RACER User's Guide and Reference Manual
Version 1.7.19

Volker Haarslev* and **Ralf Möller****
with contributions from **Michael Wessel*****

*Concordia University
Computer Science Department
1455 de Maisonneuve Blvd. W.
Montreal, Quebec H3G 1M8, Canada
haarslev@cs.concordia.ca

**Techn. Univ. Hamburg-Harburg
Inform. and Commun. Dept.
Harburger Schloßstraße 20
21079 Hamburg, Germany
r.f.moeller@tu-harburg.de

***University of Hamburg
Computer Science Department
Vogt-Kölln-Straße 30
22527 Hamburg, Germany
mwessel@informatik-uni-hamburg.de

April 26, 2004

Contents

1 Introduction

1.1 Features

The RACER[1] system is a knowledge representation system that implements a highly optimized tableau calculus for a very expressive description logic. It offers reasoning services for multiple TBoxes and for multiple ABoxes as well. The system implements the description logic $\mathcal{ALCQHI}_{R^+}$ also known as $\mathcal{SHIQ}$ (see [Horrocks et al. 2000]). This is the basic logic $\mathcal{ALC}$ augmented with qualifying number restrictions, role hierarchies, inverse roles, and transitive roles. In addition to these basic features, RACER also provides facilities for algebraic reasoning including concrete domains for dealing with:

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

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

RACER implements the HTTP-based quasi-standard DIG for interconnecting DL systems with interfaces and applications using an XML-based protocol [Bechhofer 02]. RACER also implements most of the functions specified in the older Knowledge Representation System Specification (KRSS), for details see [Patel-Schneider and Swartout 93].

RACER has been initially developed at the University of Hamburg, Germany. RACER is actively supported and future releases are developed at Concordia University in Montreal, Canada, and at the University of Applied Sciences in Wedel near Hamburg, Germany.

Given a TBox, 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 TBox: Is the set of objects described by a concept empty?
- Concept subsumption w.r.t. a TBox: Is there a subset relationship between the set of objects described by two concepts?
- Find all inconsistent concepts mentioned in a TBox. Inconsistent concepts might be the result of modeling errors.
- Determine the parents and children of a concept w.r.t. a TBox: The parents of a concept are the most specific concept names mentioned in a TBox which subsume the concept. The children of a concept are the most general concept names mentioned in a

[1] RACER stands for **R**enamed**A**Box and **C**oncept **E**xpression **R**easoner

TBox that the concept subsumes. Considering all concept names in a TBox the parent (or children) relation defines a graph structure which is often referred to as taxonomy. Note that some authors use the name taxonomy as a synonym for ontology.

Note that whenever a concept is needed as an argument for a query, not only predefined names are possible. If also an ABox is given, among others, the following types of queries are possible:

- Check the consistency of an ABox w.r.t. a TBox: Are the restrictions given in an ABox w.r.t. a TBox too strong, i.e., do they contradict each other? Other queries are only possible w.r.t. consistent ABoxes.
- Instance testing w.r.t. an ABox and a TBox: Is the object for which an individual stands a member of the set of objects described by a certain query concept? The individual is then called an instance of the query concept.
- Instance retrieval w.r.t. an ABox and a TBox: Find all individuals from an ABox 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.
- Computation of the direct types of an individual w.r.t. an ABox and a TBox: Find the most specific concept names from a TBox of which a given individual is an instance.
- Computation of the fillers of a role with reference to an individual.
- Check if certain concrete domains constraints are entailed by an ABox and a TBox.

1.2 New Features in Version 1.7.7

Among others, version 1.7 of the RACER system offers the following new features.

- Support for the DIG standard such that, for instance, graphical ontology editors such as OilEd [Bechhofer et al. 01] can be used with RACER as a reasoning engine.
- RACER 1.7 can directly read knowledge bases specified w.r.t. the OWL, DAML+OIL, RDFS or RDF standard (although there are some restrictions on some OWL and DAML+OIL language expressions).
- RACER 1.7.7 comes with an interactive graphical shell called RICE. RICE was developed by Ronald Cornet, Univ. of Amsterdam.
- Many more query functions for information about processed TBoxes (e.g., for retrieving the definition of a concept, the meta constraints, the description logic language actually used in a knowledge base, etc.).
- Modifications of TBoxes are possible (i.e., retransmission of a changed TBox to the server is no longer necessary).
- Dramatically improved performance of ABox query processing.

- Reasoning effort for answering ABox queries can be controlled (e.g., users can specifiy whether computing an index is desired or whether query subsumption should be exploited).
- RACER 1.7 offers a publish-subscribe service for subscribing instance retrieval queries (see below for details). Applications of this feature are, for instance, document management systems.
- Persistency management for TBoxes and ABoxes is now offerered. Restarting a DL inference server is now up to 10 times faster.
- Multi-user access is possible with the Racer Proxy.
- New concrete domains for cardinals (linear ineqations with order constraints and integer coefficients), complex numbers (nonlinear multivariate inequations with integer coefficients), and strings (equality and inequality) are supported.

1.3 Application Areas

There are numerous paper describing how RACER can be used to solve application problems (see the Workshops on description logics and the workshops on applications of description logics ADL-01, ADL-02). Without completeness one can summarize that applications come from the following areas:

- Semantic Web
- Electronic Business
- Medicine/Bioinformatics
- Natural Language Processing
- Knowledge-Based Vision
- Process Engineering
- Knowledge Engineering
- Software Engineering

1.4 About this Document

Developing a DL system is by no means a trivial task. However, due to our experiences, writing a usable User's Guide for a DL system might be even more complex. Writing this manual took much time. We hope that you can make use of this manual, although we know that there may be deficiencies. If you have questions do not hesitate to ask, if you find a bug send a note such that we can reproduce the problem, or send us emails if you have any suggestions.

1.5 Acknowledgments

RACER has been developed with Macintosh Common Lisp from Digitool Inc. The RACER server for Linux and Window is implemented with Lispworks from Xanalys Inc. Testing was partly performed with Allegro Common Lisp from Franz Inc.

The XML-based part of the input interface for RACER is implemented using the XML/RDF/RDFS/DAML parser Wilbur written by Ora Lassila. For more information on Wilbur see http://wilbur-rdf.sourceforge.net/. In addition, the HTTP server for the DIG interface of RACER is implemented with CL-HTTP which is developed and owned by John C. Mallery. For more information on CL-HTTP see http://www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html.

Many users have directly contributed to the functionality and stability of the RACER system by giving comments, providing ideas and test knowledge bases, implementing interfaces, or sending bug reports. In alphabetical order we would like to mention Jordi Alvarez, Carlos Areces, Franz Baader, Sean Bechhofer, Daniela Berardi, Ronald Cornet, Maarten de Rijke, Michael Eisfeld, Enrico Franconi, Günther Görz, Ian Horrocks, Alexander Huber, Sebastian, Hübner, Thomas Kleemann, Alexander Koller, Christian Lorenz, Bernd Ludwig, Carsten Lutz, Maarten Marx, Jean-Luc Metzger, Bernd Neumann, Hans-Jürgen Ohlbach, Peter Patel-Schneider, Peter Reiss, Stefan Schlobach, Michael Sintek, Kristina Striegnitz, Sergio Tessaris, Martina Timmann, Stephan Tobies, Anni-Yasmin Turhan, Mike Ushold, Ragnhild Van Der Straeten, Michael Wessel, and Cai Ziegler. We apologize if somebody who should be mentioned is not included in this list.

2 Obtaining and Running RACER

The RACER system can be obtained from the following web sites:

Europe:

http://www.fh-wedel.de/~mo/racer/

America:

http://www.cs.concordia.ca/~faculty/haarslev/racer/

2.1 System Requirements

On the one hand RACER is available as a standalone version with no additional licences required. Throughout this manual, the standalone version is also called RACER executable or RACER Server. The RACER server can be started from an operating system shell or by double-clicking the program icon. Based on either TCP sockets of HTTP streams, clients can connect to the RACER Server via a remote interface (an example client written in Java is provided with the RACER distribtion). In addition, the RACER executable can be used to process files (see below for a detailed introduction on how to use RACER). On the other hand RACER is available as a fasl file for most Common Lisp and operating systems (Linux, Macintosh, Solaris, Windows).

2.2 System Installation

Go to the RACER download page found at the URLs specied above. Download the les for the environment of your preference and use an appropriate program to unpack the archive (StuffIt Expander for Macintosh, WinZip for Windows and gzip/tar for Unix). The examples are provided as an additional archive that can be downloaded. Since you already read this manual you probably already downloaded it. New versions can be downloaded from the site specied above. In this manual the directory where you installed RACER is referred to by the name RACER:. We assume that you install the examples in the directory "RACER:Examples;"

2.3 Sample Session

In this section we present a first example for the use of description logics. We use the Lisp interface here in order to directly present the results of queries. Note, however, that all examples can be processed in a similar way with the RACER Server (and the file interface). The file `"family.racer"` contains the TBox and ABox introduced in this section. The queries are in the file `"family-queries.lisp"`. If you use the RACER executable just type `racer -f family.racer -q family-queries.lisp` in order to see the results (see Section 2.5 for details on how to use RACER from a shell).

```
;;;============================================================
;;;  the following forms are assumed to be contained in a
;;;  file "RACER:examples;family-tbox.krss".

;;; initialize the TBox "family"
(in-tbox family)


;;; supply the signature for this TBox
(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)))

;;; 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)))
```

```
;;; the concepts
(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)))
```

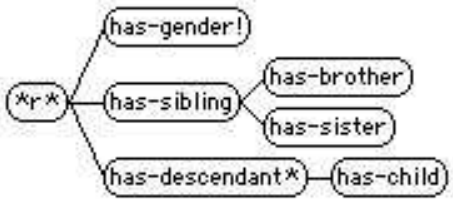

Figure 1: Role hierarchy for the `family` TBox.
r denotes the internally defined universal role.
! denotes features
∗ denotes transitive roles

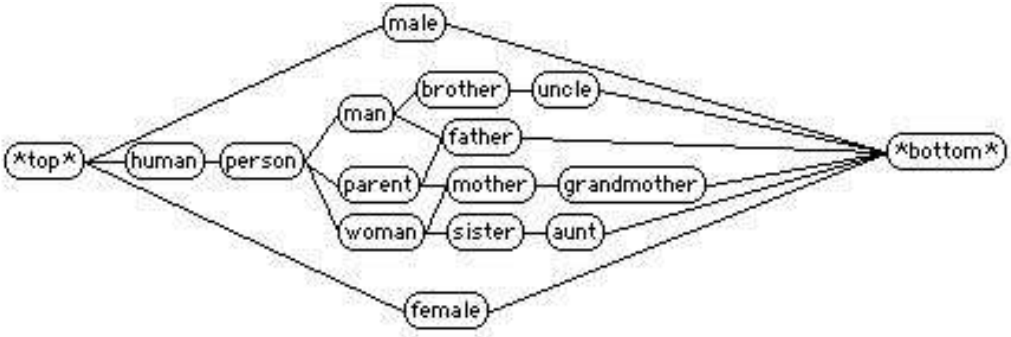

Figure 2: Concept hierarchy for the `family` TBox.

The RACER Session:

```
;;; load the TBox
CL-USER(1): (load "RACER:examples;family-tbox.krss")
;;; Loading RACER:examples;family-tbox.krss
T
;;;   some TBox queries
;;; are all uncles brothers?
CL-USER(2): (concept-subsumes? brother uncle)
T
;;; get all super-concepts of the concept mother
;;;  (This kind of query yields a list of so-called name sets
;;;   which are lists of equivalent atomic concepts.)
CL-USER(3): (concept-ancestors mother)
((PARENT) (WOMAN) (PERSON) (*TOP* TOP) (HUMAN))
;;; get all sub-concepts of the concept man
CL-USER(4): (concept-descendants man)
((UNCLE) (*BOTTOM* BOTTOM) (BROTHER) (FATHER))
;;; get all transitive roles in the TBox family
CL-USER(5): (all-transitive-roles)
(HAS-DESCENDANT)


;;;=============================================================
;;;  the following forms are assumed to be contained in a
;;;  file "RACER:examples;family-abox.krss".

;;; initialize the ABox smith-family and use the TBox family
(in-abox smith-family family)

;;; supply the signature for this ABox
(signature :individuals (alice betty charles doris eve))



;;; Alice is the mother of Betty and Charles
(instance alice mother)
(related alice betty has-child)
(related alice charles has-child)

;;; Betty is mother of Doris and Eve
(instance betty mother)
(related betty doris has-child)
(related betty eve has-child)

;;; Charles is the brother of Betty (and only Betty)
(instance charles brother)
(related charles betty has-sibling)
```

```
;;; closing the role has-sibling for Charles
(instance charles (at-most 1 has-sibling))

;;; Doris has the sister Eve
(related doris eve has-sister)

;;; Eve has the sister Doris
(related eve doris has-sister)
```

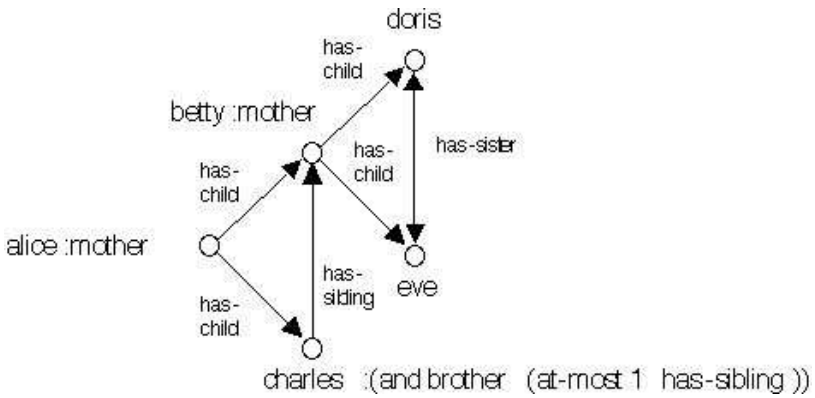

Figure 3: Depiction of the ABox `smith-family`.
(with explicitly given information being shown)

The RACER Session:

```
;;; now load the ABox
CL-USER(6): (load "RACER:examples;family-abox.krss")
;;; Loading RACER:examples;family-abox.krss
T
```

```
;;;   some ABox queries
;;; Is Doris a woman?
CL-USER(7): (individual-instance? doris woman)
T
;;; Of which concepts is Eve an instance?
CL-USER(8): (individual-types eve)
((SISTER) (WOMAN) (PERSON) (HUMAN) (*TOP* TOP))
;;; get all direct types of eve
CL-USER(9): (individual-direct-types eve)
(SISTER)
;;; get all descendants of Alice
CL-USER(10): (individual-fillers alice has-descendant)
(DORIS EVE CHARLES BETTY)
;;; get all instances of the concept sister
CL-USER(11): (concept-instances sister)
(DORIS BETTY EVE)
```

In the Appendix different versions of this knowledge base can be found. In Appendix A, on page 179, you find a version where the TBox and ABox are integrated. All example files and some additional ones (see the appendix) can also be found in the directory `"RACER:examples;"`.

2.4 Open-World Assumption and Unique Name Assumption

As other description logic systems, RACER 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 TBox and ABox of the previous subsection, a standard pitfall would be to think that RACER is wrong considering its answer to the following query:

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

RACER answers NIL. However, NIL does not means NO but just "cannot be proven w.r.t. the information given to RACER". 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 is an assertion (related alice william has-child) added to the ABox 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 ABox. Given this, the ABox will become inconsistent if another invidividual (e.g., william)) is declared to be a child of alice. Many users askedfor a switch such that RACER automatically closes roles. However, this problem is ill-dened. 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 closedthe role r for the individual i. Just determining the number of llers of r w.r.t. i and adding a corresponding assertion `(<= 1 r)` to the ABox is a bad idea because the ABox gets inconsistent. Due to the TBox, the minimum number of llers is 3. But, should we add `(at-most 1 r)` or `(at-most 1 r (and a b c))`? The rst 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 qualier concepts used in qualied number restrictions. In addition to the Open World Assumption, RACER also employs the Unique Name Assumption (UNA). This means that all individuals used in an ABox 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 ABox 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).

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 Racer, users can achieve a local closed-world (LCW) assumption (see Section 12).

2.5 The RACER Server

The RACER Server is an executable file available for Linux and Windows. It can be started from a shell or by double-clicking the correspoding program icon in a graphics-based environment. For instance, the Windows version is shown in Figure 4.

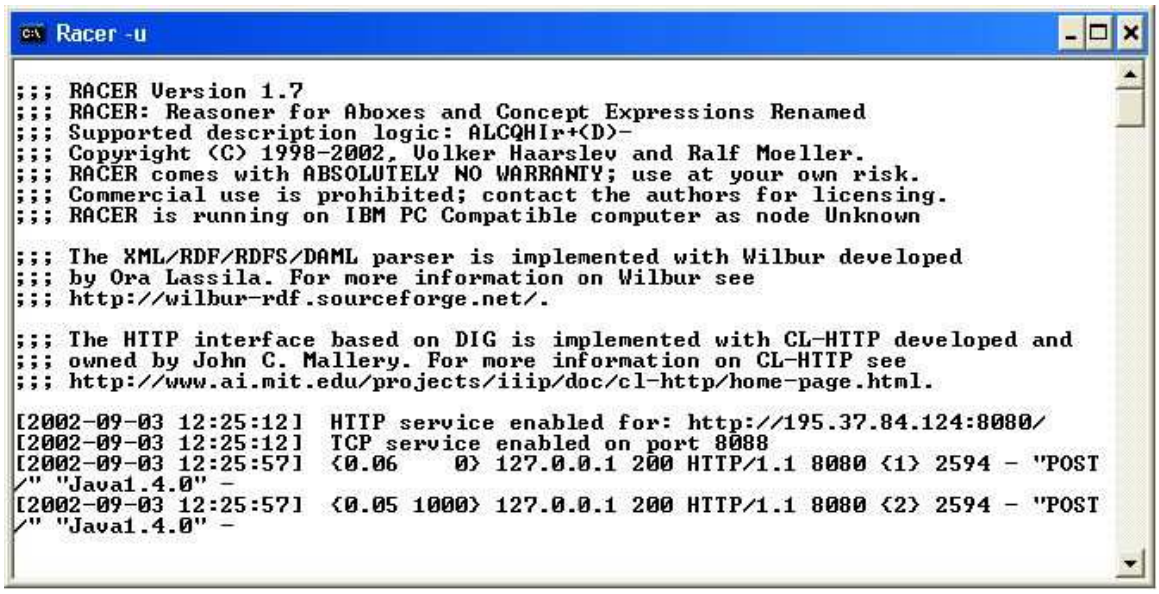

Figure 4: A screenshot of the RACER Server started under Windows.

Depending on the arguments provided at startup, the RACER executable supports different modes of operation. It offers a file-based interface, a socket-based TCP stream interface, and a HTTP-based stream interface.

2.5.1 The File Interface

If your knowledge bases and queries are available as files use the file interface of RACER, i.e. start RACER with the option `-f`. In your favorite shell just type:

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

The input file is `family.racer` and the queries file is `family-queries.lisp`. The output of RACER 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:

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

The syntax for processing input files is determined by RACER using the file type (file extension). If "`.lisp`", "`.krss`", or "`.racer`" is specified, a KRSS-based syntax is used. Other possibilities are "`.rdfs`", "`.daml`", 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`, `-daml`, and `-dig`.

The option `-xml <filename>` is provided for historical reasons. The input syntax is the older XML syntax for description logic systems. This syntax was developed for the FaCT system [Horrocks 98]. In the RACER Server, output for query results based on this syntax is also printed using an XML-based syntax. However, the old XML syntax of FaCT is now superseded by the DIG standard. Therefore, the RACER option -xml may no longer be supported once the file interface fully supports the DIG standard for queries (see above).

Currently, the file interface of RACER supports only queries given in KRSS syntax. However, DIG-based queries [Bechhofer 02] can be specified indirectly with the file interface as well. Let us assume a DIG knowledge base is given in the file `kb.xml` and corresponding DIG queries are specified in the file `queries.xml`. In order to submit this file to RACER just create a file `q.racer`, say, with contents `(dig-read-file "queries.xml")` and start RACER as follows:

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

Note the use of the option -dig for specifying that the input knowledge base is in DIG syntax. Since the le extension for the knowledge base is .xml, the option -f wouldassume the older XML syntax for knowledge bases (see above). If the queries le has the extensions .xml RACER assumes DIG syntax. For older programs this kind of backward compatibility is needed.

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.

2.5.2 TCP Socket Interface: JRACER

The socket interface of the RACER Server can be used from application programs (or graphical interfaces). If the option `-f` is not provided, the socket interface is automatically enabled. Just execute the following.

```
$ racer
```

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

```
$ racer -p 8000
```

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. An example client implemented in the Java programming language is provided with source code with the RACER distribtution. The client is based on the JRACER library developed by Jordi Alvarez. See the folder `RACER:jracer;`. 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 RACER. JRACER provides a Java layer for accessing the services of RACER using methods.

The current version of the TCP socket interface does not support logging.

The following code fragment explains how to send message to a Racer Server running at `racer.fh-wedel.de` under prot 8088.

```
public class MyProgram {
   public static void main(String[] argv) {
      RacerSocketClient client = new RacerClient("racer.fh-wedel.de", 8088);
      try {
         client.openConnection();
         try {
            String result =
               client.send
                     ("(concept-instances person)");

            ...
            }
         catch (RacerException e) {
            ...
            }
         }
         client.closeConnection();
      } catch (IOException e) {
         ...
      }
   }
}
```

2.5.3 HTTP Interface: DIG Interface

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

Clients can connect to the HTTP based RACER Server using the POST method. For details see the DIG standard [Bechhofer 02]. The default HTTP communication port used by RACER is 8080. In order to change the port number, the RACER Server should be started with the option `-http`. For instance:

```
$ racer -http 8000
```

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

A document describing the XML syntax of the DIG standard is provided with this distribution. See the folder `racer:doc;`. The DIG standard as it is defined now is just a first step towards a communication standard for connecting applications to DL systems. RACER provides many more features that are not yet standardized. These features are offered only over the TCP socket interface. However, applications using RACER can be developed with DIG as a starting point. If other facilities of RACER are to be used, the raw TCP interface of RACER can be used in a seemingless way to access a knowledge base declared with the DIG interface. If, later on, the standardization process make progress, users should be able to easily adapt their code to use the HTTP-based DIG interface.

In some applications, the RACER Server is to be addressed with persistent connections. The RACER Server provides a timeout in order to control the duration of these connections. With the option -c a timeout in seconds can be specied (the default is 10).

```
$ RACER -c 200
```

2.5.4 Additional Options for the RACER Server

Sometimes it happens that the default stack space is too small for processing certain queries. The stack space can be enlarged by starting RACER with the option `-s <value>`. The initial value is 320000.

Processing knowledge bases in a distributed system can cause security problems. The RACER Server executes statements as described in the sections below. The only 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 RACER Server. If you would like your RACER Server to support these features, startup RACER with the option `-u` (for unsafe).

If RACER 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 RACER before clients can connect.

Sometimes using the option `-verbose` is useful for debugging purposes. Use the option `-h` to get the list of possible options and a short explanation.

The option -n allows for removing the prex of the default namespace as dened for DAML les. See Chapter 4 for details.

2.6 Graphical Client Interfaces

In this section we present graphical client interfaces for the RACER Server. The examples require that RACER be started with the option `-u`. The first interface is the RICE system, which comes with the RACER distribution. Afterwards, a short presentation of the OilEd system [Bechhofer et al. 01] with RACER as a backend reasoner is given. Then, the coordinated use of OilEd and RICE is sketched.

2.6.1 RICE

RICE is an acronym for RACER Interactive Client Environment and has been developed by Ronald Cornet from the Academic Medical Center in Amsterdam. RICE

is provided with source code and uses the JRACER system (see above). The executable version is provided as a jar file. Newer versions of RICE can be found at http://www.b1g-systems.com/ronald/rice.

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

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

Then, either double-click the jar file icon or type `java -jar rice.jar` into a shell. Connect RICE to RACER by selecting Connect from the Tools menu. In addition to the default TBox (named `default`) always provided by RACER, the TBox "`FAMILY`" is displayed in the left upper window which displays the parent-children relationship between concept names of different TBoxes. An example screenshot is shown in Figure 5. Users can interactively unfold and fold the tree display.

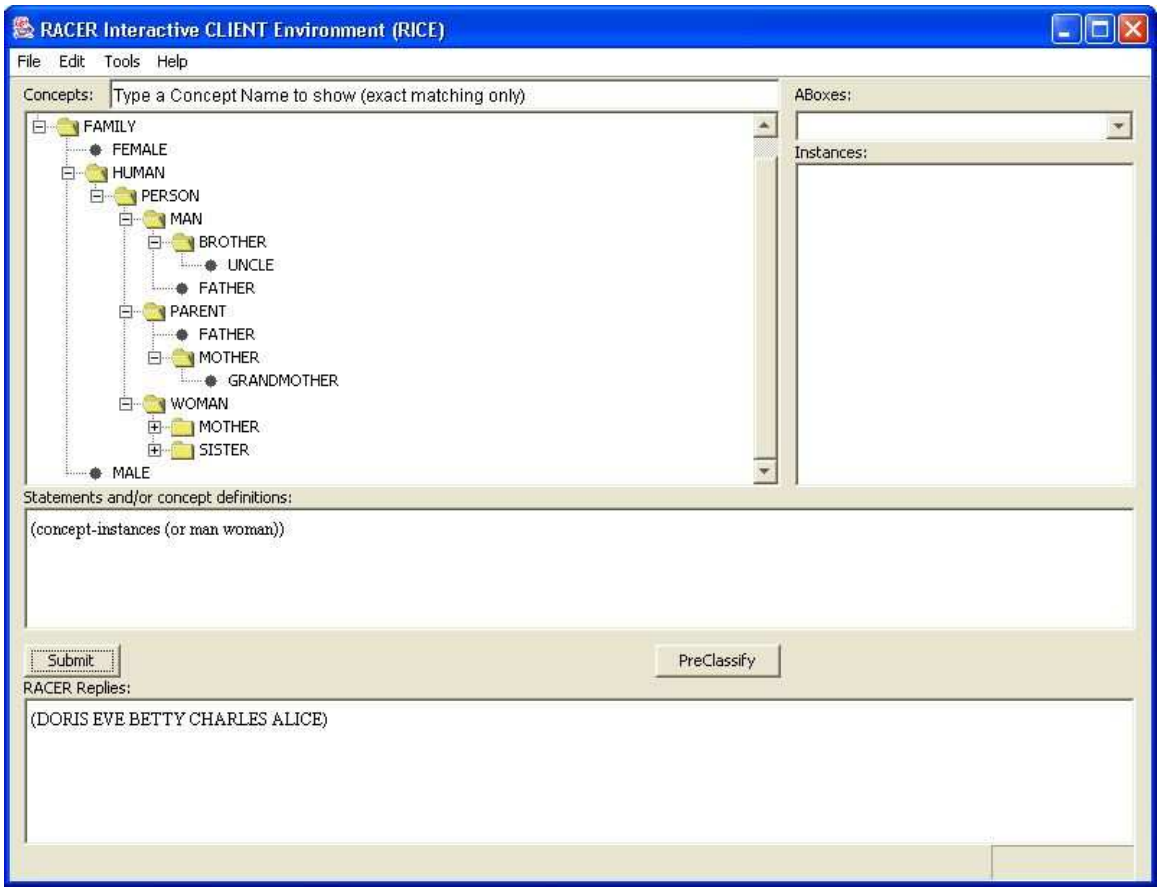

Figure 5: Screenshot of RICE.

Using the middle subwindow, statements, declarations, and queries can be typed and submitted to RACER. The answers are printed into the lower window. A query concerning the family knowledge base is presented in Figure 5. The query searches for instances of the concept `(or man woman)`. Other queries (e.g., as those shown in the previous section) can be submitted to RACER is a similar way.

An example for submitting a statement is displayed in Figure 6. The family knowledge base is exported as a DAML file. As we will see below, this file can then be processed with OilEd.

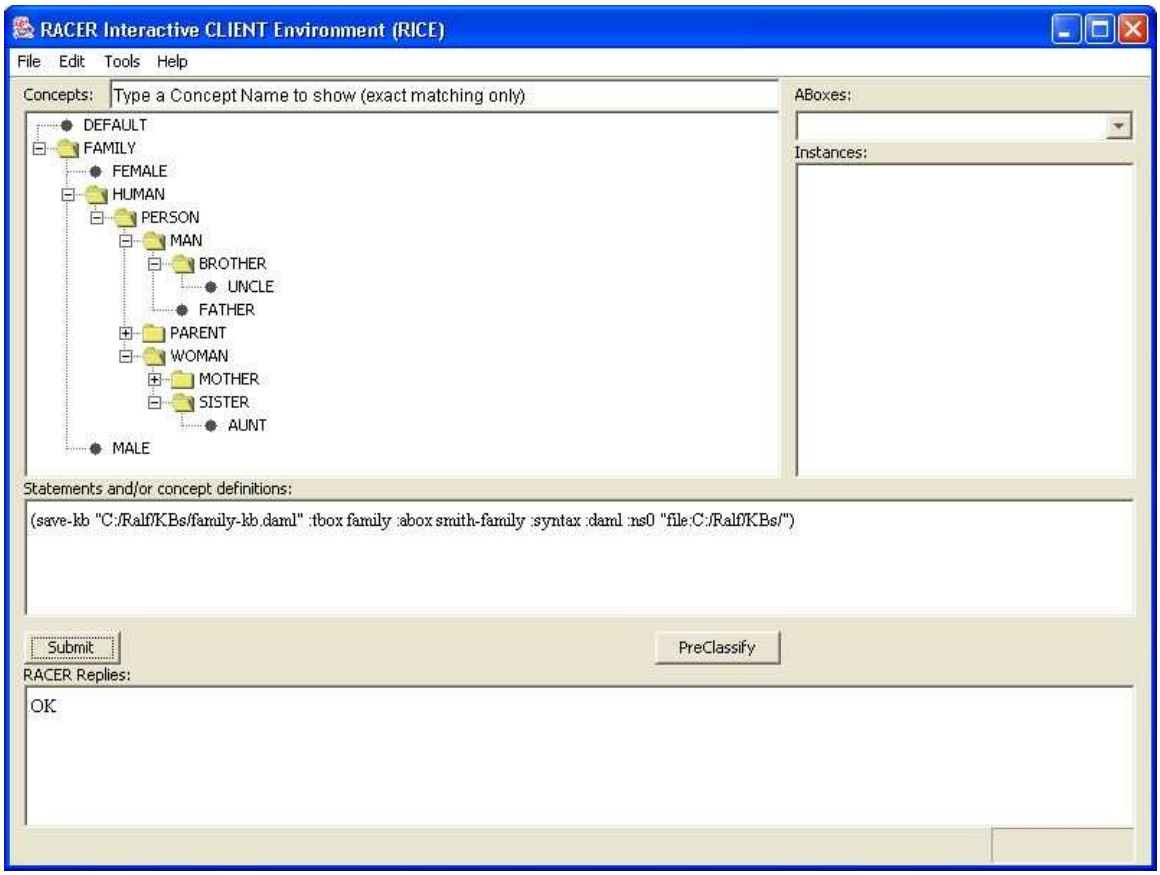

Figure 6: Screenshot of RICE.

2.6.2 OilEd

OilEd may be used as another graphical interface for RACER. While RICE can be used to pose queries, in particular for ABoxes, OilED can be used to graphically construct TBoxes (or ontologies) and ABoxes. OilEd has been developed by Sean Bechhofer and colleagues at the University of Manchester [Bechhofer et al. 01]. It is available at `http://oiled.man.ac.uk/`. You need OilEd version 3.5 or newer to access RACER as a backend reasoner.

