OF-DORIS HUMAN)
 (RELATED FIRST-CHILD-OF-DORIS DORIS HAS-MOTHER)))
```

to the ABox.

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 the rule with empty antecedence

```
(firerule () ((instance new-ind some-concept)))
```

yields

```
*** NRQL ERROR: PARSER ERROR: OBJECT $?NEW-IND NOT MENTIONED IN QUERY BODY
```

Please use

```
(firerule () ((instance (new-ind new-ind) some-concept)))
```

instead.

Rules and the Concrete Domain

As demonstrated, rules can add new concept assertions as well as new role membership assertions to an ABox.

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 rules. Moreover, 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 arguments. Please consider the following examples:

```
> (full-reset)
:OKAY-FULL-RESET

> (define-concrete-domain-attribute age :type cardinal)
AGE

> (firerule () ((instance (new-ind a) (an age)))
((INSTANCE A (AN AGE)))

> (firerule () ((instance (new-ind b) (an age)))
((INSTANCE B (AN AGE)))

> (firerule (and (?x (an age)) (?y (an age)))
((constrained ?x (new-ind age-of ?x) age)
(constrained ?y (new-ind age-of ?y) age))
:how-many 1)
(((CONSTRAINED B AGE-OF-B AGE) (CONSTRAINED A AGE-OF-A 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))
NIL

> (retrieve (?x) (?x (= age 30)))
(((?X A)))

> (retrieve (?x (told-value (age ?x))) (?x (= age 60)))
(((?X B) (:TOLD-VALUE (AGE ?X)) ((+ 30 AGE-OF-A)))))

> (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 (= (+ 30 AGE-OF-A) AGE-OF-B)))))

> (firerule (and (a (an age)) (b (an age)))
((constraints (= (told-value (age b)) (+ 30 (told-value (age a)))))))
```

```
((((CONSTRAINTS (= (+ 30 AGE-OF-A) (+ 30 30)))))

> (abox-consistent?)
T
```

Adding Pseudo-Nominals with Rules

A similar mechanism can be used to add so-called “pseudo-nominals” to an ABox. 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. The following rule achieves exactly this:

```
(firerule (?x top) ((instance ?x (new-symbol concept-for ?x))))
```

Firing the rule adds the following concept assertions to the ABox:

```
((INSTANCE CHARLES CONCEPT-FOR-CHARLES))
((INSTANCE DORIS CONCEPT-FOR-DORIS))
((INSTANCE JAMES CONCEPT-FOR-JAMES))
((INSTANCE EVE CONCEPT-FOR-EVE))
((INSTANCE BETTY CONCEPT-FOR-BETTY))
((INSTANCE ALICE CONCEPT-FOR-ALICE)))
```

Note that we have constructed a set of new atomic concepts using the `new-symbol` operator. The `new-symbol` operator takes the same arguments as the `new-ind` operator (and also has Identical semantics).

Due to the added assertions, the query

```
(retrieve (?x) (?x (some has-child concept-for-doris)))
```

now returns

```
(((?X JAMES) ((?X BETTY))).
```

The concept name `concept-for-doris` can be called a *pseudo-nominal* for the individual `doris`.

The API of the Rule Engine

There are various nRQL API functions (and macros) related to nRQL rules. Please consult the reference manual. Moreover, rules can be used in a “add all consequences at once”, or in a incremental “add one set of consequences at a time” style.

It is possible to implement different strategies of rule application. nRQL offers API functions in order to enable the user or application to control how and when rule consequences are added to an ABox. See Section 4.2.4.

4.1.6 Complex TBox Queries

nRQL can also be used to search for specific superclass-class-subclass relationship patterns in the taxonomy of a TBox. For this purpose, use `tbox-retrieve`.

The Relational Structure of a TBox Taxonomy

Suppose that we view the taxonomy of a TBox as a relational structure. The taxonomy of a TBox is a so-called *directed acyclic graph (DAG)*. A nodes in this DAG represents an *equivalence class* of *equivalent concept names*, and an edge between two nodes represents a *direct-subsumer-of relationship*.

Also assume that the following laws hold:

1. Assume that each node x has a name. The *name of the node* is the name of the equivalence class $[x]$. However, the name of this node might be any element from the equivalence class $[x]$.
2. For each node x (representing an equivalence class $[x]$) and each member $x_i[x]$ in this equivalence class, assume that the predicate x_i holds - thus, $x_i(x)$ is true. Since $x[x]$, also $x(x)$ holds.
3. The edges in this taxonomy representing the direct-subsumer-of relationship are labeled with *has_child*; that is, $\text{has_child}(x, y)$ holds iff x is a *direct subsumer* of y .
4. Let *has_parent* be the inverse relationship of *has_child*, and *has_descendant* be the transitive closure of *has_child*. Let *has_ancestor* be the inverse relationship of *has_descendant*.

Obviously, such a relational structure can also be seen as an ABox. A $\text{has_child}(x, y)$ edge is represented with a role membership axiom (`related x y has-child`), and for each x , the concept membership axiom (`instance x x`) is added, and if y is in (`concept-synonyms x`), then also (`instance x y`) is added, for all such y . But please note that the name of the node x might be any element from the equivalence class of x . We will call this ABox which is constructed according to the four laws the “*taxonomy ABox*”. Note that his taxonomy ABox is not a real ABox.

It is obvious that we can now query this very specific taxonomy ABox with nRQL - thus, the full power of nRQL can be used for TBox querying purposes. However, set of roles that can be used in role query atoms is then limited to `has-child`, `has-parent`, `has-descendant`, `has-ancestor`, and the the set of concept expressions available in concept query atoms is limited to the set of concept names from the TBox (taxony). Of course, querying for concrete domain attributes etc. does not make sense in this setting, since the “*taxonomy ABox*” contains only information according to the given four rules above.

Using TBOX-RETRIEVE to Query the Taxonomy

To give an example, suppose we want to retrieve all *direct* sub-concepts of the class `woman`:

```
(tbox-retrieve (?y) (and (?x woman) (?x ?y has-child)))
```

Answer:

```
(((?Y MOTHER)) ((?Y SISTER)))
```

Alternatively,

```
(tbox-retrieve (?y) (and (?x woman) (?y ?x has-parent)))
```

can be used. To get all sub-concepts of **woman**, use

```
(tbox-retrieve (?y) (and (?x woman) (?x ?y has-descendant)))
```

Answer:

```
(((?Y SISTER)) ((?Y AUNT)) ((?Y *BOTTOM*)) ((?Y MOTHER)) ((?Y GRANDMOTHER)))
```

Please note that, in this example,

```
(tbox-retrieve (?x) (?x woman)),
```

means actually the same as

```
(tbox-retrieve (woman) (woman woman)).
```

However, this is only true because there are no equivalent concepts for **woman** (the equivalence class of **woman** has cardinality one). See below for an example which problems arise if nodes represent equivalence classes with a cardinality greater than one.

To retrieve all concepts but **woman**, use **woman** with **neg**:

```
(tbox-retrieve (?x) (neg (?x woman))).
```

RacerPro replies:

```
(((?X *TOP*)) ((?X *BOTTOM*)) ((?X GRANDMOTHER))
((?X FATHER)) ((?X MOTHER)) ((?X UNCLE))
((?X BROTHER)) ((?X AUNT)) ((?X SISTER))
((?X PARENT)) ((?X MAN))
((?X PERSON)) ((?X HUMAN))
((?X MALE)) ((?X FEMALE)))
```

Due to the presence of the (**instance** **x** **x**) assertions, the child concepts of **woman** can also be retrieved like this:

```
(tbox-retrieve (?y) (woman ?y has-child)).
```

How to Retrieve All Concept Names

How can we retrieve *all concept names*? The following query gives us only the ***TOP*** concept:

```
(tbox-retrieve (?x) (?x top))
```

returns

```
(((?X *TOP*))).
```

Note that this is a consequence of the four rules which we used for construction of the taxonomy ABox. Thus, in order to retrieve all concept names from the taxonomy, we need a special syntax:

```
(tbox-retrieve (?x) (top ?x))
```

RacerPro replies:

```
((?X *TOP*)) ((?X *BOTTOM*)) ((?X GRANDMOTHER))
((?X FATHER)) ((?X MOTHER)) ((?X UNCLE))
((?X BROTHER)) ((?X AUNT)) ((?X SISTER))
((?X PARENT)) ((?X MAN)) ((?X WOMAN))
((?X PERSON)) ((?X HUMAN))
((?X MALE)) ((?X FEMALE)))
```

Complex TBox Queries

Now let us consider a more complex examples. Suppose we are searching for diamond-shaped super-subclass relationships in the taxonomy:

```
(tbox-retrieve (?x ?y ?z ?u)
  (and (top ?x)
    (?x ?y has-child)
    (?x ?z has-child)
    (?y ?u has-child)
    (?z ?u has-child)))
```

The answer is:

```
((?X *TOP*) (?Y MALE) (?Z FEMALE) (?U *BOTTOM*))
((?X *TOP*) (?Y FEMALE) (?Z MALE) (?U *BOTTOM*))
((?X PERSON) (?Y PARENT) (?Z MAN) (?U FATHER))
((?X PERSON) (?Y MAN) (?Z PARENT) (?U FATHER))
((?X PERSON) (?Y PARENT) (?Z WOMAN) (?U MOTHER))
((?X PERSON) (?Y WOMAN) (?Z PARENT) (?U MOTHER)))
```

Problems Caused By Equivalent Concepts

Consider the following example:

```
> (delete-all-tboxes)
> (define-concept a b)
> (define-concept c b)
> (taxonomy)
```

The returned taxonomy looks like this:

```
((TOP NIL ((A B C))) ((C A B) (TOP) (BOTTOM)) (BOTTOM ((A B C)) NIL))
```

Thus, the concepts **A**, **B**, **C** are actually equivalent. The “taxonomy ABox” will contain nodes for **TOP**, **BOTTOM**, as well as a node representing the equivalence class **A**, **B**, **C**. However, we do not know whether this node is named **C**, **A**, or **B**! Let us assume the name **C** is given to this node. Then, the following concept assertions are added to the “taxonomy ABox”:

```
(instance *TOP* TOP)
(instance *BOTTOM* BOTTOM)
(instance C A)
(instance C B)
(instance C C)
```

(Note that we do not check this imaginary ABox for consistency, so `(instance *bottom* bottom)` does no yield an inconsistent ABox; we are merely interested in the relational structure of this ABox here).

Thus, querying this ABox with

```
(tbox-retrieve (?x) (top ?x))
```

gives us

```
(((?X *TOP*)) ((?X *BOTTOM*)) ((?X C)))
```

Since there is no node named **A**, querying this ABox with

```
(tbox-retrieve (a) (a a))
```

gives us **NIL**. However,

```
(tbox-retrieve (c) (c c))
```

correctly returns `((($C C)))`, as well as

```
(tbox-retrieve (c) (c a))
(tbox-retrieve (c) (c b)).
```

Thus, it is usually better to query for **a**, **b** and **c** like this:

```
(tbox-retrieve (?x) (?x a))
(tbox-retrieve (?x) (?x b))
(tbox-retrieve (?x) (?x c))
```

For all three queries, the answer will be

```
(((?X C))).
```

Again we like to emphasize that nRQL TBox queries are basically just plain nRQL ABox queries which are posed to a very specific ABox (the “taxonomy ABox” representing the taxonomy). This taxonomy ABox is constructed from the taxonomy of the TBox according to the given four laws; the semantics of the TBox queries is a direct consequence of using nRQL on the taxonomy ABox.

4.1.7 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).

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 label). Edges must always have a description (the label).

Data Substrate Labels These descriptions are called *data substrate labels*. The exact syntax can be found on Page 150, 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” `("foo" ∨ "bar") ∧ 123.3 ∧ 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.

Data Substrate Nodes, Edges and Labels To populate a data substrate, nodes and edges with appropriate data substrate labels must be created. The macros `data-node` and `data-edge` are provided for this purpose.

For example, the statement

```
(data-node michael ("michael" age 34)
  (and man
    (= age 34)
    (some has-knowledge DL)))
```

does three things:

1. Creates a data substrate node named `michael`;
2. labels this node with the description `("michael" age 34)` which is a “conjunction” of three data literals;

3. creates an ABox individual called `michael`, and finally,
4. adds the concept assertion (`instance michael (and man (= age 34) (some has-knowledge DL))`) to the 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.

As already demonstrated, data substrate nodes can be associated with individuals from an ABox. This is done by simply giving *equal names* to the substrate node (`michael`) and the associated ABox individual (`michael`). However, creation of a corresponding ABox individual is optional. It is thus possible to create data nodes which do not have associated ABox individuals, and vice versa.

Populating the Hybrid Representation 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 to this question is that you can use nRQL to query the hybrid representation. nRQL is a hybrid query language. Let us consider the following example KB:

```
(full-reset)
(in-tbox test)

(implies man human)
(implies professor teacher)
(implies teacher human)

(define-primitive-role colleague :inverse colleague)
```

Let us now associate the ABox `test` with a data substrate. A dat a substrate is always associated with the ABox which has the same name as the data substrate. Thus, we will use (`in-data-box test`) to create a data substrate with the name `test` which will be automatically associated to the ABox we have just created:

```
(in-abox test)

(in-data-box test)
```

We can now populate the substrate and the ABox:

```
(data-node michael ("Michael Wessel" age 34)
      (and man
            (some has-knowledge DL)))

(data-node ralf ("Ralf Moeller" age 38)
      (and professor
            (some has-knowledge DL)))

;; Create an edge between Michael and Ralf
;; and label this edge with the description
;; ("this is a description" has-colleague).
;; Moreover, a "(related michael ralf colleague)"
;; axiom is added to the associated ABox "test".
```

```
(data-edge michael ralf
  ("this is a description" has-colleague)
  colleague)
```

We can ask for a description of the current substrate as well as inspect the labels of the created objects:

```
> (describe-current-substrate)
((:NAME TEST) (:TYPE THEMATIC-SUBSTRATE::DATA-SUBSTRATE)
 (:ASSOCIATED-ABOX TEST) (:ASSOCIATED-TBOX TEST) (:NO-OF-NODES 2))

> (node-label michael)
(("Michael Wessel") (AGE) (34))

> (edge-label michael ralf)
(("this is a description") (HAS-COLLEGUE))
```

Again, note that these lists of lists of data literals must be interpreted as conjunctions (of disjunctions) of data literals.

There are also API functions (and macros) for deleting a substrate node or edge, as well as various other substrate related API functions. Please consult the Reference Manual.

Querying the Hybrid Representation nRQL can now be used to query the hybrid representation. Here is our first *hybrid query*:

```
> (retrieve (?*x ?y) (and (?x (and human (some colleague teacher)))
  (?x ?y colleague)
  (?*x "Michael Wessel")))

(((?*X MICHAEL) (?Y RALF)))
```

Note that the variable `?y` has been bound to an ABox individual, whereas `?*x` has been bound to a substrate node.

The variable `?y` is an ordinary nRQL *ABox variable*, whereas `?*x` is a so-called *substrate variable* (see rule `<data-substrate-query-object>`, Page 150). Whenever we use a variable (or individual) prefixed with an asterix (*), it is assumed that this object is from the substrate realm, and must therefore be bound to a substrate node. Otherwise the object is from the ABox realm and must therefore be bound to an ABox individual. Thus, with respect to the example, the variable `?*x` ranges over the set of nodes in the data substrate, whereas the variable `?y` ranges over the set of ABox individuals.

Moreover, these ABox and substrate variables are *bound in parallel*. Whenever a binding for `?x` is made, a binding of the corresponding substrate variable `?*x` is automatically established to the associated substrate node, if an associated node exists, and vice versa. Note that you can enforce the existence of appropriate corresponding objects by putting conjuncts like `(?x top)` and `(top ?*x)` into your query.

Finally, it should be mentioned that variables which are prefixed with `$?*` also denote substrate variables. However, these variables are non-injective, whereas variables beginning with `?*` are injective variables, as for the nRQL variables ranging over the ABox individuals.

Data Substrate Query Expressions A data substrate variable is bound to a substrate node with a certain label if this label *satisfies* the given *data node query expression* (see 150, rule <data-node-query-expression>). These *substrate data query expressions* play the same role as concept and role expression for ABox query atoms.

The node *query expressions*, for example, the query expression "Michael Wessel" in the query atom (`?*x "Michael Wessel"`), have a structure similar to the node / edge labels. Again, a positive boolean description of *data substrate query items* in CNF is assumed.

As for the usual ABox concept query atoms (e.g., `(?x (and human (some colleague teacher)))`), 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 used within this substrate query atom.

For example, the data substrate node `michael` matches the query atom `(?*x "Michael Wessel")`, because the label of `michael` contains `"Michael Wessel"` as a conjunct, and thus *logically implies* the query expression `"Michael Wessel"`.

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 34)))
```

is equivalent to

```
(retrieve (?*y) (and (?*x age) (?*x 34))).
```

Referencing Data Substrate Nodes It is possible to refer to specific nodes in the substrate by naming them in the query, as it is possible to refer to specific ABox individuals in queries.

In order to distinguish an ABox individual in a query from the name of a data substrate node used within a query, the latter ones are prefixed with * as well. However, * is not part of the name of the data substrate node, but only used for referencing the node. For example, we can reference the substrate node `michael` in a nRQL query with the object name `*michael`:

```
> (retrieve (*michael) (*michael (age 34)))  
(((?*MICHAEL MICHAEL)))
```

Using Data Query Predicates So far we have only used simple data literals (like `age`, `34`, "Michael Wessel") as conjuncts in the CNF of data substrate query expressions. However, nRQL also offers you to use so-called *data predicates*. The recognized predicates are summarized on Page [150](#), see rule `<data-query-predicate>`. For example, consider the following query:

which returns

```
(((?X RALF) ($?*MICHAEL MICHAEL))).
```

Here we have been looking for the colleagues of the data node `michael` which are teachers for which a corresponding data substrate node exists which also satisfies the data substrate query expression (`age (:predicate (< 40))`).

