Fachbereich Informatik der Universität Hamburg

Vogt-Kölln-Str. 30 ◊ D-22527 Hamburg / Germany

University of Hamburg - Computer Science Department

Mitteilung Nr. 289 • Memo No. 289

RACE User's Guide and Reference Manual Version 1.1

Volker Haarslev, Ralf Möller, Anni-Yasmin Turhan

Arbeitsbereich KOGS

FBI-HH-M-289/99

Oktober 1999

RACE User's Guide and Reference Manual
Version 1.1

Volker Haarslev Ralf Möller Anni-Yasmin Turhan

{haarslev, moeller, turhan}@informatik.uni-hamburg.de
University of Hamburg, Computer Science Department
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany

October 5, 1999

Contents

1 Introduction

The RACE[1] system is a knowledge representation system that implements a highly optimized tableaux calculus for an expressive description logic. It offers reasoning services for multiple TBoxes and for multiple ABoxes as well. The system implements the description logic $\mathcal{ALCNH}_{R^+}$. This is the basic logic $\mathcal{ALC}$ augmented with number restrictions, role hierarchies and transitive roles.

RACE 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 RACE.

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

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

2 Obtaining and Running RACE

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

2.1 System Installation

For the Macintosh execute the self-extracting archive `race-1-1.sea` or unstuff the file `race-1-1.sit`.

For UNIX and Windows systems decompress the archive file after downloading. For UNIX use the command: `gzip -dc race-1-1.tar.gz | tar -xf -`
Under Windows unzip the file: `race-1-1.zip`

This creates the files and directories of the distribution. Then follow the instructions in the file `readme.txt`.

2.2 Sample Session

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

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

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

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

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

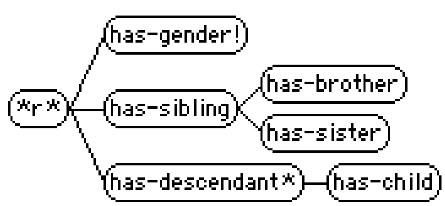

Figure 1: Role hierarchy for the `family` TBox.

r	denotes the universal role, which may not be used in knowledge bases.
!	denotes features
∗	denotes transitive roles

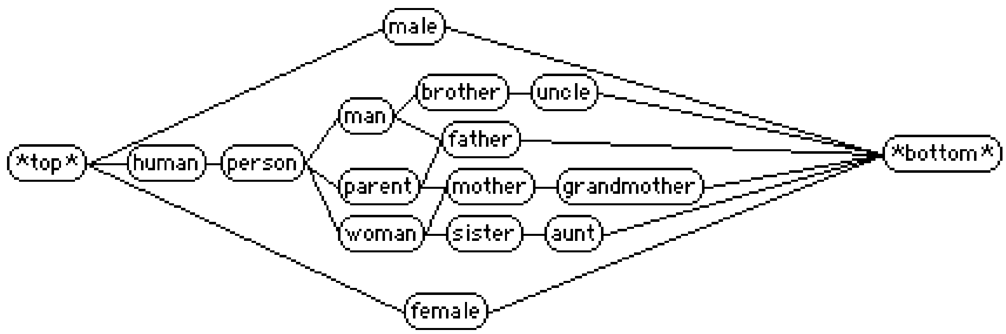

Figure 2: Concept hierarchy for the `family` TBox.

The RACE Session:

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

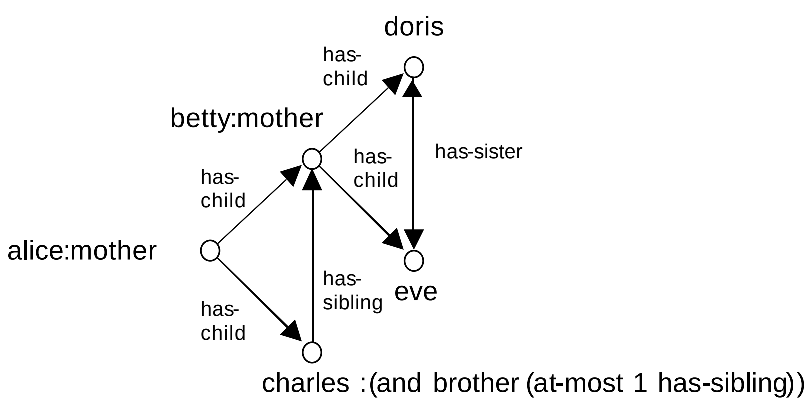

Figure 3: Depiction of the ABox `smith-family`.
(with the explicitly given information being shown)

