RACER User's Guide and Reference Manual Version 1.6.1

Volker Haarslev* and **Ralf Möller****

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

**Univ. of Appl. Sciences in Wedel
Computer Science Department
Feldstrasse 143
22880 Wedel, Germany
`rmoeller@fh-wedel.de`

November 20, 2001

Contents

1 Introduction

The RACER[1] system is a knowledge representation system that implements a highly optimized tableaux 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.

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 supports most of the functions specified in the Knowledge Representation System Specification (KRSS), for details see [Patel-Schneider and Swartout 93].

RACER is implemented in ANSI Common Lisp and has been developed at the University of Hamburg.

2 Obtaining and Running RACER

The RACER system can be obtained from the following web site:
`http://kogs-www.informatik.uni-hamburg.de/~race/`

2.1 System Installation

For the Macintosh execute the self-extracting archive `<filename>.sea`.

For UNIX and Windows systems decompress the archive file after downloading. For UNIX use the command: `gzip -dc <filename>.tar.gz | tar -xf -`
Under Windows unzip the file: `<filename>.zip`

This creates the files and directories of the distribution. Then follow the instructions in the file `readme.txt` found in the directory `<filename>`. It is important that you *load* the file `load-racer.lisp` from the Lisp Listener. Do not evaluate or compile this file. This file declares logical pathnames which are used in the example TBoxes.

2.2 Sample Session

All the files used in this example are in the directory `"racer:examples;"`. The queries are in the file `"family-queries.lisp"`.

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

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

[1]RACER stands for **R**easoner for **A**Boxes and **C**oncept **E**xpressions **R**enamed

```
;;; 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)))
```

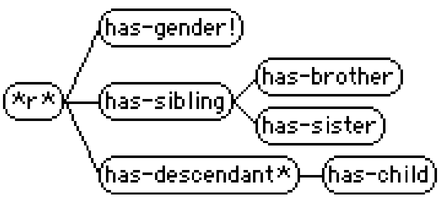

Figure 1: Role hierarchy for the `family` TBox.
r denotes the internally defined universal role.
! denotes features
∗ denotes transitive roles

The RACER Session:

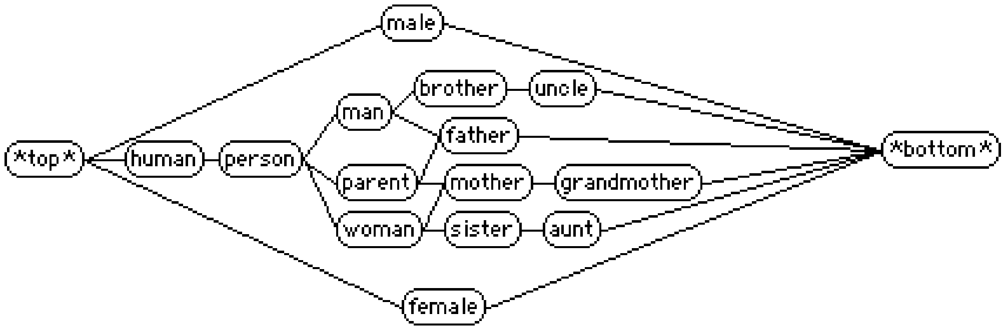

Figure 2: Concept hierarchy for the `family` TBox.

```
;;; load the TBox
CL-USER(1): (load "racer:examples;family-tbox.lisp")
;;; Loading racer:examples;family-tbox.lisp
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.lisp".

;;; 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)
```

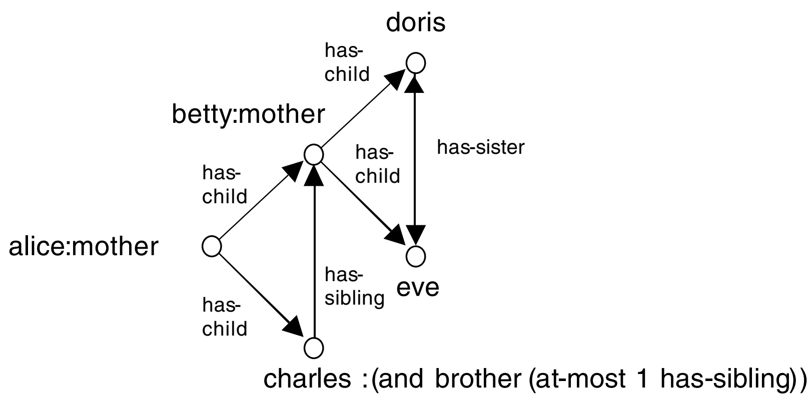

Figure 3: Depiction of the ABox `smith-family`.
(with the explicitly given information being shown)