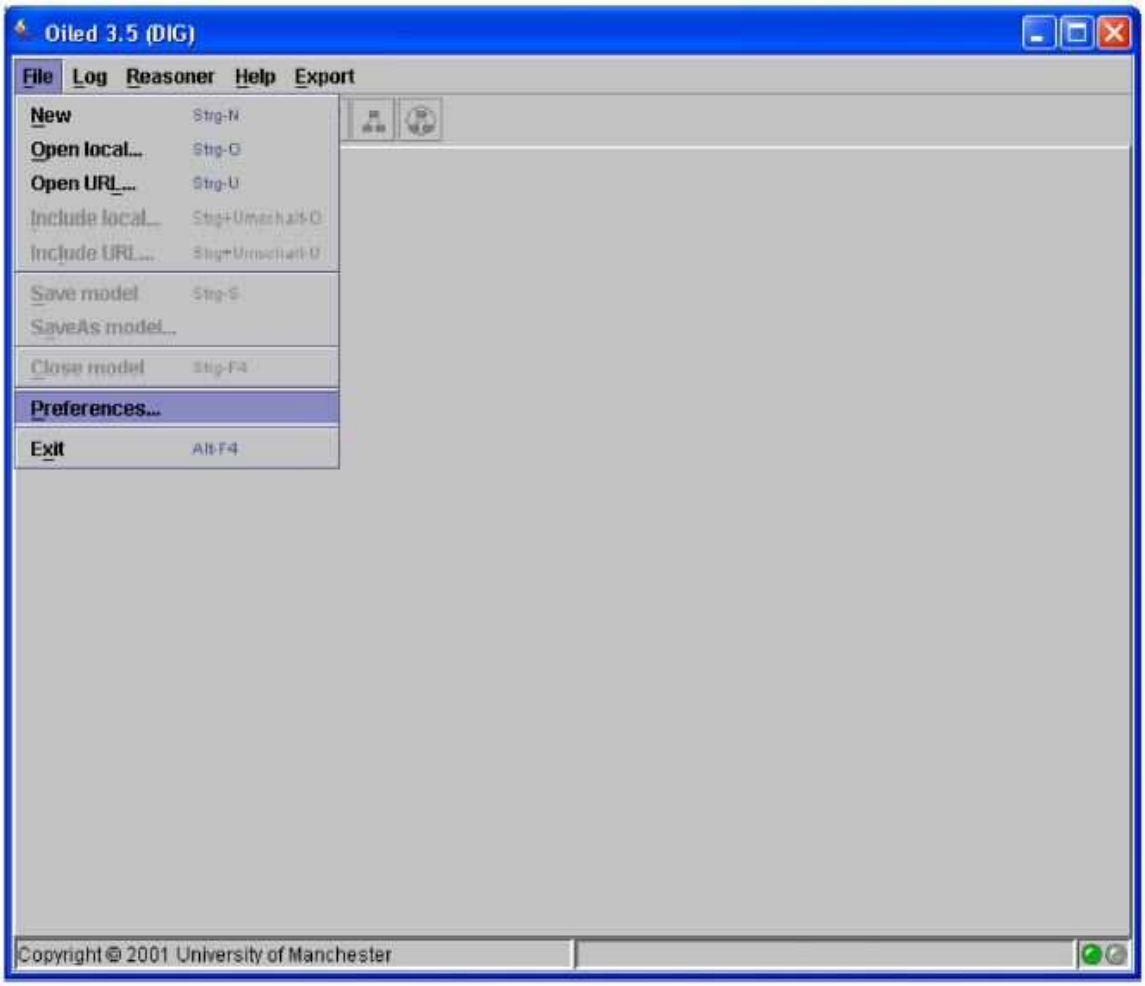

Figure 7: First step for declaring RACER as the reasoner used by OilEd.

In order to declare RACER as the standard reasoner used by OilEd, select the Preferences menu in OilEd(see Figure 7).

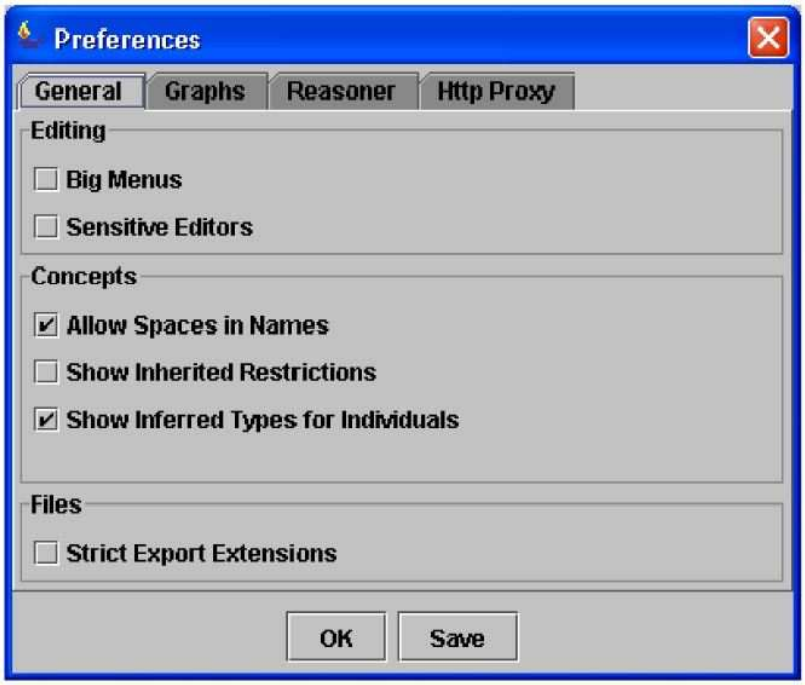

Figure 8: Second step for declaring RACER as the reasoner used by OilEd.

In the preferences dialog window select the Reasoner tab (see Figure 8).

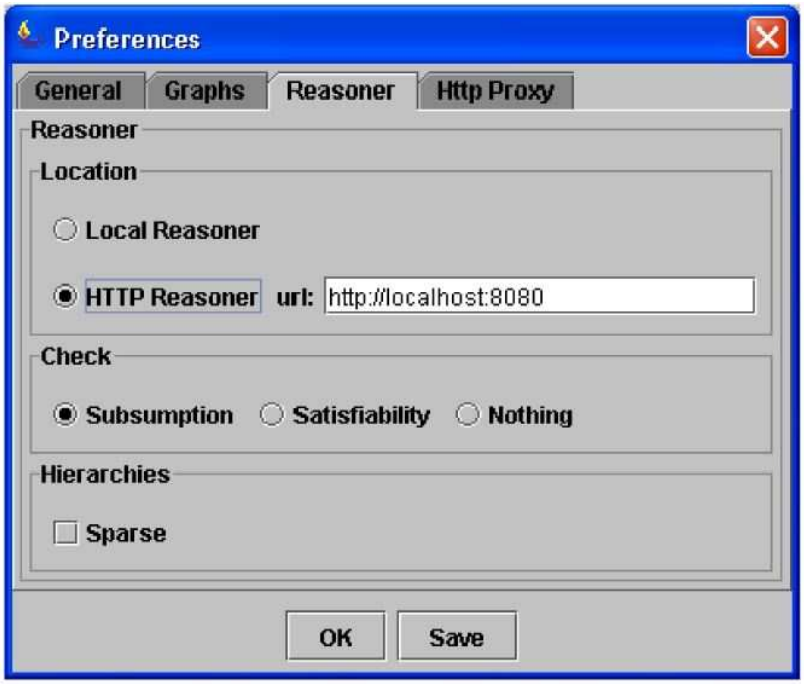

Figure 9: Third step for declaring RACER as the reasoner used by OilEd.

Then, select the HTTP Reasoner radio button in the dialog window (see Figure 9). The URL should be ok since it reects the standard port for RACER on you local host. If you would like to change the URL do not forget to press the return key in order to make the changes effective. Afterwards, press the Save button, then press the OK button. Now, RACER can be used as a reasoner for OilEd. In order to demonstrate this we discuss the Family examples from above.

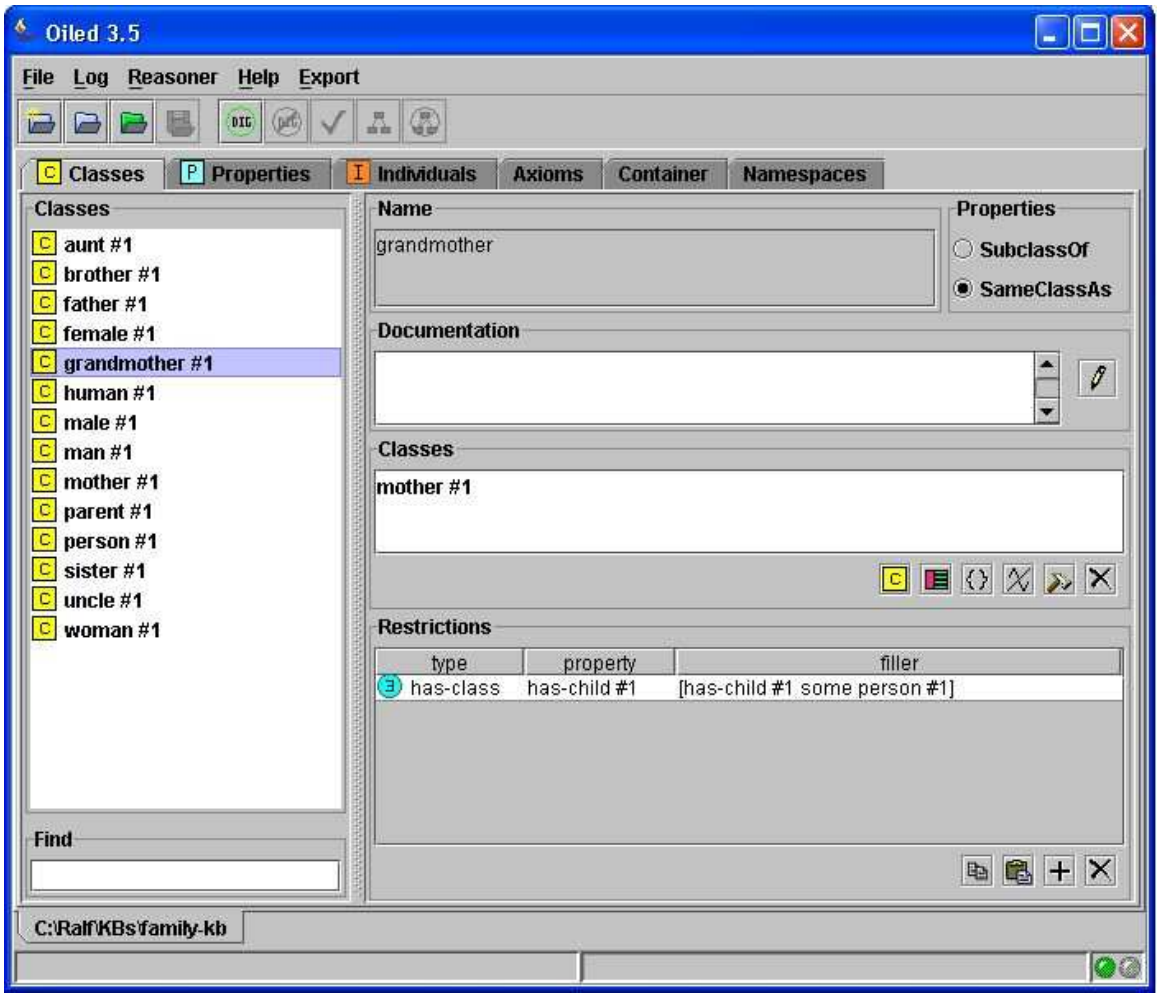

Figure 10: Oiled displaying the Family knowledge base.

The example presented in Figure 10 shows a screenshot of OilEd with the family knowledge base. The knowledge base was exported in Figure 6 using the DAML syntax. DAML files can be manipulated with OilEd.

In Figure 10 the concept `grandmother` is selected (concepts are called classes in OilEd). See the restrictions displayed is the lower right window and compare the specification with the KRSS syntax used above (see the axiom for `grandmother`).

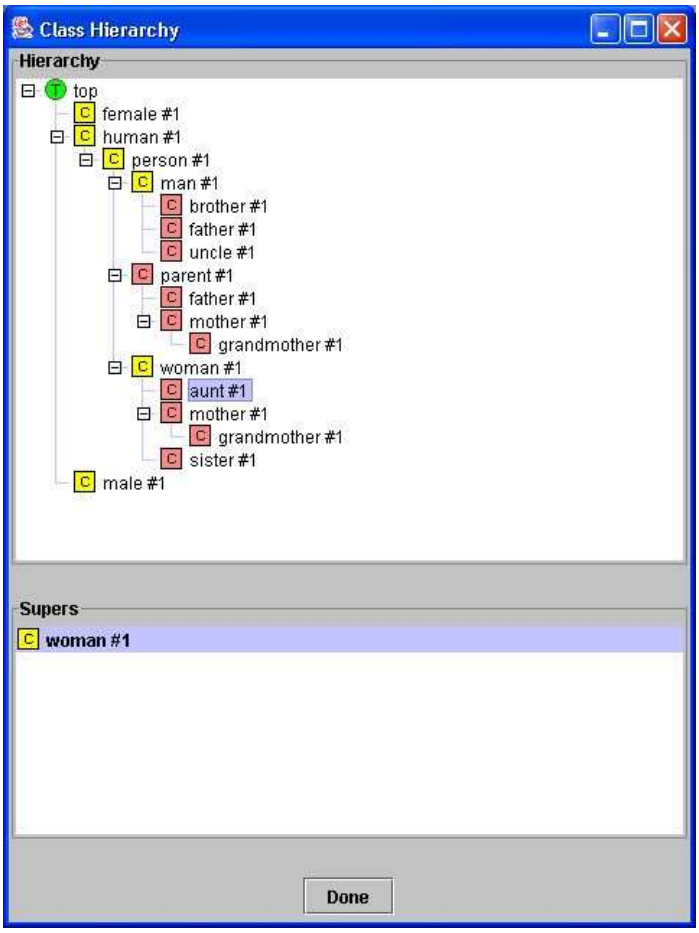

Figure 11: Concept hierarchy based on displaying explictly given superconcepts only.

Double-click a concept (class) and see the hierarchy window displayed here in Figure 11. Reasoning services are provided when OilEd is connected to RACER. Press the DIG button in the tools bar. Afterwards, select the "tick" button to let RACER check for unsatisfiable concepts in the current OilEd knowledge base and find implicit subsumption relationships between concept names.

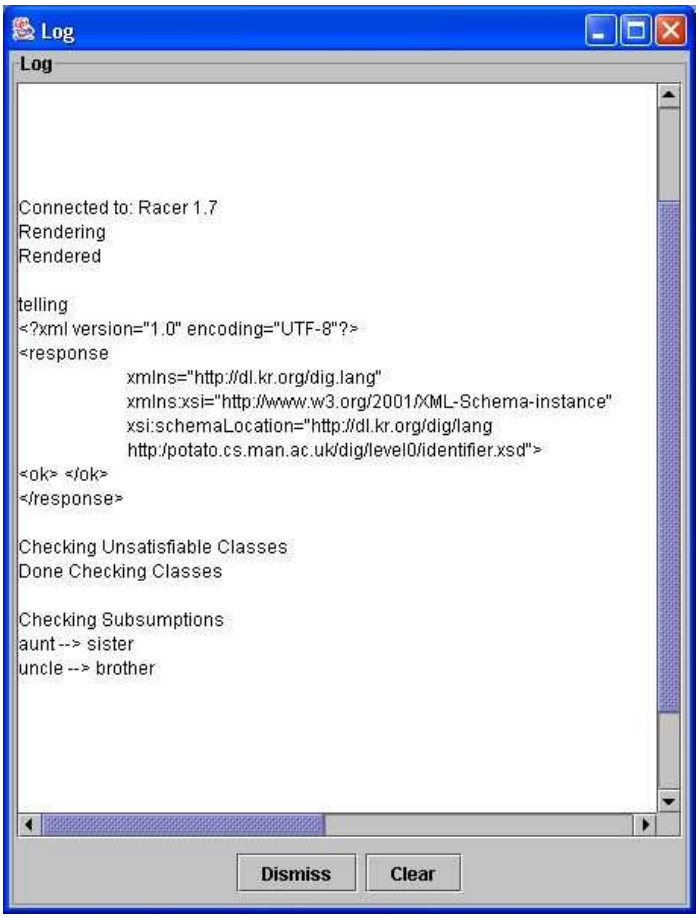

Figure 12: OilEd activity log window.

Implicit subsumption relationships are indicated in the OilEd activity log window, which can be displayed by selecting the corrosponding menu item in the Log menu (see Figure 12).

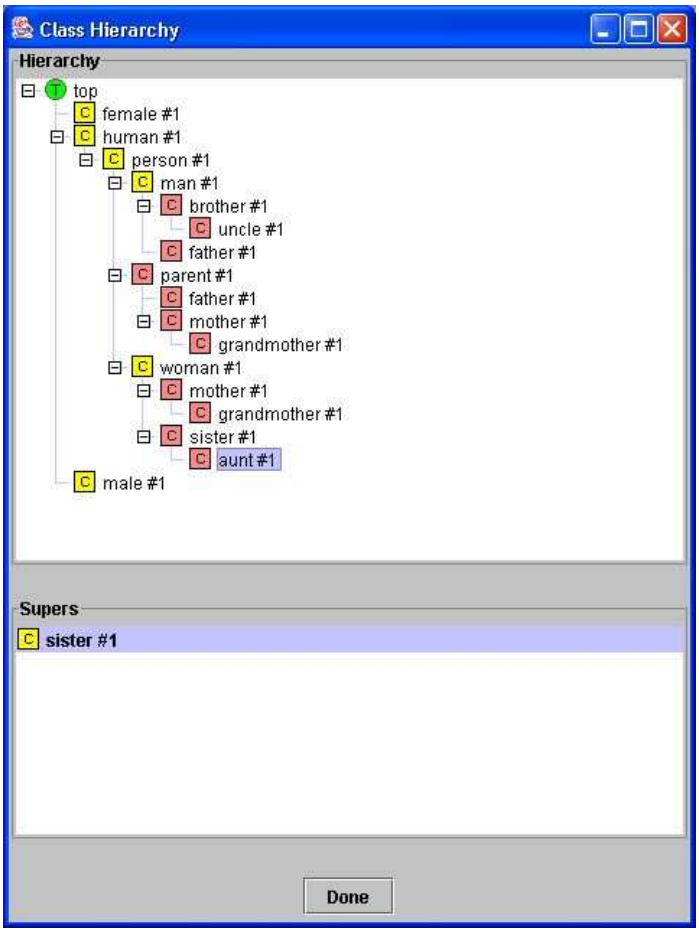

Figure 13: Concept hierarchy reflecting implicit subsumption relationships.

In Figure 12 it is indicated that RACER reveals implicit subsumption relationships. For instance, an `aunt` is also a `sister`. The same can be seen in Figure 13 using the concept hierarchy window. Compare with Figure 11.

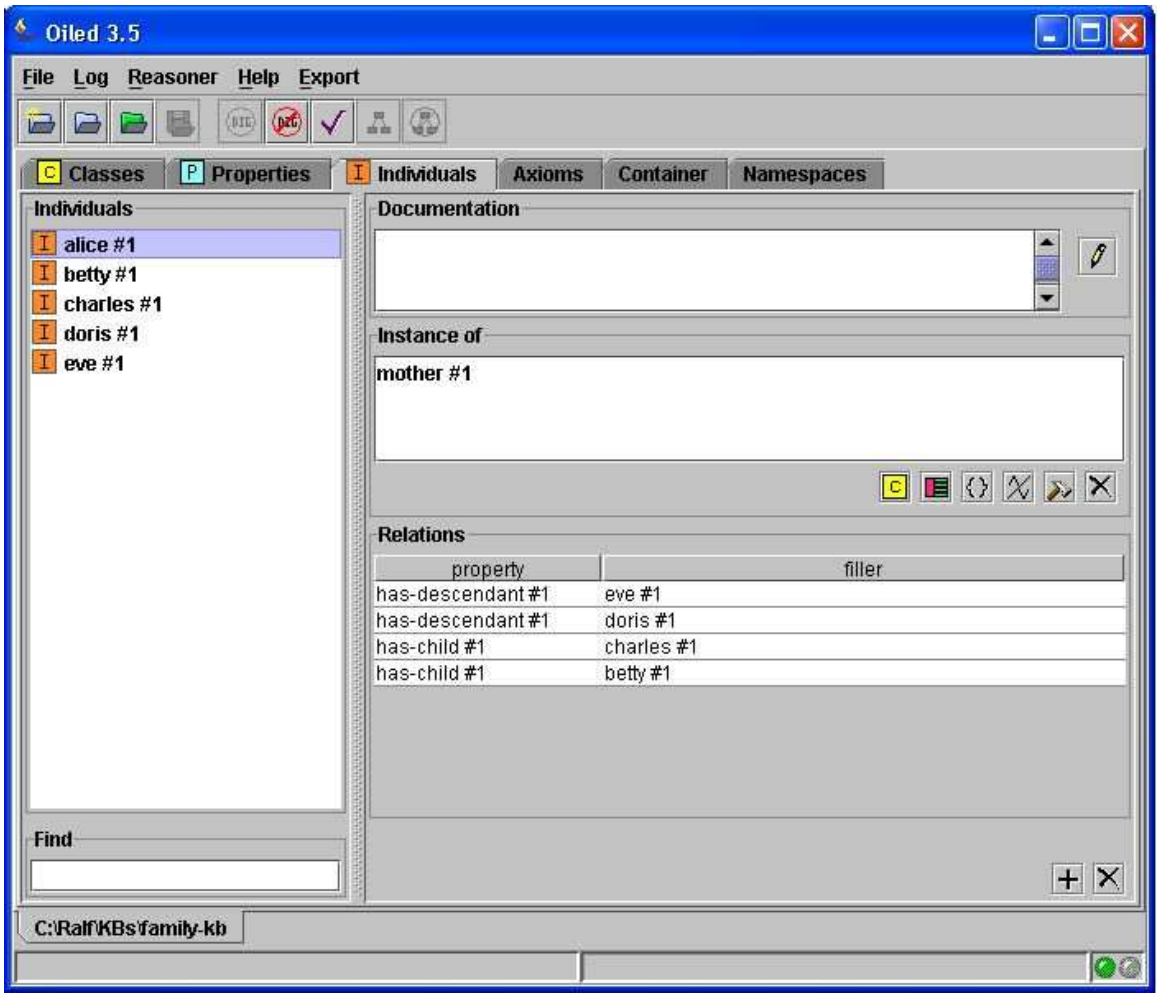

Figure 14: The individuals of the family example. Properties of the selected individual `alice` are displayed.

After the tab Individuals is selected, OilEd displays the individuals of the family knowledge base together with their properties (or relationships). This is shown here in Figure 14.

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

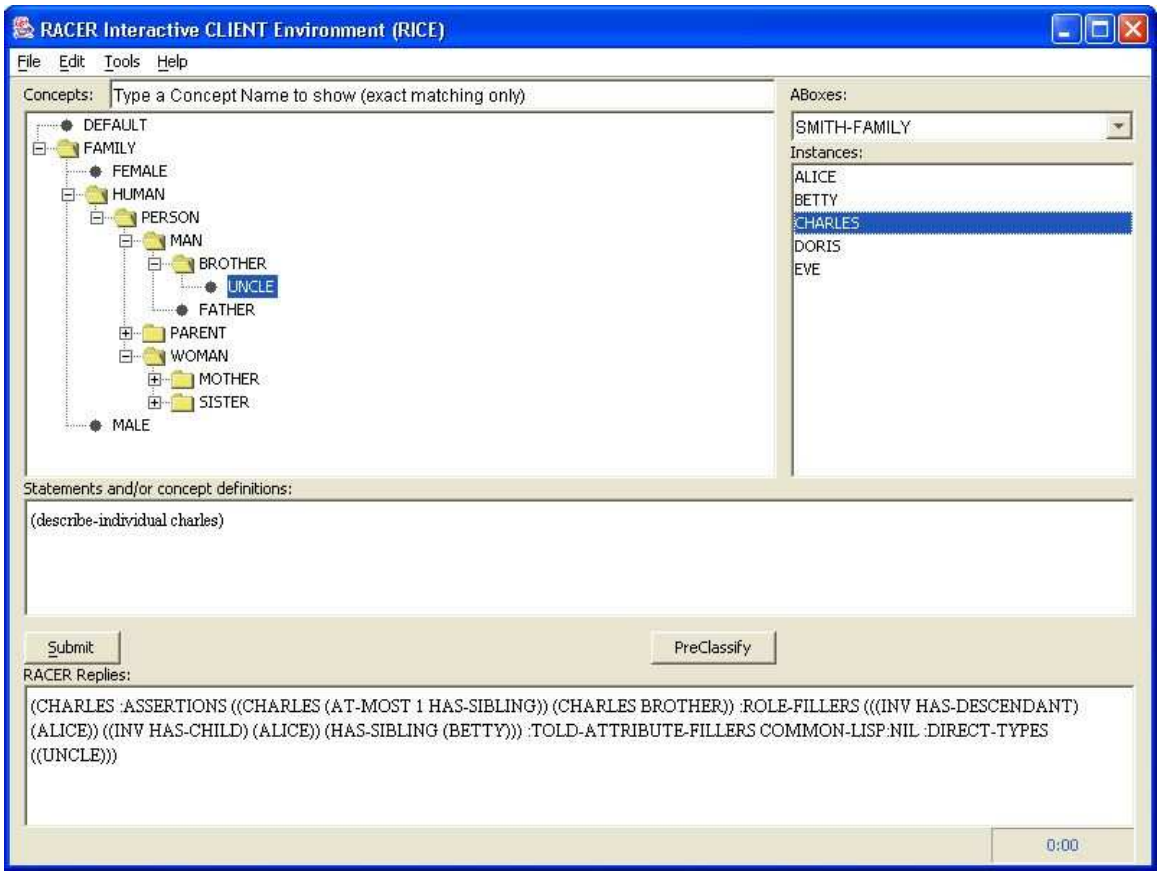

Figure 15: Screenshot of RICE indicating the direct types of the individual `charles`.

Selecting the individual `charles` causes RICE to highlight the direct types in the concept window (see Figure 15). You see that it is easy to pose often-used queries concerning individuals using RICE. In addition, query results are graphically visualized with RICE. Furthermore, RICE can be used to describe individuals. See the query `describe-individual` typed into the interaction window of RICE in Figure 15.

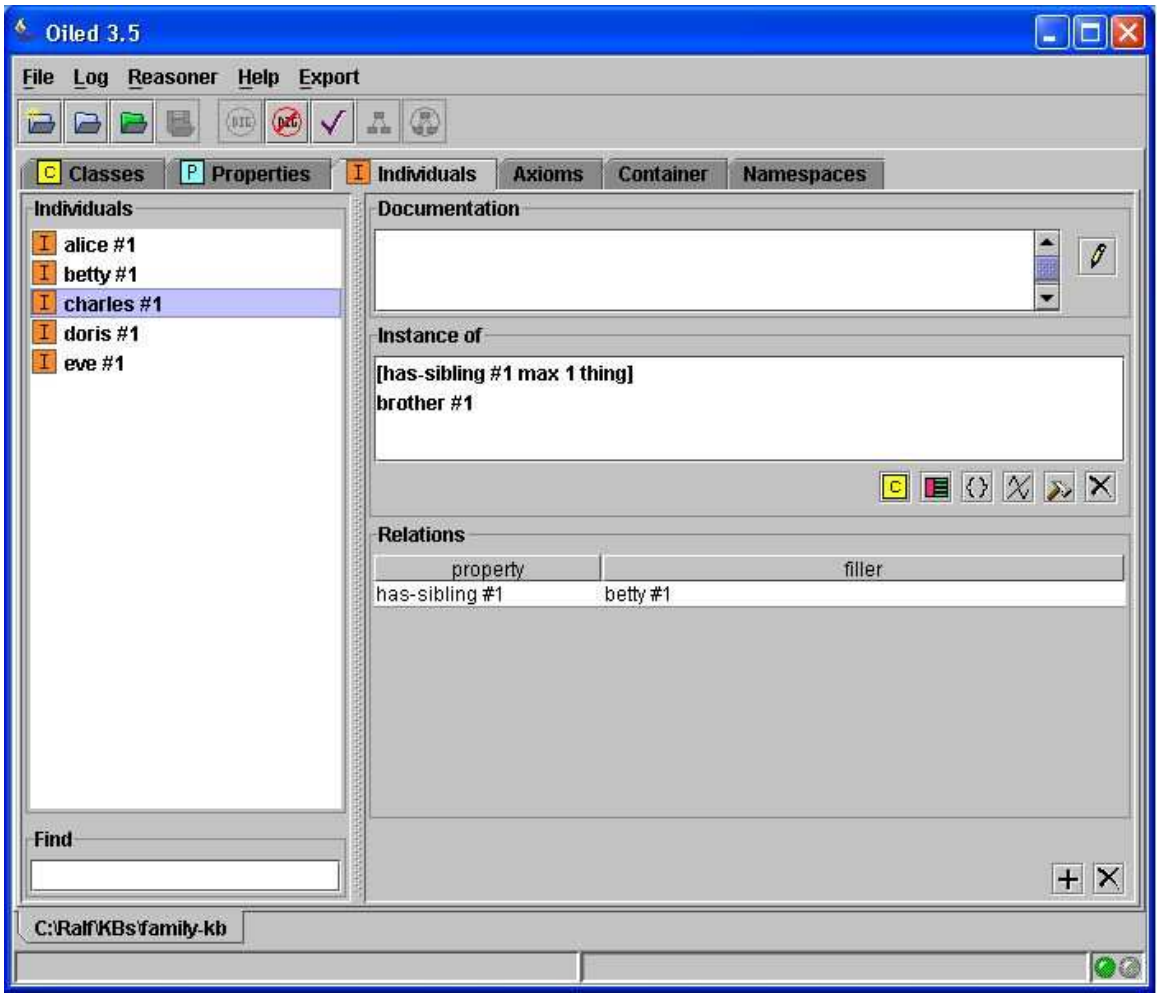

Figure 16: Screenshot of Oiled displaying only told information about `charles`.

Currenty, OilEd does not support ABox querying. Implicit information derivable about individuals is not shown in OilEd. This is indicated in Figure 16. The individual `charles` is indicated to be an instance of `brother` (rather than `uncle`).

Another example for using OilEd with RACER is presented in subsequent figures. The knowledge base is provided as one of the standard examples coming with the OilEd distribution. Start OilEd and open the example ontology `mad_cows.daml`. Then, connect OilEd to RACER and verify the knowledge base.