Thus, the label of such a node must contain the symbol `age`, and there must be an other data literal `x` which satisfies (logically implies the validity of) the expression (`< x 40`) (if `x` is substituted with a literal from the label of this node).

In general, data substrate predicate expressions can be used in data substrate node query atoms as well as in data substrate edge query atoms. Note that the ordinary data literals, e.g. `age` in this example, can also be viewed as denoting equality predicates, e.g. (`:predicate (equalp age)`) in this case.

Another example demonstrates that predicates are available which are not offered in the concrete domain of RacerPro, for example, we can search for nodes that have string on their labels that contain the substring "Michael":

```
> (retrieve (?*x) (?*x (:predicate (search "Michael"))))

((?*X MICHAEL))
```

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. The query

```
(retrieve (?*x ?*y) (?*x ?*y (:satisfies (:predicate <))))
```

returns

```
((?*X MICHAEL) (?*Y RALF)).
```

Here we have been searching for substrate nodes `?*x`, `?*y` which satisfy the binary predicate "`<`" - this means, there must be a numeric literal `a` in the label of the node bound to `?*x`, and another numeric literal `b` in the label of the node bound to `?*y` such that `a < b` holds.

A predicate used within a `:satisfies` substrate predicate edge query must always have arity two. Please note that the previous query does not require the presence of an edge between `michael` and `ralf`. Only the node labels are taken into account.

The Mirror Data Substrate

Obviously, most users do not want to populate a data substrate manually, but will prefer a *facility which automatically creates associated data substrate objects for ABox individuals*. The so-called *mirror data substrate* is provided for this purpose. It automatically creates, for every object in a given ABox (the ABox which is the associated ABox of the substrate), a corresponding and appropriately labeled substrate data object.

The functionality provided by the mirror data substrate is especially valuable for users who want to query OWL KBs. Suppose RacerPro has internally created an ABox as a result of reading in an OWL file. If the data substrate mirroring facility is enabled, RacerPro will automatically create data substrate objects for all elements in the OWL KB. However, RacerPro will also create associated data substrate if a KB which is not an OWL KB will be read in.

The associated data substrate is valuable, since nRQL can then offer additional retrieval functionality which is not available on the ABox side. For example, the additional retrieval `:predicates` can be used, which are only available in the hybrid nRQL query language.

Querying the Mirror Data Substrate Let us consider an example to demonstrate the benefits of the mirror data substrate:

```
(full-reset)
(in-tbox test)

(define-concrete-domain-attribute age :type integer)
(define-concrete-domain-attribute name :type string)
(define-primitive-role has-child :parent is-relative)

(equivalent mother (and woman (some has-child human)))
(implies woman human)

(in-abox test)
(in-mirror-data-box test)

(instance alice woman)
(instance betty woman)
(related alice betty has-child)
(constrained betty betty-age age)
(constrained alice alice-age age)
(constrained betty betty-name name)
(constrained alice alice-name name)
(constraints (equal betty-age 80))
(constraints (equal alice-age 50))
(constraints (string= betty-name "Betty"))
(constraints (string= alice-name "Alice"))
```

Due to the declaration (`in-mirror-data-box test`), mirroring is enabled. 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.

In order to be able to distinguish the “type” of a data substrate object, *special markers are added to the labels of these objects*:

- The label of a data substrate node which has been created for an ABox individual will contain the special marker `:abox-individual` as a conjunct (recall that data labels are descriptions in CNF).
- The label of a node representing a concrete domain objects will contain the marker `:abox-concrete-domain-object`.

- The label of a node representing a concrete domain value will contain the value itself, as well as the marker `:abox-concrete-domain-value`. Moreover, the node also has the value as its name.

The `related`, `constrained`, and `constraints` Abox assertions give rise to appropriately labeled edges in the substrate, connecting the different types of substrate nodes.

Whereas a `related` role assertion axiom would result in a data substrate edge connecting two nodes of type `:abox-individual`, a `constrained` assertion would create an edge between an `:abox-individual` and an `:abox-concrete-domain-object`, etc.

The following query demonstrates the use of the different markers for querying purposes:

```
(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"))))))
```

RacerPro replies:

```
(((*X ALICE) (*Y BETTY)
  (?*NAME-OF-*Y BETTY-NAME) (?*AGE-OF-*Y BETTY-AGE)
  (?*TV-1 80) (?*TV-2 "Betty")))
```

Note that, for example, the ABox assertion (`constraints (equal betty-age 80)`) was responsible for the creation of the nodes `betty-age`, `80`, as well as for the establishment of an edge between these two. Let us inspect the labels of these objects:

```
> (node-label alice)
((WOMAN) (*TOP*) (TOP) (:ABOX-OBJECT) (:ABOX-INDIVIDUAL))

> (node-label betty-age)
((:ABOX-CONCRETE-DOMAIN-OBJECT) (:ABOX-OBJECT))
```

```
> (node-label 80)
((:ABOX-CONCRETE-DOMAIN-VALUE) (80))

> (edge-label betty-age 80)
((:ABOX-RELATIONSHIP) (:ABOX-TOLD-VALUE-RELATIONSHIP))

> (edge-label alice betty)
((:ABOX-RELATIONSHIP) (:ABOX-ROLE-RELATIONSHIP) (HAS-CHILD))
```

Please note that also concept assertions have been mirrored, e.g., the information that `alice` is a `woman`.

Moreover, *the amount of information* which is mirrored for the ABox individuals and the ABox related statements is computed according to the *current nRQL mode*, see the documentation of `set-nrql-mode` in the Reference Manual.

For example, using nRQL mode 0, only the *told syntactic information* from the ABox is used for the concept membership assertions. However, for the related axioms, also the effects of the role hierarchy are taken into account. Mode 1 is like mode 0, but also the atomic concept ancestors and the atomic concept synonyms are added to the labels of the substrate nodes. In mode 2, additionally, also the concept synonyms and ancestors for complex concept membership assertions are asserted to the labels of the nodes.

Thus, if we use

```
(in-mirror-data-box test)
(enable-smart-abox-mirroring)
```

in the above example, then the label of `alice` will additionally contain `human`, and the edge label of the edge between `alice` and `betty` also contains `is-relative`, due to the role hierarchy:

```
> (node-label alice)
((WOMAN) (HUMAN) (*TOP*) (TOP) (:ABOX-OBJECT) (:ABOX-INDIVIDUAL))

> (edge-label alice betty)
((:ABOX-RELATIONSHIP) (:ABOX-ROLE-RELATIONSHIP) (HAS-CHILD) (IS-RELATIVE))
```

Markers Used in the Mirror Data Substrate In general, the following set of markers is 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-annotation,
- :owl-annotation-concept-assertion,
- :owl-annotation-relationship,
- :owl-annotation-object-relationship,
- :owl-annotation-datatype-relationship,
- :owl-annotation-datatype-object,
- :owl-annotation-value,
- :owl-datatype-value,
- :owl-datatype-role.

Using the Mirror Data Substrate on OWL Documents Especially in the case of OWL KBs, the additional querying functionality offered by means of a data mirror substrate can be valuable, since OWL KBs tend to contain a lot of annotations and told (XML Schema datatype) data values. With the data substrate mirror you can query for resources which whose fillers of an OWL annotation property contain a certain substring, etc.

The mirror data substrate allows you to evaluate certain (predefined) predicates over the set of OWL annotations. Let us (again) consider the following OWL example KB:

```
<?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:DatatypeProperty rdf:ID="age">
    <rdfs:domain rdf:resource="#person"/>
    <rdf: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"/>
  </owl:FunctionalProperty>
```

```

<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<person rdf:ID="b">
  <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">45</age>
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">betty</name>
</person>

<person rdf:ID="a">
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">betty</name>
  <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">35</age>
</person>

<person rdf:ID="c">
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">michael</name>
  <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">35</age>
</person>

</rdf:RDF>

```

Let us assume this OWL document is stored as a file under `~/dtp.owl`:

```

> (FULL-RESET)
:OKAY-FULL-RESET

> (ENABLE-DATA-SUBSTRATE-MIRRORING)
:OKAY-DATA-SUBSTRATE-MIRRORING-ENABLED

> (OWL-READ-FILE "~/dtp.owl")

Reading ~/dtp.owl...
done.

```

The following queries demonstrate the retrieval possibilities offered by the mirror data substrate:

```

> (retrieve (?*x $?*x-dtp-value)
            (?*x $?*x-dtp-value (:owl-datatype-role)))
((?*X |http://www.owl-ontologies.com/unnamed.owl#b|) ($?*X-DTP-VALUE 45))
((?*X |http://www.owl-ontologies.com/unnamed.owl#b|) ($?*X-DTP-VALUE "betty"))
((?*X |http://www.owl-ontologies.com/unnamed.owl#a|) ($?*X-DTP-VALUE 35))
((?*X |http://www.owl-ontologies.com/unnamed.owl#a|) ($?*X-DTP-VALUE "betty"))
((?*X |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*X-DTP-VALUE 35))
((?*X |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*X-DTP-VALUE "michael")))

> (retrieve (?*x $?*x-dtp-value)
            (?*x $?*x-dtp-value (:owl-datatype-role
                                  |http://www.owl-ontologies.com/unnamed.owl#age|)))
((?*X |http://www.owl-ontologies.com/unnamed.owl#b|) ($?*X-DTP-VALUE 45))
((?*X |http://www.owl-ontologies.com/unnamed.owl#a|) ($?*X-DTP-VALUE 35))
((?*X |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*X-DTP-VALUE 35)))

> (retrieve (?*x $?*x-dtp-value ?*y $?*y-dtp-value)
            (and (?*x $?*x-dtp-value
                  (:owl-datatype-role

```

```

|http://www.owl-ontologies.com/unnamed.owl#age|))
(?*y $?*y-dtp-value
 (:owl-datatype-role
 |http://www.owl-ontologies.com/unnamed.owl#age|))
($?*x-dtp-value $?*y-dtp-value (:satisfies (:predicate =)))))

(((?*X |http://www.owl-ontologies.com/unnamed.owl#a|) ($?*X-DTP-VALUE 35)
 (?*Y |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*Y-DTP-VALUE 35))
 ((?*X |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*X-DTP-VALUE 35)
 (?*Y |http://www.owl-ontologies.com/unnamed.owl#a|) ($?*Y-DTP-VALUE 35)))

> (retrieve (?*x $?*x-dtp-value ?*y $?*y-dtp-value)
 (and (?*x $?*x-dtp-value
 (:owl-datatype-role
 |http://www.owl-ontologies.com/unnamed.owl#name|))
 (?*y $?*y-dtp-value
 (:owl-datatype-role
 |http://www.owl-ontologies.com/unnamed.owl#name|))
 ($?*x-dtp-value $?*y-dtp-value (:satisfies (:predicate string<))))))

(((?*X |http://www.owl-ontologies.com/unnamed.owl#b|) ($?*X-DTP-VALUE "betty")
 (?*Y |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*Y-DTP-VALUE "michael"))
 ((?*X |http://www.owl-ontologies.com/unnamed.owl#a|) ($?*X-DTP-VALUE "betty")
 (?*Y |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*Y-DTP-VALUE "michael")))

> (retrieve (?*x $?*x-dtp-value)
 (and (?*x $?*x-dtp-value
 (:owl-datatype-role
 |http://www.owl-ontologies.com/unnamed.owl#name|))
 ($?*x-dtp-value (:predicate (search "ael")))))
 (((?*X |http://www.owl-ontologies.com/unnamed.owl#c|) ($?*X-DTP-VALUE "michael")))
```

An interesting feature of the mirror data substrate is that variables such as `$?*x-dtp-value` can now be bound to told values (of OWL datatype or annotation fillers)! This is due to the fact that the told values are accessible as nodes in the mirror data substrate. Moreover, retrieval conditions can be specified which are otherwise impossible to specify - for example, in the last query, we were searching for datatype values being fillers of the `|http://www.owl-ontologies.com/unnamed.owl#name|` property that satisfy the predicate `(:predicate (search "ael"))`; thus, we are searching for substrings. These predicates don't exist as concrete domain predicates for reasons of decidability. However, they can easily be offered in the data substrate, since query answering reduces to model checking here, which is fine for data such as "told values" in OWL documents.

Moreover, since a mirror data substrate is just a special kind of data substrate (see previous subsection), nodes and edges can of course be added manually, by using the already discussed constructors `data-node`, `data-edge`, etc. The full data substrate API applies here, please consult the Reference Manual.

The RCC Substrate - Querying RCC Networks

The so-called *Region Connection (RCC) Calculus* offers means for describing and reasoning about qualitative spatial relationships between spatial objects. To support applications which have to represent domain objects having a spatial character, we are supplying yet another type of substrate. The so-called *RCC substrate* is a special kind of data substrate.

The Purpose of the RCC Substrate An *RCC substrate* allows for the creation and querying of so-called *RCC networks*. In contrast to usual data substrates, the set of admissible edge labels is

constrained. Edges must be labeled with so-called *RCC relationships*. However, the set of allowed node labels is not constrained.

The *edges* of an RCC network are labeled with *RCC relationships*, denoting the relative qualitative spatial arrangement of two objects. For example, you can use the set of *RCC5 relationships* to distinguish whether two objects are *disjoint* (RCC5 relationship “DR”), *overlapping* (RCC5 relationship “O”), *congruent* (RCC5 relationship “EQ”), or *contained* within each other (“PP” for proper part or “inside”, and its converse, “PPI” for “contains”). Moreover, *disjunctions* of these relations can be used to represent coarser or indefinite (underspecified) knowledge regarding the spatial relationship between two objects. The RCC substrate offers the well-known set of RCC8 relationships, as well as the RCC5 relationships. The latter ones can be defined in terms of disjunctions of the former ones.

An RCC relation is specified as a single symbol (e.g., DR), or as a flat list of symbols (e.g. (PO DR)) representing a disjunction. In general, again a CNF like description is employed, as for ordinary data substrate edge labels. However, in contrast, only data literals which are symbols from the set of RCC5 or RCC8 base relations are acceptable.

For example, suppose you want to represent that object *a* contains *b*, and *b* contains *c*, but *a* and *c* are *disjoint*. The build-in RCC network consistency checker will discover that this network is inconsistent.

More importantly, it is also possible to query an RCC network using nRQL. Usually, you will not create an “isolated” RCC network, but a network in which the nodes will have *associated (corresponding) ABox individuals*. Thus, the RCC substrate can serve an ABox as an additional representation medium which is aware of the special characteristics of qualitative spatial (RCC) relationships and thus can be used for answering qualitative spatial queries by means of *constraint checking*. For example, if *a* contains *b* (PPI), and *a* and *c* are disjoint (DC), then a nRQL query asking for disjoint (DC) objects will not only retrieve *a* and *c*, but also *b* and *c*, even though only the edges (*a* *b* PPI), (*a* *c* DC) are explicitly modeled in the RCC substrate. However, (*b* *c* DC) is a logical consequence in this RCC network. Moreover, the RCC substrate will detect inconsistencies, like the one discussed above.

The RCC querying facilities are best demonstrated with yet another example:

```
(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 hamburg
  (city hamburg)
  (and city
    (some in-country germany)
    (string= has-name "Hamburg")))

;;; Create some RCC substrate edges.
;;; Note that "data-edge" can be used
;;; as well.
;;; Europe contains (= PPI) Germany:

(rcc-related europe germany PPI)

;;; Germany contains (= PPI) Hamburg:

(rcc-related germany hamburg PPI)
```

In this example, 3 RCC nodes named `europe`, `germany` and `hamburg` are created. For `europe`, no additional description and no corresponding ABox individual is created. The RCC node `germany` is described as *country* \wedge *germany*, and a corresponding ABox individual called `germany` is created as well. The ABox individual is annotated with a RacerPro concept expression. Finally, the node `hamburg` is created, and the three nodes are set into a *spatial containment* relation by using the RCC5 role `PPI` (“proper part inverse”). Note that `rcc-instance` is just syntactic sugar, and `data-node` can be used as well. The same applies to `rcc-related` and `data-edge`.

We can pose the following spatio-thematic query to the RCC substrate we have just defined:

```
(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)))
```

The answer is:

```
(((*X HAMBURG) (*Y GERMANY) (*Z EUROPE)))
```

Given the previous explanation, the meaning of the query should be obvious.

Note that nRQL has *deduced* that `(rcc-related europe hamburg PPI)` holds (thus, also `(rcc-related hamburg europe PP)`) by means of constraint checking, due to the presence of `(rcc-related europe germany PPI)` and `(rcc-related germany hamburg PPI)`.

4.1.8 Functional Lambda Expressions

As previously introduced, lambda expressions can be used in query heads and rule antecedences. nRQL allows a restricted kind of server-sided programming and can thus be used to implement, for example

1. user-defined output formats for query results (e.g., the query results can also be written into a file),
2. certain kinds of combined ABox/TBox queries,
3. efficient aggregation operators (e.g., sum, avg, like in SQL).

In the following nRQL examples we will demonstrate the points 1. to 3. For this, consider the following simple ABox:

```
(related i j r)
(related j k r)
```

Suppose you want to create a *comma separated values file* called `test.csv` which contains all the (possible implied) R role assertions. nRQL allows you to do this:

```
(retrieve (((:lambda (x y)
    (with-open-output-file ("~/test.csv")
        (format *output-stream* "~A;~A~%" x y)))
    ?x ?y))
  (?x ?y r))
```

So, the query body retrieves all `?x`, `?y` individuals which stand in an R relationship; for each `?x`, `?y` tuple, one more line is attached to the file `test.csv`.

Regarding point 2, let us illustrate how “combined” TBox/ABox queries can be used to retrieve the *direct* instances of a concept, which has been requested by many users.