The RACE Session:

```
;;; now load the ABox
CL-USER(6): (load "race:examples;family-abox.lisp")
;;; Loading race: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 are different versions of this knowledge base. In Appendix A, on page 53, you find a version in KRSS syntax and in Appendix B, on page 54, 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	*S*	List of Assertions
CN	Concept name	*GNL*	List of group names
IN	Individual name	*LCN*	List of concept names
RN	Role name	*abox*	ABox object
ABN	ABox name	*tbox*	TBox object
TBN	TBox name	*n*	a natural number
name	Name of any sort		

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 RACE 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 who 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.

3 RACE 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 a more detailed description of the concept language supported by RACE see [Haarslev and Möller 99].

3.1 Concept Language

The content of RACE 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 form the set $\mathcal{C}$ complex concept terms can be build using several operators. An overview over all concept building operators is given in Figure 4.

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

Figure 4: RACE concept terms

Boolean terms build concepts by using the boolean operators.

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

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	RACE syntax
Exists restriction	$\exists\, RN.C$	`(some` $RN \ C$`)`
Value restriction	$\forall\, RN.C$	`(all` $RN \ C$`)`

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

[2]The signature does not have to be specified explicitly in RACE 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!

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	RACE syntax
At-most restriction	$\leq n\ RN$	`(at-most` n RN`)`
At-least restriction	$\geq n\ RN$	`(at-least` n RN`)`
Exactly restriction	$= n\ RN$	`(exactly` n RN`)`

Actually, the exactly restriction `(exactly` n RN`)` is an abbreviation for the concept term: `(and (at-least` n RN`) (at-most` n RN`))`.

There are two concepts implicitly 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 RACE $\top$ is denoted as `*top*` and $\bot$ is denoted as `*bottom*`[3].

3.2 Concept Axioms and Terminology

RACE supports several kinds of concept axioms.

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

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

Concept disjointness axioms states the disjointness between several concepts. Disjoint concepts do not have instances in common.
DL notation: $C_1 \sqcap \cdots \sqcap C_n \doteq \bot$
RACE 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 RACE 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: ($CN \sqsubseteq C$)
RACE syntax: `(define-primitive-concept` CN C`)`

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

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

Concept axioms may be cyclic in RACE. 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 RACE 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 RACE. There exists just one declaration per role name in a knowledge base. If a second declaration for a role is given, an error is signalled. 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.

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 5. The table reads 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 5: 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.

RACE 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 in RACE be approximated by: `(define-primitive-role` RN `:parents (`RN_1 RN_2`))`.

RACE does not offer constructs for domain and range restrictions for roles. These restrictions for primitive roles can be simulated with GCIs, see the examples in Figure 6 (cf. [Buchheit et al. 93]).

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

Figure 6: Domain and range restrictions expressed via GCIs.

3.4 ABox Assertions

An ABox contains assertions about individuals. The set of individual names $\mathcal{I}$ 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 RN with respect to an individual IN_2.

In RACE the *unique name assumption* holds, this means that all individual names used in an ABox refer to distinct individuals, therefore two names cannot refer to the same individual.

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

3.5 Inference Modes

After the declaration of a TBox or an ABox, RACE 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 RACE ensures the following behavior: Depending on the kind of query, RACE tries to be as smart as possible to locally minimize computation time (lazy inference mode). For instance, in order to answer a subsumption query wrt. a TBox it is not necessary

[4]These concepts are assumed to be atomic concepts and 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).

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), RACE 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).

The inference behavior of RACE 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 RACE 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 separated 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, RACE ensures that the knowledge bases are in the appropriate states. This behavior usually guarantees minimum runtimes for answering queries.

3.6 Retraction and Incremental Additions

RACE 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 RACE might be forced to realize the ABox again, i.e. after retractions, some queries might take some time to answer.

RACE 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.

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* `nil))`

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

init - boolean indicating if the TBox should be initialized.

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 11.

---

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* `&optional (`*class* `'tbox))`

Arguments: *tbox* - TBox object

class - class inheriting from the class `tbox`

Values: *tbox*

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

signature *macro*

Description: Defines the signature for a knowledge base.

If the keywords *atomic-concepts* and *roles* are used. The *current-tbox* is initialized and the signature is defined for it.

If the keyword *individualnames* 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)`
`(`*individuals* `nil))`

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

roles - is a list of all role declarations, thereby also specifying $\mathcal{R}$.

individuals - is a list of individual names, specifying $\mathcal{I}$.

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 RACE 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))
    (has-coach :feature t)))
```

Signature for an ABox:

```
(signature :individuals (Charlie-Brown Snoopy ... ))
```

Signature for a TBox and an ABox:

```
(signature
  :atomic-concepts (Character Baseball-Player ... )
  :roles ((has-pet)
    (has-dog :parents (has-pet))
    (has-coach :feature t))
  :individuals (Charlie-Brown Snoopy ... ))
```

See also: Section Sample Session, on page 1 and page 3.
For role definitions see `define-primitive-role`, on page 22.

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))`

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

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

roles - is a list of all role declarations, thereby also specifying $\mathcal{R}$.

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).

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. It might as well be names of other DL systems.

transformed - if bound to `t` the TBox is saved in the format after preprocessing by RACE.