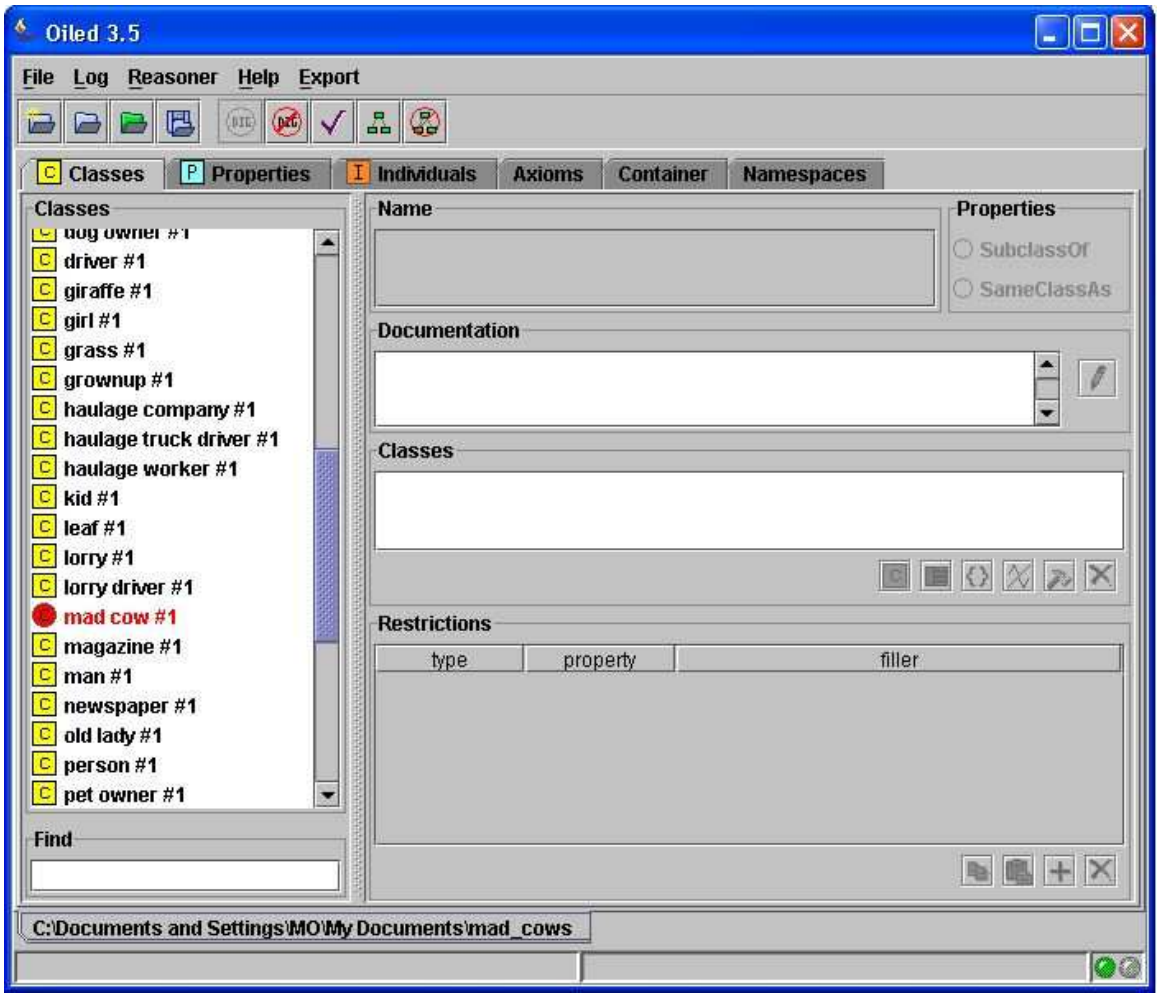

Figure 17: Inconsistent class `mad cow`.

The result of the knowledge base verification process using RACER is displayed in Figure 17. The concept (class) `mad cow` is found to be inconsistent (unsatisfiable).[2]

[2]The icon for the class `mad cow` is displayed with red color. However, this may be lost due to black and white printing. Unsatisfiable concepts are also indicated by a circle rather than by a square as used for satisfiable concepts.

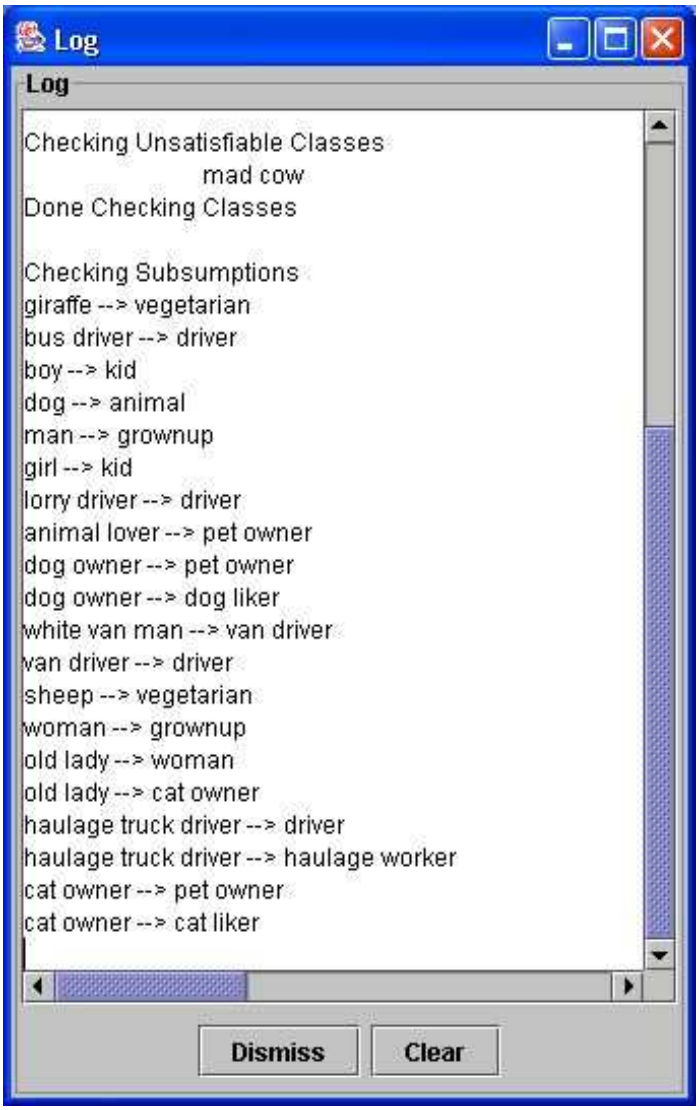

Figure 18: Implicit subsumption relationsship detected by the reasoner.

In Figure 18 implicit subsumption relationships found by the reasoner are indicated.

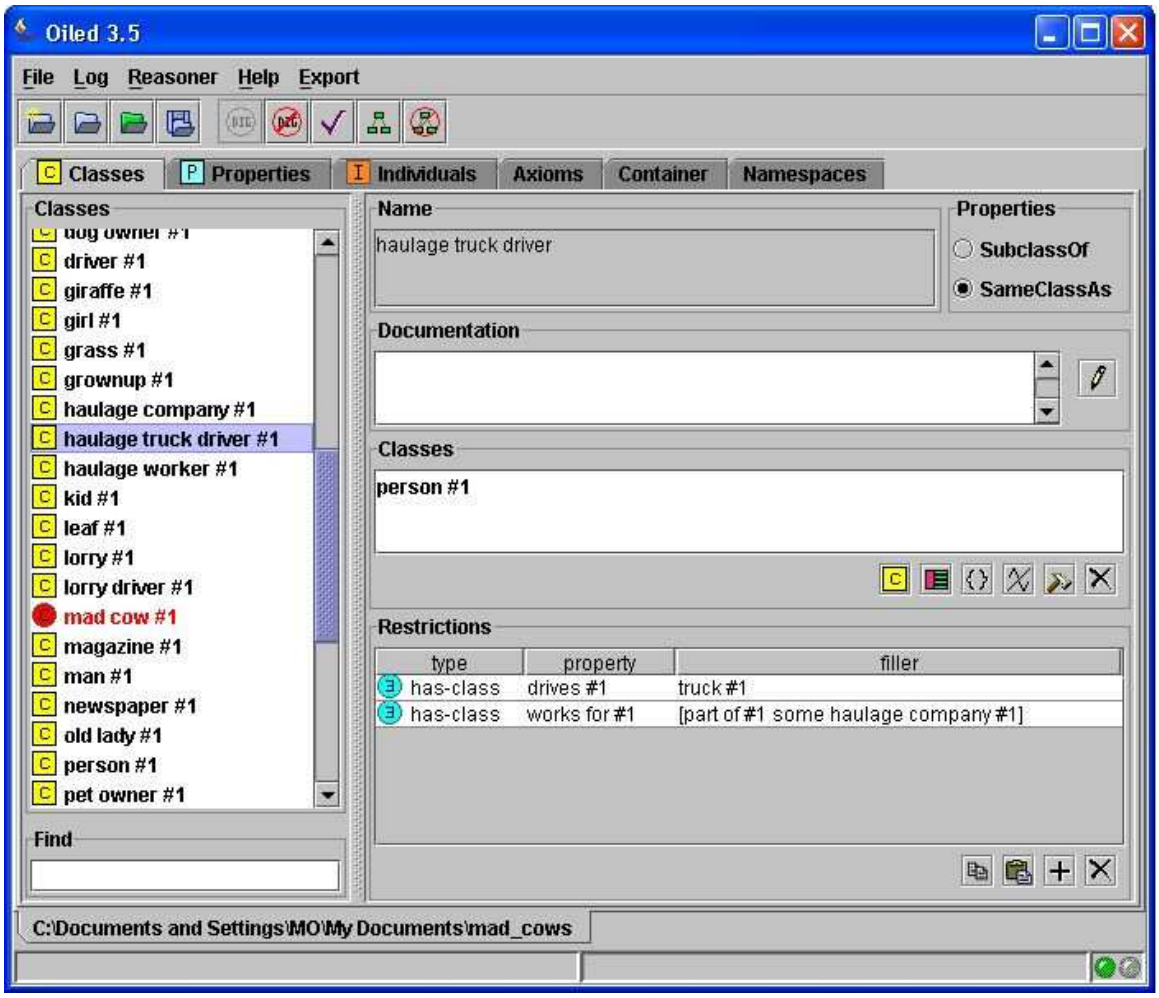

Figure 19: Restrictions for the concept `haulage truck driver`.

In particular, we consider the concept `haulage truck driver`. Its restrictions are shown in Figure 19.

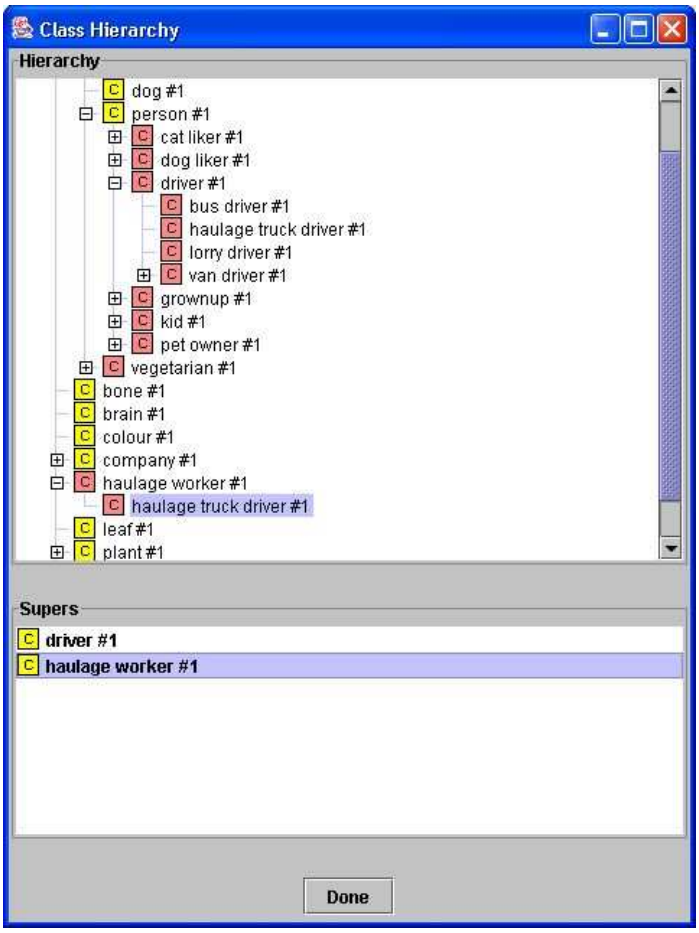

Figure 20: OilEd showing the concept hierarchy after the model is verified by the reasoner.

In Figure 20 we can see that, for instance, `haulage truck driver` is (also) subsumed by `haulage worker` and `driver`. The reasoner is used to find these implicit subsumption relationships.

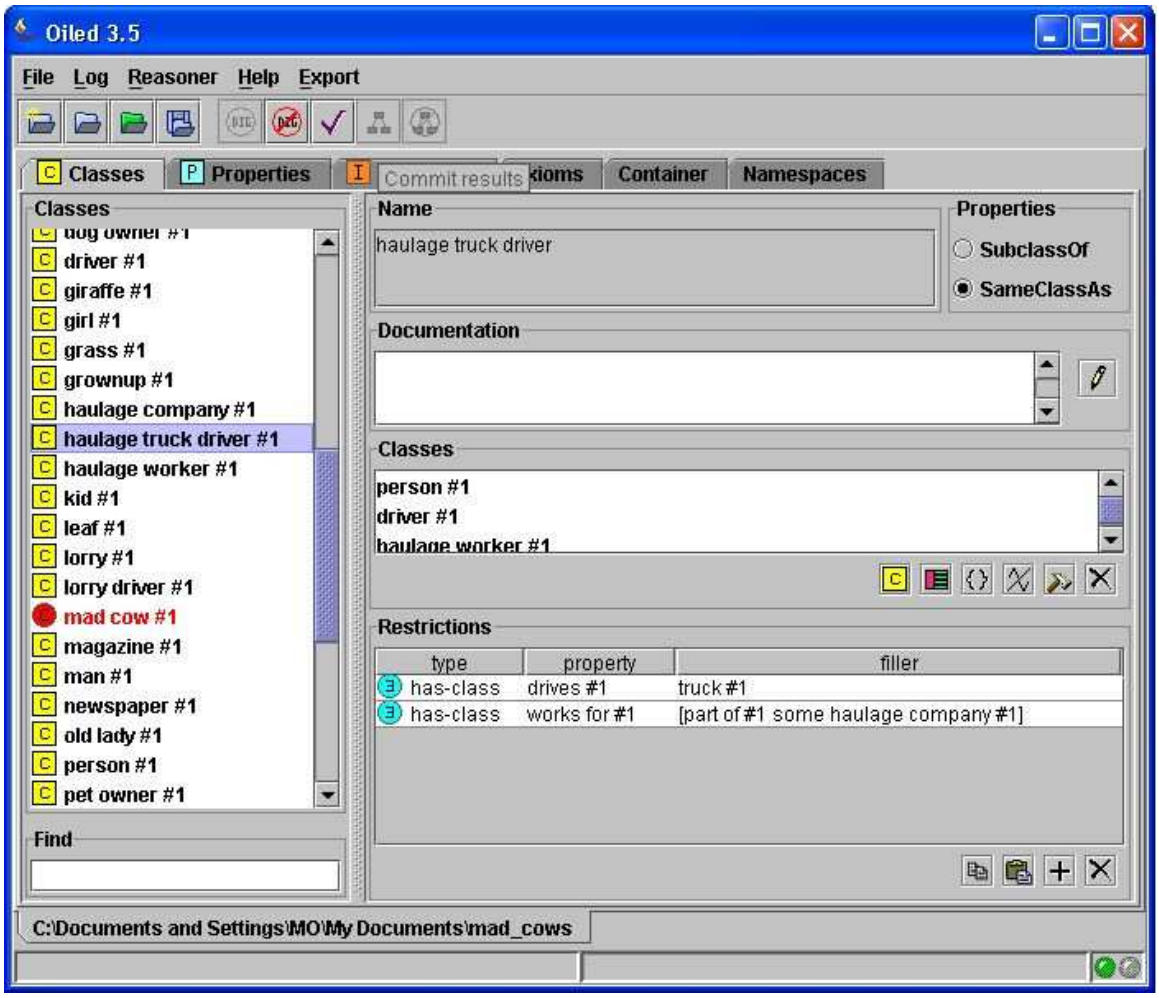

Figure 21: Making implicit subsumption explicit: committing reasoning results into the model.

OilEd can be used to commit implicit subsumption relationships found be the reasoner into the model (see the tool bar). If the augmented model is saved to a file, it can then be processed by "simple" processing engines which are not based on description logics (such as RACER).

2.6.3 Using OilEd and Rice in Combination

Let us assume, RACER is started, OilEd is connected to RACER, and some knowledge base verifications on `mad_cows.daml` have been performed. OilEd uses the default knowledge base of RACER (this is due to the DIG interface). Then, RICE is started. The user can browse through the concept hierarchy of the default knowledge base.

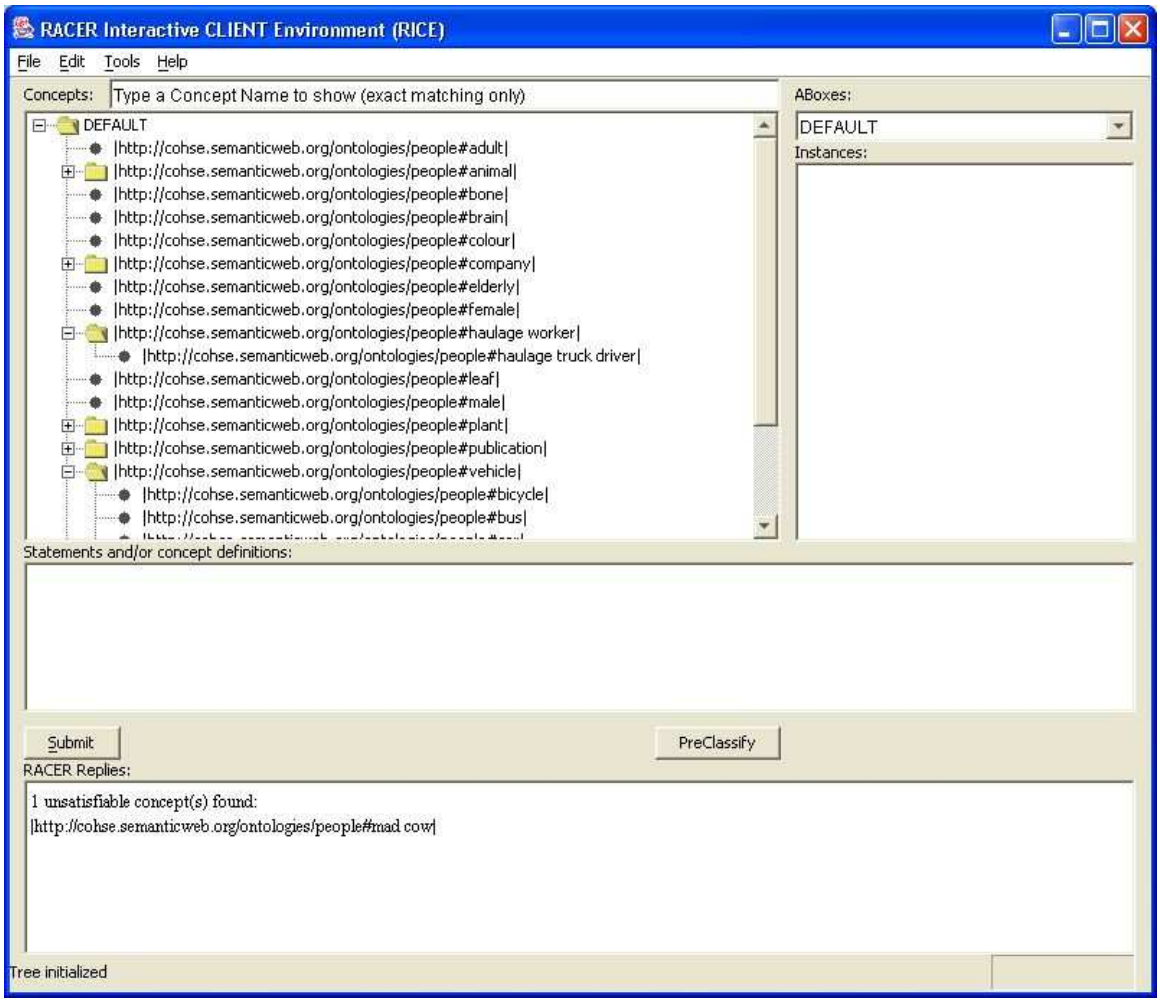

Figure 22: RICE showing the ontology `mad_cows.daml`.

The concepts from the `mad_cows` example can now be inspected with the RICE interface (Figure 22). Again, the concept `mad cow` is found to be unsatisfiable. OilEd provides namespaces for concept names. Namespaces are indicated with prefixes in concept names. The current version of RICE (and RACER) directly displays the prefixes. In addition, concept names are case-sensitive. Case-sensitive concept name should be specified surrounded with bars (|) in order to convince RACER to correctly treat these concept names.

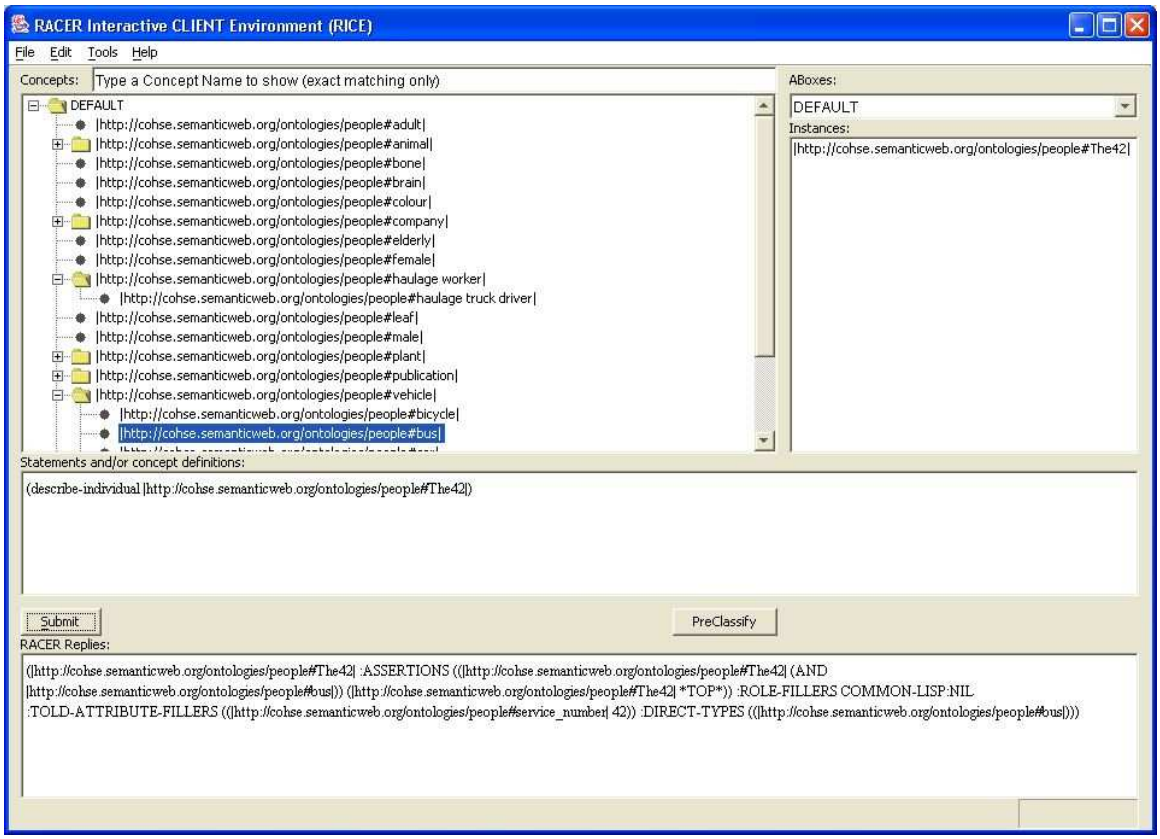

Figure 23: Dealing with individuals in RICE.

Selecting a concept instructs RICE to display its instances in the upper right instances window (see Figure 23). In addition, Figure 23 shows a query. The bus `The42` is described. Note that the filler for the attribute `service_number` is 42. Hence, we can see that Racer also supports the specification of numbers (see below for a detailed introduction on how to use concrete domains).

This concludes the interface examples. Not all features can be shown in a static text document. We encourage users to experiment with all graphical interfaces in order to find out how they can be used in combination. We now turn to the features provided by the RACER system, and we start with some naming conventions.

2.6.4 Protégé

Protégé is an ontology editor and a knowledge-base editor. Version 2.0 also supports OWL files and provides custom-tailored graphical widgets for editing description logic expressions (see http://protege.stanford.edu/ and subpages). A screenshot of Protégé is shown in Figure 24. Protégé is built by Holger Knublauch, Stanford University.

In Protégé inference services can be invoked by pressing the "Classify" button above the class tree in the OWL Classes Tab to execute classifiers. Before you can use it you must make sure that an external classifier with a DIG-compliant interface (e.g., Racer) is executing and accessible through your localhost at port 8080 (see the Protégé manual for changing the

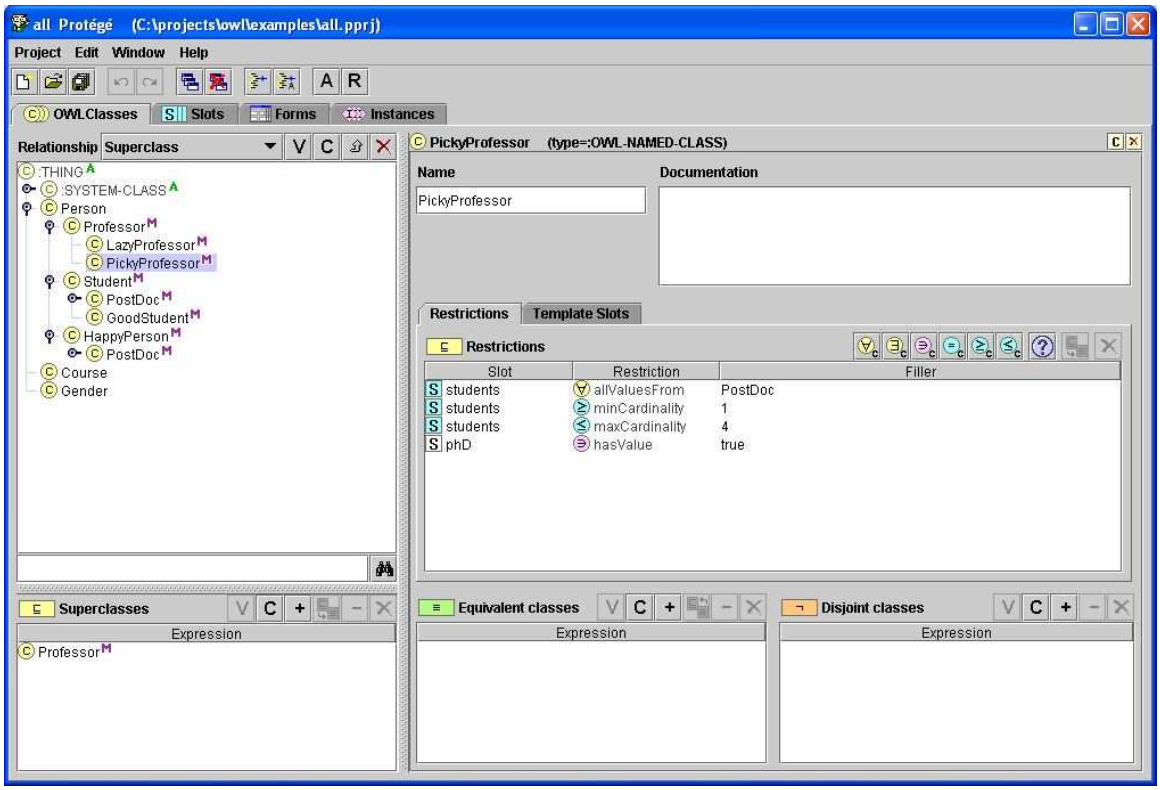

Figure 24: A screenshot of the Protégé system.

Racer host ip and port that Protégé addresses). When you hit the classify button, the system will then invoke Racer. When Racer has finished it will display a small report dialog to indicate how many new class relationships have been identifed. Using the classifier may directly change the inheritance hierarchy displayed in the Protégé window.

2.7 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*	ABox object
RN	Role name	*tbox*	TBox object
AN	Attribute name	*n*	A natural number
ABN	ABox name	*real*	A real number
TBN	TBox name	*integer*	An integer number
KBN	knowledge base name	*string*	A string

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.

The API is designed to the following conventions. For most of the services offered by RACER, macro interfaces and function interfaces are provided. For macro forms, the TBox or ABox arguments are optional. If no TBox or ABox is specified, the `*current-tbox*` or `*current-abox*` is taken, respectively. However, for the functional counterpart of a macro the TBox or ABox argument is not optional. For functions which do not have macro counterparts the TBox or ABox argument may or may not be optional. Furthermore, if an argument *tbox* or *abox* is specified in this documentation, a name (a symbol) can be used as well.

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

3 RACER Knowledge Bases

In description logic systems a knowledge base is consisting of a TBox and an ABox. The conceptual knowledge is represented in the TBox and the knowledge about the instances of a domain is represented in the ABox. For more information about the description logic $\mathcal{SHIQ}$ supported by RACER see [Horrocks et al. 2000]. Note that RACER assumes the *unique name assumption* for ABox individuals (see also [Haarslev and Möller 2000] where the logic supported by RACER's precursor RACE is described). The unique name assumption does not hold for the description logic $\mathcal{SHIQ}$ as introduced in [Horrocks et al. 2000].

3.1 Concept Language

The content of RACER TBoxes includes the conceptual modeling of concepts and roles as well. The modelling 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 names[3].

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 25.

Boolean terms build concepts by using the boolean operators.

	DL notation	RACER syntax
Negation	$\neg C$	`(not` C`)`
Conjunction	$C_1 \sqcap \ldots \sqcap C_n$	`(and` C_1 $\ldots C_n$`)`
Disjunction	$C_1 \sqcup \ldots \sqcup C_n$	`(or` C_1 $\ldots C_n$`)`

[3]The signature does not have to be specified explicitly in RACER 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!

$$
\begin{array}{lll}
C \longrightarrow & CN & | \\
& \texttt{*top*} & | \\
& \texttt{*bottom*} & | \\
& \texttt{(not } C\texttt{)} & | \\
& \texttt{(and } C_1 \ \ldots C_n\texttt{)} & | \\
& \texttt{(or } C_1 \ \ldots C_n\texttt{)} & | \\
& \texttt{(some } R \ C\texttt{)} & | \\
& \texttt{(all } R \ C\texttt{)} & | \\
& \texttt{(at-least } n \ R\texttt{)} & | \\
& \texttt{(at-most } n \ R\texttt{)} & | \\
& \texttt{(exactly } n \ R\texttt{)} & | \\
& \texttt{(at-least } n \ R \ C\texttt{)} & | \\
& \texttt{(at-most } n \ R \ C\texttt{)} & | \\
& \texttt{(exactly } n \ R \ C\texttt{)} & | \\
& \texttt{(a } AN\texttt{)} & | \\
& \texttt{(an } AN\texttt{)} & | \\
& \texttt{(no } AN\texttt{)} & | \\
& CDC & \\
 & & \\
R \longrightarrow & RN & | \\
& \texttt{(inv } RN\texttt{)} &
\end{array}
$$

Figure 25: RACER concept and role terms.

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	RACER syntax
Exists restriction	$\exists\ R.C$	`(some` R C`)`
Value restriction	$\forall\ R.C$	`(all` R C`)`

Number restrictions can specify a lower bound, an upper bound or an exact number for the amount of role fillers each instance of this concept has for a certain role. Only roles that are not transitive and do not have any transitive subroles are allowed in number restrictions (see also the comments in [Horrocks-et-al. 99a, Horrocks-et-al. 99b]).

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

Actually, the exactly restriction `(exactly` n R`)` is an abbreviation for the concept term

CDC	⟶	`(min` *AN integer*`)` |
		`(max` *AN integer*`)` |
		`(equal` *AN integer*`)` |
		`(equal` *AN AN*`)` |
		`(divisible` *AN cardinal*`)` |
		`(not-divisible` *AN cardinal*`)` |
		`(>` *aexpr aexpr*`)` |
		`(>=` *aexpr aexpr*`)` |
		`(<` *aexpr aexpr*`)` |
		`(<=` *aexpr aexpr*`)` |
		`(<>` *aexpr aexpr*`)` |
		`(=` *aexpr aexpr*`)` |
		`(string=` *AN string*`)` |
		`(string<>` *AN string*`)` |
		`(string=` *AN AN*`)` |
		`(string<>` *AN AN*`)`
string	⟶	`"` letter* `"`
aexpr	⟶	*AN* |
		real |
		`(+` *aexpr1 aexpr1*`*)` |
		aexpr1

Figure 26: RACER concrete domain concepts and attribute expressions.