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:

```
? (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)))
```

Thus, both `i` and `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)))
```

Basically, `retrieve1` is like `retrieve`, but first comes the body, and then the head. Thus, `?x` is bound to a `c` instance. Using this binding, it is checked by means of a TBox subquery (`tbox-retrieve1`) whether the individual bound to `?x` is also an instance of any subclass of `c`. If this is the case, the result tuple (resp. the current binding of `?x`) is *rejected* (see the special `:reject` token); otherwise, the result tuple is *constructed* and returned. The returned result tuples make up the final result set.

Please note that the combination of lambda expressions and `:reject` token gives you the ability to define arbitrary, user-defined *filter predicates* which are executed efficiently since they are directly on the RacerPro server.

Finally, let us consider how *aggregation operators* can be implemented. Consider the following book store scenario:

```
(full-reset)

(instance b1 book)
(instance b2 book)

(related b1 a1 has-author)
(related b1 a2 has-author)
(related b2 a3 has-author)
```

```
(define-concrete-domain-attribute price :type real)
(instance b1 (= price 10.0))
(instance b2 (= price 20.0))
```

We can now determine the number of authors of the single books with the following query:

```
? (retrieve (?x
            ((lambda (book)
               (let ((authors
                      (retrieve '(?a) '(,book ?a has-author)))
                     (length authors)))
                  ?x))
            (?x book)
            :dont-show-lambdas-p t))

> (((?x b1) 2) ((?x b2) 1))
```

Another interesting question might be to ask for the *average price* of all the books. This is a little bit more tricky, but works in nRQL as well:

```
? (retrieve
  (((lambda nil
     (let ((prices
            (flatten
              (retrieve '(((told-value-if-exists (price ?x))
                           ' (?x book)
                           :dont-show-head-projection-operators-p t)))
              ' (average-book-price
                  ,(float (/ (reduce '+ prices) (length prices)))))))
        true-query
        :dont-show-lambdas-p t))

  > (((average-book-price 15.0)))
```

The trick here is to use the always true query body `true-query` in order to let nRQL first evaluate the lambda body (since lambda bodies are only valid in nRQL heads, but not available as “first order statements” in RacerPro); thus, the lambda simply acquires all the prices of the individual books and then computes and returns the average price, as expected.

Please refer to the chapter on nRQL in the RacerPro User Guide in order to learn how to implement *even more efficient* versions of these queries.

Functional programming statements cannot only be used in Lambda terms in queries, but also at toplevel.

```
(evaluate (let ((x ...)) ...))
```

Often, user-defined query format output must be generated from query results, e.g., HTML reports. This is easy with nRQL as well. For example, the result of the query

```
(retrieve (?x ?y) (and (?x #!:person) (?x ?y #!:has_pet) (?y #!:cat)))
```

on the famous `people+pets.owl` KB is

```
((?x http://cohse.semanticweb.org/ontologies/people#Fred)
 (?y http://cohse.semanticweb.org/ontologies/people#Tibbs))
 ((?x http://cohse.semanticweb.org/ontologies/people#Minnie)
 (?y http://cohse.semanticweb.org/ontologies/people#Tom)))
```

Suppose this result shall be presented as an HTML table, together with some additional information about the query which has been posed. In principle, it is easy to generate a HTML file using `with-open-output-file` and `print (format)` statements to write some HTML. However, this results in ugly code. For this reason, some syntactic sugar is available. The following query will generate an HTML file `example.html` which is shown in Figure 4.1:

The screenshot shows a Mozilla Firefox window with the title "MiniLisp Output - Mozilla Firefox". The address bar shows "file:///home/mi.wes". The main content area displays the generated HTML page:

```
MiniLisp Output
```

Query Head

```
(?x ?y)
```

Query Body

```
(AND (?y http://cohse.semanticweb.org/ontologies/people#cat) (?x ?y
http://cohse.semanticweb.org/ontologies/people#has_pet) (?x
http://cohse.semanticweb.org/ontologies/people#person))
```

Query Answer

Result Tuple No. 1	
?x	http://cohse.semanticweb.org/ontologies/people#Fred
?y	http://cohse.semanticweb.org/ontologies/people#Tibbs

Result Tuple No. 2	
?x	http://cohse.semanticweb.org/ontologies/people#Minnie
?y	http://cohse.semanticweb.org/ontologies/people#Tom

At the bottom, there is a search bar with "Suchen: indent" and navigation buttons: "Abwärts", "Aufwärts", "Hervor", and "Fertig".

Figure 4.1: Generated HTML Page

```
(evaluate
  (let ((res
         (retrieve '(?x ?y)
                   '(and (?x \# !"person")
                         (?x ?y \# !"has_pet")))))
    (with-html ("example.html"))))
```

```
(html (head) (html (title) (content "MiniLisp Output")))
(html (body)
  (html (h1) (content "MiniLisp Output"))
  (html (h2) (content "Query Head"))
  (content (query-head :last))
  (html (h2) (content "Query Body"))
  (content (query-body :last))
  (html (h2) (content "Query Answer"))
  (html
    (table :border 1)
    (let ((count 0))
      (maplist (lambda (bindings)
        (html
          (tr)
          (html
            (th :colspan 2)
            (content
              (format nil
                "Result Tuple No. ~a"
                (incf count))))))
      (maplist (lambda
        (var-val)
        (let
          ((var (first var-val))
           (val (second var-val)))
          (html
            (tr)
            (html (td) (content var))
            (html (td) (content val))))))
        bindings))
    res))))))
```

4.1.9 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 <query-head> (if age is a concrete domain attribute, see <attribute-name>), as well as (?x betty betty \$?y |<http://a.com/ontoly#a>), and (). (nil) or (t) are invalid <query-head>s.

Moreover, the set of objects referenced within <query-head> must be a subset of the objects which are referenced in <query-body>; otherwise an error will be signaled.

Top-level syntax:

```
(retrieve <query-head>
         <query-body>

(retrieve-under-premise <query-premise>
                      <query-head>
                      <query-body>)
```

```
(tbox-retrieve <query-head>
    <query-body>)
```

Note: In TBox queries, only the roles has-child, has-parent, has-ancestor, has-descendant are meaningful in role query atoms, and only concept names are meaningful in concept query atoms

```
(defquery <query-name>
    <def-query-head>
    <query-body>)
```

```
(firerule <rule-antecedence>
    <rule-consequence>)
```

Query head for queries and defined queries. Note that also the syntax for the data substrate layer queries is included (“hybrid nRQL queries”, see manual):

```
<query-head>          -> "(" <head-entry>* ")"  

<query-premise>       -> List of RacerPro ABox assertions  

<query-name>          -> <symbol> (naming a defined query)  

<def-query-head>      -> "(" <def-head-entry>* ")"  

<head-entry>           -> <abox-query-object> |  
                         <data-substrate-query-object> |  
                         <head-projection-operator>  

<def-head-entry>       -> <abox-query-object> |  
                         <data-substrate-query-object>  

<abox-query-object>   -> <abox-query-variable> |  
                         <abox-query-individual>  

<abox-query-variable> -> "?"<symbol> | $"?"<symbol>  

<abox-query-individual> -> <symbol> (naming a RacerPro ABox individual)
```

Note: Data query objects can only be used if nRQL is put into data substrate mode.

```
<data-substrate-query-object> -> <data-substrate-query-variable> |  
                               <data-substrate-query-individual>  

<data-substrate-query-variable> -> "?*"<symbol> | $"?*"<symbol>  

<data-substrate-query-individual> -> "*"<symbol>
```

Syntax of *head* projection operators:

```
<head-projection-operator> -> "(" <attribute-name> <abox-query-object> ")" |  

                                "(TOLD-VALUE"  

                                "(" <attribute-name> <abox-query-object> "))" |  

                                "( { "TOLD-VALUE"      | "TOLD-VALUES"      |
```

```

    "DATATYPE-FILLER" | "DATATYPE-FILLERS" |
    "FILLER"          | "FILLERS"           |
    "TOLD-FILLER"     | "TOLD-FILLERS"      |
    "ANNOTATION"      | "ANNOTATIONS"       |
}
"( { <OWL-datatype-property> |
      <OWL-annotation-property> }
<abox-query-object> ))"

<symbol>      -> any LISP symbol
              (e.g., huhu, foobar, |http://a.com/ontoly#a|)
```

WITH THE FOLLOWING EXCEPTIONS:

T, NIL, BIND-INDIVIDUAL, INV, NOT, NEG, AND, INTERSECTION,
 OR, UNION, CAP, CUP, RACER, SATISFIES, TOP, BOTTOM,
 CONSTRAINT, HAS-KNOWN-SUCCESSOR, PROJECT, PROJECT-TO, PI,
 SUBSTITUTE, INSERT, SAME-AS, EQUAL, =, INTERSECTION,
 UNION, CAP, CUP, NRQL-EQUAL-ROLE

Query bodies and atoms:

```

<query-body> ->

  <empty-query-body> |
  <abox-query-atom>   |
  <substrate-query-atom> |

  "( { "PROJECT-TO" | "PROJECT" | "PI" } <def-query-head>
      <query-body> ")" |
  "( { "AND" | "CAP" | "INTERSECTION" } <query-body>* ")" |
  "( { "OR" | "CUP" | "UNION" }        <query-body>* ")" |
  "( { "NOT" | "NEG" }                <query-body> ")" |
  "( "INV"                         <query-body> ")" |

<empty-query-body> -> ""

<abox-query-atom> ->

  "( { "NOT" | "NEG" } <abox-query-atom> ")" |
  "( "INV"           <abox-query-atom> ")" |
  "( <abox-query-object> <concept-expression> ")" |
  "( <abox-query-object> <abox-query-object> <role-expression> ")" |
  "( <abox-query-object> "NIL" <role-expression> ")" |
  "( "NIL" <abox-query-object> <role-expression> ")" |
  "( "TOP <abox-query-object> ")" |
  "( "BOTTOM <abox-query-object> ")" |
  "( <abox-query-object> <abox-query-object>
      "( "CONSTRAINT"
```

```

<role-chain-followed-by-attribute>
<role-chain-followed-by-attribute>
<predicate-expression>
")"
")" | 

"(" "BIND-INDIVIDUAL" <abox-query-individual> ")"

"(" { "SUBSTITUTE" | "INSERT" }
  "(" <query-name>
    { <abox-query-object> | <data-query-object> | "NIL" }*
  ")"
")" | 

"(" { <abox-query-object> | <data-query-object> }* <query-name>
")" | 

"(" { "SAME-AS" | "=" | "EQUAL" }
  <abox-query-object> <abox-query-object>
")" | 

"(" <abox-query-object>
  "(" "HAS-KNOWN-SUCCESSOR" <role-expression> ")
")" |

```

nRQL query syntax used for querying the data substrate:

```

<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-variable> <data-query-individual>
")" |

"(" <data-query-object>
  "(" "HAS-KNOWN-SUCCESSOR" <data-edge-query-expression> ")
")" |

"(" { "SUBSTITUTE" | "INSERT" }"

```

```

"( " <query-name>
  { <abox-query-object> | <data-query-object> | "NIL" }*
")"
")"
|
```

"(" { <abox-query-object> | <data-query-object> }* <query-name>
")"

Auxiliary (Racer) syntax:

```

<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!

```

<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 for <R>!

<OWL-annotation-property> -> see <R> on page 56, Fig. 3.1

Note: role-used-as-annotation-property-p
must return T for <R>!

```

<predicate-expression> -> see <CDC> on page 57, Fig. 3.2 |  

  <CD-predicate>
```

Note: means either use <CDC> syntax,
or use <CD-predicate> syntax

```

<CD-predicate> -> "EQUAL" | "UNEQUAL" | "STRING=" | "STRING<>" |  

  ">" | "<" | ">=" | "<=" |  

  "<>" | "=" | "BOOLEAN=" | "BOOLEAN<>"
```

<concept-expression> -> see <C> on page 56, Fig. 3.1

Syntax of data substrate nodes and edge labels:

```

<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>
                                         ")"
                                         |

```

Rule consequences:

```

<constraints-assertion>    -> "(" { "CONSTRAINTS" | "CONSTRAINT" }
                                see RacerPro ``constraints'' syntax ")"

Note that you can also use projection operators
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.

```

4.2 The nRQL Query Processing Engine

The nRQL query processing engine implements the nRQL language. It is an internal part of RacerPro. The nRQL engine offers various querying modes. In this section we describe the core functionality of this engine.

4.2.1 The Query Processing Modes of nRQL

nRQL can be used in different querying modes. The two major modes are the *set at a time mode* and the *tuple-at-a-time mode*.

The Set at a Time Mode

So far we have only used the so-called *set at a time mode*. In this mode, a call to `retrieve` directly returns the answer set of a query at once, in one bunch (the answer set). Most users and/or client applications will be happy with this mode.

The Tuple at a Time Modes - Incremental Query Processing

Sometimes it is demanded to load and compute the tuples from the answer set of a query in an *incremental fashion*. The user or application requesting the tuples can look at the already retrieved tuples and decide whether it is necessary to request yet another tuple from the answer set. These incremental modes are called *tuple at a time modes*.

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`.

We demonstrate the incremental mode with an example. Consider `family.racer` loaded into RacerPro. You are already familiar with the behavior of the API in the *set at a time mode*:

```
> (retrieve (?x) (?x woman))
(((?X BETTY)) ((?X EVE)) ((?X DORIS)) ((?X ALICE)))

> (describe-query-processing-mode)
(... :SET-AT-A-TIME-MODE ...)
```

Now let us switch the nRQL engine into the *tuple at a time mode*:

```
> (process-tuple-at-a-time)
:OKAY-PROCESSING-TUPLE-AT-A-TIME

> (describe-query-processing-mode)
(... :TUPLE-AT-A-TIME-MODE :EAGER ...)
```

The meaning of `:EAGER` will be explained later. Now let us use the incremental query processing mode:

```
> (retrieve (?x) (?x woman))
(:QUERY-466 :RUNNING)

> (get-next-tuple :last)
((?X BETTY))

> (get-next-tuple :query-466)
((?X EVE))

> (get-next-tuple :query-466)
((?X DORIS))

> (get-next-tuple :query-466)
((?X ALICE))

> (get-next-tuple :query-466)
:EXHAUSTED

> (get-answer :query-456)
:NOT-FOUND

> (get-answer :query-466)
(((?X BETTY)) ((?X EVE)) ((?X DORIS)) ((?X ALICE)))
```

Thus, after the nRQL engine has been set into incremental mode, `retrieve` behaves differently: Instead of returning the whole answer set at once, it returns a so-called *query Identifier (Id)*. This query Id can then be used as an argument to API functions such as `get-next-tuple` to identify the query. Moreover, the Identifier `:last` always refers to the last submitted query.

Multiple Running Queries (Concurrent Incremental Query Processing) Conceptually, the nRQL engine is a multi-process 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-467 :RUNNING)

> (retrieve (?x) (?x uncle))
(:QUERY-468 :RUNNING)

> (get-next-tuple :query-467)
((?X CHARLES))

> (get-next-tuple :query-468)
((?X CHARLES))

> (get-next-tuple :query-467)
:EXHAUSTED

> (get-next-tuple :query-468)
:EXHAUSTED

> (retrieve (?x) (?x woman))
(:QUERY-469 :RUNNING)

> (retrieve (?x) (?x man))
(:QUERY-470 :RUNNING)

> (get-next-tuple :query-469)
((?X BETTY))

> (get-next-tuple :query-469)
((?X EVE))

> (get-next-tuple :query-470)
((?X CHARLES))

> (get-next-tuple :query-469)
((?X DORIS))

> (get-next-tuple :query-470)
:EXHAUSTED

> (get-next-tuple :query-469)
((?X ALICE))

> (get-next-tuple :query-469)
:EXHAUSTED
```

Lazy and Eager Tuple at a Time Modes The tuple at a time 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 is not put to sleep. Instead, it continues to compute tuple after tuple, even if the application has not yet requested these “future tuples”. These tuples are put into a queue for future requests.

The API behaves the same for both modes. In both cases, the query answering thread dies when all tuples have been computed, or when the query is manually aborted by the user, or if a timeout is reached, or the maximal number of requested tuples bound is reached. The query becomes inactive then.

4.2.2 The Life Cycle of a Query

A query has a life cycle: It is created, then made active (computes tuples), eventually goes to sleep, will be reactivated (compute some more tuples), etc., and eventually dies. RacerPorter provides the “Queries” tab which can be used to inspect and manage the queries as well as their current states.

Internally, the nRQL engine maintains a number of lists to maintain the life cycles of queries (similar to the scheduler found in operating systems):

1. *A list of all queries.* This list includes queries which have not yet been started, queries which are currently running or waiting, and queries which have already been processed and thus been terminated. A corresponding API function named `all-queries` returns this list. Note that queries which have been started in set at a time mode appear on this list as well. To manage this list, use `delete-query`, `delete-all-queries`, etc. As long as a query is on this list, the API functions will know the Id of the query, and as such it is possible to access the query.
2. *Queries which are ready to run*, but have not been started yet. The queries on this list are called ready queries or prepared queries. No query answering thread has been created for them yet. If you use `retrieve`, queries will be started automatically and put on the list of active queries (see below). It is possible to apply the API function `execute-query` to queries which are ready to run. In order to put a query to this list, use the API function/macro `prepare-abox-query` instead of `retrieve`. If you want to know whether a specific query is on this list, call `query-ready-p` or `query-prepared-p` on the Id of the queries. To retrieve the list of prepared queries, use the API functions `ready-queries` or `prepared-queries` (they are equivalent).
3. *A list of active queries.* These queries have been started. There is a query answering thread associated with each query which is either currently running, thus consuming CPU time, or currently sleeping. To get the list of active queries, use the API function `active-queries`. Thus, the list of active queries is further partitioned into the following two lists:
 - (a) *Queries which are currently running.* The query answering threads of these queries are consuming CPU time, computing the next tuple(s). Retrieve this list with the API function `running-queries`.

- (b) *Queries which are currently waiting (sleeping)*. A query will sleep/wait (if it has been started in lazy mode) until the next tuple is requested by the user or application. The list of waiting queries can be retrieved with the API function `waiting-queries`.
- 4. *A list of already processed (terminated, inactive) queries*. A query is put on this list if its answer set has been computed exhaustively, or if a timeout has been reached, if the query has been aborted (via the API function `abort-query`), or if the maximum number of requested tuples has been computed. To get this list, use the API functions `processed-queries`, `inactive-queries` or `terminated-queries` (they are all equivalent).