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)
```

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 signalled if the TBox is not found.

Values: TBox object

Remarks: This function can also be used to get rid of TBoxes or 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

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.

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

Usually this macro is used at top of a file containing an ABox. This macro can also be used to create new ABoxes.

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 11.

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*)`
`(`*class* `'standard-abox))`

Arguments:
abox - ABox object to initialize.

tbox - TBox object associated with the ABox

class - class of the new ABox object, that must inherit from the class `standard-abox`.

Values: ABox object

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))`

Arguments:
abox - ABox object

individuals - is a list of individual names, specifying $\mathcal{I}$.

See also: Macro `signature` on page 11 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*`)`

Arguments: *TBN* - TBox name

ABN - ABox 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 macro `in-abox`.

save-abox *function*

Description: If a pathname is specified, an ABox is saved to a file. In case a stream is specified the ABox is written to the stream (the stream must already be open).

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 ABox. It might as well be names of other DL systems.

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

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)
```

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 signalled 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 RACE.

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 ($\perp$).

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 RACE 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 RACE, 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 list 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* `t))`

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: RACE 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 RACE.

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)`
`(`*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, if *feature* is bound to `t`.

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
  :parent has-child)
```

See also: Macro `signature` on page 11.
Section 3.3 and Figure 6, on page 9 for domain and range restrictions.

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

Description: Defines an attribute.

Syntax: `(define-primitive-attribute` *AN* `&key (`*parents* `nil))`

Arguments: *AN* - attribute name

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 to the use of macro `define-primitive-role` 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 :parents (has-parents))
(define-primitive-attribute has-best-friend
  :parent has-friends)
```

See also: Macro `signature` on page 11.
Section 3.3 and Figure 6, on page 9 for domain and range restrictions.

---

add-role-axiom *function*

Description: Adds a role to a TBox.

Syntax: `(add-role-axiom` *tbox* *RN* `&key (`*transitive* `nil) (`*feature* `nil)`
`(`*parents* `nil))`

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

RN - role name

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

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

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-axiom` per TBox.

See also: Section 3.3 and Figure 6, on page 9 for domain and range restrictions.

5.4 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 RN)

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

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

add-role-assertion

function

Description: Adds a role assertion to an ABox.

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

Arguments:
abox - ABox object
IN_1 - individual name of the predecessor
IN_2 - individual name of the filler
RN - role name

Values: *abox*

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

forget-role-assertion *function*

Description: Retracts a role assertion from an ABox.

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

Arguments:
abox - ABox object
IN_1 - individual name of the predecessor
IN_2 - individual name of the filler
RN - role name

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)
  'Lucy 'Linus '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 RACE, 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).

6 Reasoning Modes

auto-classify *special-variable*

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

See also: Section 3.5 on page 9.

auto-realize *special-variable*

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

See also: Section 3.5 on page 9.

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*, the second argument must be a 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*, the first argument must be a 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*, the first argument must be a 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 30.

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*, the third argument must be a TBox.

See also: Function `atomic-concept-synonyms`, on page 39.

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*, the first argument must be a 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 39.

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*, the third argument must be a 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*, the third argument must be a TBox. If the disjointness is to be tested without reference to a TBox, `nil` can be used.

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?` RN_1 RN_2
`&optional (`*TBN* `(tbox-name *current-tbox*)))`

Arguments:
RN_1 - role name of the subsuming role
RN_2 - role name of the subsumed role
TBN - TBox name

Values: Returns `t` if RN_1 is a parent role of RN_2.

---

role-subsumes-p *function*

Description: Checks if two roles are subsuming each other.

Syntax: `(role-subsumes-p` RN_1 RN_2 *tbox*`)`

Arguments:
RN_1 - role name of the subsuming role
RN_2 - role name of the subsumed role
tbox - TBox object

Values: Returns `t` if RN_1 is a parent role of RN_2.

---

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 concept and `nil` otherwise.

---

role-p *function*

Description: Checks if *RN* is a role name for a role in the specified TBox.

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

Arguments:
RN - role name
tbox - TBox object

Values: Returns `t` if *RN* is a name of a role and `nil` otherwise.

transitive-p *function*

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

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

Arguments: *RN* - role name

tbox - TBox object

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

feature-p *function*

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

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

Arguments: *RN* - role name

tbox - TBox object

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

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-coherent`. 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-coherent` 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` *tbox*`)`

Arguments: *tbox* - TBox object

Values: Returns `t` iff 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` *tbox*`)`

Arguments: *tbox* - TBox object

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

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

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` iff the specified ABox object has been realized.

Syntax: `(abox-realized-p` *abox*`)`

Arguments: *abox* - ABox object

Values: Returns `t` if *abox* 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.

check-abox-coherence *function*

Description: Checks if the ABox is consistent. If there is a contradiction, this function print 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 *RN*
`&optional (`*abox* `*current-abox*))`

Arguments:

IN_1	- individual name of the predecessor
IN_2	- individual name of the role filler
RN	- role name
abox	- ABox object

Values: Returns `t` if IN_1 is related to IN_2 via *RN* 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 *RN* *abox*`)`

Arguments: IN_1 - individual name of the predecessor

IN_2 - individual name of the role filler

RN - role name

abox - ABox object

Values: Returns `t` if IN_1 is related to IN_2 via *RN* in *abox* and `nil` otherwise.