`(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 TBox: the concept "top" ($\top$) denotes the top-most concept in the hierarchy and the concept "bottom" ($\bot$) denotes the inconsistent concept, which is a subconcept to all other concepts. Note that $\top$ ($\bot$) can also be expressed as $C \sqcup \neg C$ ($C \sqcap \neg C$). In RACER $\top$ is denoted as `*top*` and $\bot$ is denoted as `*bottom*`[4].

[4]For KRSS compatibility reasons RACER also supports the synonym concepts `top` and `bottom`.

aexpr1	$\longrightarrow$	*aexpr2*	
	|	*aexpr3*	
	|	*aexpr5*	
aexpr2	$\longrightarrow$	*real*	
	|	*AN*	(AN of type real or complex)
	|	`(*` *real AN*`)`	(AN of type real or complex)
aexpr3	$\longrightarrow$	*real*	
	|	*AN*	(AN of type complex)
	|	`(*` *integer aexpr4 aexpr4**`)`	
aexpr4	$\longrightarrow$	*AN*	(AN of type complex)
	|	`(expt` *AN n*`)`	(AN of type complex)
aexpr5	$\longrightarrow$	*integer*	
	|	*AN*	(AN of type cardinal)
	|	`(*` *integer AN*`)`	(AN of type cardinal)

Figure 27: Specific expressions for predicates ($n > 0$ is a natural number) .

Concrete domain concepts state concrete predicate restrictions for attribute fillers (see Figure 3.1). RACER currently supports three unary predicates for integer attributes (`min`, `max`, `equal`), six nary predicates for real attributes (`>`, `>=`, `<`, `<=`, `=`, `<>`), a unary existential predicate with two syntactical variants (`a` or `an`), and a special predicate restriction disallowing a concrete domain filler (`no`). The restrictions for attributes of type `real` have to be in the form of linear inequations (with order relations) where the attribute names play the role of variables. If an expression is built with the rule for *aexpr*4 (see Figure 3.1), a so-called nonlinear constraint is specified. In this case, only equations and inequations (`=`, `<>`), but no order constraints (`>`, `>=`, `<`, `<=`) are allowed, and the attributes must be of type `complex`. If an expression is built with the rule for *aexpr*5 (see Figure 3.1) a so-called cardinal linear constraint is specified, i.e., attributes are constrainted to be a natural number (including zero). Racer also supports a concrete domain for representing equations about strings with predicaes `string=` and `string<>`. The use of concepts with concrete domain expressions is illustrated with examples in Section 3.4. For the declaration of types for attributes, see see Section 3.5.

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

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

3.2 Concept Axioms and Terminology

RACER supports several kinds of concept axioms.

General concept inclusions (GCIs) state the subsumption relation between two concept terms.
DL notation: $C_1 \sqsubseteq C_2$
RACER syntax: `(implies` C_1 C_2`)`

Concept equations state the equivalence between two concept terms.
DL notation: $C_1 \doteq C_2$
RACER syntax: `(equivalent` C_1 C_2`)`

Concept disjointness axioms state pairwise disjointness between several concepts. Disjoint concepts do not have instances in common.
DL notation: $C_1 \sqsubseteq \neg(C_2 \sqcup C_3 \sqcup \cdots \sqcap C_n)$
$C_2 \sqsubseteq \neg(C_3 \sqcup \cdots \sqcap C_n)$
$\ldots$
$C_{n-1} \sqsubseteq \neg C_n$
RACER syntax: `(disjoint` C_1 `...`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 $C_1 \ldots C_n$ 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 RACER 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: $(CN \sqsubseteq C)$
RACER syntax: `(define-primitive-concept` CN C`)`

Concept definitions state the equality between a concept name and a concept term.
DL notation: $(CN \doteq C)$
RACER syntax: `(define-concept` CN C`)`

Concept axioms may be cyclic in RACER. 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 RACER TBox 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.

3.3 Role Declarations

In contrast to concept axioms, role declarations are unique in RACER. 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 TBox 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` in Section 7.3, page 86) or referred to as `(inv` RN`)`.

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

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

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

In the current implementation the specified superrole relations may not be cyclic. If a role has a superrole, its properties are not in every case inherited by the subrole. The properties of a declared role induced by its superrole are shown in Figure 28. The table should be read as follows: For example if a role RN_1 is declared as a simple role and it has a feature RN_2 as a superrole, then RN_1 will be a feature itself.

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

Figure 28: Conflicting declared and inherited role properties.

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

RACER 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. [Buchheit et al. 93]). The KRSS-like declaration of the role `(define-primitive-role` RN `(and` RN_1 RN_2`))` can be approximated in RACER by: `(define-primitive-role` RN `:parents (`RN_1 RN_2`))`.

RACER 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 29 (cf. [Buchheit et al. 93]) or declared via the keywords `:domain` and `:range` (e.g. see the description of `define-primitive-role` in Section 7.3, page 86).

KRSS	DL notation
`(define-primitive-role` RN `(domain` C`))`	$(\exists\, RN.\top) \sqsubseteq C$
`(define-primitive-role` RN `(range` D`))`	$\top \sqsubseteq (\forall\, RN.D)$
RACER Syntax	DL notation
`(define-primitive-role` RN `:domain` C`)`	$(\exists\, RN.\top) \sqsubseteq C$
`(define-primitive-role` RN `:range` D`)`	$\top \sqsubseteq (\forall\, RN.D)$

Figure 29: Domain and range restrictions expressed via GCIs.

3.4 Concrete Domains

RACER 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
$\mathbb{C}$	nonlinear multivariate inequations with integer coefficients
Strings	equality and inequality

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

The following example is an extension of the family TBox 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-feaver (and human (>= temperature-celsius 38.5))
(equivalent seriously-ill-human (and human (>= temperature-celsius 42.0)))
...
```

Obviously, RACER determines that the concept `seriously-ill-human` is subsumed by `human-with-feaver`. For the reals, RACER 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)))
```

RACER recognizes the subsumption relationship with `human-with-feaver` and the synonym relationship with `seriously-ill-human`.

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

```
...
(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-feaver`. In the following Abox there is an inconsistency since the temperature of 102.56 Fahrenheit is identical with 39.5 Celsius.

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

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

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

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

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

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

Next, we assume some instances of February are declared.

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

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

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

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

In both cases, the answer is true.

3.5 Concrete Domain Attributes

Attributes are considered as "typed" since they can either have fillers of type `cardinal`, `integer`, `real`, `complex`, or `string`. The same attribute cannot be used in the same TBox 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 TBox/ABox. An attribute and its type can be declared with the signature form (see above and in Section 6.1, page 68)) or by using the KRSS-like form `define-concrete-domain-attribute` (see Section 7.4, page 92). If an attribute is declared to be of type `complex` is can be used in linear (in-)equations. However, if an attribute is declare to be of type `real` or `integer` it is an error to use this attribute in terms for nonlinear polynoms. In a similar way, currently, an attribute of type `integer` may not be used in a term for a linear polynoms, either. If the coefficients are integers, then `cardinal` (natural number, including 0) for the type of attributes may be used in a linear polynom. Furthermore, attributes of type `string` may not be used on polynoms, and non-strings may not be used in constraints for strings.

3.6 Algorithms for Concrete Domains

[Rychlik 2000] [Weispfenning 92]

Jaffar, incremental constraint solving

[Jaffar and Maher 94]

Gomory

3.7 ABox Assertions

An ABox contains assertions about individuals. The set of individual names (or individuals for brevity) $\mathcal{I}$ is the signature of the ABox. 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 3.1. Instead of attribute names, object names must be used.

In RACER the *unique name assumption* holds, this means that all individual names used in an ABox 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 RACER system each ABox refers to a TBox. The concept assertions in the ABox are interpreted with respect to the concept axioms given in the referenced TBox. The role assertions are also interpreted according to the role declarations stated in that TBox. When a new ABox is built, the TBox to be referenced must already exist. The same TBox may be referred to by several ABoxes. If no signature is used for the TBox, the assertions in the ABox may use new names for roles[5] or concepts[6] which are not mentioned in the TBox.

3.8 Inference Modes

After the declaration of a TBox or an ABox, RACER 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 RACER ensures the following behavior: Depending on the kind of query, RACER 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 TBox it is not necessary to classify the TBox. However, once a TBox is classified, answering subsumption queries for atomic concepts is just a lookup. Furthermore, asking whether there exists an atomic concept in a TBox that is inconsistent (`tbox-coherent-p`) does not require the TBox to be classified, either. In the lazy mode of inference (the default), RACER 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 TBox before answering a query (eager inference mode).

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

The inference behavior of RACER can be controlled by setting the value of the variables `*auto-classify*` and `*auto-realize*` for TBox and ABox inference, respectively. The lazy inference mode is activated by setting the variables to the keyword `:lazy`. Eager inference behavior can be enforced by setting the variables to `:eager`. The default value for each variable is `:lazy-verbose`, which means that RACER prints a progress bar in order to indicate the state of the current inference activity if it might take some time. If you want this for eager inferences, use the value `:eager-verbose`. If other values are encountered, the user is responsible for calling necessary setup functions (not recommended).

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

[6] These concepts are assumed to be atomic concepts.

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

During the development phase of a TBox it might be advantageous to call inference services directly. For instance, during the development phase of a TBox it might be useful to check which atomic concepts in the TBox 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 certian ABox 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, RACER ensures that the knowledge bases are in the appropriate states. This behavior usually guarantees minimum runtimes for answering queries.

3.9 Retraction and Incremental Additions

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

RACER also offers constructs for retracting ABox assertions (see `forget`, `forget-concept-assertion`, `forget-role-assertion`, and friends). If a query has been answered and some assertions are retracted, then RACER might be forced to compute the index structures for the ABox again (realization), i.e. after retractions, some queries might take some time to answer. Note that may queries are answered without index structures at all (see also Section 11).

RACER also supports incremental additions to ABoxes, i.e. assertions can be added even after queries have been answered. However, the internal data structures used for anwering queries are recomputed from scratch. This might take some time. If an ABox 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 ABox. 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 ABox are not recomputed.

To be practically useful, description logics have to be integrated into current Web software architectures. We first turn to syntax issues and deal with distributed access to inference services afterwards.

4 The RDF/RDFS/DAML interface

Racer can read RDF, RDFS, and DAML+OIL files (DAML for brevity, see the function `daml-read-file` and friends described below). Information in an RDF file is represented

using an ABox 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. RACER 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 TBox 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.

DAML files are processed in a similar way. The semantics of DAML is described elsewhere (see `http://www.w3.org/TR/daml+oil-reference`). There are a few restrictions in the RACER 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., RACER cannot read files with datatype declarations right now).

Usually, the default namespace for concept and role name is dened by the pathname of the DAML le. If the DAML le contains a specication for the default namespace (i.e., a specication xmlns="...") this URL is taken as a prex for concept and role names.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
xmlns="http://www.mycompany.com/project#"
xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
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>
```

Insteadof xmlns="..." the specication xmlns:ns0="..." also denes the default namespace (the namespace ns0 is used by OilEd). By default, RACER prepends the URL of the default namespace to all DAML names starting with the `#` sign. If you would like to instruct RACER to return abbreviatednames (i.e., to remove the prex again in output it produces), start the RACER Server with the option -n (Abbreviation is possible in the server version only).

Individual names (nominals) in class declarations introduced with `daml:oneOf` are treated as disjoint (atomic) concepts. This is similar to the behavior of other DAML inference engines. Currently, DL systems can provide only an approximation for true nominals. Note that reasoning is sound but still incomplete if `daml:oneOf` is used. In RACER, individuals used in class declarations are also represented in the ABox 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 RACER 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)
```

For those users more familiar with DAML we also give the DAML version of this knowledge base (see file `ex1.daml` in the examples folder). The file was created with OilEd.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:oiled="http://img.cs.man.ac.uk/oil/oiled#"
    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#">
    <daml:Ontology rdf:about="">
        <dc:title>&quot;An Ontology&quot;</dc:title>
        <dc:date></dc:date>
        <dc:creator></dc:creator>
        <dc:description></dc:description>
        <dc:subject></dc:subject>
        <daml:versionInfo></daml:versionInfo>
    </daml:Ontology>
    <daml:Class rdf:about="file:C:\Ralf\Ind-Examples\ex1.daml#c">
        <rdfs:label>c</rdfs:label>
        <rdfs:comment><![CDATA[]]></rdfs:comment>
        <oiled:creationDate><![CDATA[2002-09-27T19:09:25Z]]></oiled:creationDate>
        <oiled:creator><![CDATA[MO]]></oiled:creator>
        <rdfs:subClassOf>
            <daml:Restriction>
                <daml:onProperty rdf:resource="file:C:\Ralf\Ind-Examples\ex1.daml#R"/>
                <daml:hasClass>
                    <daml:Class>
                     <daml:oneOf>
                      <daml:List>
                       <daml:first>
                        <daml:Thing rdf:about="file:C:\Ralf\Ind-Examples\ex1.daml#j"/>
                       </daml:first>
                      <daml:rest>
                       <daml:nil/>
                      </daml:rest>
                     </daml:List>
                    </daml:oneOf>
                    </daml:Class>
                </daml:hasClass>
            </daml:Restriction>
        </rdfs:subClassOf>
    </daml:Class>
    <daml:ObjectProperty rdf:about="file:C:\Ralf\Ind-Examples\ex1.daml#R">
        <rdfs:label>R</rdfs:label>
        <rdfs:comment><![CDATA[]]></rdfs:comment>
        <oiled:creationDate><![CDATA[2002-09-27T19:09:49Z]]></oiled:creationDate>
        <oiled:creator><![CDATA[MO]]></oiled:creator>
```

```
    </daml:ObjectProperty>
    <rdf:Description rdf:about="file:C:\Ralf\Ind-Examples\ex1.daml#i">
        <rdfs:comment><![CDATA[]]></rdfs:comment>
        <oiled:creationDate><![CDATA[2002-09-27T19:17:36Z]]></oiled:creationDate>
        <oiled:creator><![CDATA[MO]]></oiled:creator>
        <rdf:type>
            <daml:Class rdf:about="file:C:\Ralf\Ind-Examples\ex1.daml#c"/>
        </rdf:type>
    </rdf:Description>
    <rdf:Description rdf:about="file:C:\Ralf\Ind-Examples\ex1.daml#j">
        <rdfs:comment><![CDATA[]]></rdfs:comment>
        <oiled:creationDate><![CDATA[2002-09-27T19:09:36Z]]></oiled:creationDate>
        <oiled:creator><![CDATA[MO]]></oiled:creator>
        <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
    </rdf:Description>
</rdf:RDF>
```

Dealing with indviduals 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 DAML names must be enclosed with bars (`|`).

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

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

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

The following knowledge base (for the DAML version see file `ex2.daml` 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 DAML file `ex2.daml` in the examples folder.

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

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

Due to the problems described in this section, processing DAML files with individuals in concept terms with the RACER system is not recommended. However, at the time of this writing there is no other system know that provides complete reasoning on nominals.

5 The RDF/OWL interface

RACER can also process OWL documents. Documents are interpreted w.r.t. the OWL DL languages. The implementation is prototypical. Similar restrictions as describe for DAML documents apply (see the documentation on `owl-read-file` and `owl-read-document`).

6 Knowledge Base Management Functions

A knowledge base is just a tuple consisting of a TBox and an associated ABox. Note that a TBox and its associated ABox may have the same name. This section documents the functions for managing TBoxes and ABoxes and for specifying queries.

Racer provides a default knowledge base with a TBox called `default` and an associated ABox with the same name.

---

in-knowledge-base *macro*

Description: This form is an abbreviation for the sequence:
`(in-tbox` *TBN*`)`
`(in-abox` *ABN TBN*`)`. See the appropriate documentation for these functions.

Syntax: Two forms are possible:
`(in-knowledge-base` *TBN* `&optional` *ABN*`)` or
`(in-knowledge-base` *TBN* `&key (`*init* `t))`

Arguments: *TBN* - TBox name

ABN - ABox name

init - `t` or `nil`

Remarks: If no ABox is specified an ABox with the same name as the TBox is created (or initialized if already present). The ABox is associated with the TBox. If the keyword `:init` is specified with value `nil` no new knowledge base is created but just the current TBox and ABox is set. If `:init` is specified, no ABox name may be given.

Examples:
```
(in-knowledge-base peanuts peanuts-characters)
  (in-knowledge-base peanuts)
  (in-knowledge-base peanuts :init nil)
```

racer-read-file *function*

Description: A file in RACER format (as described in this document) containing TBox and/or ABox declarations is loaded.

Syntax: `(racer-read-file` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Examples: `(racer-read-file "kbs/test.lisp")`

See also: Function `include-kb`

racer-read-document *function*

Description: A file in RACER format (as described in this document) containing TBox and/or ABox declarations is loaded.

Syntax: `(racer-read-document` *URL*`)`

Arguments: *URL* - is the URL of a text document with RACER statements.

Remarks: The URL can also be a file URL. In this case, racer-read-file is used on the pathname of the URL.

Examples:
```
(racer-read-document "http://www.fh-wedel.de/mo/test.lisp")
  (racer-read-document "file:///home/mo/kbs/test.lisp")
```

See also: Function `racer-read-file`

include-kb *function*

Description: A file in RACER format (as described in this document) containing TBox and/or ABox declarations is loaded. The function include is used for partitioning a TBox or ABox into several files.

Syntax: `(include-kb` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Examples: `(include-kb "project:onto-kb;my-knowledge-base.lisp")`

See also: Function `racer-read-file`

daml-read-file *function*

Description: A file in DAML format (e.g., produced OilEd) is loaded and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(daml-read-file` *pathname* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *pathname* - is the pathname of a file

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the file specified in the *pathname* argument (without file type).

Examples: `(daml-read-file "oiled:ontologies;madcows.daml")` reads the file `"oiled:ontologies;madcows.daml"` and creates a TBox `madcows` and an associated ABox `madcows`.

daml-read-document *function*

Description: A text document in DAML format (e.g., produced OilEd) is loaded from a web server and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(daml-read-document` *URL* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *URL* - is the URL of a text document

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the document name specified in the *URL* argument (without file type).

Examples: `(daml-read-document "http://www.fh-wedel.de/mo/madcows.daml")` reads the specified text document from the corresponding web server and creates a TBox `madcows` and an associated ABox `madcows`. A file URL may also be specified `(daml-read-document "file://mo/madcows.daml")`

owl-read-file *function*

Description: A file in OWL format (e.g., produced OilEd) is loaded and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(owl-read-file` *pathname* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *pathname* - is the pathname of a file

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the file specified in the *pathname* argument (without file type).

Examples: `(owl-read-file "oiled:ontologies;madcows.owl")` reads the file `"oiled:ontologies;madcows.owl"` and creates a TBox `madcows` and an associated ABox `madcows`.

owl-read-document *function*

Description: A text document in OWL format (e.g., produced OilEd) is loaded from a web server and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(owl-read-document` *URL* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *URL* - is the URL of a text document

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the document name specified in the *URL* argument (without file type).

Examples: `(owl-read-document "http://www.fh-wedel.de/mo/madcows.owl")` reads the specified text document from the corresponding web server and creates a TBox `madcows` and an associated ABox `madcows`. A file URL may also be specified `(owl-read-document "file://mo/madcows.owl")`

mirror *function*

Description: If you are offline, importing OWL or DAML ontologies may cause problems. However, editing documents and inserting local URLs for ontologies is inconvenient. Therefore, Racer provides a facility to declare local mirror URLs for ontology URLs

Syntax: `(mirror` *URL mirror − URL*`)`

Arguments: *URL* - a URL used to refer to an ontology in a DAML-OIL or OWL document

mirror − URL - a URL that refers to the same ontology. Possibly, a file URL may be supplied.

kb-ontologies *function*

Description: A document in DAML+OIL or OWL format can import other ontologies. With this function one can retrieve all ontologies that were imported into the specified knowledge base

Syntax: `(kb-ontologies` *KBN*`)`

Arguments: *KBN* - is the name of the knowledge base.

save-kb *function*

Description: If a pathname is specified, a TBox is saved to a file. In case a stream is specified the TBox is written to the stream (the stream must already be open) and the keywords *if-exists* and *if-does-not-exist* are ignored.

Syntax:
```
(save-kb pathname-or-stream
  &key (tbox *current-tbox*) (abox *current-abox*)
  (syntax :krss) (if-exists :supersede)
  (if-does-not-exist :create)
  (uri "")
  (ns0 ""))
```

Arguments: *pathname-or-stream* - is the pathname of a file or is an output stream

tbox - TBox name or TBox object

abox - ABox name or ABox object

syntax - indicates the syntax of the KB to be generated. Possible values for the *syntax* argmument are `:krss` (the default), `:xml`, or `:daml`. Note that concerning KRSS only a KRSS-like syntax is supported by RACER. Therefore, instead of `:krss` it is also possible to specify `:racer`.

if-exists - specifies the action taken if a file with the specified name already exists. All keywords for the Lisp function `with-open-file` are supported. The default is `:supersede`.

if-does-not-exist - specifies the action taken if a file with the specified name does not yet exist. All keywords for the Lisp function `with-open-file` are supported. The default is `:create`.

uri - The keyword `:uri` specifies the URI prefix for names. It is only available if syntax `:daml` is specified. This argument is useful in combination with OilEd. See the OilEd documentation.

ns0 - The keyword `:uri` is also provided for generating DAML files to be processed with OilEd. The keyword `:ns0` specifies the name of the OilEd namespace 0. This keyword is important for the ABox part. If the value of `:uri` is `/home/user/test#`, the value of `:ns0` should probably be `/home/user/`. Some experimentation might be necessary to find the correct values for `:uri` and `:ns0` to be used with OilEd.

Examples:
```
(save-kb "project:onto-kb;my-knowledge-base.krss"
  :syntax :krss
  :tbox 'family
  :abox 'smith-family)
```

```
(save-kb "family.daml" :syntax :daml

  :tbox 'family
  :abox 'smith-family
  :uri "http://www.fh-wedel.de/family.daml")
  :ns0 "http://www.fh-wedel.de/")
```

6.1 TBox Management

If RACER is started, there exists a TBox named DEFAULT, which is set to the current TBox.

---

in-tbox *macro*

Description: The TBox with the specified name is taken or a new TBox with that name is generated and bound to the variable `*current-tbox*`.

Syntax: `(in-tbox` *TBN* `&key (`*init* `t))`

Arguments: *TBN* - is the name of the TBox.

init - boolean indicating if the TBox should be initialized.

Values: TBox object named *TBN*

Remarks: Usually this macro is used at top of a file containing a TBox. This macro can also be used to create new TBoxes.

The specified TBox is the `*current-tbox*` until `in-tbox` is called again or the variable `*current-tbox*` is manipulated directly.

Examples:
```
(in-tbox peanuts)
(implies Piano-Player Character)
                ⋮
```

See also: Macro `signature` on page 68.

---

init-tbox *function*

Description: Generates a new TBox or initializes an existing TBox and binds it to the variable `*current-tbox*`. During the initialization all user-defined concept axioms and role declarations are deleted, only the concepts `*top*` and `*bottom*` remain in the TBox.

Syntax: `(init-tbox` *tbox*`)`

Arguments: *tbox* - TBox object

Values: *tbox*

Remarks: This is the way to create a new TBox object.

signature *macro*

Description: Defines the signature for a knowledge base.

If any keyword except *individuals* or *objects* is used, the `*current-tbox*` is initialized and the signature is defined for it.

If the keyword *individuals* or *objects* is used, the `*current-abox*` is initialized. If all keywords are used, the `*current-abox*` and its TBox are both initialized.

Syntax: `(signature &key (`*atomic-concepts*` nil) (`*roles*` nil) (`*transitive-roles*` nil) (`*features*` nil) (`*attributes*` nil) (`*individuals*` nil) (`*objects*` nil))`

Arguments: *atomic-concepts* - is a list of all the concept names, specifying $\mathcal{C}$.

roles - is a list of role declarations.

transitive-roles - is a list of transitive role declarations.

features - is a list of feature declarations.

attributes - is a list of attributes declarations.

individuals - is a list of individual names.

objects - is a list of object names.

Remarks: Usually this macro is used at top of a file directly after the macro `in-knowledge-base`, `in-tbox` or `in-abox`.

Actually it is not necessary in RACER to specify the signature, but it helps to avoid errors due to typos.

Examples: Signature for a TBox:

```
(signature
  :atomic-concepts (Character Baseball-Player...)
  :roles ((has-pet)
    (has-dog :parents (has-pet) :domain human :range dog)
    (has-coach :feature t))
  :attributes ((integer has-age) (real has-weight)))
```

Signature for an ABox:

```
(signature
  :individuals (Charlie-Brown Snoopy ...)
  :objects (age-of-snoopy ...))
```

Signature for a TBox and an ABox:

```
(signature
  :atomic-concepts (Character Baseball-Player...)
  :roles ((has-pet)
    (has-dog :parents (has-pet) :domain human :range dog)
    (has-coach :feature t))
  :attributes ((integer has-age) (real has-weight))
  :individuals (Charlie-Brown Snoopy ...)
  :objects (age-of-snoopy ...))
```

See also: Section Sample Session, on page 15 and page 17.
For role definitions see `define-primitive-role`, on page 86, for feature definitions see `define-primitive-attribute`, on page 86, for attribute definitions see `define-concrete-domain-attribute`, on page 92.

ensure-tbox-signature *function*

Description: Defines the signature for a TBox and initializes the TBox.

Syntax: `(ensure-tbox-signature` *tbox* `&key (`*atomic-concepts* `nil) (`*roles* `nil) (`*transitive-roles* `nil) (`*features* `nil) (`*attributes* `nil))`

Arguments: *tbox* - is a TBox name or a TBox object.

atomic-concepts - is a list of all the concept names.

roles - is a list of all role declarations.

transitive-roles - is a list of transitive role declarations.

features - is a list of feature declarations.

attributes - is a list of attributes declarations.

See also: Definition of macro `signature`.

tbox-signature *function*

Description: Gets the signature for a TBox.

Syntax: `(tbox-signature &optional` *tbox*`)`

Arguments: *tbox* - is a TBox name or a TBox object.

current-tbox *special-variable*

Description: The variable `*current-tbox*` refers to the current TBox object. It is set by the function `init-tbox` or by the macro `in-tbox`.

Remarks: The variable is only supported in the Lisp version of RACER. In the server version use the function `current-tbox`

See also: Definition of `current-tbox`

current-tbox *function*

Description: The function returns the value of the variable `*current-tbox*`.

Syntax: `(current-tbox)`

Arguments:

See also: Definition of `*current-tbox*`

save-tbox *function*

Description: If a pathname is specified, a TBox is saved to a file. In case a stream is specified the TBox is written to the stream (the stream must already be open) and the keywords *if-exists* and *if-does-not-exist* are ignored.

Syntax: `(save-tbox` *pathname-or-stream* `&optional (`*tbox* `*current-tbox*)`
`&key (`*syntax* `:krss) (`*transformed* `nil) (`*if-exists* `:supersede)`
`(`*if-does-not-exist* `:create)`
`(`*uri* `"")`
`(`*ns0* `""))`

Arguments: *pathname-or-stream* - is the pathname of a file or is an output stream

tbox - TBox object

syntax - indicates the syntax of the KB to be generated. Possible values for the *syntax* argmument are `:krss` (the default), `:xml`, or `:daml`. Note that only a KRSS-like syntax is supported by RACER. Therefore, instead of `:krss` it is also possible to specify `:racer`.

if-exists - specifies the action taken if a file with the specified name already exists. All keywords for the Lisp function `with-open-file` are supported. The default is `:supersede`.

if-does-not-exist - specifies the action taken if a file with the specified name does not yet exist. All keywords for the Lisp function `with-open-file` are supported. The default is `:create`.

uri - The keyword `:uri` specifies the URI prefix for names. It is only available if syntax `:daml` is specified. This argument is useful in combination with OilEd. See the OilEd documentation.

ns0 - The keyword `:uri` is also provided for generating DAML files to be processed with OilEd. The keyword `:ns0` specifies the name of the OilEd namespace 0. This keyword is important for the ABox part. If the value of `:uri` is `/home/user/test#`, the value of `:ns0` should probably be `/home/user/`. Some experimentation might be necessary to find the correct values for `:uri` and `:ns0` to be used with OilEd.

Values: TBox object

Remarks: A file may contain several TBoxes.
The usual way to load a TBox file is to use the Lisp function `load`.
If the server version is used, it must have been started with the option -u in order to have this function available.

Examples:
```
(save-tbox "project:TBoxes;tbox-one.lisp")
(save-tbox "project:TBoxes;final-tbox.lisp"
  (find-tbox 'tbox-one) :if-exists :error)
```

forget-tbox *function*

Description: Delete the specified TBox from the list of all TBoxes. Usually this enables the garbage collector to recycle the memory used by this TBox.

Syntax: `(forget-tbox` *tbox*`)`

Arguments: *tbox* - is a TBox object or TBox name.

Values: List containing the name of the removed TBox and a list of names of optionally removed ABoxes

Remarks: All ABoxes referencing the specified TBox are also deleted.

Examples:
```
(forget-tbox 'smith-family)
```

delete-tbox *macro*

Description: Delete the specified TBox from the list of all TBoxes. Usually this enables the garbage collector to recycle the memory used by this TBox.

Syntax: `(delete-tbox` *TBN*`)`

Arguments: *TBN* - is a TBox name.

Values: List containing the name of the removed TBox and a list of names of optionally removed ABoxes

Remarks: Calls `forget-tbox`

Examples: `(delete-tbox smith-family)`

delete-all-tboxes *function*

Description: Delete all known TBoxes except the default TBox called `default`. Usually this enables the garbage collector to recycle the memory used by these TBoxes.

Syntax: `(delete-all-tboxes)`

Values: List containing the names of the removed TBoxes and a list of names of optionally removed ABoxes

Remarks: All ABoxes are also deleted.

create-tbox-clone *function*

Description: Returns a new TBox object which is a clone of the given TBox. The clone keeps all declarations from its original but it is otherwise fresh, i.e., new declarations can be added. This function allows one to create new TBox versions without the need to reload the already known declarations.

Syntax: `(create-tbox-clone` *tbox* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *tbox* - is a TBox name or a TBox object.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *tbox* is generated otherwise.

overwrite - if bound to `t` an existing TBox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if a TBox with the name given by *new-name* is found.

Values: TBox object

Remarks: The variable `*current-tbox*` is set to the result of this function.

Examples:
```
(create-tbox-clone 'my-TBox)
(create-tbox-clone 'my-TBox :new-name 'my-clone :overwrite t)
```

clone-tbox *macro*

Description: Returns a new TBox object which is a clone of the given TBox. The clone keeps all declarations from its original but it is otherwise fresh, i.e., new declarations can be added. This function allows one to create new TBox versions without the need to reload the already known declarations.

Syntax: `(clone-tbox` *TBN* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *TBN* - is a TBox name.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *tbox* is generated otherwise.

overwrite - if bound to `t` an existing TBox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if a TBox with the name given by *new-name* is found.

Values: TBox object

Remarks: The function `create-tbox-clone` is called.

Examples:
```
(clone-tbox my-TBox)
(clone-tbox my-TBox :new-name my-clone :overwrite t)
```

See also: Function `create-tbox-clone` on page 72.

find-tbox *function*

Description: Returns a TBox object with the given name among all TBoxes.

Syntax: `(find-tbox` *TBN* `&optional (`*errorp* `t))`

Arguments: *TBN* - is the name of the TBox to be found.
errorp - if bound to `t` an error is signaled if the TBox is not found.

Values: TBox object

Remarks: This function can also be used to get rid of TBoxes or to rename TBoxes as shown in the examples.