It is possible to put a processed query back to the list of ready queries. This is also true for rule. This is especially useful for rules, if a rule has to be fired (applied) more than once. See `reprepare-query`, `reprepare-rule`.

4.2.3 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. `prepared-rules`, `ready-rules` (equivalent)
3. `active-rules`
 - (a) `running-rules`
 - (b) `waiting-rules`
4. `processed-rules`, `terminated-rules`, `inactive-rules` (all equivalent)

Note that there is (yet) no module which automatically checks for applicable rules and applies them automatically. The rules have to be fired manually. However, the API function `applicable-rules` returns the rules which *can* fire.

4.2.4 How to Implement Your Own Rule Application Strategy

nRQL does not offer a fixed rule application strategy other than the simple strategy “add all consequences of a rule to an ABox”. This “strategy” works only if nRQL rules are used in the so-called *set at a time mode*.

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 change the ABox by adding the memorized rule consequences produced by this rule (although nothing is really “chosen” here, please read further).

However, it is also possible to use the *incremental tuple at a time mode* for rules. In this case the nRQL API supports you with appropriate functions which you will need in order to implement domain specific rule application strategies.

As in the incremental query answering modes, instead of returning the whole set of sets of rule consequences at once in a bunch (one big set), nRQL will incrementally return one set of rule consequences constructed after the other, one at a time.

Thus, in the *tuple at a time mode*, the next set of rule consequences must be explicitly requested with `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 has been delivered (and the rule is processed). Otherwise, the API function `add-chosen-sets-of-rule-consequences` must be used to add the chosen sets of consequence manually. However, this function *cannot be called until the rule terminates*. Please note that an aborted rule never adds rule consequences to an ABox.

Let us consider an example session:

```
> (full-reset)
:OKAY-FULL-RESET

> (process-tuple-at-a-time)
:OKAY-PROCESSING-TUPLE-AT-A-TIME

> (instance betty woman)
NIL

> (instance doris woman)
NIL

> (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 DORIS MOTHER) (RELATED DORIS CHILD-OF-DORIS HAS-CHILD))

> (get-current-set-of-rule-consequences :rule-1)
((INSTANCE DORIS MOTHER) (RELATED DORIS CHILD-OF-DORIS HAS-CHILD))

> (choose-current-set-of-rule-consequences :rule-1)
(((INSTANCE DORIS MOTHER) (RELATED DORIS CHILD-OF-DORIS 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)
:EXHAUSTED

> (get-next-set-of-rule-consequences :rule-1)
:EXHAUSTED

> (active-rules)
NIL

> (all-concept-assertions)
((DORIS MOTHER) (DORIS WOMAN) (BETTY WOMAN))

> (all-role-assertions)
(((DORIS CHILD-OF-DORIS) HAS-CHILD))

> (applicable-rules)
(:RULE-1)

> (execute-rule :rule-1)
(:RULE-1 :RUNNING)

> (get-next-set-of-rule-consequences :rule-1)
((INSTANCE BETTY MOTHER) (RELATED BETTY CHILD-OF-BETTY HAS-CHILD))

> (applicable-rules)
NIL

;;; NOTE: RULES WHICH ARE ALREADY RUNNING, E.G. :RULE-1,
;;; DO NOT APPEAR ON THE LIST OF (APPLICABLE-RULES)

> (choose-current-set-of-rule-consequences :rule-1)
(((INSTANCE BETTY MOTHER) (RELATED BETTY CHILD-OF-BETTY HAS-CHILD))
 ((INSTANCE DORIS MOTHER) (RELATED DORIS CHILD-OF-DORIS 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

> (all-concept-assertions)
((BETTY MOTHER) (DORIS MOTHER) (DORIS WOMAN) (BETTY WOMAN))

> (all-role-assertions)
(((BETTY CHILD-OF-BETTY) HAS-CHILD) ((DORIS CHILD-OF-DORIS) HAS-CHILD))
```

4.2.5 Configuring the Degree of nRQL Completeness

So far we have only used the complete modes of nRQL. In these modes, nRQL uses the basic ABox querying functions of RacerPro in order to achieve completeness. However, for certain ABoxes this mode might not scale too well. ABoxes might simply become too big to be checked for consistency with RacerPro. If consistency checking becomes impossible, also the basic ABox retrieval functions of RacerPro which are called by nRQL during retrieval, such as `concept-instances`, will fail as well. If the ABox reasoning and retrieval services of RacerPro come to their limits and become practically unavailable due to high complexity, then it might be an option to consider using the *incomplete modes* of nRQL for retrieval on such an ABox.

However, for certain kinds of ABoxes which are simply structured, the incomplete modes might be complete! For example, some ABoxes only 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 modes. 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. Thus, query answering can be implemented using plain relational lookup techniques, and no (or only very cheap) reasoning is required. It 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.

The nRQL Modes in Detail

The nRQL engine offers the following degrees of completeness for query answering, which are now discussed in order of increasing completeness:

1. *Told information querying (Mode 0)*. In this setting, to answer a nRQL query, only the *syntactic, told information* from an ABox will be used which is given by the ABox assertions.

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. The mode is therefore severely incomplete. For example, for `(instance i a)` `(instance i b)`, either `(retrieve (?x) (?x a))` and `(retrieve (?x) (?x b))` will return `((?X I))`, but the query `(retrieve (?x) (?x (and a b)))` fails as well. In principle, you could write a nRQL rule to augment the ABox syntactically: `(firerule (and (?x a) (?x b)) ((instance ?x (and a b))))`.

However, in the other direction, if an ABox concept assertion such as `(instance i (and a b))` is encountered, then also `(instance i a)` and `(instance i b)` will be added in order to achieve a minimal degree of completeness. 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 (if neither transitively closed roles and role hierarchies etc. are needed).

In order to enable this mode, enter (`set-nrql-mode 0`).

2. Told information querying *plus exploited TBox information for concept names (Mode 1)*. This is like the previous setting, but now the TBox information is taken into account for concept membership assertions of the form (`instance i C`), where `C` is a concept name. If `C` is a concept name, then nRQL will use the classified TBox to compute the set of `atomic-concept-synonyms`, as well as the set of `atomic-concept-ancestors`, and put these as implied concept membership assertions into the ABox as well. For example, if the ABox is (`instance betty mother`), then also (`instance betty woman`) will be added, since `woman` is a member of (`atomic-concept-ancestors mother`). The resulting “upward saturated ABox” is then queried like in the previous setting. If a concept membership assertion uses a top-level AND, then the same process will be applied recursively for the *atomic conjuncts* of this AND.

Note that, in contrast to the previous setting, (`retrieve (?x) (?x woman)`) now correctly returns `((?X BETTY))`. However, this technique will still fail for the query (`retrieve (?x) (?x (and woman human))`). In the latter case, the next setting might help. Again, please note that you have the nRQL rules available to *syntactically augment* the ABox.

In order to enable this mode, enter (`set-nrql-mode 1`).

3. Told information querying *plus exploited TBox information for all concepts (Mode 2)*. Like the previous setting, but now the atomic concept synonyms and ancestors will also be computed for *arbitrary* concept membership assertions, not only for concept membership assertions referencing concept names.

Thus, if the ABox contains, for example, (`instance betty (OR woman mother)`), Racer-Pro will compute the equivalent concepts and concept ancestors of this concept (`OR woman mother`) from the TBox, and add these to the ABox as well – in this case, (`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, enter (`set-nrql-mode 2`).

4. 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, enter (`set-nrql-mode 3`).

5. See below for an explanation of (`set-nrql-mode 4`) and (`set-nrql-mode 5`).
6. 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 modes 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 modes will be needed.

An Example Demonstrating the Different nRQL Modes

Please edit the original `family.racer` KB file, and replace the (`instance alice mother`) with (`instance alice (and human grandmother)`), and (`instance betty mother`) with

(instance betty (or mother grandmother)). Also add (related eve jade has-child). Enter (full-reset), and reload the file into RacerPro. Then, the following answers should be reproducible:

```
> (set-nrql-mode 0)
:OKAY-MODE-0

> (retrieve (?x) (?x mother))
NIL

> (set-nrql-mode 1)
:OKAY-MODE-1

> (retrieve (?x) (?x mother))
((?X ALICE))

> (set-nrql-mode 2)
:OKAY-MODE-2

> (retrieve (?x) (?x mother))
((?X ALICE) ((?X BETTY)))

> (set-nrql-mode 3)
:OKAY-MODE-3

> (retrieve (?x) (?x mother))
((?X BETTY) ((?X EVE)) ((?X ALICE)))
```

The Two-Phase Query Processing Modes

The so-called *two-phase query processing modes* are special lazy incremental (tuple at a time) query processing modes. The modes are complete. To put nRQL into one of these modes, use (`set-nrql-mode 4`) or (`set-nrql-mode 5`). Internally, the two-phase tuple computation mode will also be exploited for the nRQL mode 6 which is a set at a time mode. Mode 6 behaves like mode 3 and is thus complete, but reduces the number of calls to RacerPro's expensive ABox retrieval functions to a minimum. See below for further discussion of this mode.

Let us describe the modes 4 and 5. If nRQL is used in these modes, then delivery of tuples will be arranged in *two phases*:

- In *phase one*, the so-called *cheap tuples* will be returned,
- followed by the *expensive tuples* in *phase two*.

The *cheap tuples* can be delivered *without* using RacerPro's ABox retrieval functions. These tuples are computed from nRQL's internal data structures and are computed as if nRQL were used in (`set-nrql-mode 1`) or (`set-nrql-mode 2`). Thus, the tuples which are delivered in *phase one* of (`set-nrql-mode 4`) are exactly the same as if nRQL were used in (`set-nrql-mode 1`). The same applies to the relationship between (`set-nrql-mode 5`) and (`set-nrql-mode 2`).

In phase two, after the cheap tuples are exhausted, nRQL will switch to the complete ABox retrieval mode. In this mode, computation of tuples might take considerably longer. These tuples are therefore called the *expensive tuples*. The expensive tuples are computed using RacerPro's ABox retrieval functions.

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 expensive ABox retrieval functions and might thus eventually take longer. However, delivery of this warning token is optional. Please refer to the Reference Manual for more details and inspect the following example session.

Completeness of Modes 4 and 5 Please note that, like mode 3, modes 4 and 5 are complete. The overall set of tuples delivered is the same as in mode 3. The only difference is that delivery of tuples is arranged in two phases, and that mode 4 and 5 are lazy and incremental. However, tuples will not be duplicated. Mode 5 will eventually return more tuples in phase 1 than mode 4, but the overall returned set of tuples is Identical.

Mode 4 and 5 Cannot be Used for Queries with NEG Please note that only queries that do not make use of the `NEG` operator can take advantage of the two-phase query processing mode (the reason is that if an incomplete answer is computed for the argument query body of `NEG` in mode 1, then the set-difference yields already too many tuples in mode 1; thus, the answer would be incorrect).

If nRQL is in mode 4, 5, or 6, and a query is posed containing `NEG`, then nRQL will answer this query using mode the incremental (tuple at a time) mode 3.

Two-Phase Tuple Computation Example Session Please modify the original `family.racer` file as follows: Replace the assertions (`instance alice mother`) with (`instance alice (and human grandmother)`), (`instance betty mother`) with (`instance betty (or mother grandmother)`), and add (`related eve jane has-child`). Enter (`full-reset`) and reload the KB.

Using mode 4, nRQL behaves like this:

```
> (set-nrql-completeness 4)
:OKAY-MODE-4

> (describe-query-processing-mode)
(... :MODE-4 :TUPLE-AT-A-TIME-MODE
    :LAZY :TWO-PHASE-QUERY-PROCESSING-MODE ...
    :DELIVER-PHASE-TWO-WARNING-TOKENS ...)

> (retrieve (?x) (?x mother))
(:QUERY-9 :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))
```

```
> (get-next-tuple :last)
:EXHAUSTED
```

We can also switch to mode 5. Note that phase 1 using mode 5 is now able to produce one more tuple; thus, the `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` warning token is delivered slightly later:

```
> (set-nrql-completeness 5)
:OKAY-MODE-5

> (retrieve (?x) (?x mother))
(:QUERY-10 :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))

> (get-next-tuple :last)
:EXHAUSTED
```

Mode 6 vs. Mode 3

If you use `(set-nrql-mode 6)`, then nRQL will basically behave as if `(set-nrql-mode 3)` were used.

Thus, mode 6 is a *complete set at a time mode*. However, unlike mode 3, nRQL will exploit the two phase query processing scheme. nRQL will thus ensure that an expensive ABox test on a tuple (in phase two) will only be made if that same tuple has not already been computed in phase one. Thus, expensive ABox retrieval functions are avoided whenever possible. However, since phase one is incomplete, it might be the case that no answer 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.

4.2.6 Automatic Deadlock Prevention

Certain types of queries (or rules) must make temporary changes to 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 and references the same ABox, then these different queries must be isolated from one another. nRQL uses locking techniques to achieve isolation and preserve ABox consistency. Thus, an ABox which must be modified by a query (or rule) in order to be answered will be locked until that query has been terminated.

This means that nRQL will not allow you to start (execute) a query (rule) which is about to make changes to an ABox if there are still other queries active that refer to the same ABox. Such a “delay situation” for a query can only happen if nRQL is used in incremental (tuple at a time) mode.

But there is a problem. If some of these active queries had been started in *lazy* incremental tuple computation mode, then these queries will not terminate automatically (unless a timeout had been set or the last tuple is requested). Please note that queries which have been started in *eager* incremental mode will terminate automatically after some time, since the query processing thread is not put to sleep. Thus, in order to avoid deadlock situations, nRQL will eventually not allow you to start a query at all. If such a deadlock situation is encountered by nRQL, `execute-query` will return immediately with an appropriate error message.

```

> (FULL-RESET)
:OKAY-FULL-RESET

> (DEFINE-PRIMITIVE-ROLE R :DOMAIN C :RANGE (NOT C))
36:OKAY

> (INSTANCE I C)
:OKAY

> (INSTANCE J C)
:OKAY

> (INSTANCE K D)
:OKAY

> (INSTANCE L C)
:OKAY

> (PROCESS-TUPLE-AT-A-TIME)
:OKAY-PROCESSING-TUPLE-AT-A-TIME

> (ENABLE-LAZY-TUPLE-COMPUTATION)
:OKAY-LAZY-MODE-ENABLED

> (RETRIEVE (?X) (?X C))
(:QUERY-1 :RUNNING)

> (GET-NEXT-TUPLE :LAST)
:WARNING-KB-HAS-CHANGED

> (GET-NEXT-TUPLE :LAST)
((?X I))

> (ACTIVE-QUERIES)
(:QUERY-1)

> (RETRIEVE (?X ?Y) (?X ?Y (NOT R)))

*** 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).
(:QUERY-2 :DENIED-DUE-TO-DEADLOCK-PREVENTION)

```

```
> (GET-NEXT-TUPLE :LAST)
((?X J))

> (GET-NEXT-TUPLE :LAST)
((?X L))

> (GET-NEXT-TUPLE :LAST)
:EXHAUSTED

> (RETRIEVE (?X ?Y) (?X ?Y (NOT R)))
(:QUERY-3 :RUNNING)

> (GET-NEXT-TUPLE :LAST)
((?X L) (?Y J))

> (GET-NEXT-TUPLE :LAST)
:WARNING-KB-HAS-CHANGED

> (GET-NEXT-TUPLE :LAST)
((?X L) (?Y I))

> (GET-NEXT-TUPLE :LAST)
((?X K) (?Y L))

> (GET-NEXT-TUPLE :LAST)
((?X K) (?Y J))

> (GET-NEXT-TUPLE :LAST)
((?X K) (?Y I))

> (GET-NEXT-TUPLE :LAST)
((?X J) (?Y L))

> (GET-NEXT-TUPLE :LAST)
((?X J) (?Y I))

> (GET-NEXT-TUPLE :LAST)
((?X I) (?Y L))

> (GET-NEXT-TUPLE :LAST)
((?X I) (?Y J))

> (GET-NEXT-TUPLE :LAST)
:EXHAUSTED

> (ACTIVE-QUERIES)
NIL

> (PROCESSED-QUERIES)
(:QUERY-3 :QUERY-1)
```

4.2.7 Reasoning with Queries

The services described in this Section should be considered as non-essential add-ons and are still experimental. If you are wondering why a certain query never returns any tuples and you think it should, consider using the consistency checking service (it might find an inconsistency in your query).

Reporting Inconsistent and Tautological Queries

The nRQL engine also offers (yet experimental!) reasoning services for queries. Consistency checking of queries is such a service. An inconsistent query must necessarily produce an empty answer on all ABoxes; thus, CPU cycles can be saved if such queries are recognized before starting them. In contrast, a *tautological* query always returns *all possible combinations of tuples*. In most cases, such a query does not return useful information. Thus, it is good to recognize such queries as well. Even though the reasoning mechanism offered by nRQL for queries are incomplete (and yet experimental), they are still useful.

The incompleteness in the query consistency checker is caused by NAF. If you don't use NAF in your query, then the consistency check is "quite complete". However, the dual situation arises for the tautology checker.