See also: Function `retrieve-individual-relations`, on page 49,
Function `retrieve-related-individuals`, on page 49.

individual-equal? *KRSS macro*

Description: Checks if two individual names refer to the same individual.

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 RACE 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 individual.

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 RACE 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.

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.

8 Retrieval

If the retrieval refers to concept names, RACE 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 41.

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 30.

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 30.

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 is 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: This function is the transitive closure of the function `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 is 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: This function is the transitive closure of the function `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 RACE.

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

Arguments: *RN* - role name

TBN - TBox name

Values: List of role names

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` *RN tbox*`)`

Arguments: *RN* - role name

tbox - TBox object

Values: List of role names

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

Arguments: *RN* - role name

TBN - TBox name

Values: List of role names

atomic-role-ancestors *function*

Description: Gets all roles from the TBox, that subsume the given role in the role hierarchy.

Syntax: `(atomic-role-ancestors` *RN* *tbox*`)`

Arguments: *RN* - role name

tbox - TBox object

Values: List of role names

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

Arguments: *RN* - role name

TBN - TBox name

Values: List of role names

Remarks: This is the equivalent macro to the KRSS macro `role-offspring`, which is also supplied by the RACE 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` *RN* *tbox*`)`

Arguments: *RN* - role name

tbox - TBox object

Values: List of role names

role-parents

KRSS macro

Description: Gets the roles from the TBox that directly subsume the given role in the role hierarchy.

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

Arguments: *RN* - role name

TBN - TBox name

Values: List of role names

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` *RN* *tbox*`)`

Arguments: *RN* - role name

tbox - TBox object

Values: List of role names

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 all TBoxes.

Syntax: `(all-tboxes)`

Values: List of TBox objects

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 roles names

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 names

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 names

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` *RN*
`&optional (`*tbox-or-abox* `*current-tbox*)`
`(`*stream* `*standard-output*))`

Arguments: *tbox-or-abox* - TBox object or ABox object

RN - role name (or feature name)

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* *RN*
`&optional (`*ABN* `(abox-name *current-abox*)))`

Arguments: *IN* - individual name of the predecessor

RN - role name

ABN - ABox name

Values: List of individual names

Examples: `(individuals-fillers Charlie-Brown 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 RN abox*`)`

Arguments:

IN	- individual name of the predecessor
RN	- role name
abox	- ABox object

Values: List of individual names

Examples:
```
(individuals-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` *RN abox*`)`

Arguments:

abox	- ABox object
RN	- role 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 36.

retrieve-individual-relations *function*

Description: This function gets all roles that hold between the specified pair of individuals.

Syntax: `(retrieve-individual-relations` 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 names

Examples:
```
(retrieve-individual-relations 'Charlie-Brown 'Snoopy
  (find-abox 'peanuts-characters))
```

See also: Function `individuals-related-p`, on page 36.

retrieve-direct-predecessors *function*

Description: Gets all individuals that are predecessors of a role for a specified individual.

Syntax: `(retrieve-direct-predecessors` *RN IN abox*`)`

Arguments: *RN* - role name

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 all ABoxes.

Syntax: `(all-aboxes)`

Values: List of ABox objects

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 52.

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 52.

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 52.

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 51.

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 "race: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 "race: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 "race: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)
```

C Web Interface

C.1 Introduction

The Web interface for the RACE system offers a convenient way to use a powerful taxonomical reasoning system. Based on a set of atomic concepts $\mathcal{C}$ and roles $\mathcal{R}$, it allows one to specify a set of axioms in the so called TBox. It represents the terminological knowledge about the domain. In the ABox one can assert that individuals are instances of specified concepts and there you can assert relations between these instances as well. The reasoning services can be used via the query interface. They include, for example, the computation of those elements of $\mathcal{C}$ which are direct sub- and superconcepts of a concept or the computation of the most-specific concept an individual is instance of. The TBoxes and the ABoxes together with the queries can be saved separately and used when the user logs in again using the same account.

C.2 How to use the interface

On the start page (see Fig. 7) you can

- Register as a new user.
- Use the RACE Prover if you are already registered.
- Maintain the TBoxes and ABoxes you have built.