Examples: `(find-tbox 'my-TBox)`

Getting rid of a TBox:
`(setf (find-tbox 'tbox1) nil)`

Renaming a TBox:
`(setf (find-tbox 'tbox2) tbox1)`

tbox-name *function*

Description: Finds the name of the given TBox object.

Syntax: `(tbox-name` *tbox*`)`

Arguments: *tbox* - TBox object

Values: TBox name

Remarks: This function is only needed in the Lisp version.

clear-default-tbox *function*

Description: This function initializes the default TBox.

Syntax: `(clear-default-tbox)`

Arguments:

associated-aboxes *function*

Description: Returns a list of ABoxes or ABox names which are defined wrt. the TBox specified as a parameter.

Syntax: `(associated-aboxes` *TBN*`)`

Arguments: *TBN* - is the name of a TBox.

Values: List of ABox objects

xml-read-tbox-file *function*

Description: A file in XML format containing TBox declarations is parsed and the resulting TBox is returned.

Syntax: `(xml-read-tbox-file` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Values: TBox object

Remarks: Only XML descriptions which correspond the so-called FaCT DTD are parsed, everyting else is ignored.

Examples: `(xml-read-tbox-file "project:TBoxes;tbox-one.xml")`

rdfs-read-tbox-file *function*

Description: A file in RDFS format containing TBox declarations is parsed and the resulting TBox is returned. The name of the TBox is the filename without file type.

Syntax: `(rdfs-read-tbox-file` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Values: TBox object

Remarks: If the file to be read also contains RDF descriptions, use the function `daml-read-file` instead. The RDF descriptions are represented using appropriate ABox assertions. The function `rdfs-read-tbox-file` is supported for backward compatibility.

Examples: `(rdfs-read-tbox-file "project:TBoxes;tbox-one.rdfs")`

6.2 ABox Management

If RACER is started, there exists a ABox named DEFAULT, which is set to the current ABox.

in-abox *macro*

Description: The ABox with this name is taken or generated and bound to `*current-abox*`. If a TBox is specified, the ABox is also initialized.

Syntax: `(in-abox` *ABN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *ABN* - ABox name

TBN - name of the TBox to be associated with the ABox.

Values: ABox object named *ABN*

Remarks: If the specified TBox does not exist, an error is signaled.

Usually this macro is used at top of a file containing an ABox. This macro can also be used to create new ABoxes. If the ABox is to be continued in another file, the TBox must not be specified again.

The specified ABox is the `*current-abox*` until `in-abox` is called again or the variable `*current-abox*` is manipulated directly. The TBox of the ABox is made the `*current-tbox*`.

Examples:
```
(in-abox peanuts-characters peanuts)
(instance Schroeder Piano-Player)
```
⋮

See also: Macro `signature` on page 68.

init-abox *function*

Description: Initializes an existing ABox or generates a new ABox and binds it to the variable `*current-abox*`. During the initialization all assertions and the link to the referenced TBox are deleted.

Syntax: `(init-abox` *abox* `&optional (`*tbox* `*current-tbox*))`

Arguments: *abox* - ABox object to initialize

tbox - TBox object associated with the ABox

Values: *abox*

Remarks: The *tbox* has to already exist before it can be referred to by `init-abox`.

ensure-abox-signature *function*

Description: Defines the signature for an ABox and initializes the ABox.

Syntax: `(ensure-abox-signature` *abox* `&key (`*individuals* `nil) (`*objects* `nil))`

Arguments: *abox* - ABox object
individuals - is a list of individual names.
objects - is a list of concrete domain object names.

See also: Macro `signature` on page 68 is the macro counterpart. It allows to specify a signature for an ABox and a TBox with one call.

abox-signature *function*

Description: Gets the signature for an ABox.

Syntax: `(abox-signature &optional` *ABN*`)`

Arguments: *ABN* - is an ABox name

kb-signature *function*

Description: Gets the signature for a knowledge base.

Syntax: `(kb-signature &optional` *KBN*`)`

Arguments: *KBN* - is a name for a knowledge base.

current-abox *special-variable*

Description: The variable `*current-abox*` refers to the current ABox object. It is set by the function `init-abox` or by the macros `in-abox` and `in-knwoledge-base`.

Remarks: The variable is only provided in the Lisp version.

See also: Definition of `current-abox`

current-abox *function*

Description: Returns the value of the variable `*current-abox*`

Syntax: `(current-abox)`

Arguments:

See also: Definition of `*current-abox*`

save-abox *function*

Description: If a pathname is specified, an ABox is saved to a file. In case a stream is specified, the ABox is written to the stream (the stream must already be open) and the keywords *if-exists* and *if-does-not-exist* are ignored.

Syntax: `(save-abox` *pathname-or-stream* `&optional (`*abox* `*current-abox*) &key (`*syntax* `:krss) (`*transformed* `nil) (`*if-exists* `:supersede) (`*if-does-not-exist* `:create))`

Arguments: *pathname-or-stream* - is the name of the file or an output stream.

abox - ABox object

syntax - indicates the syntax of the TBox. Possible value for the *syntax* argmument are :krss (the default), :xml, or :daml.

transformed - if bound to `t` the ABox is saved in the format it has after preprocessing by RACER.

if-exists - specifies the action taken if a file with the specified name already exists. All keywords for the Lisp function `with-open-file` are supported. The default is `:supersede`.

if-does-not-exist - specifies the action taken if a file with the specified name does not yet exist. All keywords for the Lisp function `with-open-file` are supported. The default is `:create`.

Values: ABox object

Remarks: A file may contain several ABoxes.
The usual way to load an ABox file is to use the Lisp function `load`.
If the server version is used, it must have been started with the option -u in order to have this function available.

Examples:
```
(save-abox "project:ABoxes;abox-one.lisp")
(save-abox "project:ABoxes;final-abox.lisp"
  (find-abox 'abox-one) :if-exists :error)
```

forget-abox *function*

Description: Delete the specified ABox from the list of all ABoxes. Usually this enables the garbage collector to recycle the memory used by this ABox.

Syntax: `(forget-abox` *abox*`)`

Arguments: *abox* - is a ABox object or ABox name.

Values: The name of the removed ABox

Examples:
```
(forget-abox 'family)
```

delete-abox *macro*

Description: Delete the specified ABox from the list of all ABoxes. Usually this enables the garbage collector to recycle the memory used by this ABox.

Syntax: `(delete-abox` *ABN*`)`

Arguments: *ABN* - is a ABox name.

Values: The name of the removed ABox

Remarks: Calls `forget-abox`

Examples:
```
(delete-abox family)
```

delete-all-aboxes *function*

Description: Delete all known ABoxes. Usually this enables the garbage collector to recycle the memory used by these ABoxes.

Syntax: `(delete-all-aboxes)`

Values: List containing the names of the removed ABoxes

create-abox-clone *function*

Description: Returns a new ABox object which is a clone of the given ABox. The clone keeps the assertions and the state from its original but new declarations can be added without modifying the original ABox. This function allows one to create new ABox versions without the need to reload (and reprocess) the already known assertions.

Syntax: `(create-abox-clone` *abox* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *abox* - is an ABox name or an ABox object.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *abox* is generated otherwise.

overwrite - if bound to `t` an existing ABox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if an ABox with the name given by *new-name* is found.

Values: ABox object

Remarks: The variable `*current-abox*` is set to the result of this function.

Examples:
```
(create-abox-clone 'my-ABox)
(create-abox-clone 'my-ABox :new-name 'abox-clone :overwrite t)
```

clone-abox *macro*

Description: Returns a new ABox object which is a clone of the given ABox. The clone keeps the assertions and the state from its original but new declarations can be added without modifying the original ABox. This function allows one to create new ABox versions without the need to reload (and reprocess) the already known assertions.

Syntax: `(clone-abox` *ABN* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *ABN* - is an ABox name.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *abox* is generated otherwise.

overwrite - if bound to `t` an existing ABox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if an ABox with the name given by *new-name* is found.

Values: ABox object

Remarks: The function `create-abox-clone` is called.

Examples:
```
(clone-abox my-ABox)
(clone-abox my-ABox :new-name abox-clone :overwrite t)
```

See also: Function `create-abox-clone` on page 79.

find-abox *function*

Description: Finds an ABox object with a given name among all ABoxes.

Syntax: `(find-abox` *ABN* `&optional (`*errorp* `t))`

Arguments: *ABN* - is the name of the ABox to be found.

errorp - if bound to `t` an error is signaled if the ABox is not found.

Values: ABox object

Remarks: This function can also be used to delete ABoxes or rename ABoxes as shown in the examples.

Examples:
```
(find-tbox 'my-ABox)
```

Get rid of an ABox, i.e. make the ABox garbage collectible:
```
(setf (find-abox 'abox1) nil)
```

Renaming an ABox:
```
(setf (find-abox 'abox2) abox1)
```

abox-name *function*

Description: Finds the name of the given ABox object.

Syntax: `(abox-name` *abox*`)`

Arguments: *abox* - ABox object

Values: ABox name

Remarks: Only available in the Lisp version.

Examples: `(abox-name (find-abox 'my-ABox))`

tbox *function*

Description: Gets the associated TBox for an ABox.

Syntax: `(tbox` *abox*`)`

Arguments: *abox* - ABox object

Values: TBox object

Remarks: This function is provided in the Lisp version only.

associated-tbox *function*

Description: Gets the associated TBox for an ABox.

Syntax: `(associated-tbox` *abox*`)`

Arguments: *abox* - ABox object

Values: TBox object

Remarks: This function is provided in the server version only.

set-associated-tbox *function*

Description: Sets the associated TBox for an ABox.

Syntax: `(set-associated-tbox` *ABN TBN*`)`

Arguments: *ABN* - ABox name
TBN - TBox name

Values: TBox object

Remarks: This function is provided in the server version only.

7 Knowledge Base Declarations

Knowledge base declarations include concept axioms and role declarations for the TBox and the assertions for the ABox. The TBox object and the ABox object must exist before the functions for knowledge base declarations can be used. The order of axioms and assertions does not matter because forward references can be handled by RACER.

The macros for knowledge base declarations add the concept axioms and role declarations to the `*current-tbox*` and the assertions to the `*current-abox*`.

7.1 Built-in Concepts

---

top, top *concept*

Description: The name of most general concept of each TBox, the top concept ($\top$).

Syntax: `*top*`

Remarks: The concepts `*top*` and `top` are synonyms. These concepts are elements of every TBox.

---

bottom, bottom *concept*

Description: The name of the incoherent concept, the bottom concept ($\bot$).

Syntax: `*bottom*`

Remarks: The concepts `*bottom*` and `bottom` are synonyms. These concepts are elements of every TBox.

7.2 Concept Axioms

This section documents the macros and functions for specifying concept axioms. The different concept axioms were already introduced in section 3.2.

Please note that the concept axioms `define-primitive-concept`, `define-concept` and `define-disjoint-primitive-concept` have the semantics given in the KRSS specification only if they are the only concept axiom defining the concept *CN* in the terminology. This is not checked by the RACER system.

implies *macro*

Description: Defines a GCI between C_1 and C_2.

Syntax: `(implies` C_1 C_2`)`

Arguments: C_1, C_2 - concept term

Remarks: C_1 states necessary conditions for C_2. This kind of facility is an addendum to the KRSS specification.

Examples:
```
(implies Grandmother (and Mother Female))
(implies
  (and (some has-sibling Sister) (some has-sibling Twin)
    (exactly 1 has-sibling))
  (and Twin (all has-sibling Twin-sister)))
```

equivalent *macro*

Description: States the equality between two concept terms.

Syntax: `(equivalent` C_1 C_2`)`

Arguments: C_1, C_2 - concept term

Remarks: This kind of concept axiom is an addendum to the KRSS specification.

Examples:
```
(equivalent Grandmother
  (and Mother (some has-child Parent)))
(equivalent
  (and polygon (exactly 4 has-angle))
  (and polygon (exactly 4 has-edges)))
```

disjoint *macro*

Description: This axiom states the disjointness of a set of concepts.

Syntax: `(disjoint` CN_1 `...`CN_n`)`

Arguments: CN_1,..., CN_n - concept names

Examples:
```
(disjoint Yellow Red Blue)
(disjoint January February ...November December))
```

define-primitive-concept *KRSS macro*

Description: Defines a primitive concept.

Syntax: `(define-primitive-concept` *CN C*`)`

Arguments: *CN* - concept name

C - concept term

Remarks: *C* states the necessary conditions for *CN*.

Examples:
```
(define-primitive-concept Grandmother (and Mother Female))
(define-primitive-concept Father Parent)
```

define-concept *KRSS macro*

Description: Defines a concept.

Syntax: `(define-concept` *CN C*`)`

Arguments: *CN* - concept name

C - concept term

Remarks: Please note that in RACER, definitions of a concept do not have to be unique. Several definitions may be given for the same concept.

Examples:
```
(define-concept Grandmother
  (and Mother (some has-child Parent)))
```

define-disjoint-primitive-concept *KRSS macro*

Description: This axiom states the disjointness of a group of concepts.

Syntax: `(define-disjoint-primitive-concept` *CN GNL C*`)`

Arguments: *CN* - concept name

GNL - group name list, which lists all groups to which *CN* belongs to (among other concepts). All elements of each group are declared to be disjoint.

C - concept term, that is implied by *CN*.

Remarks: This function is just supplied to be compatible with the KRSS.

Examples:
```
(define-disjoint-primitive-concept January
  (Month) (exactly 31 has-days))
(define-disjoint-primitive-concept February
  (Month) (and (at-least 28 has-days) (at-most 29 has-days)))
                    ⋮
```

add-concept-axiom *function*

Description: This function adds a concept axiom to a TBox.

Syntax: `(add-concept-axiom` *tbox* C_1 C_2 `&key (`*inclusion-p* `nil))`

Arguments: *tbox* - TBox object

C_1, C_2 - concept term

inclusion-p - boolean indicating if the concept axiom is an inclusion axiom (GCI) or an equality axiom. The default is to state an inclusion.

Values: *tbox*

Remarks: RACER imposes no constraints on the sequence of concept axiom declarations with `add-concept-axiom`, i.e. forward references to atomic concepts for which other concept axioms are added later are supported in RACER.

add-disjointness-axiom *function*

Description: This function adds a disjointness concept axiom to a TBox.

Syntax: `(add-disjointness-axiom` *tbox CN GN*`)`

Arguments: *tbox* - TBox object

CN - concept name

GN - group name

Values: *tbox*

7.3 Role Declarations

Roles can be declared with the following statements.

define-primitive-role *KRSS macro (with changes)*

Description: Defines a role.

Syntax: `(define-primitive-role` *RN* `&key (`*transitive* `nil) (`*feature* `nil) (`*symmetric* `nil) (`*reflexive* `nil) (`*inverse* `nil) (`*domain* `nil) (`*range* `nil) (`*parents* `nil))`

Arguments: *RN* - role name

transitive - if bound to `t` declares that the new role is transitive.

feature - if bound to `t` declares that the new role is a feature.

symmetric - if bound to `t` declares that the new role is a symmetric. This is equivalent to declaring that the new role's inverse is the role itself.

reflexive - if bound to `t` declares that the new role is reflexive (currently only supported for $\mathcal{ALCH}$). If *feature* is bound to `t`, the value of *reflexive* is ignored.

inverse - provides a name for the inverse role of *RN*. This is equivalent to `(inv` *RN*`)`. The inverse role of *RN* has no user-defined name, if *inverse* is bound to `nil`.

domain - provides a concept term defining the domain of role *RN*. This is equivalent to adding the axiom `(implies (at-least 1` *RN*`)` *C*`)` if *domain* is bound to the concept term *C*. No domain is declared if *domain* is bound to `nil`.

range - provides a concept term defining the range of role *RN*. This is equivalent to adding the axiom `(implies *top* (all` *RN* *D*`))` if *range* is bound to the concept term *D*. No range is declared if *range* is bound to `nil`.

parents - provides a list of superroles for the new role. The role *RN* has no superroles, if *parents* is bound to `nil`.
If only a single superrole is specified, the keyword `:parent` may alternatively be used, see the examples.

Remarks: This function combines several KRSS functions for defining properties of a role. For example the conjunction of roles can be expressed as shown in the first example below.

A role that is declared to be a feature cannot be transitive. A role with a feature as a parent has to be a feature itself. A role with transitive subroles may not be used in number restrictions.

Examples:
```
(define-primitive-role conjunctive-role :parents (R-1 ...R-n))
(define-primitive-role has-descendant :transitive t
  :inverse descendant-of :parent has-child)
(define-primitive-role has-children :inverse has-parents
  :domain parent :range children))
```

See also: Macro `signature` on page 68.
Section 3.3 and Figure 29, on page 51 for domain and range restrictions.

define-primitive-attribute *KRSS macro (with changes)*

Description: Defines an attribute.

Syntax: `(define-primitive-attribute` *AN* `&key (`*symmetric* `nil)`
`(`*inverse* `nil) (`*domain* `nil) (`*range* `nil) (`*parents* `nil))`

Arguments: *AN* - attribute name

symmetric - if bound to `t` declares that the new role is a symmetric. This is equivalent to declaring that the new role's inverse is the role itself.

inverse - provides a name for the inverse role of *AN*. This is equivalent to `(inv` *AN*`)`. The inverse role of *AN* has no user-defined name, if *inverse* is bound to `nil`.

domain - provides a concept term defining the domain of role *AN*. This is equivalent to adding the axiom `(implies (at-least 1` *AN*`)` *C*`)` if *domain* is bound to the concept term *C*. No domain is declared if *domain* is bound to `nil`.

range - provides a concept term defining the range of role *AN*. This is equivalent to adding the axiom `(implies *top* (all` *AN* *D*`))` if *range* is bound to the concept term *D*. No range is declared if *range* is bound to `nil`.

parents - provides a list of superroles for the new role. The role *AN* has no superroles, if *parents* is bound to `nil`.
If only a single superrole is specified, the keyword `:parent` may alternatively be used, see examples.

Remarks: This macro is supplied to be compatible with the KRSS specification. It is redundant since the macro `define-primitive-role` can be used with :*feature* `t`. This function combines several KRSS functions for defining properties of an attribute.

An attribute cannot be transitive. A role with a feature as a parent has to be a feature itself.

Examples:
```
(define-primitive-attribute has-mother
  :domain child :range mother :parents (has-parents))
(define-primitive-attribute has-best-friend
  :inverse best-friend-of :parent has-friends)
```

See also: Macro `signature` on page 68.
Section 3.3 and Figure 29, on page 51 for domain and range restrictions.

add-role-axioms *function*

Description: Adds a role to a TBox.

Syntax: `(add-role-axioms` *tbox RN* `&key (`*cd-attribute* `nil) (`*transitive* `nil) (`*feature* `nil) (`*symmetric* `nil) (`*reflexive* `nil) (`*inverse* `nil) (`*domain* `nil) (`*range* `nil) (`*parents* `nil))`

Arguments: *tbox* - TBox object to which the role is added.

RN - role name

cd-attribute - may be either `integer` or `real`.

transitive - if bound to `t` declares that *RN* is transitive.

feature - if bound to `t` declares that *RN* is a feature.

symmetric - if bound to `t` declares that *RN* is a symmetric. This is equivalent to declaring that the new role's inverse is the role itself.

reflexive - if bound to `t` declares that *RN* is reflexive (currently only supported for $\mathcal{ALCH}$). If *feature* is bound to `t`, the value of *reflexive* is ignored.

inverse - provides a name for the inverse role of *RN* (is equivalent to `(inv` *RN*`)`). The inverse role of *RN* has no user-defined name, if *inverse* is bound to `nil`.

domain - provides a concept term defining the domain of role *RN* (equivalent to adding the axiom `(implies (at-least 1` *RN*`)` *C*`)` if *domain* is bound to the concept term *C*. No domain is declared if *domain* is bound to `nil`.

range - provides a concept term defining the range of role *RN* (equivalent to adding the axiom `(implies *top* (all` *RN D*`))` if *range* is bound to the concept term *D*. No range is declared if *range* is bound to `nil`.

parents - providing a single role or a list of superroles for the new role. The role *RN* has no superroles, if *parents* is bound to `nil`.

Values: *tbox*

Remarks: For each role *RN* there may be only one call to `add-role-axioms` per TBox.

See also: Section 3.3 and Figure 29, on page 51 for domain and range restrictions.

functional *macro*

Description: States that a role is to be interpreted as functional.

Syntax: `(functional` *RN*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
TBN - TBox name

Remarks: States that a role is to be interpreted as functional.

role-is-functional *function*

Description: States that a role is to be interpreted as functional.

Syntax: `(role-is-functional` *RN*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
TBN - TBox name

transitive *macro*

Description: States that a role is to be interpreted as transitive.

Syntax: `(transitive` *RN*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
TBN - TBox name

role-is-transitive *function*

Description: States that a role is to be interpreted as transitive.

Syntax: `(role-is-transitive` *RN*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
TBN - TBox name

inverse *macro*

Description: Defines a name for the inverse of a role.

Syntax: `(inverse` *RN inverse − role*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
inverse − role - inverse role of the Form `(inv` *RN*`)`
TBN - TBox name

inverse-of-role *function*

Description: Defines a name for the inverse of a role.

Syntax: `(inverse-of-role` *RN invese − role*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
inverse − role - inverse role of the Form `(inv` *RN*`)`
TBN - TBox name

roles-equivalent *macro*

Description: Declares two roles to be equivalent.

Syntax: `(roles-equivalent` *RN1 RN1* `arguTBN)`

Arguments: *RN1* - role name
RN2 - role name
TBN - TBox name

roles-equivalent-1 *function*

Description: Declares two roles to be equivalent.

Syntax: `(roles-equivalent-1` *RN1 RN2* `arguTBN)`

Arguments: *RN1* - role name
RN2 - role name
TBN - TBox name

domain *macro*

Description: Declares the domain of a role.

Syntax: `(domain` *RN C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
C - concept
TBN - TBox name

role-has-domain *function*

Description: Declares the domain of a role.

Syntax: `(role-has-domain` *RN C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
C - concept
TBN - TBox name

attribute-has-domain *function*

Description: Declares the domain of an attribute.

Syntax: `(attribute-has-domain` *AN C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *AN* - attribute name
C - concept
TBN - TBox name

range *macro*

Description: Declares the range of a role.

Syntax: `(range` *RN C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
C - concept
TBN - TBox name

role-has-range *function*

Description: Declares the range of a role.

Syntax: `(role-has-range` *RN C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments:
RN - role name
C - concept
TBN - TBox name

attribute-has-range *function*

Description: Declares the range of an attribute.

Syntax: `(attribute-has-range` *AN D*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments:
AN - attribute name
C - concept
D - either cardinal, integer, real, complex, or string

implies-role *macro*

Description: Defines a parent of a role.

Syntax: `(implies-role` RN_1 RN_2
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments:
RN_1 - role name
RN_2 - parent role name
TBN - TBox name

role-has-parent *function*

Description: Defines a parent of a role.

Syntax: `(role-has-parent` RN_1 RN_2
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments:
RN_1 - role name
RN_2 - parent role name
TBN - TBox name

7.4 Concrete Domain Attribute Declaration

define-concrete-domain-attribute *macro*

Description: Defines a concrete domain attribute.

Syntax: `(define-concrete-domain-attribute` *AN* `&key` *type domain*`)`

Arguments: *AN* - attribute name

type - can be either bound to `cardinal`, `integer`, `real`, `complex`, or `string`. The type must be supplied.

domain - a concept describing the domain of the attribute.

Remarks: Calls `add-role-axioms`

Examples:
```
(define-concrete-domain-attribute has-age :type integer)
(define-concrete-domain-attribute has-weight :type real)
```

See also: Macro `signature` on page 68 and Section 3.5.

7.5 Assertions

instance *KRSS macro*

Description: Builds a concept assertion, asserts that an individual is an instance of a concept.

Syntax: `(instance` *IN C*`)`

Arguments: *IN* - individual name

C - concept term

Examples:
```
(instance Lucy Person)
(instance Snoopy (and Dog Cartoon-Character))
```

add-concept-assertion *function*

Description: Builds an assertion and adds it to an ABox.

Syntax: `(add-concept-assertion` *abox IN C*`)`

Arguments: *abox* - ABox object
IN - individual name
C - concept term

Values: *abox*

Examples:
```
(add-concept-assertion (find-abox 'peanuts-characters)
  'Lucy 'Person)
(add-concept-assertion (find-abox 'peanuts-characters)
  'Snoopy '(and Dog Cartoon-Character))
```

forget-concept-assertion *function*

Description: Retracts a concept assertion from an ABox.

Syntax: `(forget-concept-assertion` *abox IN C*`)`

Arguments: *abox* - ABox object
IN - individual name
C - concept term

Values: *abox*

Remarks: For answering subsequent queries the index structures for the ABox will be recomputed, i.e. some queries might take some time (e.g. those queries that require the realization of the ABox).

Examples:
```
(forget-concept-assertion (find-abox 'peanuts-characters)
  'Lucy 'Person)
(forget-concept-assertion (find-abox 'peanuts-characters)
  'Snoopy '(and Dog Cartoon-Character))
```

related *KRSS macro*

Description: Builds a role assertion, asserts that two individuals are related via a role (or feature).

Syntax: `(related` IN_1 IN_2 R`)`

Arguments:
- IN_1 - individual name of the predecessor
- IN_2 - individual name of the filler
- R - a role term or a feature term.

Examples:
```
(related Charlie-Brown Snoopy has-pet)
(related Linus Lucy (inv has-brother))
```

add-role-assertion *function*

Description: Adds a role assertion to an ABox.

Syntax: `(add-role-assertion` *abox* IN_1 IN_2 R`)`

Arguments:
- *abox* - ABox object
- IN_1 - individual name of the predecessor
- IN_2 - individual name of the filler
- R - role term

Values: *abox*

Examples:
```
(add-role-assertion (find-abox 'peanuts-characters)
  'Charlie-Brown 'Snoopy 'has-pet)
(add-role-assertion (find-abox 'peanuts-characters)
  'Linus 'Lucy '(inv has-brother))
```

forget-role-assertion *function*

Description: Retracts a role assertion from an ABox.

Syntax: `(forget-role-assertion` *abox* IN_1 IN_2 *R*`)`

Arguments: *abox* - ABox object

IN_1 - individual name of the predecessor

IN_2 - individual name of the filler

R - role term

Values: *abox*

Remarks: For answering subsequent queries the index structures for the ABox will be recomputed, i.e. some queries might take some time (e.g. those queries that require the realization of the ABox).

Examples:
```
(forget-role-assertion (find-abox 'peanuts-characters)
  'Charlie-Brown 'Snoopy 'has-pet)
(forget-role-assertion (find-abox 'peanuts-characters)
  'Linus 'Lucy '(inv has-brother))
```

forget-disjointness-axiom *function*

Description: This function is used to forget declarations with `define-disjoint-primitive-concept`.

Syntax: `(forget-disjointness-axiom` *tbox CN group − name*`)`

Arguments: *tbox* - TBox object

CN - concept-name

group − name - name of the disjointness group

forget-disjointness-axiom-statement *function*

Description: This function is used to forget statements of the form `(disjoint a b c)`

Syntax: `(forget-disjointness-axiom-statement` *tbox* `&rest` *concepts*`)`

Arguments: *tbox* - TBox object

concepts - List of concepts

define-distinct-individual *KRSS macro*

Description: This statement asserts that an individual is distinct to all other individuals in the ABox.

Syntax: `(define-distinct-individual` *IN*`)`

Arguments: *IN* - name of the individual

Values: *IN*

Remarks: Because the unique name assumption holds in RACER, all individuals are mapped to distinct domain objects by definition. Thus, the function is essentially a no-op. This function is supplied to be compatible with the KRSS specification.

state *KRSS macro*

Description: This macro asserts a set of ABox statements.

Syntax: `(state &body forms)`

Arguments: *forms* - is a sequence of `instance` or `related` assertions.

Remarks: This macro is supplied to be compatible with the KRSS specification. It realizes an implicit `progn` for assertions.

forget *macro*

Description: This macro retracts a set of TBox/ABox statements. Note that statement to be forgotten must be literally identical to the ones previously asserted, i.e., only explicitly given information can be forgotten.

Syntax:
```
(forget (&key (tbox *current-tbox*) (abox *current-abox))
  &body forms)
```

Arguments: *forms* - is a sequence of assertions.

Remarks: For answering subsequent queries the index structures for the TBox/ABox will probably be recomputed, i.e. some queries might take some time (e.g. those queries that require the reclassification of the TBox or realization of the ABox).

Examples:
```
(forget (:tbox family) (implies c d) (implies a b ))
(forget (:abox smith-family) (instance i d))
```

forget-statement *function*

Description: Functional interface for the macro `forget`

Syntax: `(forget-statement` *tbox abox* `&rest statements)`

Arguments: *tbox* - TBox

tbox - ABox

statements - statement previously asserted

7.6 Concrete Domain Assertions

add-constraint-assertion *function*

Description: Builds a concrete domain predicate assertion and adds it to an ABox.

Syntax: `(add-constraint-assertion` *abox constraint*`)`

Arguments: *abox* - ABox object

constraint - constraint form

Remarks: The syntax of concrete domain constraints is described in Section 3.1, page 45, and in Figure **??**, page **??**.

Examples:
```
(add-constraint-assertion (find-abox 'family)
  '(= temp-eve 102.56))
```

constraints *macro*

Description: This macro asserts a set of concrete domain predicates for concrete domain objects.

Syntax: `(constraints &body forms)`

Arguments: *forms* - is a sequence of concrete domain predicate assertions.

Remarks: Calls `add-constraint-assertion`. The syntax of concrete domain constraints is described in Section 3.1, page 45, and in Figure **??**, page **??**.

Examples:
```
(constraints
  (= temp-eve 102.56)
  (= temp-doris 38.5)
  (> temp-eve temp-doris))
```

add-attribute-assertion *function*

Description: Adds a concrete domain attribute assertion to an ABox. Asserts that an individual is related with a concrete domain object via an attribute.

Syntax: `(add-attribute-assertion` *abox IN ON AN*`)`

Arguments:
- *abox* - ABox object
- *IN* - individual name
- *ON* - concrete domain object name as the filler
- *AN* - attribute name

Examples:
```
(add-attribute-assertion (find-abox 'family)
  'eve 'temp-eve 'temperature-fahrenheit))
```

constrained *macro*

Description: Adds a concrete domain attribute assertion to an ABox. Asserts that an individual is related with a concrete domain object via an attribute.

Syntax: `(constrained` *IN ON AN*`)`

Arguments:
- *IN* - individual name
- *ON* - concrete domain object name as the filler
- *AN* - attribute name

Remarks: Calls `add-attribute-assertion`

Examples:
```
(constrained eve temp-eve temperature-fahrenheit)
```

8 Reasoning Modes

auto-classify *special-variable*

Description: Possible values are `:lazy`, `:eager`, `:lazy-verbose`, `:eager-verbose`, `nil`

Remarks: This variable is available in the Lisp version only.

See also: Section 3.8 on page 55.

auto-realize *special-variable*

Description: Possible values are `:lazy`, `:eager`, `:lazy-verbose`, `:eager-verbose`, `nil`

Remarks: This variable is available in the Lisp version only.

See also: Section 3.8 on page 55.

9 Evaluation Functions and Queries

9.1 Queries for Concept Terms

concept-satisfiable? *macro*

Description: Checks if a concept term is satisfiable.

Syntax: `(concept-satisfiable?` *C* `&optional (`*tbox* `*current-tbox*))`

Arguments: *C* - concept term.
tbox - TBox object

Values: Returns `t` if *C* is satisfiable and `nil` otherwise.

Remarks: For testing whether a concept term is satisfiable *with respect to a TBox tbox*. If satisfiability is to be tested without reference to a TBox, `nil` can be used.

concept-satisfiable-p *function*

Description: Checks if a concept term is satisfiable.

Syntax: `(concept-satisfiable-p` *C* *tbox*`)`

Arguments: *C* - concept term.
tbox - TBox object

Values: Returns `t` if *C* is satisfiable and `nil` otherwise.