The RACER Session:

```
;;; now load the ABox
CL-USER(6): (load "racer:examples;family-abox.lisp")
;;; Loading racer:examples;family-abox.lisp
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 71, you find a version in KRSS syntax and in Appendix B, on page 73, a version where the TBox and ABox are integrated.

2.3 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
IN	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

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.

$$
\begin{array}{lll}
C \longrightarrow & CN & | \\
& \texttt{*top*} & | \\
& \texttt{*bottom*} & | \\
& \texttt{(not } C\texttt{)} & | \\
& \texttt{(and } C_1 \ \ldots \ C_n\texttt{)} & | \\
& \texttt{(or } C_1 \ \ldots \ C_n\texttt{)} & | \\
& \texttt{(some } R \ C\texttt{)} & | \\
& \texttt{(all } R \ C\texttt{)} & | \\
& \texttt{(at-least } n \ R\texttt{)} & | \\
& \texttt{(at-most } n \ R\texttt{)} & | \\
& \texttt{(exactly } n \ R\texttt{)} & | \\
& \texttt{(at-least } n \ R \ C\texttt{)} & | \\
& \texttt{(at-most } n \ R \ C\texttt{)} & | \\
& \texttt{(exactly } n \ R \ C\texttt{)} & | \\
& CDC & \\
& & \\
R \longrightarrow & RN & | \\
& \texttt{(inv } RN\texttt{)} &
\end{array}
$$

Figure 4: RACER concept and role terms (for concrete domain concepts (*CDC*) see Figure 3).

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[2].

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 4.

[2]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!

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 ... C_n`)`
Disjunction	$C_1 \sqcup \ldots \sqcup C_n$	`(or` C_1 ... C_n`)`

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

	DL notation	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 `(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*`[3].

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

```
CDC  ⟶   (a AN)                  |
          (an AN)                 |
          (no AN)                 |
          (min AN integer)        |
          (max AN integer)        |
          (> aexpr aexpr)         |
          (>= aexpr aexpr)        |
          (< aexpr aexpr)         |
          (<= aexpr aexpr)        |
          (= aexpr aexpr)

aexpr  ⟶  AN                      |
          real                    |
          (+ aexpr1 aexpr1*)      |
          aexpr1

aexpr1  ⟶ real                    |
          AN                      |
          (* real AN)
```

Figure 5: RACER concrete domain concepts and attribute expressions.

Concrete domain concepts state concrete predicate restrictions for attribute fillers (see Figure 3). RACER currently supports two unary predicates for integer attributes (`min`, `max`), five 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 where the attribute names play the role of variables. The use of these concepts is illustrated in Section 3.4.

	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	$\exists A.min_z$	`(min A` z`)`
with $z \in \mathbb{Z}$	$\exists A.max_z$	`(max A` z`)`
Real predicate exists restriction	$\exists A_1, \ldots, A_n.P$	`(`P *aexpr aexpr*`)`
with $P \in \{>, >=, <, <=, =\}$		

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 the disjointness between several concepts. Disjoint concepts do not have instances in common.
DL notation: $C_1 \sqcap \cdots \sqcap C_n \doteq \bot$
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 5.3, page 33) 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 6. 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 6: 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`))`.

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 7: Domain and range restrictions expressed via GCIs.

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 7 (cf. [Buchheit et al. 93]) or declared via the keywords `:domain` and `:range` (e.g. see the description of `define-primitive-role` in Section 5.3, page 33).

3.4 Concrete Domains

Racer supports reasoning over the integers ($\mathbb{Z}$) and the rationals ($\mathbb{Q}$) in combination with linear inequations. However, for the convenience of the users the rational numbers are specified in floating point notation and automatically transformed into their rational equivalents (e.g., 0.75 is transformed into 3/4). Therefore, the term ‘real’ is used as type designator although the reasoning is based on $\mathbb{Q}$. Names for values from these 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.