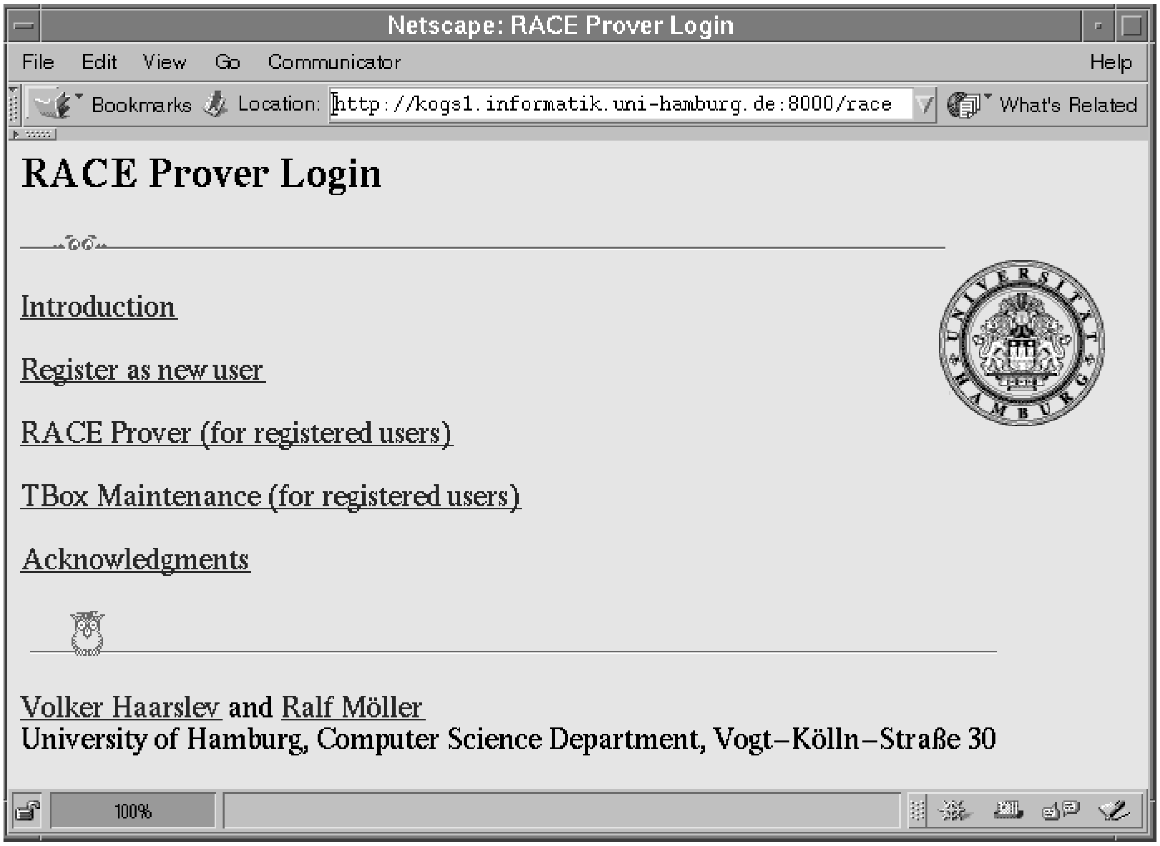

Figure 7: The RACE interface start page.

C.2.1 Register as a new user

When you use RACE for the first time you must register as a new user with the register page (see Fig. 8). In order to do this you must fill in the fields for login name and password. If you want to receive your TBoxes or your ABoxes via email, the fields for name and email address must be filled in as well.

Building a TBox

On the main page (see Fig. 9)[5] TBoxes and ABoxes can be constructed and queries can be posed. The TBox and the ABox are saved under the specified name when the button "Eval" is pressed, see Fig. 9.

[5]The example is is taken from [Buchheit et al. 93].