Remarks: For testing whether a concept term is satisfiable *with respect to a TBox tbox*. If satisfiability is to be tested without reference to a TBox, `nil` can be used.

concept-subsumes? *KRSS macro*

Description: Checks if two concept terms subsume each other.

Syntax: `(concept-subsumes?` C_1 C_2 `&optional (`*tbox* `*current-tbox*))`

Arguments: C_1 - concept term of the subsumer
C_2 - concept term of the subsumee
tbox - TBox object

Values: Returns `t` if C_1 subsumes $C2$ and `nil` otherwise.

concept-subsumes-p *function*

Description: Checks if two concept terms subsume each other.

Syntax: `(concept-subsumes-p` C_1 C_2 *tbox*`)`

Arguments: C_1 - concept term of the subsumer
C_2 - concept term of the subsumee
tbox - TBox object

Values: Returns `t` if C_1 subsumes $C2$ and `nil` otherwise.

Remarks: For testing whether a concept term subsumes the other *with respect to a TBox tbox*. If the subsumption relation is to be tested without reference to a TBox, `nil` can be used.

See also: Function `concept-equivalent-p`, on page 101, and function `atomic-concept-synonyms`, on page 122.

concept-equivalent? *macro*

Description: Checks if the two concepts are equivalent in the given TBox.

Syntax: `(concept-equivalent?` C_1 C_2 `&optional (`*tbox* `*current-tbox*))`

Arguments: C_1, C_2 - concept term
tbox - TBox object

Values: Returns `t` if C_1 and C_2 are equivalent concepts in *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are equivalent *with respect to a TBox tbox*.

See also: Function `atomic-concept-synonyms`, on page 122, and function `concept-subsumes-p`, on page 100.

concept-equivalent-p *function*

Description: Checks if the two concepts are equivalent in the given TBox.

Syntax: `(concept-equivalent-p` C_1 C_2 *tbox*`)`

Arguments: C_1, C_2 - concept terms
tbox - TBox object

Values: Returns `t` if C_1 and C_2 are equivalent concepts in *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are equivalent *with respect to a TBox* *tbox*. If the equality is to be tested without reference to a TBox, `nil` can be used.

See also: Function `atomic-concept-synonyms`, on page 122, and function `concept-subsumes-p`, on page 100.

concept-disjoint? *macro*

Description: Checks if the two concepts are disjoint, e.g. no individual can be an instance of both concepts.

Syntax: `(concept-disjoint?` C_1 C_2 `&optional (`*tbox* `*current-tbox*))`

Arguments: C_1, C_2 - concept term
tbox - TBox object

Values: Returns `t` if C_1 and C_2 are disjoint with respect to *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are disjoint *with respect to a TBox* *tbox*. If the disjointness is to be tested without reference to a TBox, `nil` can be used.

concept-disjoint-p *function*

Description: Checks if the two concepts are disjoint, e.g. no individual can be an instance of both concepts.

Syntax: `(concept-disjoint-p` C_1 C_2 *tbox*`)`

Arguments: C_1, C_2 - concept term
tbox - TBox object

Values: Returns `t` if C_1 and C_2 are disjoint with respect to *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are disjoint *with respect to a TBox* *tbox*. If the disjointness is to be tested without reference to a TBox, `nil` can be used.

concept-p *function*

Description: Checks if *CN* is a concept name for a concept in the specified TBox.

Syntax: `(concept-p` *CN* `&optional (`*tbox* `*current-tbox*))`

Arguments: *CN* - concept name

tbox - TBox object

Values: Returns `t` if *CN* is a name of a known concept and `nil` otherwise.

concept? *macro*

Description: Checks if *CN* is a concept name for a concept in the specified TBox.

Syntax: `(concept?` *CN* `&optional (`*TBN* `*current-tbox*))`

Arguments: *CN* - concept name

TBN - TBox name

Values: Returns `t` if *CN* is a name of a known concept and `nil` otherwise.

concept-is-primitive-p *function*

Description: Checks if *CN* is a concept name of a so-called *primitive* concept in the specified TBox.

Syntax: `(concept-is-primitive-p` *CN* `&optional (`*tbox* `*current-tbox*))`

Arguments: *CN* - concept name

tbox - TBox object

Values: Returns `t` if *CN* is a name of a known primitive concept and `nil` otherwise.

concept-is-primitive? *macro*

Description: Checks if *CN* is a concept name of a so-called *primitive* concept in the specified TBox.

Syntax: `(concept-is-primitive-p` *CN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *CN* - concept name

TBN - TBox name

Values: Returns `t` if *CN* is a name of a known primitive concept and `nil` otherwise.

alc-concept-coherent *function*

Description: Tests the satisfiability of a $\mathrm{K}_{(m)}$, $\mathrm{K4}_{(m)}$ or $\mathrm{S4}_{(m)}$ formula encoded as an $\mathcal{ALC}$ concept.

Syntax: `(alc-concept-coherent` *C* `&key (`*logic* `:K))`

Arguments: *C* - concept term

logic - specifies the logic to be used.

`:K` - modal $\mathbf{K_{(m)}}$,
`:K4` - modal $\mathbf{K4_{(m)}}$ all roles are transitive,
`:S4` - modal $\mathbf{S4_{(m)}}$ all roles are transitive and reflexive.

If no logic is specified, the logic `:K` is chosen.

Remarks: This function can only be used for $\mathcal{ALC}$ concept terms, so number restrictions are not allowed.

9.2 Role Queries

role-subsumes? *KRSS macro*

Description: Checks if two roles are subsuming each other.

Syntax: `(role-subsumes?` R_1 R_2
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: R_1 - role term of the subsuming role

R_2 - role term of the subsumed role

TBN - TBox name

Values: Returns `t` if R_1 is a parent role of R_2.

role-subsumes-p *function*

Description: Checks if two roles are subsuming each other.

Syntax: `(role-subsumes-p` R_1 R_2 *tbox*`)`

Arguments: R_1 - role term of the subsuming role

R_2 - role term of the subsumed role

tbox - TBox object

Values: Returns `t` if R_1 is a parent role of R_2.

role-p *function*

Description: Checks if R is a role term for a role in the specified TBox.

Syntax: `(role-p` R `&optional (`*tbox* `*current-tbox*))`

Arguments: R - role term
tbox - TBox object

Values: Returns `t` if R is a known role term and `nil` otherwise.

role? *macro*

Description: Checks if R is a role term for a role in the specified TBox.

Syntax: `(role?` R `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: R - role term
TBN - TBox name

Values: Returns `t` if R is a known role term and `nil` otherwise.

transitive-p *function*

Description: Checks if R is a transitive role in the specified TBox.

Syntax: `(transitive-p` R `&optional (`*tbox* `*current-tbox*))`

Arguments: R - role term
tbox - TBox object

Values: Returns `t` if the role R is transitive in *tbox* and `nil` otherwise.

transitive? *macro*

Description: Checks if R is a transitive role in the specified TBox.

Syntax: `(transitive?` R `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: R - role term
TBN - TBox name

Values: Returns `t` if the role R is transitive in *TBN* and `nil` otherwise.

feature-p *function*

Description: Checks if R is a feature in the specified TBox.

Syntax: `(feature-p` R `&optional (`*tbox* `*current-tbox*))`

Arguments: R - role term
tbox - TBox object

Values: Returns `t` if the role R is a feature in *tbox* and `nil` otherwise.

feature? *macro*

Description: Checks if R is a feature in the specified TBox.

Syntax: `(feature?` R `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: R - role term
TBN - TBox name

Values: Returns `t` if the role R is a feature in *TBN* and `nil` otherwise.

cd-attribute-p *function*

Description: Checks if AN is a concrete domain attribute in the specified TBox.

Syntax: `(cd-attribute-p` AN `&optional (`*tbox* `*current-tbox*))`

Arguments: AN - attribute name
tbox - TBox object

Values: Returns `t` if AN is a concrete domain attribute in *tbox* and `nil` otherwise.

cd-attribute? *macro*

Description: Checks if AN is a concrete domain attribute in the specified TBox.

Syntax: `(cd-attribute?` AN `&optional`
`(`*TBN* `(tbox-name *current-tbox*)))`

Arguments: AN - attribute name
TBN - TBox name

Values: Returns `t` if the role AN is a concrete domain attribute in *TBN* and `nil` otherwise.

symmetric-p *function*

Description: Checks if R is symmetric in the specified TBox.

Syntax: `(symmetric-p` R `&optional (`*tbox* `*current-tbox*))`

Arguments: R - role term

tbox - TBox object

Values: Returns `t` if the role R is symmetric in *tbox* and `nil` otherwise.

symmetric? *macro*

Description: Checks if R is symmetric in the specified TBox.

Syntax: `(symmetric?` R `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: R - role term

TBN - TBox name

Values: Returns `t` if the role R is symmetric in *TBN* and `nil` otherwise.

reflexive-p *function*

Description: Checks if R is reflexive in the specified TBox.

Syntax: `(reflexive-p` R `&optional (`*tbox* `*current-tbox*))`

Arguments: R - role term

tbox - TBox object

Values: Returns `t` if the role R is reflexive in *tbox* and `nil` otherwise.

reflexive? *macro*

Description: Checks if R is reflexive in the specified TBox.

Syntax: `(reflexive?` R `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: R - role term

TBN - TBox name

Values: Returns `t` if the role R is reflexive in *TBN* and `nil` otherwise.

atomic-role-inverse *function*

Description: Returns the inverse role of role term R.

Syntax: `(atomic-role-inverse` R *tbox*)

Arguments: R - role term
tbox - TBox object

Values: Role name or term for the inverse role of R.

role-inverse *macro*

Description: Returns the inverse role of role term R.

Syntax: `(role-inverse` R `&optional (`TBN `(tbox-name *current-tbox*)))`

Arguments: R - role term
TBN - TBox name

Values: Role name or term for the inverse role of R.

Remarks: This macro uses `atomic-role-inverse`.

role-domain *macro*

Description: Returns the domain of role name RN.

Syntax: `(role-domain` RN `&optional (`TBN `(tbox-name *current-tbox*)))`

Arguments: RN - role name
TBN - TBox name

Remarks: This macro uses `atomic-role-domain`.

atomic-role-domain *function*

Description: Returns the domain of role name RN.

Syntax: `(atomic-role-domain` RN `&optional (`TBN `(tbox-name *current-tbox*)))`

Arguments: RN - role name
TBN - TBox name

role-range *macro*

Description: Returns the range of role name *RN*.

Syntax: `(role-range` *RN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
TBN - TBox name

Remarks: This macro uses `atomic-role-range`.

atomic-role-range *function*

Description: Returns the range of role name *RN*.

Syntax: `(atomic-role-range` *RN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *RN* - role name
TBN - TBox name

attribute-domain *macro*

Description: Returns the domain of attribute name *AN*.

Syntax: `(attribute-domain` *AN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *AN* - attribute name
TBN - TBox name

attribute-domain-1 *function*

Description: Returns the domain of attribute name *AN*.

Syntax: `(attribute-domain-1` *AN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *AN* - attribute name
TBN - TBox name

9.3 TBox Evaluation Functions

classify-tbox *function*

Description: Classifies the whole TBox.

Syntax: `(classify-tbox &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Remarks: This function needs to be executed before queries can be posed.

check-tbox-coherence *function*

Description: This function checks if there are any unsatisfiable atomic concepts in the given TBox.

Syntax: `(check-tbox-coherence &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Values: Returns a list of all atomic concepts in *tbox* that are not satisfiable, i.e. an empty list (`NIL`) indicates that there is no additional synonym to bottom.

Remarks: This function does not compute the concept hierarchy. It is much faster than `classify-tbox`, so whenever it is sufficient for your application use `check-tbox-coherence`. This function is supplied in order to check whether an atomic concept is satisfiable during the development phase of a TBox. There is no need to call the function `check-tbox-coherence` if, for instance, a certain ABox is to be checked for consistency (with `abox-consistent-p`).

tbox-classified-p *function*

Description: It is checked if the specified TBox has already been classified.

Syntax: `(tbox-classified-p &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox has been classified, otherwise it returns `nil`.

tbox-classified? *macro*

Description: It is checked if the specified TBox has already been classified.

Syntax: `(tbox-classified? &optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *TBN* - TBox name

Values: Returns `t` if the specified TBox has been classified, otherwise it returns `nil`.

tbox-prepared-p *function*

Description: It is checked if internal index structures are already computed for the specified TBox.

Syntax: `(tbox-prepared-p &optional (`*tbox*` *current-tbox*))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The function is used to determine whether Racer has spent some effort in processing the axioms of the TBox.

tbox-prepared? *macro*

Description: It is checked if internal index structures are already computed for the specified TBox.

Syntax: `(tbox-prepared? &optional (`*TBN*` (tbox-name *current-tbox*)))`

Arguments: *TBN* - TBox name

Values: Returns `t` if the specified TBox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The form is used to determine whether Racer has spent some effort in processing the axioms of the TBox.

tbox-cyclic-p *function*

Description: It is checked if cyclic GCIs are present in a TBox

Syntax: `(tbox-cyclic-p &optional (`*tbox*` *current-tbox*))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox contains cyclic GCIs otherwise it returns `nil`.

Remarks: Cyclic GCIs can be given either directly as a GCI or can implicitly result from processing, for instance, disjointness axioms.

tbox-cyclic? *macro*

Description: It is checked if cyclic GCIs are present in a TBox

Syntax: `(tbox-cyclic? &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox contains cyclic GCIs otherwise it returns `nil`.

Remarks: Cyclic GCIs can be given either directly as a GCI or can implicitly result from processing, for instance, disjointness axioms.

tbox-coherent-p *function*

Description: This function checks if there are any unsatisfiable atomic concepts in the given TBox.

Syntax: `(tbox-coherent-p &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Values: Returns `nil` if there is an inconsistent atomic concept, otherwise it returns `t`.

Remarks: This function calls `check-tbox-coherence` if necessary.

tbox-coherent? *macro*

Description: Checks if there are any unsatisfiable atomic concepts in the current or specified TBox.

Syntax: `(tbox-coherent? &optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *TBN* - TBox name

Values: Returns `t` if there is an inconsistent atomic concept, otherwise it returns `nil`.

Remarks: This macro uses `tbox-coherent-p`.

get-tbox-language *function*

Description: Returns a specifier indicating the description logic language used in the axioms of a given TBox.

Syntax: `(get-tbox-language &optional (`*TBN*` (tbox-name *current-tbox*)))`

Arguments: *TBN* - TBox name

Values: The language is indicated with the quasi-standard scheme using letters. Note that the language is identified for selecting optimization techniques. Since RACER does not exploit optimization techniques for sublanguages of $\mathcal{ALC}$, the language indicator starts always with `ALC`. Then `f` indicates whether features are used, `Q` indicates qualified number restrictions, `N` indicates simple number restrictions, `H` stands for a role hierarchy, `I` indicates inverse roles, `r+` indicates transitive roles, the suffix `-D` indicates the use of concrete domain language constructs.

get-meta-constraint *function*

Description: Optimized DL systems perform a static analysis of given terminological axioms. The axioms of a TBox are usually transformed in such a way that processing promises to be faster. In particular, the idea is to transform GCIs into (primitive) concept definitions. Since it is not always possible to "absorb" GCIs completely, a so-called meta constraint might remain. The functions `get-meta-constraint` returns the remaining constraint as a concept.

Syntax: `(get-meta-constraint &optional (`*TBN*` (tbox-name *current-tbox*)))`

Arguments: *TBN* - TBox name

Values: A concept term.

Remarks: The absorption process uses heuristics. Changes to a TBox might have dramatic effects on the value returned by `get-meta-constraint`.

get-concept-definition *macro*

Description: Optimized DL systems perform a static analysis of given terminological axioms. The axioms of a TBox are usually transformed in such a way that processing promises to be faster. In particular, the idea is to transform GCIs into (primitive) concept definitions. For a given concept name the function `get-concept-definition` returns the definition compiled by RACER during the absorption phase.

Syntax: `(get-concept-definition` *CN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *CN* - concept name

TBN - TBox name

Values: A concept term.

Remarks: The absorption process uses heuristics. Changes to a TBox might have dramatic effects on the value returned by `get-concept-definition`. Note that it might be useful to test whether the definition is primitive. See the function `concept-primitive-p`. RACER does not introduce new concept names for primitive definitions.

get-concept-definition-1 *function*

Description: Functional interface for `get-concept-definition`

Syntax: `(get-concept-definition-1` *CN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *CN* - concept name

TBN - TBox name

Remarks: The absorption process uses heuristics. Changes to a TBox might have dramatic effects on the value returned by `get-concept-negated-definition`. Note that it might be useful to test whether the definition is primitive. See the function `concept-primitive-p`. RACER does not introduce new concept names for primitive definitions.

Examples: Assume the following TBox:

```
(in-tbox test)
  (implies top (or a b c))
```

Then, `(get-concept-negated-definition c)` returns `(OR A B)`. Thus, RACER has transformed the GCI into the form `(implies (not C) (OR A B))` which can be handled more effectively be lazy unfolding. Note that the absorption process is heuristic. RACER could also transform the GCI into `(implies (not B) (OR A C))` or something similar depending on the current version and strategy.

get-concept-negated-definition *macro*

Description: Optimized DL systems perform a static analysis of given terminological axioms. The axioms of a TBox are usually transformed in such a way that processing promises to be faster. In particular, the idea is to transform GCIs into (primitive) concept definitions. For a given concept name the function `get-concept-negated-definition` returns the definition of the negated concept compiled by RACER during the absorption phase.

Syntax: `(get-concept-negated-definition` *CN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *CN* - concept name

TBN - TBox name

get-concept-negated-definition-1 *function*

Description: Functional interface for `get-concept-negated-definition`.

Syntax: `(get-concept-negated-definition-1` *CN* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *CN* - concept name

TBN - TBox name

9.4 ABox Evaluation Functions

realize-abox *function*

Description: This function checks the consistency of the ABox and computes the most-specific concepts for each individual in the ABox.

Syntax: `(realize-abox &optional (`*abox* `*current-abox*))`

Arguments: *abox* - ABox object

Values: *abox*

Remarks: This Function needs to be executed before queries can be posed. If the TBox has changed and is classified again the ABox has to be realized, too.

abox-realized-p *function*

Description: Returns `t` if the specified ABox object has been realized.

Syntax: `(abox-realized-p &optional (`*abox* `*current-abox*))`

Arguments: *abox* - ABox object

Values: Returns `t` if *abox* has been realized and `nil` otherwise.

abox-realized? *macro*

Description: Returns `t` if the specified ABox object has been realized.

Syntax: `(abox-realized? &optional (`*ABN* `(abox-name *current-abox*))`

Arguments: *ABN* - ABox name

Values: Returns `t` if *ABN* has been realized and `nil` otherwise.

abox-prepared-p *function*

Description: It is checked if internal index structures are already computed for the specified abox.

Syntax: `(abox-prepared-p &optional (`*abox* `*current-abox*))`

Arguments: *abox* - abox object

Values: Returns `t` if the specified abox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The function is used to determine whether Racer has spent some effort in processing the assertions of the abox.

abox-prepared? *macro*

Description: It is checked if internal index structures are already computed for the specified abox.

Syntax: `(abox-prepared? &optional (`*TBN* `(abox-name *current-abox*)))`

Arguments: *ABN* - abox name

Values: Returns `t` if the specified abox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The form is used to determine whether Racer has spent some effort in processing the assertions of the abox.

compute-all-implicit-role-fillers *function*

Description: Instruct RACER to use compute all implicit role fillers. After computing these fillers, the function all-role-assertions returns also the implicit role fillers.

Syntax: `(compute-all-implicit-role-fillers &optional (`*ABN* `*current-abox*))`)

Arguments: *ABN* - ABox name

compute-implicit-role-fillers *function*

Description: Instruct RACER to use compute all implicit role fillers for the individual specified. After computing these fillers, the function all-role-assertions returns also the implicit role fillers for the individual specified.

Syntax: `(compute-implicit-role-fillers individual &optional (`*ABN* `*current-abox*))`)

Arguments: *individual* - individual name

ABN - ABox name

get-abox-language *function*

Description: Returns a specifier indicating the description logic language used in the axioms of a given ABox.

Syntax: `(get-abox-language &optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *ABN* - ABox name

Values: The language is indicated with the quasi-standard scheme using letters. Note that the language is identified for selecting optimization techniques. Since RACER does not exploit optimization techniques for sublanguages of $\mathcal{ALC}$, the language indicator starts always with `ALC`. Then `f` indicates whether features are used, `Q` indicates qualified number restrictions, `N` indicates simple number restrictions, `H` stands for a role hierarchy, `I` indicates inverse roles, `r+` indicates transitive roles, the suffix `-D` indicates the use of concrete domain language constructs.

9.5 ABox Queries

abox-consistent-p *function*

Description: Checks if the ABox is consistent, e.g. it does not contain a contradiction.

Syntax: `(abox-consistent-p &optional (`*abox*` *current-abox*))`

Arguments: *abox* - ABox object

Values: Returns `t` if *abox* is consistent and `nil` otherwise.

abox-consistent? *macro*

Description: Checks if the ABox is consistent.

Syntax: `(abox-consistent? &optional (`*ABN*` (abox-name *current-abox*)))`

Arguments: *ABN* - ABox name

Values: Returns `t` if the ABox *ABN* is consistent and `nil` otherwise.

Remarks: This macro uses `abox-consistent-p`.

check-abox-coherence *function*

Description: Checks if the ABox is consistent. If there is a contradiction, this function prints information about the culprits.

Syntax: `(check-abox-coherence &optional (`*abox*` *current-abox*)`
`(stream *standard-output*)`

Arguments: *abox* - ABox object

stream - Stream object

Values: Returns `t` if *abox* is consistent and `nil` otherwise.

individual-instance? *KRSS macro*

Description: Checks if an individual is an instance of a given concept with respect to the `*current-abox*` and its TBox.

Syntax: `(individual-instance?` *IN C*
`&optional (`*abox*` (abox-name *current-abox*)))`

Arguments: *IN* - individual name

C - concept term

abox - ABox object

Values: Returns `t` if *IN* is an instance of *C* in *abox* and `nil` otherwise.

individual-instance-p *function*

Description: Checks if an individual is an instance of a given concept with respect to an ABox and its TBox.

Syntax: `(individual-instance-p` *IN C abox*`)`

Arguments: *IN* - individual name
C - concept term
abox - ABox object

Values: Returns `t` if *IN* is an instance of *C* in *abox* and `nil` otherwise.

constraint-entailed? *macro*

Description: Checks a specified constraint is entailed by an ABox (and its associated TBox).

Syntax: `(constraint-entailed? constraint &optional (`*abox* `*current-abox*))`

Arguments: *constraint* - A constraint
abox - ABox object

Values: Returns `t` if *abox* the constraint and `nil` otherwise.

Remarks: See Figure 3.1 for the syntax of the constraint argument.

constraint-entailed-p *function*

Description: Checks a specified constraint is entailed by an ABox (and its associated TBox).

Syntax: `(constraint-entailed-p constraint &optional (`*abox* `*current-abox*))`

Arguments: *constraint* - A constraint
abox - ABox object

Values: Returns `t` if *abox* the constraint and `nil` otherwise.

Remarks: See Figure 3.1 for the syntax of the constraint argument.

individuals-related? *macro*

Description: Checks if two individuals are directly related via the specified role.

Syntax: `(individuals-related?` IN_1 IN_2 R
`&optional (`*abox* `*current-abox*))`

Arguments:
- IN_1 - individual name of the predecessor
- IN_2 - individual name of the role filler
- R - role term
- *abox* - ABox object

Values: Returns `t` if IN_1 is related to IN_2 via R in *abox* and `nil` otherwise.

individuals-related-p *function*

Description: Checks if two individuals are directly related via the specified role.

Syntax: `(individuals-related-p` IN_1 IN_2 R *abox*`)`

Arguments:
- IN_1 - individual name of the predecessor
- IN_2 - individual name of the role filler
- R - role term
- *abox* - ABox object

Values: Returns `t` if IN_1 is related to IN_2 via R in *abox* and `nil` otherwise.

See also: Function `retrieve-individual-filled-roles`, on page 135,
Function `retrieve-related-individuals`, on page 134.

individual-equal? *KRSS macro*

Description: Checks if two individual names refer to the same domain object.

Syntax: `(individual-equal?` IN_1 IN_2 `&optional (`*abox* `*current-abox*))`

Arguments:
- IN_1, IN_2 - individual name
- *abox* - abox object

Remarks: Because the unique name assumption holds in RACER this macro always returns `nil` for individuals with different names. This macro is just supplied to be compatible with the KRSS.

individual-not-equal? *KRSS macro*

Description: Checks if two individual names do not refer to the same domain object.

Syntax: `(individual-not-equal?` IN_1 IN_2
`&optional (`*abox* `*current-abox*))`

Arguments: IN_1, IN_2 - individual name

abox - abox object

Remarks: Because the unique name assumption holds in RACER this macro always returns `t` for individuals with different names. This macro is just supplied to be compatible with the KRSS.

individual-p *function*

Description: Checks if *IN* is a name of an individual mentioned in an ABox *abox*.

Syntax: `(individual-p` *IN* `&optional (`*abox* `*current-abox*))`

Arguments: *IN* - individual name

abox - ABox object

Values: Returns `t` if *IN* is a name of an individual and `nil` otherwise.

individual? *macro*

Description: Checks if *IN* is a name of an individual mentioned in an ABox *ABN*.

Syntax: `(individual?` *IN* `&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name

ABN - ABox name

Values: Returns `t` if *IN* is a name of an individual and `nil` otherwise.

cd-object-p *function*

Description: Checks if *ON* is a name of a concrete domain object mentioned in an ABox *abox*.

Syntax: `(cd-object-p` *ON* `&optional (`*abox* `*current-abox*))`

Arguments: *ON* - concrete domain object name

abox - ABox object

Values: Returns `t` if *ON* is a name of a concrete domain object and `nil` otherwise.

cd-object? *macro*

Description: Checks if *ON* is a name of a concrete domain object mentioned in an ABox *ABN*.

Syntax: `(cd-object? ON &optional (ABN (abox-name *current-abox*)))`

Arguments: *ON* - concrete domain object name

ABN - ABox name

Values: Returns `t` if *ON* is a name of a concrete domain object and `nil` otherwise.

10 Retrieval

If the retrieval refers to concept names, RACER always returns a set of names for each concept name. A so called name set contains all synonyms of an atomic concept in the TBox.

10.1 TBox Retrieval

taxonomy *function*

Description: Returns the whole taxonomy for the specified TBox.

Syntax: `(taxonomy &optional (tbox *current-tbox*))`

Arguments: *tbox* - TBox object

Values: A list of triples, each of it consisting of:

a name set - the atomic concept *CN* and its synonyms

list of concept-parents name sets - each entry being a list of a concept parent of *CN* and its synonyms

list of concept-children name sets - each entry being a list of a concept child of *CN* and its synonyms.

Examples: `(taxonomy my-TBox)`
may yield:

```
(((*top*) () ((quadrangle tetragon)))
  ((quadrangle tetragon) ((*top*)) ((rectangle) (diamond)))
  ((rectangle) ((quadrangle tetragon)) ((*bottom*)))
  ((diamond) ((quadrangle tetragon)) ((*bottom*)))
  ((*bottom*) ((rectangle) (diamond)) ()))
```

See also: Function `atomic-concept-parents`,
function `atomic-concept-children` on page 125.

concept-synonyms *macro*

Description: Returns equivalent concepts for the specified concept in the given TBox.

Syntax: `(concept-synonyms` *CN*
`&optional (`*tbox* `(tbox-name *current-tbox*)))`

Arguments: *CN* - concept name
tbox - TBox object

Values: List of concept names

Remarks: The name *CN* is not included in the result.

See also: Function `concept-equivalent-p`, on page 101.

atomic-concept-synonyms *function*

Description: Returns equivalent concepts for the specified concept in the given TBox.

Syntax: `(atomic-concept-synonyms` *CN* *tbox*`)`

Arguments: *CN* - concept name
tbox - TBox object

Values: List of concept names

Remarks: The name *CN* is included in the result.

See also: Function `concept-equivalent-p`, on page 101.

concept-descendants *KRSS macro*

Description: Gets all atomic concepts of a TBox, which are subsumed by the specified concept.

Syntax: `(concept-descendants` *C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *C* - concept term
TBN - TBox name

Values: List of name sets

Remarks: This macro return the transitive closure of the macro `concept-children`.

atomic-concept-descendants *function*

Description: Gets all atomic concepts of a TBox, which are subsumed by the specified concept.

Syntax: `(atomic-concept-descendants` *C tbox*`)`

Arguments: *C* - concept term

tbox - TBox object

Values: List of name sets

Remarks: Returns the transitive closure from the call of `atomic-concept-children`.

concept-ancestors *KRSS macro*

Description: Gets all atomic concepts of a TBox, which are subsuming the specified concept.

Syntax:
```
(concept-ancestors C
  &optional (TBN (tbox-name *current-tbox*)))
```

Arguments: *C* - concept term

TBN - TBox name

Values: List of name sets

Remarks: This macro return the transitive closure of the macro `concept-parents`.

atomic-concept-ancestors *function*

Description: Gets all atomic concepts of a TBox, which are subsuming the specified concept.

Syntax: `(atomic-concept-ancestors` *C tbox*`)`

Arguments: *C* - concept term

tbox - TBox object

Values: List of name sets

Remarks: Returns the transitive closure from the call of `atomic-concept-parents`.

concept-children *KRSS macro*

Description: Gets the direct subsumees of the specified concept in the TBox.

Syntax: `(concept-children` *C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *C* - concept term
TBN - TBox name

Values: List of name sets

Remarks: Is the equivalent macro for the KRSS macro `concept-offspring`, which is also supplied in RACER.

atomic-concept-children *function*

Description: Gets the direct subsumees of the specified concept in the TBox.

Syntax: `(atomic-concept-children` *C tbox*`)`

Arguments: *C* - concept term
tbox - TBox object

Values: List of name sets

concept-parents *KRSS macro*

Description: Gets the direct subsumers of the specified concept in the TBox.

Syntax: `(concept-parents` *C*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *C* - concept term
TBN - TBox name

Values: List of name sets

atomic-concept-parents *function*

Description: Gets the direct subsumers of the specified concept in the TBox.

Syntax: `(atomic-concept-parents` *C tbox*`)`

Arguments: *C* - concept term
tbox - TBox object

Values: List of name sets

role-descendants *KRSS macro*

Description: Gets all roles from the TBox, that the given role subsumes.

Syntax: `(role-descendants` *R*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *R* - role term
TBN - TBox name

Values: List of role terms

Remarks: This macro is the transitive closure of the macro `role-children`.

atomic-role-descendants *function*

Description: Gets all roles from the TBox, that the given role subsumes.

Syntax: `(atomic-role-descendants` *R tbox*`)`

Arguments: *R* - role term
tbox - TBox object

Values: List of role terms

Remarks: This function is the transitive closure of the function `atomic-role-descendants`.

role-ancestors *KRSS macro*

Description: Gets all roles from the TBox, that subsume the given role in the role hierarchy.

Syntax: `(role-ancestors` *R*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *R* - role term
TBN - TBox name

Values: List of role terms

atomic-role-ancestors *function*

Description: Gets all roles from the TBox, that subsume the given role in the role hierarchy.

Syntax: `(atomic-role-ancestors` *R tbox*`)`

Arguments: *R* - role term

tbox - TBox object

Values: List of role terms

role-children *macro*

Description: Gets all roles from the TBox that are directly subsumed by the given role in the role hierarchy.

Syntax: `(role-children` *R*
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *R* - role term

TBN - TBox name

Values: List of role terms

Remarks: This is the equivalent macro to the KRSS macro `role-offspring`, which is also supplied by the RACER system.

atomic-role-children *function*

Description: Gets all roles from the TBox that are directly subsumed by the given role in the role hierarchy.