Here is an example session, demonstrating the utility of nRQL's reasoning facilities. Again we have used the original `family.racer` KB, but we have entirely removed the `signature` statement, and added the following role declarations instead:

```
(define-primitive-role has-descendant
  :transitive t
  :inverse descendant-of)

(define-primitive-role has-child
  :parent has-descendant
  :inverse has-parent
  :domain parent
  :range person)

(define-primitive-role has-parent
  :parent descendant-of)

(disjoint man woman)
```

Then, nRQL should behave as follows:

```
> (report-inconsistent-queries)
:OKAY-REPORTING-INCONSISTENT-QUERIES

> (report-tautological-queries)
:OKAY-REPORTING-TAUTLOGICAL-QUERIES

> (retrieve (?x) (and (?x woman) (?x man)))

*** NRQL WARNING: QUERY-1 IS INCONSISTENT
:INCONSISTENT
```

```

> (prepare-abox-query (?x) (and (?x woman) (?x (not mother))
                                 (?x ?y has-child) (?y person)))

*** NRQL WARNING: QUERY-3 IS INCONSISTENT
(:QUERY-3 :READY-TO-RUN)

> (execute-query :query-3)
: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-5 IS INCONSISTENT
(:QUERY-5 :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)))

*** NRQL WARNING: QUERY-7 IS INCONSISTENT
(:QUERY-7 :READY-TO-RUN)

> (prepare-abox-query (?x) (neg (?x bottom)))

*** NRQL WARNING: QUERY-9 IS TAUTOLOGICAL
(:QUERY-9 :READY-TO-RUN)

> (prepare-abox-query (?x) (union (neg (?x man)) (neg (?x woman)))))

*** NRQL WARNING: QUERY-11 IS TAUTOLOGICAL
(:QUERY-11 :READY-TO-RUN)

```

Reporting Inconsistent and Tautological 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)
:OKAY-REPORTING-INCONSISTENT-QUERIES

> (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)
```

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 offered query entailment check is incomplete as well, but T answers can be trusted.

Query entailment is more difficult to understand than query consistency. For each query, an associated *vector of variables and individuals* exists. This vector is called the *object vector*, and should not be confused with the *head* of a query. Even queries with empty head have an object vector.

The *object vector* lists all variables and individuals which are referenced in the body of the query, but in *lexicographic order*, not in syntactic order. Recall that `union` queries are internally rewritten such that each disjunct references the same variables (see Section 4.1.3).

For example, the associated object vector of the query `(and (?x person) (?y top))` is `<?x,?y>`. The vector of `(and (?y person) (?x top))` is also `<?x,?y>`. This vector specifies the *internal format* of the answer tuples, which are projected and reordered according to an eventually given head to yield the answer set which is finally returned to the user (as a result of a call to `retrieve` etc.)

When two queries are checked for entailment, then these *internal* sets of answer tuples are compared, and not the resulting answer sets which are returned to the user. The tuples of the *internal* answer set of the body `(and (?x person) (?y top))` are also “typed”, in this case the type is `<person,top>` This means, the first element of each tuple has type `person`, and the second element has type `top`, since the object vector is `<?x,?y>`.

For the query body `(and (?y person) (?x top))`, the object vector is also `<?x,?y>`. Thus, the internal result tuples have type `<top,person>`.

Given this example, it is obvious that these internal tuple sets are not in a subset relationship to one another, or even equal. Thus, there holds no query entailment (subsumption) relationship between the two queries.

The situation is even more complex when checking for entailment of two queries having *different* object vectors. In this case, the variables from the second query (which is assumed to be the more general one) are renamed in order to match the names in the vector of the first query (the more specific one). However, *individuals* are never renamed. Given the first query and the variable-renamed second query, then a query `(and query1 (neg query2))` is constructed, and checked for consistency with the query consistency checker. If this query is inconsistent, then `query1` entails `query2`.

Renaming of variables is done according to the positions of the variables in the object vectors, and as such, the lexicographic order matters. Consider the following examples:

```
> (full-reset)
:OKAY-FULL-RESET

> (prepare-abox-query () (and (?x woman) (?y person)))
(:QUERY-1 :READY-TO-RUN)

> (prepare-abox-query () (and (?b woman) (?a person)))
(:QUERY-2 :READY-TO-RUN)

> (query-equivalent-p :query-1 :query-2)
NIL

;; Note: The object vector of :QUERY-1 is <?x,?y>; and the vector
;; of :QUERY-2 is <?a,?b> (due to lexicographic reordering).
;; Thus, the INTERNAL answer sets of these queries are not
;; equivalent. According to these vectors, the tuples in
;; the internal answerset of :QUERY-1 have type
;; <WOMAN,PERSON>, whereas the tuples in :QUERY-2
;; have type <PERSON,WOMAN>. Thus, the queries are
;; not equivalent, since the sets are unequal.
;; Lets rename the variables in the query:
```

```
> (prepare-abox-query () (and (?a woman) (?b person)))
(:QUERY-4 :READY-TO-RUN)

> (query-equivalent-p :query-1 :query-4)
T
```

Finally, some more complex examples, demonstrating the utility of the query entailment checker. Please ensure that `family.racer` has been loaded:

```
> (prepare-abox-query () (betty woman))
(:QUERY-1 :READY-TO-RUN)

> (prepare-abox-query () (betty mother))
(:QUERY-2 :READY-TO-RUN)

> (query-entails-p :query-1 :query-2)
NIL

> (query-entails-p :query-2 :query-1)
T

> (prepare-abox-query () (charles human))
(:QUERY-3 :READY-TO-RUN)

> (query-entails-p :query-1 :query-3)
NIL
```

```

> (query-entails-p :query-2 :query-3)
NIL

;; Note: (betty woman) DOES NOT ENTAIL (charles human),
;; since charles will not be renamed into betty !

> (prepare-abox-query () (?x woman))
(:QUERY-4 :READY-TO-RUN)

> (query-entails-p :query-4 :query-1)
NIL

> (query-entails-p :query-1 :query-4)
T

;; Note: (betty woman) ENTAILS (?x woman),
;; but not the other way around

> (query-entails-p :query-2 :query-4)
T

;; Note: (betty mother) ENTAILS (?x woman)

> (prepare-abox-query () (?x human))
(:QUERY-5 :READY-TO-RUN)

> (query-entails-p :query-2 :query-5)
T

> (query-entails-p :query-5 :query-4)
NIL

> (query-entails-p :query-4 :query-5)
T

> (define-concept woman (and person (some has-gender female)))
NIL

> (prepare-abox-query () (and (?x person) (?x ?y has-gender)
                               (?y female)))
(:QUERY-6 :READY-TO-RUN)

> (query-entails-p :query-6 :query-5)
T

;; Note: (and (?x person) ...) ENTAILS (?x human)

> (query-entails-p :query-6 :query-4)
T

;; Note: (and (?x person) ...) even ENTAILS (?x woman), due
;; to redefinition of concept woman!

```

```

> (query-entails-p :query-4 :query-6)
NIL

> (prepare-abox-query () (and (?x person) (?x ?f has-gender)
                               (?f female) (?x ?y has-child)
                               (?y person)))
(:QUERY-7 :READY-TO-RUN)

> (prepare-abox-query () (?x mother))
(:QUERY-8 :READY-TO-RUN)

> (query-entails-p :query-7 :query-8)
NIL

;; Note: In fact we have expected T here, but then we
;; see that the object vector of :QUERY-7 is
;; <?f,?x,?y>, thus of type <FEMALE,PERSON,PERSON>,
;; which is compared to the vector of :QUERY-8 which is
;; <?x> : <MOTHER>. Due to the modelling in the TBox
;; female and mother are unrelated.

> (prepare-abox-query () (and (?x person) (?x ?y has-gender)
                               (?y female) (?x ?z has-child)
                               (?z person)))
(:QUERY-9 :READY-TO-RUN)

;; Note: The object vector of :QUERY-9
;; is now <?x,?y,?z> : <PERSON,FEMALE,PERSON>
;; Thus, we get the expected result:

> (query-entails-p :query-9 :query-8)
T

;; <PERSON,FEMALE,PERSON> is in fact <MOTHER,FEMALE,PERSON>, thus
;; this is compared to <MOTHER>, and the entailment is detected!
;; Note that this is a non-trivial inference.

> (query-entails-p :query-8 :query-9)
NIL

;; Note: This is correct, since :QUERY-8 = (?x mother) DOES NOT
;; enforce the existence of KNOWN children, but :QUERY-9 does!
;; Thus, :QUERY-9 is more specific than :QUERY-8.

```

4.2.8 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, similar to the taxonomy which is computed from a TBox. Once a query has been classified, *cached answer sets* from its parent and children queries in that QBox will be exploited for optimization purposes (as tuple caches). The incompleteness of the query entailment check is not an issue here, since the QBox is only used as a cache and thus for optimization purposes.

If a classified query must be executed, then the cached answer sets of parent queries (direct subsuming queries) can be utilized as *superset caches*, and the cached answers sets of child queries (direct subsumed queries) will be exploited 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 stricter than the one discussed in the previous Section. Here we additionally require that the *object vectors* must have equal length. Otherwise the queries cannot be in a QBox subsumption (entailment) relationship to one another. This is due to the fact that we want to exploit cached tuples. The tuples from the cache must therefore have the same “format” as the requested tuples. Therefore, a stronger notion of query entailment is required for this purpose.

It is possible to ask for the parents, children as well as for equivalent queries of queries which have been classified. The QBox can also be printed graphically. Please consult the Reference Manual.

Please consider the following example session to get familiar with the offered functionality and API functions:

```
> (full-reset)
:OKAY-FULL-RESET

> (load "family.racer")
; Loading text file /home/mi.wessel/family.racer
#P"/home/mi.wessel/family.racer"

> (define-primitive-role has-parent :inverse has-child)
HAS-PARENT

> (enable-query-repository)
:OKAY-QUERY-REPOSITORY-ENABLED

> (retrieve (?x) (?x top))

Concept (BROTHER) causes a cycle in TBox FAMILY.
Concept (SISTER) causes a cycle in TBox FAMILY.
Concept (PARENT) causes a cycle in TBox FAMILY.
(((?X ALICE)) ((?X BETTY)) ((?X DORIS)) ((?X CHARLES)) ((?X EVE)))

> (retrieve (?a) (?a human))
(((?A EVE)) ((?A CHARLES)) ((?A DORIS)) ((?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 EVE)) ((?Y DORIS)) ((?Y CHARLES)) ((?Y BETTY))

> (show-current-qbox t)

;;;
;;; QBOX FOR RACER-DUMMY-SUBSTRATE FOR ABOX SMITH-FAMILY
;;;
;;; MASTER-TOP-QUERY
;;; |---(AND (?X-AN01 WOMAN) (?Y HUMAN) (?X-AN01 ?Y HAS-CHILD))
;;; |   \___MASTER-BOTTOM-QUERY
;;; \___(?X TOP)
;;;     \___(?A HUMAN)
;;;         \___(?B WOMAN)
;;;             \___MASTER-BOTTOM-QUERY

:SEE-OUTPUT-ON-STDOUT

> (retrieve (?a) (?a mother))
((?A BETTY)) ((?A ALICE))

> (retrieve (?y) (and (?x mother) (?x ?y has-child) (?y human)))
((?Y BETTY)) ((?Y CHARLES)) ((?Y DORIS)) ((?Y EVE)))

> (show-current-qbox t)

;;;
;;; QBOX FOR ABOX SMITH-FAMILY
;;;
;;; MASTER-TOP-QUERY
;;; |---(AND (?X-AN01 WOMAN) (?Y HUMAN) (?X-AN01 ?Y HAS-CHILD))
;;; |   \___MASTER-BOTTOM-QUERY
;;; \___(?X TOP)
;;;     \___(?A HUMAN)
;;;         \___(?B WOMAN)
;;;             \___(?A MOTHER)
;;;                 \___MASTER-BOTTOM-QUERY

:SEE-OUTPUT-ON-STDOUT

> (show-current-qbox)

;;;
;;; QBOX FOR RACER-DUMMY-SUBSTRATE FOR ABOX SMITH-FAMILY
;;;
;;; O:MASTER-TOP-QUERY
;;; |---4:QUERY-9 = (QUERY-48)
;;; |   \___O:MASTER-BOTTOM-QUERY
;;; \___1:QUERY-1
;;;     \___2:QUERY-2 = (SUBQUERY-2-OF-QUERY-48 SUBQUERY-2-OF-QUERY-9)

```

```

;;;
      \___3:QUERY-5 = (SUBQUERY-1-OF-QUERY-9)
;;;
      \___5:QUERY-43 = (SUBQUERY-1-OF-QUERY-48)
;;;
      \___0:MASTER-BOTTOM-QUERY

:SEE-OUTPUT-ON-STDOUT

> (query-equivalents :last)
(:QUERY-9)

> (retrieve (?y) (and (?x mother) (?y ?x has-parent) (?y man)))
((?Y CHARLES)))

> (show-current-qbox t)

;;;
;;; QBOX FOR ABOX SMITH-FAMILY
;;;
;;; MASTER-TOP-QUERY
;;; |---(AND (?X-AN01 WOMAN) (?Y HUMAN) (?X-AN01 ?Y HAS-CHILD))
;;; |   \___(AND (?X-AN01 MOTHER) (?Y MAN) (?Y ?X-AN01 HAS-PARENT))
;;; |       \___MASTER-BOTTOM-QUERY
;;; \___(?X TOP)
;;;     \___(?A HUMAN)
;;;         \___(?B WOMAN)
;;;             \___(?A MOTHER)
;;;                 \___MASTER-BOTTOM-QUERY

:SEE-OUTPUT-ON-STDOUT

> (query-children :last)
(MASTER-BOTTOM-QUERY)

> (query-children :query-9)
(:QUERY-72)

> (query-parents :query-72)
(:QUERY-9)

> (query-equivalents :query-37)
(:QUERY-9)

> (query-equivalents :query-9)
(:QUERY-37)

```

4.2.9 The Query Realizer

Query realization can be called a *semantic query optimization* which will enhance the amount of information available for the guidance of the search process used for query answering. The service is experimental as well.

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

> (load "family.racer")

; Loading text file /home/mi.wessel/family.racer
#P"/home/mi.wessel/family.racer"

> (enable-query-realization)

:OKAY-QUERY-REALIZATION-ENABLED

> (retrieve (?x) (and (?x woman) (?x ?y has-child) (?y human)))

(((?X ALICE) ((?X BETTY)))

;; 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!

> (describe-query :last)

(:QUERY-1 (:ACCURATE :PROCESSED)
  (RETRIEVE (?X)
    (AND (?X ?Y-ANO1 HAS-CHILD)
      (?Y-ANO1 (AND HUMAN PERSON))
      (?X (AND WOMAN MOTHER)))))

;; This is the realized query body.
;; Note that nRQL has used the TBox to
;; add logically implied conjuncts syntactically
;; to the query expression.
;;
;; You can also get to original query:

> (describe-query :last nil)

(:QUERY-1 (:ACCURATE :PROCESSED)
  (RETRIEVE (?X) (AND (?X WOMAN)
    (?X ?Y HAS-CHILD))
```

```

        (?Y HUMAN)))))

> (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 ?Y-ANO1 HAS-CHILD)
         (?Y-ANO1 (AND HUMAN MAN))
         (?X (AND WOMAN (ALL HAS-DESCENDANT MAN) MOTHER)))))

;; Note that nRQL has deduced that ?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.

4.2.10 The nRQL Persistency Facility

Internally, nRQL maintains a so-called *substrate* data structure for each RacerPro ABox which has been 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. The API functions `store-substrate-for-abox` and `store-substrate-for-current-abox` do this for you. 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 4.1.4) defined for the corresponding TBox, as well as an *Query Repository (QBox)* (see Section 4.2.8) associated with this substrate. The queries in the QBox serve as an additional cache. All this information is automatically included in the dump file. 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 the *define* these queries (see Section 4.1.4). Restored definitions can be reused. See the example below.

We have also discussed in Section 4.1.7 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-load-kb "family")
; Loading text file /home/mi.wessel/family.racer
#P"/home/mi.wessel/family.racer"

> (enable-query-repository)
:OKAY-QUERY-REPOSITORY-ENABLED

> (retrieve (?x) (?x woman))
Concept (BROTHER) causes a cycle in TBox FAMILY.
Concept (SISTER) causes a cycle in TBox FAMILY.
Concept (PARENT) causes a cycle in TBox FAMILY.
(((?X EVE)) ((?X DORIS)) ((?X BETTY)) ((?X ALICE)))

> (defquery mother-with-son (?x ?y)
  (and (?x woman) (?x ?y has-child) (?y man)))
MOTHER-WITH-SON

> (show-current-qbox)
;;;
;;; QBOX FOR ABOX SMITH-FAMILY
;;;
;;; O: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-current-abox "test")
SMITH-FAMILY
```

The substrate can now, for example, be restored on a different RacerPro server:

```
> (full-reset)
:OKAY-FULL-RESET

> (restore-substrate "test")

SMITH-FAMILY

> (describe-all-definitions)

((DEFQUERY MOTHER-WITH-SON (?X ?Y)
  (AND (?X WOMAN) (?X ?Y HAS-CHILD) (?Y MAN)))))

> (current-abox)

SMITH-FAMILY

> (show-current-qbox)

;;;
;;; QBOX FOR ABOX SMITH-FAMILY
;;;
;;; NIL:MASTER-TOP-QUERY
;;; \___NIL:QUERY-1
;;; \___NIL:MASTER-BOTTOM-QUERY

:SEE-OUTPUT-ON-STDOUT

> (retrieve (?x ?y) (?x ?y mother-with-son))

(((?X ALICE) (?Y CHARLES)))

> (enable-query-repository)

:OKAY-QUERY-REPOSITORY-ENABLED

> (retrieve (?x) (?x mother))

(((?X ALICE) ((?X BETTY)))

> (show-current-qbox)

;;;
;;; QBOX FOR ABOX SMITH-FAMILY
;;;
;;; NIL:MASTER-TOP-QUERY
;;; \___NIL:QUERY-1
;;; \___2:QUERY-3
;;; \___NIL:MASTER-BOTTOM-QUERY

:SEE-OUTPUT-ON-STDOUT
```


Chapter 5

Additional Facilities

5.1 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 queries are answered without index structures at all (see also Section 3.9).

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.