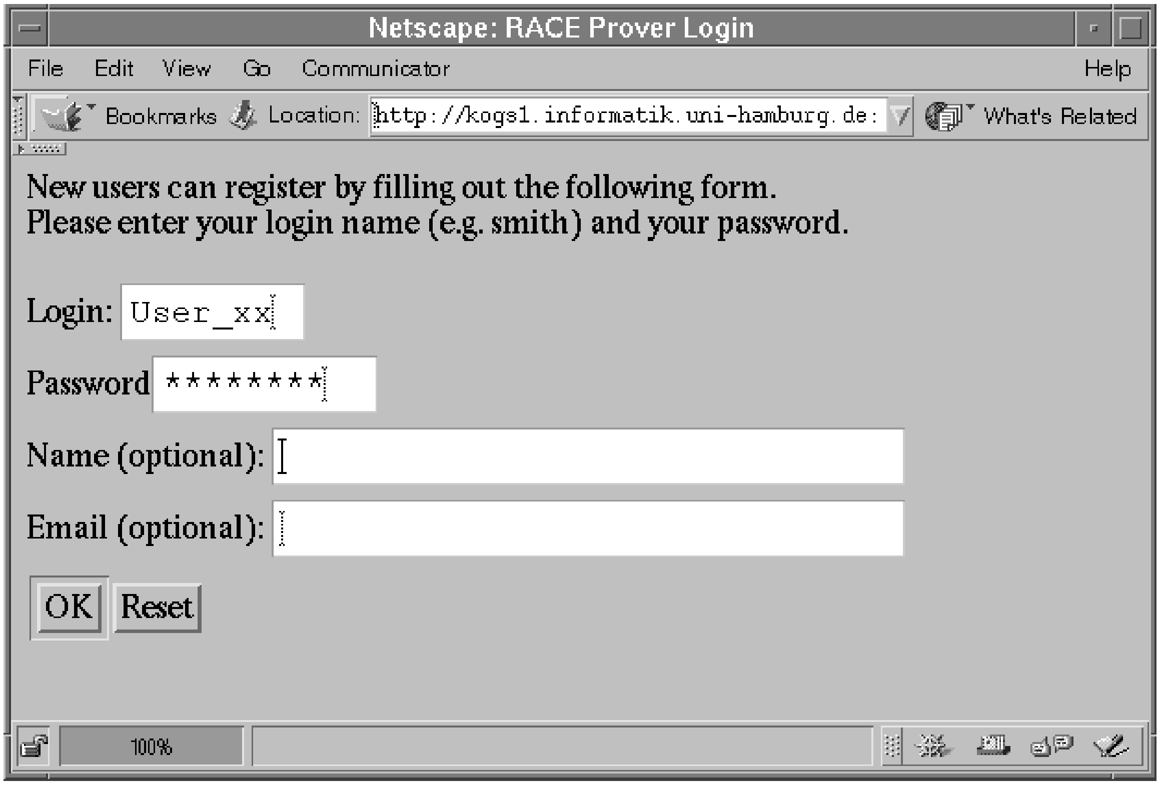

Figure 8: The RACE interface register page.

A TBox contains role declarations, role axioms and concept axioms. The set of concept terms used in the concept axioms of a TBox is always based on the set of concept names, the atomic concepts. In the RACE system the set of concept names $\mathcal{C}$ is extracted automatically from the concept axioms mentioned in the TBox. The set of roles $\mathcal{R}$ consists of three disjoint sets: The set of primitive roles, the set of transitive roles and the set of features. The set of roles $\mathcal{R}$ is also extracted automatically from the TBox by the RACE system. The transitive roles and the features are given by role declarations. The undeclared roles are assumed to be primitive roles. Hierarchical relations between roles are stated by role axioms (s.b.). In the following the syntax for concept axioms is explained.

C.2.2 Concept axioms

RACE supports two kinds of concept axioms. We first give the usual German style DL notation and afterwards the ASCII counterpart of the operators explained. In the following C with index denotes a concept term.