```
...
(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))
```

3.5 Concrete Domain Attributes

Attributes are considered as "typed" since they can either have fillers of type integer or real. 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)`. 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 4.1, page 17)) or by using the KRSS-like form `define-concrete-domain-attribute` (see Section 5.4, page 35).

3.6 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 two kinds of assertions:

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

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

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

Constraints state relationships between objects of the concrete domain.

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[4] or concepts[5] which are not mentioned in the TBox.

3.7 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.

[4]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).

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

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).

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.

3.8 Retraction and Incremental Additions

RACER offers constructs for retracting ABox assertions (see `forget`, `forget-concept-assertion` and `forget-role-assertion`). If a query has been answered and some assertions are retracted, then RACER might be forced to realize the ABox again, i.e. after retractions, some queries might take some time to answer.

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.

4 Knowledge Base Management Functions

This section documents the functions for managing TBoxes and ABoxes and for specifying queries.

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

Remarks: This function is only available in the server version of Racer. It is intended to load TBox and/or ABox declarations using Racer's Lisp syntax. Only the statements described in this document are accepted.

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

4.1 TBox Management

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 17.

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 2 and page 3.
For role definitions see `define-primitive-role`, on page 33, for feature definitions see `define-primitive-attribute`, on page 33, for attribute definitions see `define-concrete-domain-attribute`, on page 35.

---

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`.

---

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`.

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))`

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

tbox - TBox object

syntax - indicates the syntax of the TBox (only `:krss` is currently implemented).

transformed - if bound to `t` the TBox is saved in the format 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: TBox object

Remarks: A file may contain several TBoxes.
The usual way to load a TBox file is to use the Lisp function `load`.

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. 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 20.

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

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.

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

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

Values: TBox object

Remarks: Only RDFS descriptions are parsed, everyting else is ignored.

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

4.2 ABox Management

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 17.

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 17 is the macro counterpart. It allows to specify a signature for an ABox and a TBox with one call.

in-knowledge-base *macro*

Description: This form is an abbreviation for the sequence:

`(in-tbox` *TBN*`)`
`(in-abox` *ABN TBN*`)`

Syntax: `(in-knowledge-base` *TBN ABN* `&optional (`*name* `'cl-user))`

Arguments: *TBN* - TBox name

ABN - ABox name

name - Package name

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

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`.

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 (only `:krss` is currently implemented).

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`.

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 26.

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

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

5 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*`.

5.1 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.

5.2 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: $CN_1, \ldots, CN_n$ - 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*

5.3 Role Declarations

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 17.
Section 3.3 and Figure 7, on page 10 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 17.
Section 3.3 and Figure 7, on page 10 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 - if bound to `t` declares that *RN* is a concrete domain attribute.

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 7, on page 10 for domain and range restrictions.

5.4 Concrete Domain Attribute Declaration

define-concrete-domain-attribute *macro*

Description: Defines a concrete domain attribute.

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

Arguments: *AN* - attribute name

type - can be either bound to `integer` or `real`.

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 17 and Section 3.5.

5.5 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))
```

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 distinct. 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 ABox statements.

Syntax: `(forget &body forms)`

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

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).

5.6 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 6, and in Figure 3, page 8.

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 6, and in Figure 3, page 8.

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)`

6 Reasoning Modes

auto-classify *special-variable*

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

See also: Section 3.7 on page 13.

auto-realize *special-variable*

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

See also: Section 3.7 on page 13.

7 Evaluation Functions and Queries

7.1 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 43, and function `atomic-concept-synonyms`, on page 57.

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 57, and function `concept-subsumes-p`, on page 42.

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 57, and function `concept-subsumes-p`, on page 42.

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 $K_{(m)}$, $K4_{(m)}$ or $S4_{(m)}$ 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.

7.2 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`.

7.3 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-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`.

7.4 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.

7.5 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.

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 68,
Function `retrieve-related-individuals`, on page 67.

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.

8 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.

8.1 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 60.

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 43.

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 not included in the result.

See also: Function `concept-equivalent-p`, on page 43.

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

loop-over-tboxes *function*

Description: Iterator function for all TBoxes.

Syntax: (loop-over-tboxes (*tbox-variable*)
loop-clause
⋮)

Arguments: *tbox-variable* - variable for a TBox object

loop-clause - loop clause

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

Remarks: `(all-atomic-concepts (find-tbox 'my-tbox))`

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-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*.

8.2 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*)))

Arguments: *C* - concept term

ABN - ABox name

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*)

Arguments: *C* - concept term

abox - ABox object

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))
```

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 54.

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 54.

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 54.

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))
```

loop-over-aboxes *function*

Description: Iterator function for all ABoxes.

Syntax: `(loop-over-aboxes (`*abox-variable*`)`
  loop-clause
  ⋮ `)`

Arguments: *abox-variable* - variable for a ABox object

loop-clause - loop clause

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 70.

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 70.

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 70.

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: *IN* - individual name

abox - ABox object

Values: List of assertions

See also: Function `all-concept-assertions-for-individual` on page 69.

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*.

A KRSS Sample Knowledge Base

The following knowledge base is specified in KRSS syntax. It is a version of the knowledge base used in the Sample Session, on page 1.

A.1 KRSS Sample TBox

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

;;; initialize the TBox family
(in-tbox family :init t)

;;; the roles
(define-primitive-role has-child :parents (has-descendant))
(define-primitive-role has-descendant :transitive t)
(define-primitive-role has-sibling)
(define-primitive-role has-sister :parents (has-sibling))
(define-primitive-role has-brother :parents (has-sibling))
(define-primitive-attribute has-gender)

;;; 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
(define-primitive-concept person
    (and human (some has-gender (or female male))))
(define-disjoint-primitive-concept female (gender) top)
(define-disjoint-primitive-concept male (gender) top)
(define-primitive-concept woman (and person (some has-gender female)))
(define-primitive-concept man (and person (some has-gender male)))
(define-concept parent (and person (some has-child person)))
(define-concept mother (and woman parent))
(define-concept father (and man parent))
(define-concept grandmother
    (and mother
         (some has-child
               (some has-child person))))


(define-concept aunt (and woman (some has-sibling parent)))
(define-concept uncle (and man (some has-sibling parent)))
(define-concept brother (and man (some has-sibling person)))
(define-concept sister (and woman (some has-sibling person)))
```

A.2 KRSS Sample ABox

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

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

;;; 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 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-kb.lisp".

(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)
```

References

[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-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

Index