5.2 Rule Formalism Extended to Support Abduction

In RacerPro, A-box rules can be specified that are applied w.r.t. the autoepistemic semantics. In a nutshell this means the following: if for the variables in the rule body (precondition), some bindings with individuals of the A-box can be found such that all instantiated query atoms of the body are entailed, then the assertion of the rule head (conclusion) is added to the A-box with variables instantiated appropriately (forward-chaining). All rules whose preconditions match the assertions of the A-box in the way just described are applied until nothing new can be added. The rule mechanism allows for a convenient augmentation of A-boxes with assertions. For details of the formalism see the nRQL User's Guide. An example for such a rule is the following statement.

```
(define-rule (?x ?y has-sibling)
  (and (?z ?x has-child)
```

```
(?z ?y has-child)
(?x human))
```

The form ($?x$ $?y$ has-sibling) is the head and the rest is the body.

Rules may also be specified as part of an OWL document in SWRL syntax (see Figure 5.1 for a graphical specification of SWRL rules in Protégé).

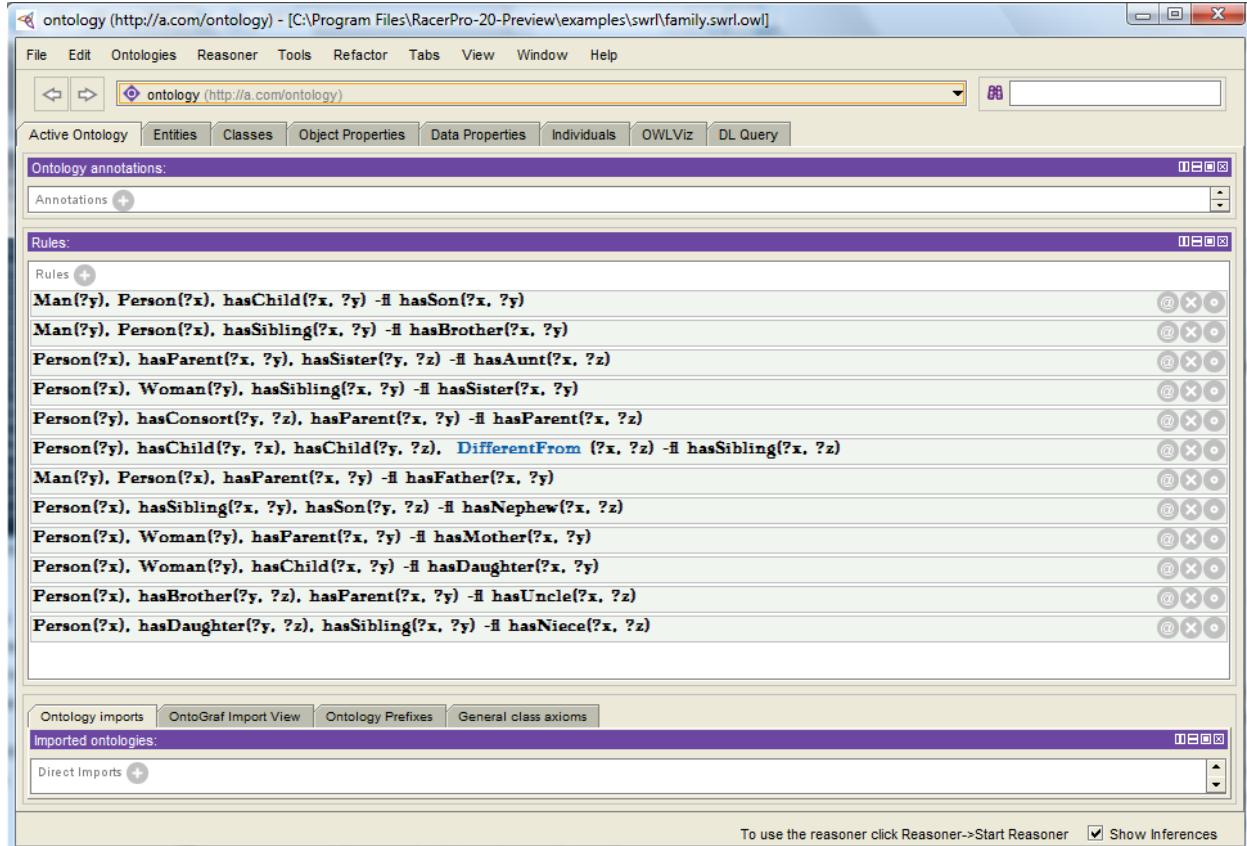


Figure 5.1: Rules in Protégé.

Rules are required because for some purposes, OWL is not expressive enough. Rather than just adding statements to the A-box whenever preconditions are satisfied as suggested above, in some cases it makes sense to just ask whether the head is entailed by an A-box (rules are applied in a backward-chaining way). This facility can also be used for computing what should be added to an A-box to make an assertion be entailed (abduction). Depending on the rule, this could involve the addition of new individuals to the A-box. In the following we present an example involving the detection of a pole-vault event.

```
(full-reset)

(in-knowledge-base test)

(define-primitive-role near)
(define-primitive-role has-part)

(disjoint pole bar)
(disjoint pv-in-start-phase pv-in-turn-phase)
```

```
(define-rule (?x ?y near)
  (and (?z ?x has-part)
    (?z ?y has-part)
    (?x pole)
    (?y human)
    (?z pole-vault)
    (?z pv-in-start-phase)))

(define-rule (?x ?y near)
  (and (?z ?x has-part)
    (?z ?y has-part)
    (?x bar)
    (?y human)
    (?z pole-vault)
    (?z pv-in-turn-phase)))

(instance p1 pole)
(instance h1 human)
```

Given the statements in this knowledge base one can ask the following query.

```
(retrieve-with-explanation () (p1 h1 near) :final-consistency-checking-p t)
```

The answer is

```
(t (((:tuple)
  (:new-inds IND-2)
  (:hypothesized-assertions
    (related IND-2 p1 has-part)
    (instance IND-2 pole-vault)
    (instance IND-2 pv-in-start-phase)
    (related IND-2 h1 has-part))))
```

The answer is **t** (as expected), and the result returned by RacerPro suggests that there is one way to achieve this answer. One new individual is required (**IND-2**). In addition, a set of hypothesized assertions to achieve the positive answer is indicated. Multiple ways to (positively) answer the query are possibly generated in other cases. If the optional keyword argument **:final-consistency-checking-p t** is used, RacerPro checks if the A-box would remain consistent if the hypothesized assertion were added to the A-box. The additional option **:incremental-consistency-checking-p t** might be more efficient in some cases (intermediate checks) but there is some overhead with multiple consistency checks.

Now, assume that there are some additional assertions added as in the following example.

```
(full-reset)

(in-knowledge-base test)

(define-primitive-role near)
(define-primitive-role has-part)

(disjoint pole bar)
```

```
(disjoint pv-in-start-phase pv-in-turn-phase)

(define-rule (?x ?y near)
  (and (?z ?x has-part)
    (?z ?y has-part)
    (?x pole)
    (?y human)
    (?z pole-vault)
    (?z pv-in-start-phase)))

(define-rule (?x ?y near)
  (and (?z ?x has-part)
    (?z ?y has-part)
    (?x bar)
    (?y human)
    (?z pole-vault)
    (?z pv-in-turn-phase)))

(instance p1 pole)
(instance h1 human)
(instance e1 pole-vault)
(instance e1 pv-in-start-phase)
(related e1 p1 has-part)
(related e1 h1 has-part)
```

Given the statements in this knowledge base one can ask the query from above again.

```
(retrieve-with-explanation () (p1 h1 near) :final-consistency-checking-p t)
```

The answer is now:

```
(t (((:tuple) (:new-inds) (:hypothesized-assertions))))
```

This means, **(p1 h1 near)** would be added if the rule was applied in a forward-chaining way, i.e., no hypothesized assertions are required. The **retrieve-with-explanation** facility can also be used for explaining to a user what must be added to an A-box in order to positively answer a query.

Racer Systems has applied the abduction operator to media interpretation problems (image interpretation and natural language text interpretation). We offer consulting to support industrial application projects that could use these facilities.

5.3 Reasoning with Triples in Secondary Memory

In practical applications, not all parts always require reasoning with expressive T-boxes (specified for instance in OWL ontologies). Indeed, for some purposes, classical data retrieval is just fine. However, in almost all practical applications there will be a large amount of data. Hence, data access must scale at least for the standard retrieval tasks. At the same time, it must be possible to apply expressive reasoning to (parts of) the data without producing copies of the data. For this purpose, we propose to use a triple store for RDF.

Using a triple store has several advantages. On the one hand, triples may be manipulated and retrieved from programs (Turing-complete representations), e.g. Java programs or Common Lisp

programs, without reasoning as usual in industrial applications. On the other hand, the same triples can be queried w.r.t. a background a background ontology. This involves reasoning and might result in additional, implicit triples to be found.

RacerPro allows for accessing existing AllegroGraph triple stores as well as for the creation of new ones. In the following example, a triple store is created, afterwards it is opened, and the triples are read into the knowledge base. nRQL queries are executed over the knowledge base. There is no need to write long-winded data extraction programs that move triples to OWL files on which, in turn, reasoning is then applied.

To create the triple store the following commands are used on the shell of RacerPorter, consider that AllegroGraph should be started before the following commands are executed:

```
(create-triple-store "test" :directory "C:/test-ts/" :if-exists :supersede)
(triple-store-read-file "C:/Program Files/RacerPro-20-Preview/examples/semweb/
                        lumb/university/university0-0.owl" :init nil :ignore-import t)
(index-all-triples)
(close-triple-store)
```

The commands above create a triple store on the provided directory, notice that the directory folder should be created in advanced. The content of the ontology “university0-0.owl” found in the RacerPro distribution is added to the triple store. The (`index-all-triples`) command is necessary to increase performance when working on secondary memory. Figure 5.2 shows the files created on the system containing the triple store. Now we can open the triple store and use it for querying on a knowledge base called “test-kb”.

```
(evaluate
  (let ((db (open-triple-store "test" :directory "C:/test-ts/")))
    (use-triple-store db :kb-name 'test-kb)))

(retrieve (?x ?y)
  (and (?y http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Publication)
       (?y ?x http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#publicationAuthor)
       (?x http://www.Department0.University0.edu http://www.lehigh.edu/%7Ezhp2/
           2004/0401/univ-bench.owl#memberOf)
       (?x http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#GraduateStudent)))

(retrieve (?x ?y ?z)
  (and (?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 http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#GraduateStudent)
       (?y http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#AssistantProfessor)
       (?z http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#GraduateCourse)))
```

The examples should illustrate the flavor of how triples can be accessed from RacerPro. Currently, for reasoning, the triples are loaded into main memory by RacerPro. Thus, only a limited number of triples should be in the store. In a future version, reasoning will be done also on secondary memory.

The result tuples can be selected in a separate presentation tab (Query IO) of the editor as illustrated in Figure 5.3. Afterwards, e.g., the corresponding inferred relational (role filler) A-box structure can be viewed using one of the graph panes of RacerPorter (see Figure 5.4).

As mentioned before, it is not always the case that reasoning is required. Therefore, one can pose the same nRQL queries to secondary memory for very fast access (but without reasoning). RacerPro optimizes the queries in order to provide good average-case performance.

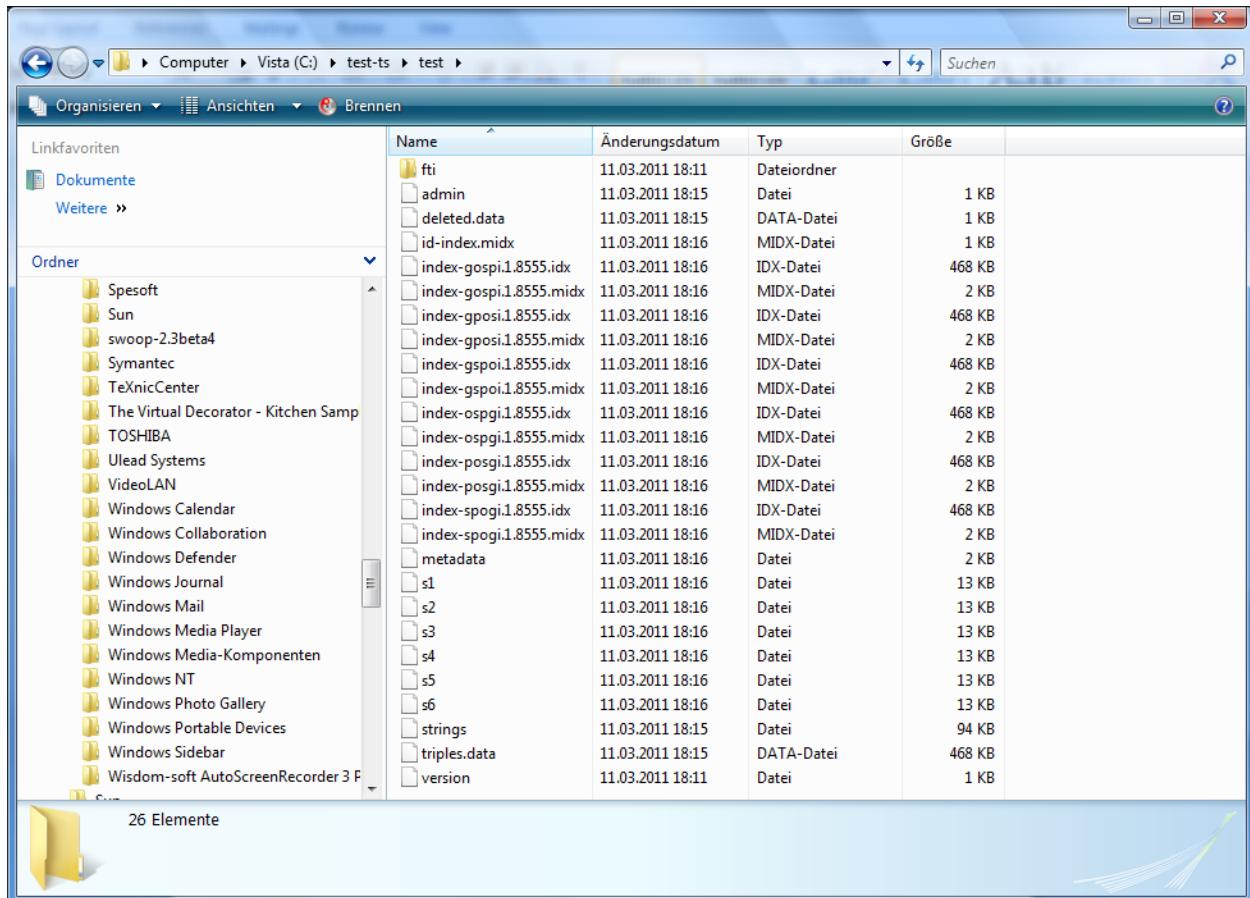


Figure 5.2: The triple store “test” created with RacerPorter.

```
(open-triple-store "test" :directory "C:/test-ts/")

(pretrieve (?x ?y)
  (and (?y http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Publication)
       (?y ?x http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#publicationAuthor)
       (?x http://www.Department0.University0.edu http://www.lehigh.edu/%7Ezhp2/
           2004/0401/univ-bench.owl#memberOf)
       (?x http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#GraduateStudent)))
```