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

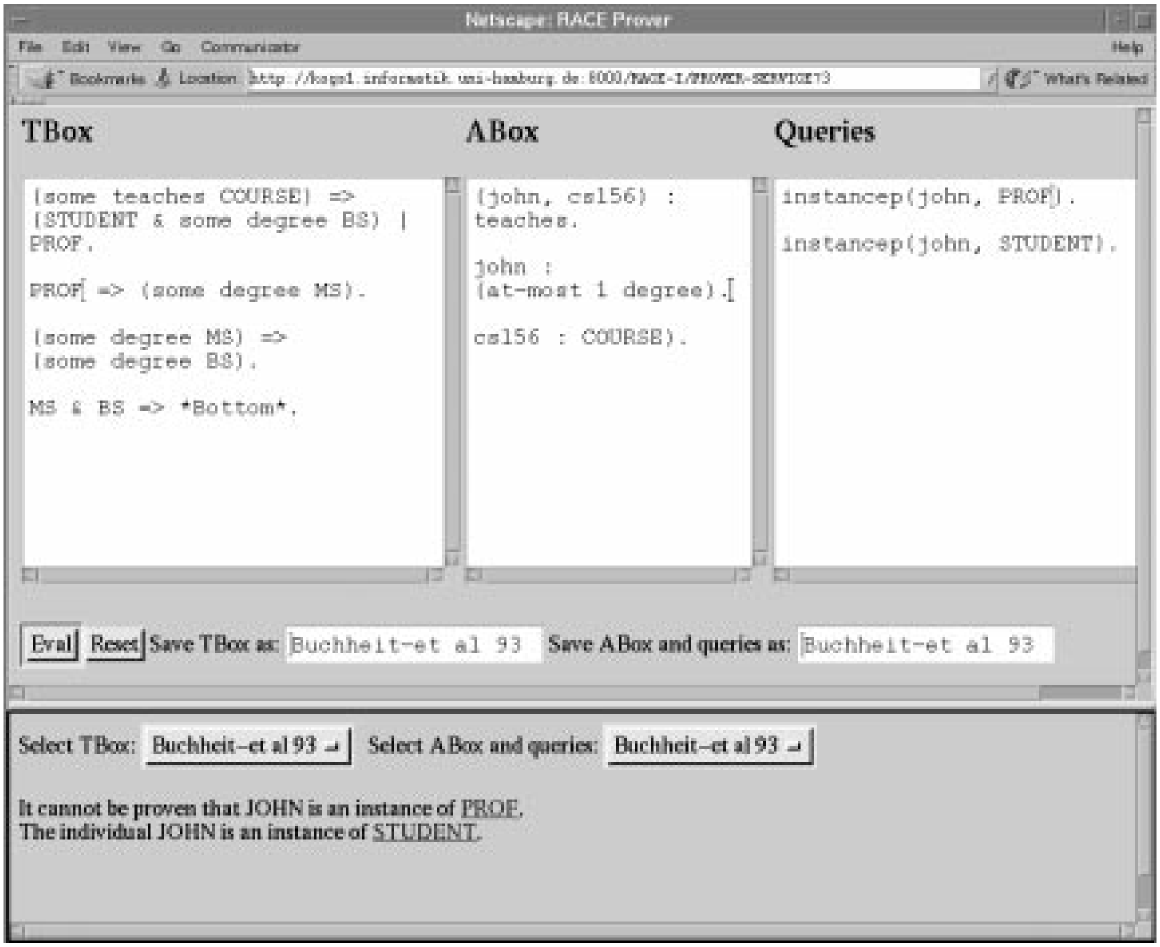

Figure 9: The RACE main page.

DL notation:	$C_1 \sqsubseteq C_2$	
RACE syntax:	C_1 `=>` C_2`.`	
Examples:	`Human => Animal & Biped.`	
	`mammal => all has-parent mammal.`	
	`some teaches Course => (Student & some degree BS)`	
	`| Prof.`	

For the users convenience RACE also supports another kind of concept axiom.

Concept equations state the equality between two concept terms.

DL notation:	$C_1 \doteq C_2$	
RACE syntax:	$C1$ `=` $C2$.	
Examples:	`Woman = Human & Female.`	
	`Primary-Color & /Yellow = Red	Blue.`	

Actually, a concept equation $C_1 \doteq C_2$ can be expressed by the two GCIs $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. Concept axioms are separated by a dot.

C.2.3 Concept Terms

There are two predefined concepts in RACE. The concept `*top*`, denotes the most general concept and the concept `*bottom*` denotes the inconsistent concept. Besides concept names the following terms are also concept terms.

Boolean terms build concepts by using the boolean operators.

	DL notation	RACE syntax
Negation	$\neg\ C$	`/` C
Conjunction	$C \sqcap D$	C `&` D
Disjunction	$C \sqcup D$	C `|` D

Example: `Tree = Plant | (Graph & /(some contains Cycle)).`

Qualified restrictions state that role fillers have to be of a certain concept. Value restictions 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	RACE syntax
Exists restriction	$\exists\ R.C$	`some` R C
Value restriction	$\forall\ R.C$	`all` R C

Example: `human = (all has-ancestors human).`

Number restrictions can specify a lower and upper bound for the number of role fillers each instance of this concept has for a certain role.

	DL notation	RACE syntax
At-most restriction	$\leq$ n R	`at-most` n R
At-least restriction	$\geq$ n R	`at-least` n R

Example: `Week = (at-most 7 has-days) & (at-least 7 has-days).`

The precedence of the operators is: `/`, `&`, —, `=>`, `=`.
The comment sign is: `%`, after this sign the rest of the line is ignored. The RACE system is not case-sensitive. Concepts may be put in parentheses.

C.2.4 Role Declarations

As we have seen before in RACE the set of roles $\mathcal{R}$ consists of primitive roles, transitive roles and features. The features and transitive roles have to be declared. In the following R (possibly with index) denotes a role name.

Feature declaration Features (also called Attributes) restrict a role to be a functional role. That means each individual can only have up to one filler for this role.

RACE syntax: `feature(`R`).`
Example: `feature(has-mother).`

Transitive Role declaration states that a role is transitively closed.
Please note that the transitive roles cannot be used in number restrictions and that features can not be transitive.

RACE syntax: `transitive(`R`).`
Example: `transitive(ancestor-of).`

Besides these kinds of special roles, role hierarchies can be established.

C.2.5 Role Axioms

Role Hierarchies consist of 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. All subroles of a feature are implicitly declared as features. Sub- or superroles of a transitive role are not neccessarily transitive, unless declared as being so.

RACE syntax: `parent(`R_{sub}`, `R_{super}`).`
Example: `parent(is-mother-of, is-parent-of).`

C.2.6 Building an ABox

In the ABox pane (see Fig. 9) assertions can be made about individuals. It can be stated that an individual is an instance of a concept and that a relation between two individuals holds.

Concept assertions state that an individual i is an instance of the specified concept C.
RACE syntax: i `:` C.

Role assertions state that an individual j is a role filler for a role R with respect to an individual i.
RACE syntax: `(`i`, `j`) : `R`.`

C.2.7 Building and Executing a Query

In the query field, queries can be posed concering the TBox and the ABox. Some example queries are shown in Fig. 10.