Syntax: `(atomic-role-children` *R tbox*`)`

Arguments: *R* - role term

tbox - TBox object

Values: List of role terms

role-parents *KRSS macro*

Description: Gets the roles from the TBox that directly subsume the given role in the role hierarchy.

Syntax: `(role-parents` *R* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *R* - role term
TBN - TBox name

Values: List of role terms

atomic-role-parents *function*

Description: Gets the roles from the TBox that directly subsume the given role in the role hierarchy.

Syntax: `(atomic-role-parents` *R tbox*`)`

Arguments: *R* - role term
tbox - TBox object

Values: List of role terms

role-synonyms *KRSS macro*

Description: Gets the synonyms of a role including the role itself.

Syntax: `(role-synonyms` *R* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *R* - role term
TBN - TBox name

Values: List of role terms

atomic-role-synonyms *function*

Description: Gets the synonyms of a role including the role itself.

Syntax: `(atomic-role-synonyms` *R tbox*`)`

Arguments: *R* - role term
tbox - TBox object

Values: List of role terms

all-tboxes *function*

Description: Returns the names of all known TBoxes.

Syntax: `(all-tboxes)`

Values: List of TBox names

all-atomic-concepts *function*

Description: Returns all atomic concepts from the specified TBox.

Syntax: `(all-atomic-concepts &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Values: List of concept names

all-equivalent-concepts *function*

Description: xx

Syntax: `(all-equivalent-concepts &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Values: List of name sets

all-roles *function*

Description: Returns all roles and features from the specified TBox.

Syntax: `(all-roles &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox object

Values: List of role terms

Examples: `(all-roles (find-tbox 'my-tbox))`

all-features *function*

Description: Returns all features from the specified TBox.

Syntax: `(all-features &optional (`*tbox* `*current-tbox*))`

Arguments: *tbox* - TBox

Values: List of feature terms

all-attributes *function*

Description: Returns all attributes from the specified TBox.

Syntax: `(all-attributes &optional (`*tbox*` *current-tbox*))`

Arguments: *tbox* - TBox

Values: List of attributes names

attribute-type *function*

Description: Returns the attribute type declared for a given attribute name in a specified TBox.

Syntax: `(attribute-type` *AN* `&optional (`*tbox*` *current-tbox*))`

Arguments: *AN* - attribute name

tbox - TBox

Values: Either cardinal, integer, real, or complex.

all-transitive-roles *function*

Description: Returns all transitive roles from the specified TBox.

Syntax: `(all-transitive-roles &optional (`*tbox*` *current-tbox*))`

Arguments: *tbox* - TBox object

Values: List of transitive role terms

describe-tbox *function*

Description: Generates a description for the specified TBox.

Syntax: `(describe-tbox &optional (`*tbox*` *current-tbox*)`
`(`*stream*` *standard-output*))`

Arguments: *tbox* - TBox object or TBox name

stream - open stream object

Values: *tbox*
The description is written to *stream*.

describe-concept *function*

Description: Generates a description for the specified concept used in the specified TBox or in the ABox and its TBox.

Syntax: `(describe-concept` *CN* `&optional (`*tbox-or-abox* `*current-tbox*)` `(`*stream* `*standard-output*))`

Arguments: *tbox-or-abox* - TBox object or ABox object

CN - concept name

stream - open stream object

Values: *tbox-or-abox*
The description is written to *stream*.

describe-role *function*

Description: Generates a description for the specified role used in the specified TBox or ABox.

Syntax: `(describe-role` *R* `&optional (`*tbox-or-abox* `*current-tbox*)` `(`*stream* `*standard-output*))`

Arguments: *tbox-or-abox* - TBox object or ABox object

R - role term (or feature term)

stream - open stream object

Values: *tbox-or-abox*
The description is written to *stream*.

10.2 ABox Retrieval

individual-direct-types *KRSS macro*

Description: Gets the most-specific atomic concepts of which an individual is an instance.

Syntax: `(individual-direct-types` *IN*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name

ABN - ABox name

Values: List of name sets

most-specific-instantiators *function*

Description: Gets the most-specific atomic concepts of which an individual is an instance.

Syntax: `(most-specific-instantiators` *IN abox*`)`

Arguments: *IN* - individual name
abox - ABox object

Values: List of name sets

individual-types *KRSS macro*

Description: Gets *all* atomic concepts of which the individual is an instance.

Syntax: `(individual-types` *IN*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name
ABN - ABox name

Values: List of name sets

Remarks: This is the transitive closure of the KRSS macro `individual-direct-types`.

instantiators *function*

Description: Gets *all* atomic concepts of which the individual is an instance.

Syntax: `(instantiators` *IN abox*`)`

Arguments: *IN* - individual name
abox - ABox object

Values: List of name sets

Remarks: This is the transitive closure of the function `most-specific-instantiators`.

concept-instances *KRSS macro*

Description: Gets all individuals from an ABox that are instances of the specified concept.

Syntax: `(concept-instances` *C*
`&optional (`*ABN* `(abox-name *current-abox*)) (`*candidates*`)`

Arguments: *C* - concept term
ABN - ABox name
candidates - a list of individual names

Values: List of individual names

retrieve-concept-instances *function*

Description: Gets all individuals from an ABox that are instances of the specified concept.

Syntax: `(retrieve-concept-instances` *C abox candidates*`)`

Arguments: *C* - concept term
abox - ABox object
candidates - a list of individual names

Values: List of individual names

individual-fillers *KRSS macro*

Description: Gets all individuals that are fillers of a role for a specified individual.

Syntax: `(individual-fillers` *IN R*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name of the predecessor
R - role term
ABN - ABox name

Values: List of individual names

Examples:
```
(individual-fillers Charlie-Brown has-pet)
(individual-fillers Snoopy (inv has-pet))
```

retrieve-individual-fillers *function*

Description: Gets all individuals that are fillers of a role for a specified individual.

Syntax: `(retrieve-individual-fillers` *IN R abox*`)`

Arguments: *IN* - individual name of the predecessor
R - role term
abox - ABox object

Values: List of individual names

Examples:
```
(retrieve-individual-fillers 'Charlie-Brown 'has-pet
  (find-abox 'peanuts-characters))
```

individual-attribute-fillers *macro*

Description: Gets all object names that are fillers of an attribute for a specified individual.

Syntax:
```
(individual-attribute-fillers IN AN
  &optional (ABN (abox-name *current-abox*)))
```

Arguments: *IN* - individual name of the predecessor
AN - attribute-name
ABN - ABox name

Values: List of object names

retrieve-individual-attribute-fillers *function*

Description: Gets all object names that are fillers of an attribute for a specified individual.

Syntax:
```
(retrieve-individual-attribute-fillers IN AN
  &optional (ABN (abox-name *current-abox*)))
```

Arguments: *IN* - individual name of the predecessor
AN - attribute-name
ABN - ABox name

Values: List of object names

told-value *function*

Description: Returns an explicitly asserted value for an object that is declared as filler for a certain attribute w.r.t. an individual.

Syntax: `(told-value` *ON*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *ON* - object name

ABN - ABox name

Values: Concrete domain value

retrieve-related-individuals *function*

Description: Gets all pairs of individuals that are related via the specified relation.

Syntax: `(retrieve-related-individuals` *R abox*`)`

Arguments: *R* - role term

abox - ABox object

Values: List of pairs of individual names

Examples:
```
(retrieve-related-individuals 'has-pet
  (find-abox 'peanuts-characters))
```
may yield:
```
((Charlie-Brown Snoopy) (John-Arbuckle Garfield))
```

See also: Function `individuals-related-p`, on page 119.

related-individuals *macro*

Description: Gets all pairs of individuals that are related via the specified relation.

Syntax: `(related-individuals` *R*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *R* - role term

ABN - ABox name

Values: List of pairs of individual names

Examples:
```
(retrieve-related-individuals 'has-pet
  (find-abox 'peanuts-characters))
```
may yield:
```
((Charlie-Brown Snoopy) (John-Arbuckle Garfield))
```

See also: Function `individuals-related-p`, on page 119.

retrieve-individual-filled-roles *function*

Description: This function gets all roles that hold between the specified pair of individuals.

Syntax: `(retrieve-individual-filled-roles` IN_1 IN_2 *abox*`)`.

Arguments:
- IN_1 - individual name of the predecessor
- IN_2 - individual name of the role filler
- *abox* - ABox object

Values: List of role terms

Examples:
```
(retrieve-individual-filled-roles 'Charlie-Brown 'Snoopy
  (find-abox 'peanuts-characters))
```

See also: Function `individuals-related-p`, on page 119.

retrieve-direct-predecessors *function*

Description: Gets all individuals that are predecessors of a role for a specified individual.

Syntax: `(retrieve-direct-predecessors` *R IN abox*`)`

Arguments:
- *R* - role term
- *IN* - individual name of the role filler
- *abox* - ABox object

Values: List of individual names

Examples:
```
(retrieve-direct-predecessors 'has-pet 'Snoopy
  (find-abox 'peanuts-characters))
```

all-aboxes *function*

Description: Returns the names of all known ABoxes.

Syntax: `(all-aboxes)`

Values: List of ABox names

all-individuals *function*

Description: Returns all individuals from the specified ABox.

Syntax: `(all-individuals &optional (`*abox*` *current-abox*))`

Arguments: *abox* - ABox object

Values: List of individual names

all-concept-assertions-for-individual *function*

Description: Returns all concept assertions for an individual from the specified ABox.

Syntax: `(all-concept-assertions-for-individual` *IN*
`&optional (`*abox*` *current-abox*))`

Arguments: *IN* - individual name
abox - ABox object

Values: List of concept assertions

See also: Function `all-concept-assertions` on page 137.

all-role-assertions-for-individual-in-domain *function*

Description: Returns all role assertions for an individual from the specified ABox in which the individual is the role predecessor.

Syntax: `(all-role-assertions-for-individual-in-domain` *IN*
`&optional (`*abox*` *current-abox*))`

Arguments: *IN* - individual name
abox - ABox object

Values: List of role assertions

Remarks: Returns only the role assertions explicitly mentioned in the ABox, not the inferred ones.

See also: Function `all-role-assertions` on page 137.

all-role-assertions-for-individual-in-range *function*

Description: Returns all role assertions for an individual from the specified ABox in which the individual is a role successor.

Syntax: `(all-role-assertions-for-individual-in-range` *IN* `&optional (`*abox* `*current-abox*))`

Arguments: *IN* - individual name
abox - ABox object

Values: List of assertions

See also: Function `all-role-assertions` on page 137.

all-concept-assertions *function*

Description: Returns all concept assertions from the specified ABox.

Syntax: `(all-concept-assertions &optional (`*abox* `*current-abox*))`

Arguments: *abox* - ABox object

Values: List of assertions

all-role-assertions *function*

Description: Returns all role assertions from the specified ABox.

Syntax: `(all-role-assertions &optional (`*abox* `*current-abox*))`

Arguments: *abox* - ABox object

Values: List of assertions

See also: Function `all-concept-assertions-for-individual` on page 136.

all-constraints *function*

Description: Returns all constraints from the specified ABox which refer to a list of object names.

Syntax: `(all-constraints &optional (`*abox* `*current-abox*)` *ONs*`)`

Arguments: *abox* - ABox object
ONs - list of object names

Values: List of constraints

Remarks: If ONs is not specified, all constraints of the ABox are returned.

all-attribute-assertions *function*

Description: Returns all attribute assertions from the specified ABox.

Syntax: `(all-attribute-assertions &optional (`*abox*` *current-abox*))`

Arguments: *abox* - ABox object

Values: List of assertions

describe-abox *function*

Description: Generates a description for the specified ABox.

Syntax: `(describe-abox &optional (`*abox*` *current-abox*)`
`(`*stream*` *standard-output*))`

Arguments: *abox* - ABox object

stream - open stream object

Values: *abox*
The description is written to *stream*.

describe-individual *function*

Description: Generates a description for the individual from the specified ABox.

Syntax: `(describe-individual `*IN*` &optional (`*abox*` *current-abox*)`
`(`*stream*` *standard-output*))`

Arguments: *IN* - individual name

abox - ABox object

stream - open stream object

Values: *IN*
The description is written to *stream*.

10.3 The Racer Query Language - RQL

In this section of the manual we will describe an extended query language for Racer, called RQL (for Racer Query Language). The RQL can be seen as a straightforward extension and combination of the ABox querying mechanisms already described in chapter 10.2. Users who are familiar with the basic ABox retrieval functions of Racer will easily understand the new RQL. Unlike the ABox retrieval function described in Chapter 10.2, the RQL allows the use of variables within queries, as well as much more *complex queries*. The variables in the queries are to be bound against those ABox individuals that satisfy the specified query. Queries will make use of concept and role terms; also the current TBox is taken into account. However, it is possible to use ABox individuals in query expressions as well. ABox individuals and variables will be commonly referenced as *objects.*

We will guide the user step-by-step through the RQL; then give a formal BNF syntax description.

10.3.1 Step by step: RQL by example

As a running example, we will use the knowledge base `family-1.racer` which is contained in the `examples.tar.gz` file that can be download from the Racer download page. Please load this KB and start your favorite Racer interface (e.g., RICE).

10.3.1.1 Simple Queries and Query Atoms

Unary Concept Query Atoms To pose an RQL query to the Racer system, the function `retrieve` is provided as part of the functional Racer API. To give a first example, consider the query

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

asking Racer for all instances of type woman from the current ABox to be bound to the variable `?x`.

Racer replies:

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

i.e., the query 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.

Within the call to retrieve, (`?x`) represents the list of *result objects*, and `(?x woman)` is the *query body* or *query expression.* In this case, the query body is a so-called *unary query atom.* Query atoms are the most basic query expressions offered by the RQL.

The *list of result objects* specifies the form of the returned binding lists. Note that the concept `woman` might be replaced by an arbitrary concept term (not just a concept name).

Moreover, a so-called *active domain semantics* is employed for the variables: variables can only be bound to *explicitly present ABox individuals* in the current ABox.

Basically, the result is the same for `(concept-instances woman)`:

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

Thus, the unary query atom `(?x woman)` has basically the same semantics as `(concept-instances woman)`.

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

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

Racer replies: `T`,

which means "yes".

In this case, the list of supplied result objects is empty. Such a query never returns any bindings, but only `T` or `NIL` (true or false). `T` is returned if any binding possibility has been found making the query body true; and `NIL` otherwise.

It is also possible to use ABox individuals within queries. Suppose we want to know if Betty is a woman - we can pose the query

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

Racer replies: `T`,

and consequently, for

```
(retrieve () (betty man))
```

Racer replies: `NIL`.

If ABox individuals are used as result objects, they will be listed in the bindings as well:

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

Racer replies:

```
(((BETTY BETTY)))
```

However, for

```
(retrieve (betty) (betty man))
```

Racer replies:

```
NIL.
```

If an ABox individual is used which is not present in the ABox, Racer signals an error:

```
(retrieve () (jane woman))
```

yields

```
RACER Exception while submitting command:
Undefined individual name JANE in ABox SMITH-FAMILY.
```

Binary Role Query Atoms Now, let us consider a more complex example. Suppose we are looking for all explicitly modeled mother-child-pairs in the ABox. In the example KB, the role `has-child` is used. We can therefore pose the following query:

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

Racer replies:

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

The query expression `(?mother ?child has-child)` is an example of a so-called *binary 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 Racer replies:

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

Actually, we might use a *role term* instead of a simple role:

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

Again, Racer replies:

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

Actually, we might use a *role term* instead of a simple role:

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

Again, Racer replies:

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

It should be noted that the list of result objects must be a *subset* of the objects which are actually used within the query body. For example,

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

or

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

yields the error `Racer Exception while submitting command: Bad result variables given!`, since neither `?x` nor `charles` are referenced within the query body `(betty ?child-of-betty has-child)`. On the contrary, it is possible to duplicate or "reorder" objects:

```
(retrieve (?x ?x ?x) (?x man))
```

yields

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

which is a binding list containing a single triple. Note that Charles is the only known man in the example KB.

Binary Constraint Query Atoms There is one more type of binary query atom, which is related to the concrete domain part of a KB.

Before we can proceed, please load `family-1.racer` into your favorite text editor and modify the file as follows:

```
(signature ...
           :attributes ((integer age))
           ; :individuals (alice betty charles doris eve)
           )
```

That means, add the `:attributes` part and put a "`;`" before `:individuals` (or kill this line). Please see the note below for an explanation why this is necessary. Then add, at the end of the file, the following assertions representing ages of the family members:

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

Every person should now have an age:

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

Racer replies:

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

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

```
(retrieve (?x) (?x (min age 75)))
```

Racer replies:

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

Note that `(min age 75)` is an ordinary Racer concept expression.

Now for the only remaining type of query atoms - If we want to know who is older than whom we can pose the following query:

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

Racer 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)))
```

That means, Charles is older than Eve, Doris, and Betty; etc. Of course, individuals can also be used. Let us verify that nobody is older than Alice:

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

Racer replies: `NIL`.

A binary constraint query atom such as `(?x ?y (:constraint (age) (age) >))` retrieves the set of all tuples such that the age attribute of the first argument in the tuples, `?x`, is greater (`>`) than the age attribute of the second argument of the tuple, `?y`. Moreover, the two lists appearing after `:constraint`, here `(age) (age)`, can be *feature chains of arbitrary length such that the last feature in each chain is a (concrete domain) attribute.* Let us consider a more complex example with such feature chains of length greater than one. Please modify the KB as follows:

```
(signature ...

           :roles (  ...

                   (has-child :parent has-descendant

                              :inverse has-parent

                              :domain parent
                              :range person)
                     ...
                  )

           ...

           :features ((has-father :parent has-parent)
                      (has-mother :parent has-parent)))
```

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 axioms 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) ; just for the sake of KB brevity ;-)
```

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) >)))
```

Racer replies:

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

Since, for the sake of KB brevity, Betty and Charles are actually siblings but also have a common child (Eve), the following query succeeds:

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

Racer replies:


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

Currently, the predicates cannot be complex expressions (e.g., expressions like `(:constraint (age) (age) (= ?age1 (+ 10 ?age2)))` are currently not supported, but will be in a future version of Racer).

IMPORTANT IMPLEMENTATION LIMITATION / NOTE: CURRENTLY, THESE `:CONSTRAINT`-ATOMS WORK ONLY IF THE SIGNATURE DOES NOT CONTAIN AN `:INDIVIDUALS`-SECTION.

Unary `:has-known-successor` Query Atoms *Sometimes one just wants to ensure that there are certain explicitly modeled role successors in an ABox without actually retrieving them.*

For example, suppose we want to know for which individuals we have explicitly modeled children in the ABox. Thus, the query

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

gives us

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

since these are the individuals which have appropriate successors of type `has-child` in the ABox. Alternatively we could have used

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

So what is the difference between these two queries? Well, the latter query would unnecessarily also bind `?y` to all present children of `?x`. Therefore, the former query is *computationally cheaper* to answer.

However, there is a more subtle difference which will come into play if we consider *negated query atoms*, see below.

Also note that

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

is *not* equivalent to

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

Suppose we add the axiom

```
(individual-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)))
```

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.

10.3.2 Complex Queries

AND Queries Suppose we want to list all mothers of male persons in the KB. Whereas the previous queries were all *simple* (a single unary or binary *query atom* was sufficient for expressing them), we will now need a *compound (or complex) query*:

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

Racer replies:

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

In this query, we have used the `AND` operator.

Understanding the Query Results It should be noted that

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

is *not* equivalent to

```
(retrieve (?x) (?x mother)),
```

since the first query (internally) also binds the variable `?y` and ensures that `(?y man) (?x ?y has-child)` holds as well, even if the possible bindings of `?y` are not returned to the user.

The objects (variables and individuals) which are referenced within a *query body are always bound in every possible way;* then the list of *result objects* is used to determine the format of the output tuples of the query. This can be seen as a *projection operation* (if we ignore the possibility to duplicate or reorder objects in the output binding lists). However, the projection to the result objects is always the *last step* in the query processing chain, and not the first one. Consequently, if the specified list of result objects is empty, we get

- `T` iff any binding possibility has been found making the query body true, and
- `NIL` otherwise.

The Unique Name Assumption Racer uses the *unique name assumption (UNA) for variables:* consider the query

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

Racer replies: `NIL`,

since there is only one known man in the ABox (Charles). Due to the UNA for variables, `?x` and `?y` are required to be bound to *different* men in the ABox. But, since Charles is the only man, the answer is `NIL`.

Note that, if ABox individuals are used within a query, they are excluded as possible bindings for the other variables as well. E.g., for the query

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

Racer replies: `NIL`

as well, since `?x` has to be bound to a man *different* from Charles.

However, it is possible to switch off the UNA for certain (or all) variables: simply prefix a variable with a "$":

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

Racer replies:

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

and for

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

Racer replies:

```
((($?X CHARLES) (CHARLES CHARLES)))
```

A More Complex Example RQL 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)))
```

Racer 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:

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

Racer replies:

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

Such a query would be hard to formulate without the RQL. In fact, quite some *manual programming* would be necessary for the user if he had to use the basic ABox retrieval operations described in Chapter 10.2.

OR Queries Moreover, Racer also offers an `OR` operator:

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

Racer replies:

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


However, the `OR` operator is more subtle to understand, since the *names* of the variables matter. If disjuncts within an `OR` reference different variables, then the system will ensure that each disjunct references the same variables. For example, the query

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

The result 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 the union of the two 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 new tuples in this example.

Consider the query

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

Again, it is important to note that this query is *not* equivalent to

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

As already described, Racer will rewrite this query into

```
(retrieve (?y) (or (and (?x woman) (?y top))
                   (and (?x top) (?y man)))),
```

Thus, the possible bindings for `?y` are from the union of `top` and `man`, which is of course `top`, and not `man`.
Racer therefore replies:

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

whereas

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

returns

```
(((?Y CHARLES))),
```

as expected.

10.3.3 Negated Query Atoms

Negated Unary Concept Query Atoms A `NOT` is provided which implements a *Negation as Failure Semantics*.
Consider the query

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

Racer replies:

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

If we query with a `NOT` within an ordinary Racer concept term, e.g.

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

where `(not grandmother)` is just an ordinary Racer concept term, we get

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

Since Charles is a man, he can obviously never be a mother, and therefore Racer *can prove* that Charles is *not a grandmother*. However, due to the open world semantics, this doesn't apply to the other individuals - for example, Betty might very well be a grandmother; we simply would have to add some additional ABox axioms.

However, sometimes one wants to know which individuals are currently *not known* to be grandmothers. In this case, we would like to retrieve all the individuals for which Racer currently *cannot prove* that they are grandmothers. Consequently, *all individuals but Alice* should be returned. This is exactly the semantics of a negation as failure atom:

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

Note that the `NOT` is placed "around" the entire atom. Racer replies:

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

This is simply the complement query of

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

w.r.t. the set of all individuals in the ABox.

Note that `(?x (not grandmother))` implies `(not (?x grandmother))`, but not the other way around.

A further example:

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

yields

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

Suppose we are looking for persons without children. The query

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

yields `NIL`,

since, again due to the open world semantics, any of the currently present persons who do not have an explicitly modeled child in the ABox yet could still be a parent (simply add some role assertions to the ABox). However, currently, Doris and Betty do not have an explicitly modeled child in the ABox. How do we retrieve them? One way might be to use the query

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

which correctly yields

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

However, sometimes this approach doesn't work. Suppose we add the following axiom:

```
(individual-instance doris mother)
```

If we query again with

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

we only get

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

since Racer can now prove that Doris has a child (since Doris is an instance of the concept mother), even though her child is not explicitly modeled in the ABox. If we want to get a *positive answer* that Doris does not have any explicitly modeled children in the ABox, we can use the query atom `(?x NIL has-child)`, which we borrowed from the query language of the LOOM system:

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

retrieves the individuals for which there is no known (explicitly modeled) child in the ABox:

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

Equivalently, the query

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

could be used, which yields the same result.

Internally, atoms such as `(?x NIL has-child)` are rewritten into unary atoms of the kind `(not (?x (:has-known-successor has-child)))`. This explains why we mention them in this section of the manual. Note that the atoms of type `:has-known-successor` have already been introduced.

Negated Binary Role Query Atoms The `NOT` can also be applied in front of *binary role query atoms.* In the following examples we will use the role `has-descendant`, which is transitive:

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

yields

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

 ((?X BETTY) (?Y DORIS))
 ((?X BETTY) (?Y EVE))

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

If we want to retrieve all tuples of persons that are currently *not related by descendance,* i.e. such that Racer *cannot prove* that the two persons are related by descendance, we can use the following query:

```
(retrieve (?x ?y) (and (not (?x ?y has-descendant))
                       (not (?y ?x has-descendant))))
```

Racer replies:

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

 ((?X CHARLES) (?Y DORIS))
 ((?X DORIS) (?Y CHARLES))

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

Thus, Eve and Doris are siblings; as well as Betty and Charles. In contrast, Charles is the uncle of Doris.

Suppose we want to know which person does *not have any explicitly modeled descendants in the ABox.* A first idea might be to use the following query:

```
(retrieve (?x) (and (not (?x ?y has-descendant))
                    (not (?y ?x has-descendant))))
```

However, since this is just the projection of the previous query to the first argument of its result tuples, we will get

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

which is not what we intended. Fortunately, as already mentioned, the query

```
(retrieve (?x) (and (?x NIL has-descendant)
                    (NIL ?x has-descendant)))
```

can be used - Racer replies `NIL`, since every modeled individual is somehow set into relation with some other individual.

A further, more complex example:

```
(retrieve (?x ?y) (and (not (?x ?y has-descendant))
                       (not (?y ?x has-descendant))
                       (not (?x ?y has-sibling))
                       (not (?y ?x has-sibling))))
```

If `has-sibling` had been declare as symmetric in the KB, we could have omitted the last conjunct in the query. Racer replies:

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

Negated Binary Constraint Query Atoms The `NOT` can also be applied in front of *binary constraint query atoms.* We can easily list the complement of

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

Answer:

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

with

```
(retrieve (?x)
          (not (?x ?x (:constraint (has-father has-mother age)
                                   (has-mother has-mother age) =))))
```

which consequently yields

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

Negated Query Atoms Referencing Individuals and `bind-individual` Atoms What is the semantics of a negated query atom referencing ABox individuals?

First note that a query atom and its negated counterpart always behave dually: if `(?x man)` yields all men, then `(not (?x man))` yields all ABox individuals minus the set of all men. If `(?x ?y has-child)` yields a set of pairs of individuals, then `(not (?x ?y has-child))` yields the cross products of all ABox individuals with all ABox individuals minus the answer set of `(?x ?y has-child)`.

Since `(betty top)` returns a singleton answer set (only Betty), we should expect that `(not (betty top))` behaves in fact like a variable, enumerating all individuals that are not Betty:

```
(retrieve (betty) (not (betty top)))
```

Racer replies:

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

since this is the complement of

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

which yields

```
(((BETTY BETTY))).
```

Since

```
(retrieve (betty) (betty man))
```

retrieves `NIL`, its complement query

```
(retrieve (betty) (not (betty man)))
```

gives us

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

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

However, sometimes this behavior is unwanted. Suppose we want to verify whether 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:

```
(retrieve (eve) (eve mother))
```

yields `NIL` (a *negative answer*), since Racer cannot prove that Eve is a mother, and

```
(retrieve (eve) (eve (not mother)))
```

yields `NIL` as well. However, these are *negative answers.* Unfortunately, if we query with

```
(retrieve (eve) (not (eve mother))),
```

we get

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

as described above, since `eve` turned into a variable. **However, if we are only interested in Eve, we can use an additional conjunct which ensures that `eve` can only be bound to the ABox individual Eve. This additional query atom is `(bind-individual eve)`:**

```
(retrieve (eve) (and (bind-individual eve) (not (eve mother))))
```

yields

```
(((EVE EVE))
```

and thus we get a *positive answer that it cannot be proven that Eve is a mother.* Consequently,

```
(retrieve () (and (bind-individual eve) (not (eve mother))))
```

returns `T`.

10.3.4 Boolean Complex Queries

In fact, it is possible to combine arbitrarily nested `NOT`, `AND` and `OR` query expressions. We therefore might call the queries *boolean.*

Internally, the system will bring the queries into Negation Normal Form (NNF) such that `NOT` appears only in front of query *atoms.*

Moreover, the queries are even brought into Disjunctive Normal Form (DNF). Since the DNF might be exponentially larger than the original query and thus result in very big queries, we would like to inform the user of this potential performance pitfall.

In order to understand the query results of some complex queries better it might help if the user reminds him / herself about these internal transformations.

10.4 Formal Syntax of the RQL

A call to `retrieve` looks as follows:

```
(retrieve <list-of-objects> <query-body>)
```

where `<list-of-objects>` is specified in EBNF (`*` means zero or more occurrences; `"X"` denotes a literal):

```
<list-of-objects>  -> "(" <query-object>* ")"
<query-object>     -> <query-variable> | <query-individual>
<query-variable>   -> "?"<symbol> | "$?"<symbol>
<query-individual> -> <symbol>
<symbol>           -> any LISP symbol (huhu, foobar, but not t or nil)
```

For example, the list `(?x betty)` is a valid `<list-of-objects>`, as well as `(?x betty betty $?y)`, and `()`. `(nil)` or `(t)` are invalid `<list-of-objects>`.

Moreover, `<list-of-objects>` should be a subset of the query objects which are referenced in `<query-body>`; otherwise, a `Racer Exception while submitting command: Bad result variables given!` will be signaled.

Query bodies are defined as follows:

```
<query-body> -> <query-atom> |
                "(" "AND" <query-body>* ")" |
                "(" "OR"  <query-body>* ")" |
                "(" "NOT" <query-body>   ")"

<query-atom> -> "(" "NOT" <query-atom> ")" |
                "(" <query-object> <concept-expression> ")" |
                "(" "bind-individual" <query-individual> ")" |
                "(" <query-object>
                   "(" ":has-known-successor" <role-expression> "))" |
                "(" <query-object> <query-object> <role-expression> ")" |
                "(" <query-object> "NIL" <role-expression> ")" |
                "(" "NIL" <query-object> <role-expression> ")" |
                "(" <query-object> <query-object>
                   "(" ":constraint"
                       <chain>
                       <chain>
                       <CD-predicate> ")"
                ")"

<chain> -> <attribute-name> |
           "(" <feature-name>* <attribute-name> ")"

<attribute-name> -> see "AN" on page 46, Fig. 25

<feature-name> -> see "R" on page 46, Fig. 25

<concept-expression> -> see "C" on page 46, Fig. 25

<role-expression> -> see "R" on page 46, Fig. 25

<CD-prediate> -> "equal" | "unequal" | "string=" | "string<>" |
                 ">" | "<" | ">=" | "<=" | "<>" | "="
```

10.5 Acknowledgments

We greatly acknowledge the feedback and comments we have received from Ragnhild van der Straeten and Rolf de By. Both have tested the new RQL. Both have reported deficiencies and errors in the RQL implementation and in this document as well. Thanks a lot for that!

11 Configuring Optimizations

The standard configuration of RACER 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 TBox must be classified. However, for answering an instance retrieval query this is not necessary and, therefore, RACER does not classify the TBox in the standard inference mode (but see the documentation of the special variable `*auto-classify*`). Nevertheless, if multiple instance retrieval queries are to be answered by RACER, it might be useful to have the TBox classified in order to be able to compute an index for query answering. Considering a single query RACER cannot determine whether computing an index is worth the required computational ressources. Therefore, RACER can be instructed about answering strategies for subsequent queries. The corresponding functions are documented in this chapter.

---

compute-index-for-instance-retrieval *function*

Description: Let RACER create an index for subsequent instance retrieval queries wrt. the specified ABox.