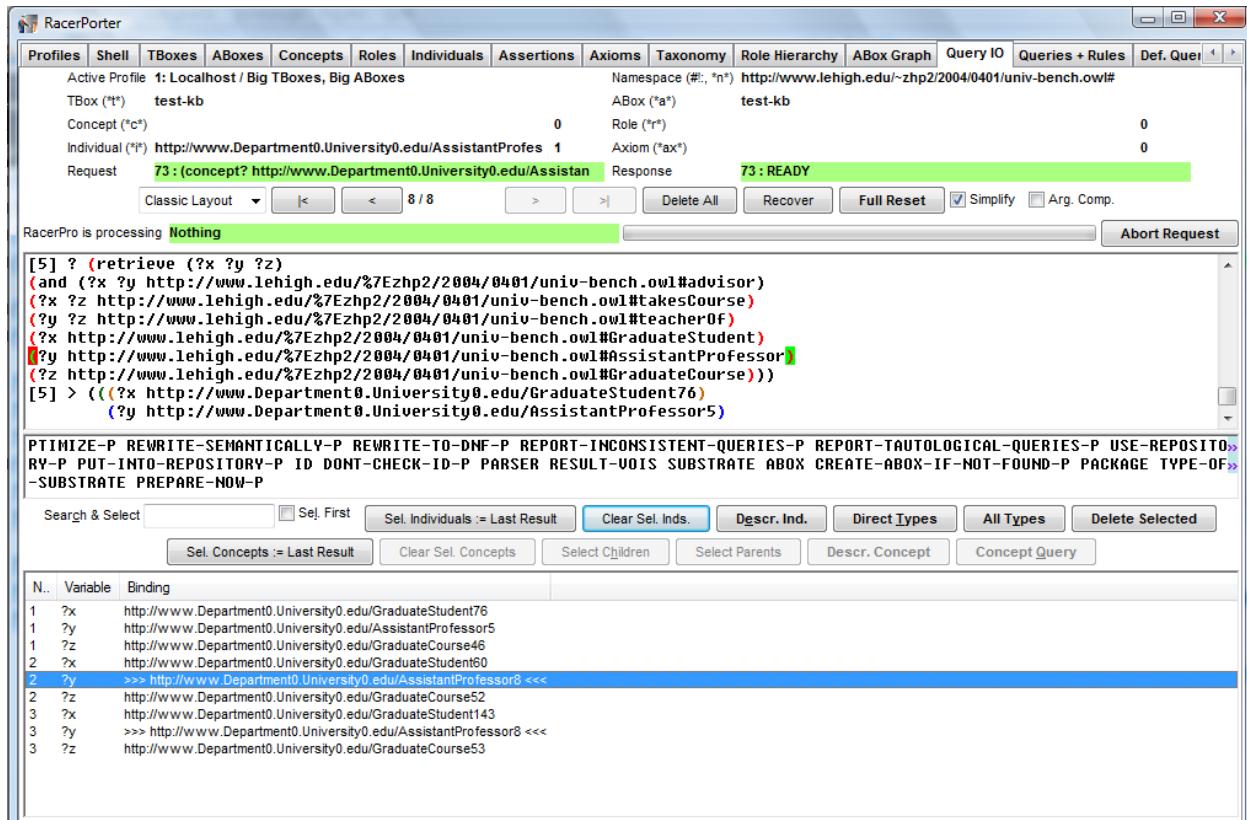


Figure 5.3: Query results

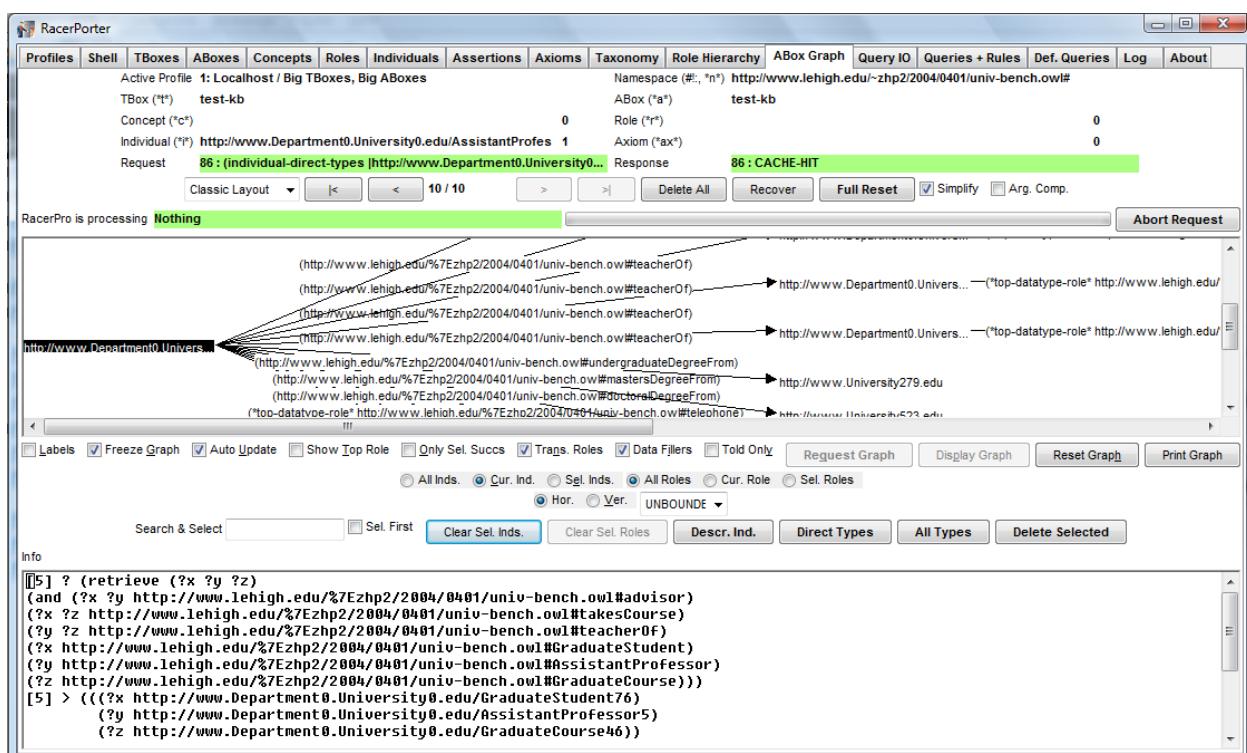


Figure 5.4: A-box Graph

```
(pretrieve (?x ?y ?z)
  (and (?x ?y http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#advisor)
    (?x ?z http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse)
    (?y ?z http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#teacherOf)
    (?x http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent)
    (?y http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#AssistantProfessor)
    (?z http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateCourse)))
  (close-triple-store))
```

In addition, it is possible to materialize in a triple store what can be computed by applying reasoning w.r.t. a background ontology such that later on the results are available to all applications which may or may not use reasoning. It is possible to optimized index data structures for subsequent query answering.

```
(evaluate
  (let ((db (open-triple-store "test" :directory "C:/test-ts/")))
    (use-triple-store db :kb-name 'test-kb :ignore-import t)
    (materialize-inferences 'test-kb :db db
      :directory "C:/test-ts/"
      :A-box t
      :index-p t)))
```

Ad-hoc Querying with SPARQL Queries can also be specified in the SPARQL language and can be executed with or without reasoning w.r.t. background ontologies.

```
(sparql-retrieve "
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?x, ?y
WHERE
  (?y rdf:type lubm:Publication)
  (?y lubm:publicationAuthor ?x)
  (?x lubm:memberOf "http://www.Department0.University0.edu" )
  (?x rdf:type lubm:GraduateStudent)
")
```

This is the SPARQL pendant to the first query in the example above.

Note that there is no need to store the OWL ontology in the triple store. With RacerPro you can access your existing triple store with various different OWL ontologies. An ontology can be put into a file or can possibly be retrieved from the Web. A triple store is then opened and queries are answered w.r.t. this triple store. The triple store corresponds to the A-box in this case.

5.4 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 are 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.

5.4.1 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)
(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.

```
(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.

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.

```
(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 and 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).

5.4.2 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 racer = new RacerServer("www.racer-systems.com",8088);
        String res;
        try {
            racer.openConnection();
            res = racer.send("(subscribe q_1 Book \"rm.sts.tu-harburg.de\" 8080)");
            racer.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.

5.5 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.² 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:

²In LRacer or RacerMaster you have to quote the symbol dinner.

```
(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. In RacerPorter you can abbreviate the display of OWL names with the check box "Simplify Names".

5.6 Explanation

The development of ontologies that use highly expressive constructs is by no means a trivial task, thus, the implicit consequences that a newly added axiom might cause to other axioms in the ontology are not obvious. For this reason, an explanation facility is helpful to support the ontology engineer in understanding which axioms of the ontology, both in the terminological (i.e., T-box part) as well as in the assertional part (i.e., A-box part) are causing incoherences.

RacerPro provides explanations for unsatisfiable concepts, for subsumption relationships (either unwanted or unexpected), and for unsatisfiable A-boxes. An example for an unsatisfiable A-box is shown in the following example.

```
(in-knowledge-base test)

(implies a b)
(equivalent c (some r a))
(equivalent d (some r b))

(instance i (and c (not d)))

(check-abox-coherence)
```

The lines below are obtained from the shell of the RacerPorter and show the result of the explanation facility. It is indicated which axioms in the T-box and which assertions in the A-box are the culprits for the inconsistency of the A-box. The explanation facilities are provided through the commands `(check-abox-coherence)`, `(check-tbox-coherence)` and `(check-ontology)`.

```
[1] ? (in-knowledge-base)
[1] > (NIL NIL)

[2] ? (implies a b)
[2] > :OKAY

[3] ? (equivalent c (some r a))
[3] > :OKAY

[4] ? (equivalent d (some r b))
[4] > :OKAY

[5] ? (instance i (and c (not d)))
[5] > :OKAY

[6] ? (check-abox-coherence)
```

```
[6] > (NIL
      ((i (and (not d) c))
       (:inference-step
        :assertion
        (i c)
        :axiom
        (equivalent c (some r a)))
       (:inference-step
        :assertion
        (i (not d))
        :axiom
        (equivalent (not d) (not (SOME r b))))))
```

One can use the operator (`retrieve-with-explanation`) in order to use the explanation facility for query answering, which is available in case a boolean query is answered with false but the expected answer is true.

5.7 Non-standard Inferences

RacerPro 2.0 provides an implementation of the LCS operator (least-common subsumer) for the description logic \mathcal{ALC} with unfoldable T-boxes. Furthermore, the MSC-k (most-specific concept up to k levels of nested existentials) is supported. In order to demonstrate the application of these operators, the following knowledge base is used.

```
(in-knowledge-base test)

(equivalent x (some r (and a d)))
(equivalent y (some r b))
(implies a b)
(implies c b)

(instance j a)
(instance i (all r (and (some r b) d)))
(related i j r)
(related j k r)
(related k i r)
```

We no apply the LCS operator to two input concepts.

```
(lcs-unfold x (and (all r d) (some r c)))
```

The result is (`some r (and d b)`). If no unfolding is desired, use `lcs` instead of `lcs-unfold`. It is possible to specify the T-box as an optional last argument (default is the current T-box).

For some purposes, an extract of the information contained in an A-box might be useful. RacerPro 2.0 provides an implementation of the MSC-k operator. MSC-k is applied to an individual and a nesting depth. For instance, given the knowledge base above, the following form is executed.

```
(msc-k i 3)
```

The result is

```
(and (some r (and a
               (some r (and (some r (and (all r (and (some r b) d))
                                         (some r (and a top))))))))))
```

It is possible to supply an additional boolean argument which indicates whether for each individual, the direct types are included in the MSC approximation. For instance,

```
(msc-k i 3 t)
```

returns

```
(and (some r
       (and x
             a
             (some r
                   (and a
                         d
                         y
                         (some r (and *top*
                                       (all r (and (some r b) d))
                                       (some r (and x a top))))))))))
```

The default for the third argument is `nil` (indicating that direct types are not to be included in the MSC-k output). Given this boolean argument is specified, there can be a fourth argument, the A-box (which defaults to the current A-box).

5.8 Event Recognition

In some applications it might be interesting to state that assertions are valid only within a certain time interval. This is not possible with standard description logic systems. There are several research results available on qualitative temporal reasoning in the context of description logics. RacerPro can support qualitative temporal reasoning as part of the nRQL system (compare the section on RCC substrates in the User's Guide). In addition, in some cases, quantitative information about time intervals might be available and could be relevant for query answering. RacerPro now also supports A-box query answering w.r.t. assertions that are associated with specifications for time intervals. The following example shows the main idea.

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

(define-primitive-role r :inverse r)

(implies car vehicle)
(implies volkswagen car)

(instance vw1 volkswagen)
(instance vw2 volkswagen)
(instance ralf pedestrian)
(related vw2 vw1 r)

(define-event-assertion ((move vw1) 7 80))
```

```
(define-event-assertion ((move vw2) 3 70))
(define-event-assertion ((move ralf) 3 70))
(define-event-assertion ((approach vw1 vw2) 10 30))
(define-event-assertion ((behind vw1 vw2) 10 30))
(define-event-assertion ((beside vw1 vw2) 30 40))
(define-event-assertion ((in-front-of vw1 vw2) 40 80))
(define-event-assertion ((recede vw1 vw2) 40 60))

(define-event-rule ((overtake ?obj1 ?obj2) ?t1 ?t2)
  ((?obj1 car) ?t0 ?tn)
  ((?obj1 ?obj2 r) ?t0 ?tn)
  ((move ?obj1) ?t0 ?t2)
  ((move ?obj2) ?t1 ?t2)
  ((approach ?obj1 ?obj2) ?t1 ?t3)
  ((behind ?obj1 ?obj2) ?t1 ?t3)
  ((beside ?obj1 ?obj2) ?t3 ?t4)
  ((in-front-of ?obj1 ?obj2) ?t4 ?t2)
  ((recede ?obj1 ?obj2) ?t4 ?t2))
```

The following query check whether there exists an overtaking event hidden in the A-box and the event assertions.

```
(timenet-retrieve ((overtake ?obj1 ?obj2) ?t1 ?t2))
```

The query returns a set of binding for variables if an event can be detected (and nil otherwise). For time variables an interval for the lower-bound and upper-bound are returned. Thus, the `overtake` event start at time unit 10 at the earliest and 29 at the latest. It ends at time unit 29 at the earliest and 60 at the latest. In a future version of RacerPro, redundant binding specifications will be removed.

```
(((?obj1 vw1) (?obj2 vw2) (?t1 (10 29)) (?t2 (41 60)))
 ((?obj1 vw1) (?obj2 vw2) (?t1 (10 29)) (?t2 (41 60))))
```

Note that the example involves reasoning. The fact that `vw1` is a car is only implicitly stated. In addition, temporal constraint have to be checked for the time intervals.

In the previous queries, variables are used. However, one might very well use constants in event queries as shown in the following example.

```
(timenet-retrieve ((overtake ?obj1 vw2) ?t1 ?t2))
```

Now, only bindings for the variable `?obj1` are returned.

```
(((?obj1 vw1) (?t1 (10 29)) (?t2 (41 60)))
 ((?obj1 vw1) (?t1 (10 29)) (?t2 (41 60))))
```

In a future version, a forward-chaining application of event rules will be supported, so there is no need to repeatedly cycle through all known events using respective queries. Racer Systems offers consulting for industrial partners who need event recognition w.r.t. expressive background ontologies.

Appendix A

Editions and Licenses

A.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.

But RacerPlus goes even further: RacerPlus has a built-in knowledge base editor that has direct access to the data structures of the integrated RacerPro! You can open, inspect, verify and edit your T-Boxes and A-Boxes in a user interface which instantly reflects the state of the embedded RacerPro server. Of course RacerPlus offers all the features of the integrated products: You can connect from RacerPlus to any RacerPro-server on the network or your data in RacerPlus can be accessed via client applications over the network. RacerPlus is available as a separate product positioned above the RacerPro kernel license.

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. We offer RacerMaster on request for Allegro Common Lisp, Lispworks, MCL, and other Lisp development environments.

A.2 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. We offer the following license schemes.

“Network” license Is the standard license of RacerPro. It allows you to run RacerPro on a single system (server) with an unlimited number of users from the network. This license covers the complete product including the RacerPorter. Available for 32bit versions of Windows, Linux or Mac OS X, for Sun and other branded UNIX workstations with 32bit or 64bit, as well as 64bit Linux environments. A Network license also includes one year of software updates and upgrades (see “Terms and Conditions” below). The 32-bit versions for Linux, Windows and Mac OS X are available for 2,990.00 Euro; all other versions start at 9,990.00 Euro.

“Desktop” license A desktop license allows one user to run RacerPro on a single machine, while all computation takes place on that local host. Usage from another machine via the network is prohibited. This license is ideal for evaluation in a commercial environment as well as for an academic researcher who wants to have a non time-limited license at hand. RacerPorter is included in this license. Available for 32bit versions of Windows, Linux or Mac OS X. No software maintenance is included or available for a Desktop license, however bug-fixes will be provided (see “Terms and Conditions” below). The license fee is 990.00 Euro; a volume discount is available starting with five licenses when purchased with the same order.

“Classroom” license If you are a teacher or trainer you probably want to have your personal license for your daily business, e.g., research, preparation of courses and the like. But you also have to have a license of RacerPro for each computer system used by the attendees of your seminars. Hence you have demand for a “classroom license” which covers your personal Desktop License as well as up to 15 additional systems typically used only for attendees of lectures. Available for 32bit versions of Windows, Linux or Mac OS X. A Classroom license also includes one year of software updates and upgrades (see “Terms and Conditions” below). The 32-bit versions for Linux, Windows and Mac OS X are available for 3,990.00 Euro.

“Site” license For environments where more desktops and/or servers will be installed we offer the site licensing of RacerPro applications. The licensing organization may use an arbitrary number of desktop or network servers in one building. If the licensee controls more than one building, one site license covers locations only within reasonable proximity to each other (given that they are located in the same state/country). Available for 32bit versions of Windows, Linux or Mac OS X, for Sun and other branded UNIX workstations with 32bit or 64bit, as well as 64bit Linux environments. A Site license also includes one year of software updates and upgrades (see “Terms and Conditions” below). The 32-bit versions for Linux, Windows and Mac OS X are available for 9,990.00 Euro; all other versions start at 25,000.00 Euro.

RacerPro on other platforms RacerPro is available for a wide variety of platforms. While the standard licensing schemes cover typical 32bit systems, we provide individual licensing options for other 32bit systems and also for 64bit large scale computing environments. Typically available for

Sun Sparc (Solaris) and AMD Opteron (Linux) systems. Please contact us for more information about other supported platforms. See above for prices for Network and Site Licenses for Sun and Linux. A site license for RacerPro for the target platform is necessary. More conditions and additional fees may apply. Please contact us for more information about the availability RacerPro on other platforms like IBM or HP.

Terms and Conditions All prices are in Euro and without taxes, shipping costs and expenses. Given prices are calculated for electronic delivery (download). Furthermore value added tax and other excise dues may be applicable to your individual purchase. If you are in doubt or in case need an official offer please contact us for a quote.

All active licenses are entitled to receive bug-fix releases of the licensed version as long as this version is the current release version.

Owners of Network, Classroom or Site licenses additionally get software updates free of charge during the first year after purchase (“software maintenance”). Following that period software updates are offered for an additional fee. Alternatively we offer software maintenance contracts for a yearly charge of about 25% of the initial license fee which will cover updates and upgrades within the second and subsequent years after purchase. Neither on-site service nor individual consulting or support are included in software maintenance.

Before ordering please check the intended operating system with us. Before the purchase we will provide access to the General Terms and Customer License Agreement documents.

Appendix B

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, [5], [13], [38], [42], [47], [48], [43], [45], [46], [40].@

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/>).

Special thanks go to Michael Hülsen, Britta Hummel, Thorsten Liebig, Marko Luther, and Ragnhild van der Straeten, for testing RacerPro in demanding applications. We also would like to thank Olaf Noppens for providing the Protégé plugin for RacerPro and to Ronald Cornet for providing an earlier interface for Racer called RICE.

The XML/RDF-based part of the input interface for RacerPro is implemented using the Wilbur Semantic Web Toolkit written by Ora Lassila. For more information on Wilbur and the Wilbur source code see <http://wilbur-rdf.sourceforge.net/>. RacerPorter is built with Lispworks Common Lisp.

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>.

Finally we would like to thank all customers and users that 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.