C.2.8 Queries concerning the concept hierarchy

The following queries always refer to the current TBox implicitly. They depend on set $\mathcal{C}$ of concept names extracted from the current TBox.

Superconcepts returns the most-specific subsumers of CN found in $\mathcal{C}$.

RACE syntax: `superconcepts(`CN`).`

Subconcepts returns the most-specific subsumees of CN found in $\mathcal{C}$.

RACE syntax: `subconcepts(`CN`).`

Equivalentp checks wether the two concepts from $\mathcal{C}$ are equivalent.
RACE syntax: `equivalentp(`CN_1`, `CN_2`).`

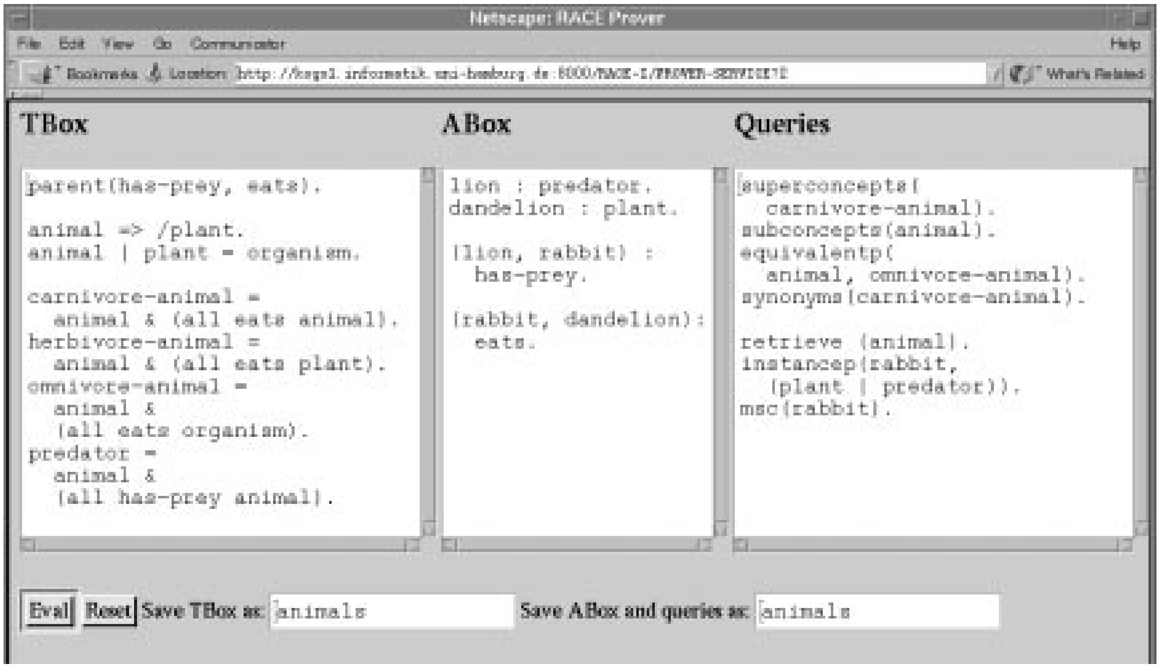

Figure 10: Some example queries.

Synonyms finds all concept names in $\mathcal{C}$ that are equivalent to CN.

RACE syntax: `synonyms(`CN`).`

C.2.9 Queries concerning the instances

Instancep checks, if the specified individual i is an instance of the specified concept C.

RACE syntax: `instancep(`i`, `C`).`

Retrieve finds all individuals in the current ABox that are instances of a concept C.

RACE syntax: `retrieve(`C`).`

Most-specific concepts finds the most-specific concept names in $\mathcal{C}$, of which the specified individual i is an instance of.

RACE syntax: `msc(`i`).`

Figure 11: The RACE maintenance page.

The queries in the pane are executed, after the ”Eval” button is clicked. The response of the RACE system is displayed in the lower part of the main dialog as you can see in Fig. 9.

A negative answer such as ”It cannot be proven that I is an instance of C.” for queries like ”`instancep` (i, C)`.`” may seem surprising for someone who expected ”No”. RACE avoids ”No” because this might suggest that in this case i is an instance of $/A$. Considering the TBox $\{D \doteq A \sqcup B\}$ with the ABox $\{i : D\}$ it cannot be inferred that the individual is an instance of A or not.

C.2.10 Maintaining TBoxes and ABoxes

From the RACE start page you can go to the maintenance page (see Fig. 11). There you find all the names of your TBoxes and ABoxes (with their associated Queries) listed. In this dialog you can delete TBoxes or ABoxes. You can also send a TBox or an ABox to yourself via email if you have filled in your email address for this account.

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 99] V. Haarslev and R. Möller, "Expressive ABox Reasoning with Number Restrictions, Role Hierarchies, and Transitively Closed Roles", Technical Report FBI-HH-M-288/99, University of Hamburg, 1999.

[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.

[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