Syntax: `(compute-index-for-instance-retrieval &optional (`*ABN* `*current-abox*))`)

Arguments: *ABN* - ABox object

Remarks: Computing an index requires the associated TBox be classified and the input ABox be realized. Thus, it may take some time for this function to complete. Use the function `abox-realized-p` to check whether index-based instance retrieval is enabled.

---

ensure-subsumption-based-query-answering *function*

Description: Instruct RACER to use caching strateties and to exploit query subsumption for answering instance retrieval queries.

Syntax: `(ensure-subsumption-based-query-answering &optional (`*ABN* `*current-abox*))`)

Arguments: *ABN* - ABox object

Remarks: Subsumption-based query answering requires the associated TBox to be classfied. Thus, the function might require computational ressources that are not negligible. Instructing RACER to perform reasoning in this mode pays back if one and the same instance retrieval query might be posed several times or if the concepts in subsequent instance retrieval queries subsumes each other (in other words: if queries are more and more refined). Use the function `tbox-classified-p` to check whether index-based instance retrieval is enabled.

12 The Publish-Subscribe Mechanism

Instance retrieval (see the function `concept-instances`) is one of the main inference services for ABoxes. 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.[7] Furthermore, if some assertions are added to an ABox, 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, RACER 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 ABox in question) that have been "publised" previously. If information about a new individuals is added to an ABox 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.

12.1 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 RACER 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 TBox `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)
```

[7] Certainly, 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 RACER.

In order to manage assertions about specific documents, an ABox `current-documents` is defined with the following statements. The ABox `current-documents` is the "current ABox" to which subsequent statements and queries refer. The set of subscriptions (w.r.t. the current ABox) 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.

```
(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 RACER is the following

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

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

An application receiving this message from RACER as a return result can then decide how to inform the client appropriately. In future releases of RACER, subscriptions can be extended with information about how the retrieval events are to be signalled 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, RACER 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 TBox `document-ontology`.

Now, we have information about seven documents declared in the ABox `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 ABox, it is possible to use RACER's forget facility. The instance assertion can be removed from the ABox 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`. RACER'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 ABox.

```
(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 RACER system handles all publish statements within a `state` as a single `publish` statement and answers the follwing 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 intances 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 RACER.

```
(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)))
```

RACER returns the following information:

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

Notice again that subscriptions might be considered when new information is added to the ABox.

```
(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 RACER 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 subscriptons (i.e. event handlers can be introduced).

12.2 Using JRacer for Publish and Subscribe

The following code fragment demonstrates how to interact with a Racer 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 Racer system to direct the channel to computer `"mo.fh-wedel.de"` at port `8080`. Before the subscription is sent to a Racer Server, the agent should make sure that at `"mo.fh-wedel.de"`, the assumed agent base station, a so-called listener process is started at port 8080. This can be easily accomplished:

```
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 Racer runs at node `"racer.fh-wedel.de"` on port 8088).

```
public class Subscription {
   public static void main(String[] argv) {
      RacerSocketClient client = new RacerClient("racer.fh-wedel.de", 8088);
      try {
         client.openConnection();
         try {
            String result =
               client.send
                     ("(subscribe q_1 Book \"mo.fh-wedel.de\" 8080))");
            }
         catch (RacerException e) {
            ...
            }
         }
         client.closeConnection();
      } catch (IOException e) {
         ...
      }
   }
}
```

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

12.3 Realizing Local Closed World Assumptions

Feedback from many users of the Racer 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 Racer, 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 Racer into two variables A and B, respectively, and, in order to compute the set of books for which there does not exist an author, it can consider the complement of B wrt. A. We see this strategy as an implementation of a local closed-world (LCW) assumption.

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

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

12.4 Publish and Subscribe Functions

In the following the functions offered by the publish-subscribe facility are explained in detail.

---

publish *macro*

Description: Publish an ABox individual.

Syntax: `(publish` *IN*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name
ABN - ABox name

Values: A list of tuples consisting of subscriber and individuals names.

---

publish-1 *macro*

Description: Functional interface for `publish`.

Syntax: `(publish-1` *IN*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name
ABN - ABox name

---

unpublish *macro*

Description: Withdraw a publish statement.

Syntax: `(unpublish` *IN*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name
ABN - ABox name

unpublish-1 *function*

Description: Functional interface for `unpublish`.

Syntax: `(unpublish-1` *IN*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name
ABN - ABox name

subscribe *macro*

Description: Subscribe to an instance retrieval query.

Syntax: `(subscribe` *subscriber-name C*
`&optional (`*ABN* `(abox-name *current-abox*))`
`host port)`

Arguments: *subscriber-name* - subscriber name
C - concept term
ABN - ABox name
host - ip number of the host to which results are to be sent as a string
port - port number (integer)

Values: A list of tuples consisting of subscriber and individuals names.

subscribe-1 *function*

Description: Functional interface for `subscribe`.

Syntax: `(subscribe-1` *subscriber-name C*
`&optional (`*ABN* `(abox-name *current-abox*))`
`host port)`

Arguments: *subscriber-name* - subscriber name
C - concept term
ABN - ABox name
host - ip number of the host to which results are to be sent as a string
port - port number (integer)

unsubscribe *macro*

Description: Retract a subscription.

Syntax: `(unsubscribe` *subscriber-name*
`&optional` *C* `(`*ABN* `(abox-name *current-abox*)))`

Arguments: *subscriber-name* - subscriber name

C - concept term

ABN - ABox name

unsubscribe-1 *function*

Description: Functional interface for `unsubscribe`.

Syntax: `(unsubscribe` *subscriber-name*
`&optional` *C* `(`*ABN* `(abox-name *current-abox*)))`

Arguments: *subscriber-name* - subscriber name

C - concept term

ABN - ABox name

init-subscriptions *macro*

Description: Initialize the subscription database.

Syntax: `(init-subscriptions &optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *ABN* - ABox name

init-subscriptions-1 *function*

Description: Functional interface for `init-subscriptions`

Syntax: `(init-subscriptions-1 &optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *ABN* - ABox name

init-publications *macro*

Description: Initialize the set of published individuals.

Syntax: `(init-publications &optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *ABN* - ABox name

init-publications-1 *function*

Description: Functional interface for `init-subscription`.

Syntax: `(init-publications &optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *ABN* - ABox name

check-subscriptions *macro*

Description: Explicitly check for new instance retrieval results w.r.t. the set of subscriptions.

Syntax: `(check-subscriptions` *ABN*`)`

Arguments: *ABN* - ABox name

Values: A list of tuples consisting of subscriber and individuals names.

13 The Racer Persistency Services

If you load some knowledge bases into Racer and ask some queries, Racer builds internal data structure that enables the system to provide for faster response times. However, generating these internal data structures takes some time. So, if the Racer Server is shut down, all this work is usually lost, and data structures have to be rebuilt when the server is restarted again. In order to save time at server startup, Racer provides a facility to "dump" the server state into a file and restore the state from the file at restart time. The corresponding functions form the Persistency Services of a Racer Server. The Persistency Services can also be used to "prepare" a knowledge base at a specific server and use it repeatedly at multiple clients (see also the documentation about the Racer Proxy). For instance, you can classify a TBox or realize an ABox and dump the resulting data structures into a file. The file(s) can be reloaded and multiple servers can restart with much less computational ressources (time and space). Starting from a dump file is usually about ten times faster than load the corresponding text files and classifying the TBox (or realizing the ABox) again.

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

---

store-tbox-image *function*

Description: Store an image of a TBox.

Syntax: `(store-tbox-image` *filename* `&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments: *filename* - filename

TBN - tbox name

---

store-tboxes-image *function*

Description: Store an image of a list of TBoxes.

Syntax: `(store-tboxes-image` *tboxes* *filename*`)`

Arguments: *tboxes* - a list of TBox names

filename - filename

---

restore-tbox-image *function*

Description: Restore an image of a TBox.

Syntax: `(restore-tbox-image` *filename*`)`

Arguments: *filename* - filename

restore-tboxes-image *function*

Description: Restore an image of a set of TBoxes.

Syntax: `(restore-tboxes-image` *filename*`)`

Arguments: *filename* - filename

store-abox-image *function*

Description: Store an image of an Abox.

Syntax: `(store-abox-image` *filename* `&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *filename* - filename
ABN - abox name

store-aboxes-image *function*

Description: Store an image of a list of Aboxes.

Syntax: `(store-aboxes-image` *aboxes filename*`)`

Arguments: *aboxes* - a list of abox names
filename - filename

restore-abox-image *function*

Description: Restore an image of an Abox.

Syntax: `(restore-abox-image` *filename*`)`

Arguments: *filename* - filename

restore-aboxes-image *function*

Description: Restore an image of a set of aboxes.

Syntax: `(restore-aboxes-image` *filename*`)`

Arguments: *filename* - filename

store-kb-image *function*

Description: Store an image of an kb.

Syntax: `(store-kb-image` *filename* `&optional (`*KBN* `(tbox-name *current-tbox*)))`

Arguments: *filename* - filename

KBN - kb name

store-kbs-image *function*

Description: Store an image of a list of kbs.

Syntax: `(store-kbs-image` *kbs filename*`)`

Arguments: *kbs* - a list of knowledge base names

filename - filename

restore-kb-image *function*

Description: Restore an image of an kb.

Syntax: `(restore-kb-image` *filename*`)`

Arguments: *filename* - filename

restore-kbs-image *function*

Description: Restore an image of a set of kbs.

Syntax: `(restore-kbs-image` *filename*`)`

Arguments: *filename* - filename

14 The Racer Proxy

The Racer Proxy is a program controlling the communication between multiple client programs and a Racer Server. In addition, the Racer Proxy provides new services for client programs. The Racer Proxy is written in Java and is provided with source code for non-commercial research purposes.

14.1 Installation and Configuration

14.2 Multiuser-Access to a Racer Server

14.3 Load Balancing Using Multiple Racer Servers

14.4 Extension of the Publish-Subscribe Mechanism

14.5 Persistency and Logging

15 Reporting Errors and Inefficiencies

Although RACER has been used in some application projects and version 1.7 has been extensively tested, it might be the case that you detect a bug. In this case, please send us the knowledge base together with the query. 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 RACER.

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

The following function provide a way for you to collect the statements sent to the RACER server.

logging-on *macro*

Description: Start logging of expressions to the Racer server.

Syntax: `(logging-on` *filename*`)`

Arguments: *filename* - filename

Values: None.

Remarks: RACER must have been startedin unsafe mode (option -u) to use this facility. Logging is only available in the RACER server version.

logging-off *macro*

Description: Start logging of expressions to the Racer server.

Syntax: `(logging-off )`

Arguments:

Values: None.

Remarks: Logging is only available in the RACER server version.

16 What comes next?

Future releases of RACER will provide:

- Role equality (in particular for the DAML interface)
- Feature chains for $\mathcal{ALC}(\mathcal{D})$ knowledge bases
- Feature chain equality for $\mathcal{ALCF}(\mathcal{D})$ knowledge bases
- Support for default reasoning and support for iteratively finding models for concepts and ABoxes.
- Support for (additional) datatypes in DAML and OWL
- Support for complete reasoning on SHOQ knowledge bases
- Option to switch off the unique name assumption in ABoxes.
- Support for exporting and importing knowledge bases to and from the XMI format used by UML-based software engineering tools (see Figure 30). Although UML can represent only part of, e.g., DAML knowledge bases, even just exporting class hiearchies using XMI might be interesting because with tools such as ArgoUML (see Figure 30) Java code can be generated and models based on description logic may be used in Java-based environments.

The order in this list says nothing about priority.

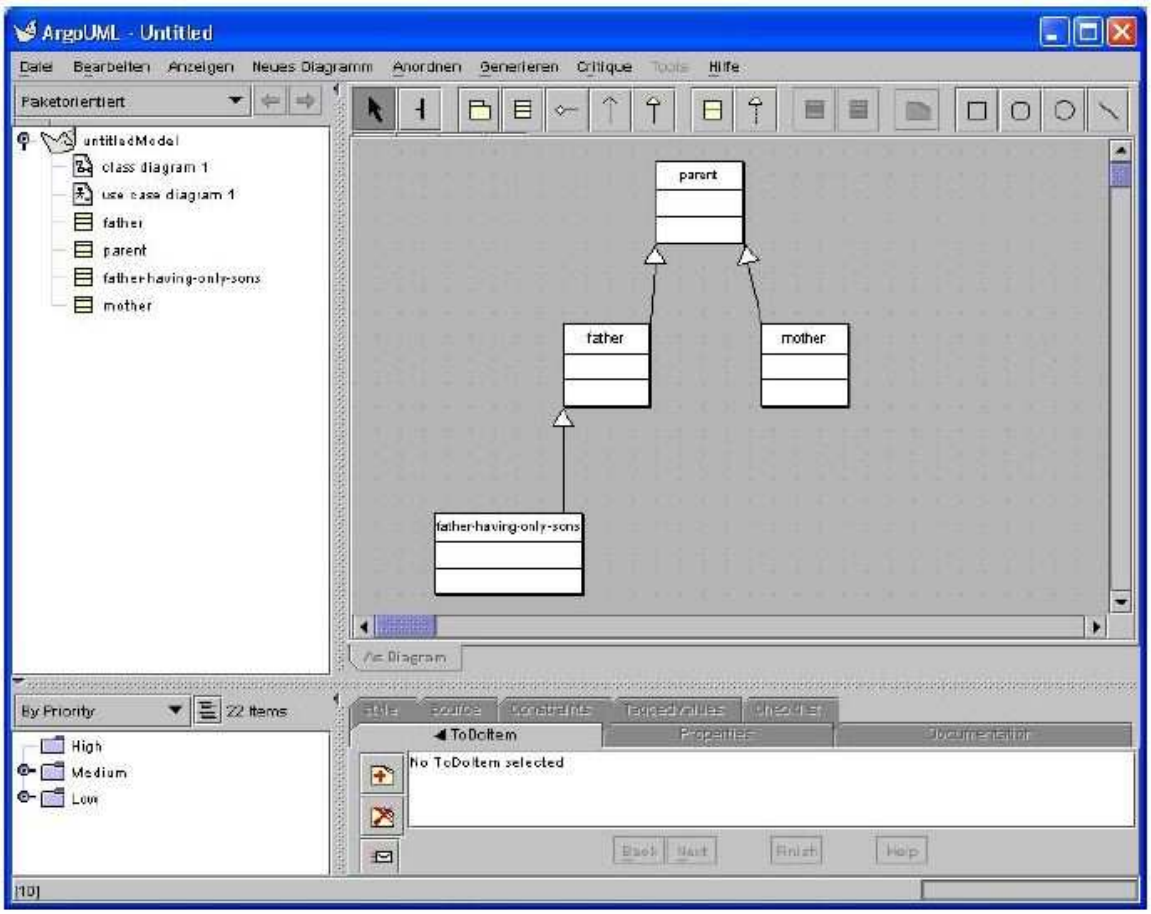

Figure 30: Concept hierarchy for the family TBox in ArgoUML.

A Integrated Sample Knowledge Base

This section shows an integrated version of the family knowledge base.

```
;;;============================================================
;;;  the following forms are assumed to be contained in a
;;;  file "RACER:examples;family.racer".

(in-knowledge-base family smith-family)

(signature :atomic-concepts (person human female male woman man
                             parent mother father grandmother
                             aunt uncle sister brother)
           :roles ((has-descendant :transitive t)
                   (has-child :parent has-descendant)
                   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)))

;;; the concepts
(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)))
```

```
;;; Alice is the mother of Betty and Charles
(instance alice mother)
(related alice betty has-child)
(related alice charles has-child)

;;; Betty is mother of Doris and Eve
(instance betty mother)
(related betty doris has-child)
(related betty eve has-child)

;;; Charles is the brother of Betty (and only Betty)
(instance charles brother)
(related charles betty has-sibling)
;;; closing the role has-sibling for charles
(instance charles (at-most 1 has-sibling))

;;; Doris has the sister Eve
(related doris eve has-sister)

;;; Eve has the sister Doris
(related eve doris has-sister)
```

B An Excerpt of the Family Example in DAML Syntax

```
;;;============================================================
;;;  the following forms are assumed to be contained in a
;;;  file "file:c:/ralf/family-2/family.daml".

<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:ns0="file:c:/ralf/family-2/"
   xmlns:oiled="http://img.cs.man.ac.uk/oil/oiled#"
   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#">
   <daml:Ontology rdf:about="">
      <dc:title>FAMILY</dc:title>
      <dc:date>5.6.2002 14:1</dc:date>
      <dc:creator></dc:creator>
      <dc:description></dc:description>
      <dc:subject></dc:subject>
      <daml:versionInfo></daml:versionInfo>
   </daml:Ontology>
   <daml:ObjectProperty rdf:about="file:c:/ralf/family-2/has-descendant">
      <rdfs:label>HAS-DESCENDANT</rdfs:label>
      <rdfs:comment></rdfs:comment>
      <oiled:creationDate></oiled:creationDate>
      <oiled:creator>RACER</oiled:creator>
   </daml:ObjectProperty>
   <daml:TransitiveProperty rdf:about="file:c:/ralf/family-2/has-descendant"/>
   <daml:ObjectProperty rdf:about="file:c:/ralf/family-2/has-child">
      <rdfs:label>HAS-CHILD</rdfs:label>
      <rdfs:comment></rdfs:comment>
      <oiled:creationDate></oiled:creationDate>
      <oiled:creator>RACER</oiled:creator>
      <rdfs:subPropertyOf rdf:resource="file:c:/ralf/family-2/has-descendant"/>
      <rdfs:domain>
         <daml:Class rdf:about="file:c:/ralf/family-2/parent"/>
      </rdfs:domain>
      <rdfs:range>
         <daml:Class rdf:about="file:c:/ralf/family-2/person"/>
      </rdfs:range>
   </daml:ObjectProperty>
```

```
<daml:ObjectProperty rdf:about="file:c:/ralf/family-2/has-gender">
   <rdfs:label>HAS-GENDER</rdfs:label>
   <rdfs:comment></rdfs:comment>
   <oiled:creationDate></oiled:creationDate>
   <oiled:creator>RACER</oiled:creator>
</daml:ObjectProperty>
<daml:UniqueProperty rdf:about="file:c:/ralf/family-2/has-gender"/>
<daml:Class rdf:about="file:c:/ralf/family-2/male">
   <daml:disjointWith>
   <daml:Class rdf:about="file:c:/ralf/family-2/female"/>
   </daml:disjointWith>
</daml:Class>
<daml:Class rdf:about="file:c:/ralf/family-2/person">
  <rdfs:subClassOf>
    <daml:Class>
      <daml:intersectionOf>
        <daml:List>
          <daml:first>
            <daml:Class rdf:about="file:c:/ralf/family-2/human"/>
          </daml:first>
          <daml:rest>
            <daml:List>
              <daml:first>
                <daml:Restriction>
                  <daml:onProperty rdf:resource="file:c:/ralf/family-2/has-gender"/>
                  <daml:hasClass>
                    <daml:Class>
                      <daml:unionOf>
                        <daml:List>
                          <daml:first>
                            <daml:Class rdf:about="file:c:/ralf/family-2/female"/>
                          </daml:first>
                          <daml:rest>
                            <daml:List>
                              <daml:first>
                                <daml:Class rdf:about="file:c:/ralf/family-2/male"/>
                              </daml:first>
                              <daml:rest>
                                <daml:nil/>
                              </daml:rest>
                            </daml:List>
```

```
                        </daml:rest>
                      </daml:List>
                    </daml:unionOf>
                  </daml:Class>
                </daml:hasClass>
              </daml:Restriction>
            </daml:first>
            <daml:rest>
              <daml:nil/>
            </daml:rest>
          </daml:List>
        </daml:rest>
      </daml:List>
    </daml:intersectionOf>
  </daml:Class>
 </rdfs:subClassOf>
</daml:Class>

<daml:Class rdf:about="file:c:/ralf/family-2/woman">
  <rdfs:subClassOf>
    <daml:Class>
      <daml:intersectionOf>
        <daml:List>
          <daml:first>
            <daml:Class rdf:about="file:c:/ralf/family-2/person"/>
          </daml:first>
          <daml:rest>
            <daml:List>
              <daml:first>
                <daml:Restriction>
                  <daml:onProperty rdf:resource="file:c:/ralf/family-2/has-gender"/>
                  <daml:hasClass>
                    <daml:Class rdf:about="file:c:/ralf/family-2/female"/>
                  </daml:hasClass>
                </daml:Restriction>
              </daml:first>
              <daml:rest>
                <daml:nil/>
              </daml:rest>
            </daml:List>
          </daml:rest>
        </daml:List>
      </daml:intersectionOf>
    </daml:Class>
  </rdfs:subClassOf>
</daml:Class>
```

```
   <rdf:Description rdf:about="file:c:/ralf/family-2/charles">
      <rdf:type rdf:resource="file:c:/ralf/family-2/only-child"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/charles">
      <rdf:type rdf:resource="file:c:/ralf/family-2/brother"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/betty">
      <rdf:type rdf:resource="file:c:/ralf/family-2/mother"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/alice">
      <rdf:type rdf:resource="file:c:/ralf/family-2/mother"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/alice">
      <ns0:has-descendant rdf:resource="file:c:/ralf/family-2/eve"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/alice">
      <ns0:has-descendant rdf:resource="file:c:/ralf/family-2/doris"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/eve">
      <ns0:has-sister rdf:resource="file:c:/ralf/family-2/doris"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/doris">
      <ns0:has-sister rdf:resource="file:c:/ralf/family-2/eve"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/charles">
      <ns0:has-sibling rdf:resource="file:c:/ralf/family-2/betty"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/betty">
      <ns0:has-child rdf:resource="file:c:/ralf/family-2/eve"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/betty">
      <ns0:has-child rdf:resource="file:c:/ralf/family-2/doris"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/alice">
      <ns0:has-child rdf:resource="file:c:/ralf/family-2/charles"/>
   </rdf:Description>
   <rdf:Description rdf:about="file:c:/ralf/family-2/alice">
      <ns0:has-child rdf:resource="file:c:/ralf/family-2/betty"/>
   </rdf:Description>
</rdf:RDF>
```

C Another Family Knowledge Base

In this section we present another family knowledge base (see the file `family-2.racer` in the examles folder).

```
(in-knowledge-base family)

(define-primitive-role descendants :transitive t)

(define-primitive-role children :parents (descendants))

(implies (and male female) *bottom*)
(equivalent man (and male human))
(equivalent woman (and female human))
(equivalent parent (at-least 1 children))
(equivalent grandparent (some children parent))
(equivalent mother (and woman parent))
(equivalent father (and man parent))
(implies (some descendants human) human)
(implies human (all descendants human))
(equivalent father-having-only-male-children (and father human (all children male)))
(equivalent father-having-only-sons (and man
                                         (at-least 1 children)
                                         (all children man)))
(equivalent grandpa (and male (some children (and parent human))))
(equivalent great-grandpa (and male
                               (some children (some children (and parent human)))))


(instance john male)
(instance mary female)
(related john james children)
(related mary james children)
(instance james (and human male))
(instance john (at-most 1 children))


(individual-direct-types john)
(individual-direct-types mary)
(individual-direct-types james)
```

D A Knowledge Base with Concrete Domains

In this section we present another family knowledge base with concrete domains (see the file `family-3.racer` in the examles folder).

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

(signature :atomic-concepts (human female male woman man
                                parent mother father
                                mother-having-only-female-children
                                mother-having-only-daughters
                                mother-with-children
                                mother-with-siblings
                                mother-having-only-sisters
                                grandpa great-grandpa
                                grandma great-grandma
                                aunt uncle
                                sister brother sibling
                                young-parent normal-parent old-parent
                                child teenager teenage-mother
                                young-human adult-human
                                old-human young-child
                                human-with-fever
                                seriously-ill-human
                                human-with-high-fever)
           :roles ((has-descendant :domain human :range human
                                   :transitive t)
                   (has-child :domain parent
                              :range child
                              :parent has-descendant)
                   (has-sibling :domain sibling :range sibling)
                   (has-sister :range sister
                               :parent has-sibling)
                   (has-brother :range brother
                                :parent has-sibling))
           :features ((has-gender :range (or female male)))
           :attributes ((integer has-age)
                        (real temperature-fahrenheit)
                        (real temperature-celsius))
           :individuals (alice betty charles doris eve)
           :objects (age-of-alice age-of-betty age-of-charles
                                  age-of-doris age-of-eve
                                  temperature-of-doris
                                  temperature-of-charles))
```

```
;; the concepts
(disjoint female male human)
(implies human (and (at-least 1 has-gender) (a has-age)))
(implies human (= temperature-fahrenheit
                  (+ (* 1.8 temperature-celsius) 32)))

(equivalent young-human (and human (max has-age 20)))
(equivalent teenager (and young-human (min has-age 10)))
(equivalent adult-human (and human (min has-age 21)))
(equivalent old-human (and human (min has-age 60)))
(equivalent woman (and human (all has-gender female)))
(equivalent man (and human (all has-gender male)))
(implies child human)
(equivalent young-child (and child (max has-age 9)))

(equivalent human-with-fever
            (and human (>= temperature-celsius 38.5)))
(equivalent seriously-ill-human
            (and human (>= temperature-celsius 42.0)))
(equivalent human-with-high-fever
            (and human (>= temperature-fahrenheit 107.5)))


(equivalent parent (at-least 1 has-child))
(equivalent young-parent (and parent (max has-age 21)))
(equivalent normal-parent (and parent (min has-age 22) (max has-age 40)))
(equivalent old-parent (and parent (min has-age 41)))
(equivalent mother (and woman parent))
(equivalent father (and man parent))
(equivalent teenage-mother (and mother (max has-age 20)))

(equivalent mother-having-only-female-children
            (and mother
                 (all has-child (all has-gender (not male)))))
(equivalent mother-having-only-daughters
            (and woman
                 (at-least 1 has-child)
                 (all has-child woman)))
(equivalent mother-with-children
            (and mother (at-least 2 has-child)))
(equivalent grandpa (and man (some has-child parent)))
(equivalent great-grandpa
            (and man (some has-child (some has-child parent))))
```

```
(equivalent grandma (and woman (some has-child parent)))
(equivalent great-grandma
            (and woman (some has-child (some has-child parent))))
(equivalent aunt (and woman (some has-sibling parent)))
(equivalent uncle (and man (some has-sibling parent)))

(equivalent sibling (or sister brother))
(equivalent mother-with-siblings (and mother (all has-child sibling)))
(equivalent brother (and man (at-least 1 has-sibling)))
(equivalent sister (and woman (at-least 1 has-sibling)))

(implies (at-least 2 has-child) (all has-child sibling))
;(implies (some has-child sibling) (at-least 2 has-child))

(implies sibling (all (inv has-child) (and (all has-child sibling)
                                           (at-least 2 has-child))))
(equivalent mother-having-only-sisters
            (and mother
                 (all has-child (and sister
                                     (all has-sibling sister)))))


;; Alice is the mother of Betty and Charles
(instance alice (and woman (at-most 2 has-child)))
;; Alice's age is 45
(constrained alice age-of-alice has-age)
(constraints (equal age-of-alice 45))
(related alice betty has-child)
(related alice charles has-child)

;; Betty is mother of Doris and Eve
(instance betty (and woman (at-most 2 has-child)))
;; Betty's age is 20
(constrained betty age-of-betty has-age)
(constraints (equal age-of-betty 20))
(related betty doris has-child)
(related betty eve has-child)
(related betty charles has-sibling)
;; closing the role has-sibling for charles
(instance betty (at-most 1 has-sibling))
```

```
; Charles is the brother of Betty (and only Betty)
(instance charles brother)
;; Charles's age is 39
(constrained charles age-of-charles has-age)
(constrained charles temperature-of-charles temperature-fahrenheit)
(constraints (equal age-of-charles 39) (= temperature-of-charles 107.6))
(related charles betty has-sibling)
;; closing the role has-sibling for charles
(instance charles (at-most 1 has-sibling))

;; Doris has the sister Eve
(related doris eve has-sister)
(instance doris (at-most 1 has-sibling))
;; Doris's age is 2
(constrained doris age-of-doris has-age)
(constrained doris temperature-of-doris temperature-celsius)
(constraints (equal age-of-doris 2) (= temperature-of-doris 38.6))

;; Eve has the sister Doris
(related eve doris has-sister)
(instance eve (at-most 1 has-sibling))
;; Eve's age is 1
(constrained eve age-of-eve has-age)
(constraints (equal age-of-eve 1))
```

```
;;; some TBox queries
;; are all uncles brothers?
(concept-subsumes? brother uncle)

;; get all super-concepts of the concept mother
(concept-ancestors mother)

;; get all sub-concepts of the concept man
(concept-descendants man)

;; get all transitive roles in the TBox family
(all-transitive-roles)

;;; some ABox queries
;; Is Doris a woman?
(individual-instance? doris woman)

;; Of which concepts is Eve an instance?
(individual-types eve)

;; get all descendants of Alice
(individual-fillers alice has-descendant)

(individual-direct-types eve)

(concept-instances sister)

(describe-individual doris)

(describe-individual charles)
```

References

[Bechhofer et al. 01] S. Bechhofer, I. Horrocks, C. Goble, R. Stevens, "OilEd: a Reasonable Ontology Editor for the Semantic Web", in *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, September 19-21, Vienna, Springer-Verlag LNAI Vol. 2174, pp 396–408, 2001

[Bechhofer 02] S. Bechhofer, "The DIG Description Logic Interface: DIG/1.0", Technical Report, University of Manchester, 2000. http://potato.cs.man.ac.uk/dig/interface1.0.pdf

[Buchheit et al. 93] M. Buchheit, F.M. Donini & A. Schaerf, "Decidable Reasoning in Terminological Knowledge Representation Systems", in *Journal of Artificial Intelligence Research*, 1, pp. 109-138, 1993.

[Haarslev and Möller 2000] Haarslev, V. and Möller, R. (2000), "Expressive ABox reasoning with number restrictions, role hierarchies, and transitively closed roles", in *Proceedings of Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR'2000), Cohn, A., Giunchiglia, F., and Selman, B., editors, Breckenridge, Colorado, USA, April 11-15, 2000, pages 273–284.*

[Horrocks 98] I. Horrocks, "Using an Expressive Description Logic: FaCT or Fiction?", in *Proceedings of Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy*, Editors: Cohn, T. and Schubert, L. and Shapiro, S. 1998, pages 636–647.

[Horrocks-et-al. 99a] I. Horrocks, U. Sattler, S. Tobies, "Practical Reasoning for Description Logics with Functional Restrictions, Inverse and Transitive Roles, and Role Hierarchies", Proceedings of the 1999 Workshop Methods for Modalities (M4M-1), Amsterdam, 1999.

[Horrocks-et-al. 99b] I. Horrocks, U. Sattler, S. Tobies, "A Description Logic with Transitive and Converse Roles, Role Hierarchies and Qualifying Number Restrictions", Technical Report LTCS-99-08, RWTH Aachen, 1999.

[Horrocks et al. 2000] Horrocks, I., Sattler, U., and Tobies, S. (2000). Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In MacAllester, D., editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, Lecture Notes in Computer Science, Germany. Springer Verlag.

[Patel-Schneider and Swartout 93] P.F. Patel-Schneider, B. Swartout "Description Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing Effort", November 1993. The paper is available as: http://www-db.research.bell-labs.com/user/pfps/papers/krss-spec.ps

[Rychlik 2000] Rychlik, M., Complexity and applications of parametric algorithms of computational algebraic geometry. In: R. del la Llave, L. Petzold, and J. Lorenz, editors, *Dynamics of Algorithms*, volume 118 of The IMA Volumes in Mathematics and its Applications, pp. 1-29, New York, 2000. Springer-Verlag.

[Weispfenning 92] Weispfenning, V., Comprehensive Groebner Bases, Journal of Symbolic Computation, volume 14, 1992, pp 1-29.

[Jaffar and Maher 94] Constraint Logic Programming: A Survey. J. Jaffar, M. Maher, In: Journal of Logic Programming, Vol. 19/20, pp. 503-581, 1994.

Index