Bibliography

- [1] Allegrograph user guide. <http://www.franz.com/agraph/allegrograph/>.
- [2] F. Baader, A. Bauer, P. Baumgartner, A. Cregan, A. Gabaldon, K. Ji, K. Lee, D. Rajaratnam, and R. Schwitter. A novel architecture for situation awareness systems. In Martin Giese and Arild Waaler, editors, *Proceedings of the 18th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux 2009)*, volume 5607 of *Lecture Notes in Computer Science*, pages 77–92. Springer-Verlag, 2009.
- [3] F. Baader, S. Brandt, and C. Lutz. Pushing the \mathcal{EL} envelope. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence IJCAI-05, Edinburgh, UK*. Morgan-Kaufmann Publishers., 2005.
- [4] F. Baader, D. Calvanese, D. MacGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, Cambridge, UK, 2003.
- [5] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making \mathcal{KRIS} get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.
- [6] F. Baader and B. Hollunder. \mathcal{KRIS} : Knowledge Representation and Inference System. *SIGART Bull.*, 2(3):8–14, 1991.
- [7] 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.
- [8] S. Bechhofer, R. Möller, and P. Crowther. The DIG description logic interface. In *Proceedings of the International Workshop on Description Logics (DL-2003), Rome, Italy, September 5-7, 2003.*
- [9] A. Borgida. On the relationship between description logics and predicate logic queries. Technical Report LCSR-TR-295-A, Rutgers University, New Brunswick (NJ, USA), 1992.
- [10] A. Borgida. Description logics in data management. *IEEE Trans. on Knowledge and Data Engineering*, 7(5):671–682, 1995.
- [11] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. Alperin Resnick. CLASSIC: A structural data model for objects. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 59–67, 1989.
- [12] A. Borgida and P.F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *J. of Artificial Intelligence Research*, 1:277–308, 1994.
- [13] M. Buchheit, F.M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

- [14] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Eql-lite: effective first-order query processing in description logics. In *IJCAI'07: Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 274–279, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.
- [15] S. Castano, S. Espinosa, A. Ferrara, V. Karkaletsis, A. Kaya, R. Möller, S. Montanelli, G. Petasis, and M. Wessel. Multimedia interpretation for dynamic ontology evolution. *Journal of Logic and Computation*, 19(5):859–897, 2008.
- [16] A.K. Chandra. Theory of database queries. In *Proc. of the 7th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'88)*, pages 1–9, 1988.
- [17] S. Espinosa, A. Kaya, and R. Möller. Formalizing multimedia interpretation based on abduction over description logic Aboxes. In *Proc. of the 2009 International Workshop on Description Logics DL- 2009, 27 to 30 July 2009, Oxford, United Kingdom*, 2009. CEUR Workshop Proceedings (Vol. 477).
- [18] J. Faddoul and V. Haarslev. Algebraic tableau reasoning for the description logic \mathcal{SHOQ} . In *Journal of Applied Logic, Special Issue on Hybrid Logics*, page 31 pages, 2010. Accepted for publication.
- [19] J. Faddoul and V. Haarslev. Optimizing algebraic tableau reasoning for \mathcal{SHOQ} : First experimental results. In *Proceedings of the 2010 International Workshop on Description Logics (DL-2010), Waterloo, Canada, May 4-7*, pages 161–172., 2010.
- [20] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
- [21] J. W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81:183–198, 1996.
- [22] J. Gómez-Romero, J. Garca, M.A. Patricio, and J.M. Molina. Towards the implementation of an ontology-based reasoning system for visual information fusion. In *3rd Annual Skövde Workshop on Information Fusion Topics (SWIFT 2009), Skövde, Sweden: Oct.*, pages pp. 5–10, 2009.
- [23] V. Haarslev and R. Möller. Expressive ABox reasoning with number restrictions, role hierarchies, and transitively closed roles. In *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'2000)*, pages 273–284. Morgan Kaufmann, Los Altos, 2000.
- [24] V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In B. Nebel, editor, *Proceedings of Seventeenth International Joint Conference on Artificial Intelligence, IJCAI-01*, pages 161–166, 2001.
- [25] V. Haarslev and R. Möller. Optimizing reasoning in description logics with qualified number restrictions. In *Proceedings International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August*, pages 142–151, 2001.
- [26] V. Haarslev and R. Möller. Optimization strategies for instance retrieval. In *Proceedings of the International Workshop on Description Logics (DL-2002)*. CEUR Electronic Workshop Proceedings, <http://ceur-ws.org/Vol-53/>, 2002.
- [27] V. Haarslev and R. Möller. Incremental query answering for implementing document retrieval services. In *Proceedings of the International Workshop on Description Logics (DL-2003), Rome, Italy, September 5-7*, pages 85–94, 2003.

- [28] V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *Proc. of the 8th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'2004)*, 2004.
- [29] V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5*, pages 163–173, 2004.
- [30] V. Haarslev and R. Möller. The revival of structural subsumption in tableau-based description logic reasoners. In *Proceedings of the 2008 International Workshop on Description Logics (DL-2008)*, 2008.
- [31] V. Haarslev, R. Möller, and A.Y. Turhan. Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, pages 29–44. Springer-Verlag, 2001.
- [32] V. Haarslev, R. Möller, and M. Wessel. The description logic ALCNHR+ extended with concrete domains: A practically motivated approach. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, pages 29–44. Springer-Verlag, 2001.
- [33] V. Haarslev, R. Möller, and M. Wessel. The description logic 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.
- [34] V. Haarslev, R. Möller, and M. Wessel. Querying the semantic web with Racer + nRQL. In *Proceedings of the KI-2004 International Workshop on Applications of Description Logics (ADL'04), Ulm, Germany, September 24*, 2004.
- [35] V. Haarslev, R. Möller, and M. Wessel. Description logic inference technology: Lessons learned in the trenches. In I. Horrocks, U. Sattler, and F. Wolter, editors, *Proc. International Workshop on Description Logics*, 2005.
- [36] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In *Proc. of the 2nd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'91)*, pages 335–346, 1991.
- [37] M. Horridge and S. Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. In *OWLED 2009, 6th OWL Experienced and Directions Workshop, Chantilly, Virginia*, 2009.
- [38] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.
- [39] I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.
- [40] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible \mathcal{SROIQ} . In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67. AAAI Press, 2006.

- [41] I. Horrocks and U. Sattler. A tableaux decision procedure for \mathcal{SHIQ} . In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence IJCAI-05, Edinburgh, UK*, Edinburgh (UK), 2005. Morgan Kaufmann, Los Altos.
- [42] 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.
- [43] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic \mathcal{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.
- [44] I. Horrocks, U. Sattler, and St. Tobies. Reasoning with individuals for the description logic \mathcal{SHIQ} . In David McAllester, editor, *Proc. of the 17th Int. Conf. on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Computer Science*, pages 482–496. Springer-Verlag, 2000.
- [45] 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.
- [46] Ian Horrocks and Ulrike Sattler. Optimised reasoning for \mathcal{SHIQ} . In *Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002)*, pages 277–281, July 2002.
- [47] 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.
- [48] 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.
- [49] Michael Hülsen, J. Marius Zöllner, Nino Haeberlen, and Christian Weiss. Asynchronous real-time framework for knowledge-based intersection assistance. In *14th International IEEE Conference on Intelligent Transportation Systems, Washington, DC, USA. October 5-7,, 2011*.
- [50] Michael Hülsen, J. Marius Zöllner, and Christian Weiss. Traffic intersection situation description ontology for advanced driver assistance. In *2011 IEEE Intelligent Vehicles Symposium (IV), Baden-Baden, Germany, June 5-9, 2011*.
- [51] B. Hummel. *Description Logic for Scene Understanding at the example of Urban Road Intersections*. Südwestdeutscher Verlag für Hochschulschriften, 2010.
- [52] R.L.G. Lemmens. *Semantic interoperability of distributed geo-services*. PhD thesis, International Institute for Geo-Information Science and Earth Observations (CT), Delft, Netherlands Geodetic Commission NCG, 2006.
- [53] T. Liebig, M. Luther, O. Noppens, M. Rodrigues, D. Calvanesez, M. Wessel, R. Möller, M. Horridge, S. Bechhofer, D. Tsarkov, and E. Sirin. OWLlink: DIG for OWL 2. In *Fifth International Workshop on OWL: Experiences and Directions (OWLED08), Karlsruhe, Germany*, 2008.
- [54] R. Möller and B. Neumann. Ontology-based Reasoning Techniques for Multimedia Interpretation and Retrieval. In *Semantic Multimedia and Ontologies : Theory and Applications*. Springer, 2008.

- [55] O. Noppens. RACER reasoning support for OWLAPI and Protégé. <http://www.uni-ulm.de/in/ki/semantics/owltools>.
- [56] O. Noppens, M. Luther, and Th. Liebig. The OWLlink protocol: Infrastructure for interfacing and managing OWL 2 reasoning systems. In *OWLED 2009, 6th OWL Experienced and Directions Workshop, Chantilly, Virginia, October 2009*, volume 529 of *CEUR Workshop Proceedings*, 2009.
- [57] A. Pease and I. Niles. IEEE Standard Upper Ontology: A progress report. *Knowledge Engineering Review, Special Issue on Ontologies and Agents*, 17:65–70, 2002. <http://www.ontologyportal.org/>.
- [58] Evren Sirin. From wine to water: Optimizing description logic reasoning for nominals. In *In: Proceedings of KR-2006. (2006) 9099*, pages 90–99, 2006.
- [59] R. Van Der Straeten. *Inconsistency Management in Model-Driven Engineering: An Approach using Description Logics*. PhD thesis, Vrije Universiteit Brussel, 2005.
- [60] Sudoku in OWL. <http://www.mindswap.org/~aditkal/sudoku>.
- [61] D. Tsarkov, I. Horrocks, and P. F. Patel-Schneider. Optimizing terminological reasoning for expressive description logics. *J. of Automated Reasoning*, 39(3):277–316, 2007.
- [62] A.-Y. Turhan. *On the Computation of Common Subsumers in Description Logics*. PhD thesis, TU Dresden, Institute for Theoretical Computer Science, 2008.
- [63] M. Wessel. Solving Sudoku problems using Abox reasoning. <http://www.sts.tu-harburg.de/people/mi.wessel/papers/sudoku.racer>.
- [64] M. Wessel. *Flexible und konfigurierbare Software-Architekturen für datenintensive ontologiebasierte Informationssysteme*. PhD thesis, Hamburg University of Technology, 2008. ISBN 978-3-8325-2162-2.
- [65] M. Wessel, M. Luther, and R. Möller. What happened to bob? semantic data mining of context histories. In *Proc. of the 2009 International Workshop on Description Logics DL-2009, Oxford, United Kingdom*, volume 477 of *CEUR Workshop Proceedings*, 2009.
- [66] M. Wessel, M. Luther, and M. Wagner. The difference a day makes - recognizing important events in daily context logs. In *Proceedings of the International Workshop on Contexts and Ontologies: Representation and Reasoning (C&O:RR)*, volume 298 of *CEUR Workshop Proceedings*, 2007.
- [67] M. Wessel and R. Möller. A high performance semantic web query answering engine. In I. Horrocks, U. Sattler, and F. Wolter, editors, *Proc. International Workshop on Description Logics*, 2005.
- [68] M. Wessel and R. Möller. A flexible DL-based architecture for deductive information systems. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proc. IJCAR-06 Workshop on Empirically Successful Computerized Reasoning (ESCoR)*, pages 92–111, 2006.
- [69] M. Wessel and R. Möller. Design principles and realization techniques for user friendly, interactive, and scalable ontology browsing and inspection tools. In *International Workshop on OWL: Experiences and Directions (OWLED 2007)*, 2007.
- [70] J. Wu and V. Haarslev. Planning of axiom absorption. In *Proceedings of the 2008 International Workshop on Description Logics (DL-2008), Dresden, Germany*, 2008.

- [71] M. Zuo and V. Haarslev. High performance absorption algorithms for terminological reasoning. In *Proceedings of the 2006 International Workshop on Description Logics (DL-2006), Lake District, UK*, pages 159–166, 2006.

Index

Protégé, 55
RacerEditor, 45
RacerMaster, 205
RacerPlus, 205
RacerPorter, 11, 41, 205
RacerPro, 205
AND query, 109
HAS-KNOWN-SUCCESSOR query atom, 100
NEG query, 113
PROJECT-TO operator, 119
SAME-AS query atom, 99
UNION query, 110

A-box Graph tab, 53
A-boxes tab, 50
abduction, 185
ABox augmentation, 125
ABox modification, 125
active domain semantics, 86
APIs, 20
assertion, 69
Assertions tab, 51
associated ABox individual, 134
associated Abox individual, 134
associated substrate node, 134
Axioms tab, 51

binary query atom, 85
body projection operator, 119
boolean query, 118
bug reports, 37

cheap query, 166
cheap rule, 166
cheap tuple, 166
Classroom license, 206
closed-world assumption, 77
complex query, 108
complex TBox query, 129
concept axioms, 64
concept definition, 65
concept equation, 64

concept query atom, 86
concept term, 62
Concepts tab, 50
concrete domain restriction, 64
concrete domain value query, 101
concrete domains, 66
concurrent query processing, 159
conjunction of roles, 66
conjunctive query, 109
consistency of qualitative spatial networks, 143
constraint query atom, 94
creating individuals with a rule, 126

data representation, 133
data substrate, 133
data substrate edge, 133
data substrate edge query expression, 136
data substrate label, 133
data substrate node, 133
data substrate node query expression, 136
data substrate predicate, 136
data substrate query expression, 136
deadlock prevention, 168
Defined Queries tab, 54
defined query, 120
defined query and NAF, 122
defined query and PROJECT-TO, 122
Desktop license, 206
disjoint concepts, 64
disjunctive query, 110
domain restriction, 66

eager mode, 160
equal role, 91
event recognition, 203
exists restriction, 62
expensive query, 166
expensive rule, 166
expensive tuple, 166
explicit role filler query, 93
extended OWL query, 137, 141

feature, 65
feature chains in queries, 95
Functional Lambda Expressions, 146

GCI, 64
graphical client interfaces, 55

head projection operators, 101
hybrid ABox individual, 134
hybrid OWL query, 137, 141
hybrid query, 135, 138

hybrid representation, 133
implicit role filler query, 93
incomplete mode, 164
incremental mode, 157
incremental query processing, 159
individual, 85
Individuals tab, 51
inference modes, 78
injective variable, 85

JRacer, 20

lazy mode, 160
Log tab, 54
LRacer, 34

macro query, 120
marker, 140
mirror data substrate, 137
mirror of an ABox, 137
mirror substrate, 137
mirror substrate marker, 140
mirror substrate query, 138
mirroring an ABox, 137
mirroring an OWL file, 137
mirroring OWL documents, 141

NAF, 113
NAF in constraint query atom, 116
NAF in query with individuals, 117
NAF in SAME-AS query atom, 117
NAF query, 113
negation as failure, 113
Network license, 206
nominals, 75, 128
nominals in queries, 128
nRQL API, 157
nRQL EBNF, 150
nRQL engine, 157
nRQL grammar, 150
nRQL mode, 164
nRQL mode 0, 164
nRQL mode 1, 165
nRQL mode 2, 165
nRQL mode 3, 165
nRQL mode 4, 165
nRQL mode 5, 165
nRQL mode 6, 165
nRQL modes, 157, 165
nRQL persistency, 181
nRQL syntax, 150
number restriction, 63

object, 85
 open-world assumption, 77
 optimization strategies, 78
 options, 35
 OWL, 199
 OWL annotation property filler query, 105
 OWL annotation property value query, 105
 OWL constraint checking, 97
 OWL constraint query atom, 97
 OWL datatype property filler query, 105
 OWL datatype property value query, 105
 OWL instances query, 87
 OWL mirror, 137, 141
 OWL object property filler query, 92
 OWL query, 87, 92, 97, 105, 141
 predicate, 136
 predicate query, 136
 preferences, 44
 primitive concept, 65
 Profiles tab, 47
 projection, 119
 projection operator, 119
 pseudo-nominal, 128
 publish-subscribe mechanism, 193
 QBox, 177
 qualitative spatial reasoning, 143
 Queries + Rules tab, 54
 query API, 157
 query atom, 86
 query body, 86
 query cache, 177
 query conjunct, 109
 query consistency, 171
 query disjunct, 110
 query engine, 157
 query entailment, 173
 query head, 86, 101
 query inconsistency, 171
 query inference, 171
 Query IO tab, 54
 query life cycle, 160
 query predicate, 136
 query realization, 180
 query realizer, 180
 query reasoning, 171
 query repository, 177
 query subsumption, 173
 querying qualitative spatial networks, 143
 range restriction, 66
 RCC consistency, 143

RCC constraint checking, 144
RCC query, 144
RCC substrate, 143
RDF edge query, 92
RDF instance query, 87
RDF node query, 87
Reasoning in Secondary Memory, 188
referencing a data substrate node, 136
region connection calculus, 143
retraction, 185
retrieving concrete domain values, 101
role chains in queries, 95
role hierarchy, 65
Role Hierarchy tab, 53
role query atom, 88
role query atom with features, 91
role query atom with negated role, 89
Roles tab, 51
rule, 125
rule antecedent, 125
rule application, 125, 161
rule application strategy, 161
rule consequence, 125
rule engine, 125, 157
rule firing, 161
rule life cycle, 161
rule postcondition, 125
rule precondition, 125
rule strategy, 161
rules and pseudo-nominals, 128
rules and the concrete domain, 127

scalable query answering, 164
semantic cache, 177
semantic web, 5
semantically identical individuals, 91
semi-structured data, 133
services, 6, 7
set at a time mode, 157
Shell tab, 48
signature, 62
Site license, 206
socket interface, 20
spatial query, 144
spatial reasoning, 143
spatial reasoning substrate, 143
spatio-thematic query, 144
stored defined query, 181
stored QBox, 181
stored substrate, 181
substrate, 133
substrate layer, 133

substrate persistency, 181
substrate query, 135
SWRL, 71

T-boxes tab, 50
Taxonomy tab, 52
TBox query, 129
Terms and Conditions, 207
told value query, 101
transitive role, 65
tuple at a time mode, 157
two-phase query processing, 166

unary query atom, 85
unique name assumption, 70, 78

value restriction, 62
variable, 85

warning token, 166