MOTEL User Manual
Version 0.8.2 (February 1994)

Ullrich Hustadt
Andreas Nonnengart
Renate Schmidt
Jan Timm

MPI–I–93–236 December 1993

Addresses

Ullrich Hustadt
Max-Planck-Institut für Informatik
Im Stadtwald
W-6600 Saarbrücken
Germany
E-mail: Ullrich.Hustadt@mpi-sb.mpg.de

Andreas Nonnengart
Max-Planck-Institut für Informatik
Im Stadtwald
W-6600 Saarbrücken
Germany
E-mail: Andreas.Nonnengart@mpi-sb.mpg.de

Renate Schmidt
Max-Planck-Institut für Informatik
Im Stadtwald
W-6600 Saarbrücken
Germany
E-mail: Renate.Schmidt@mpi-sb.mpg.de

Acknowledgements

This research was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102 and the Deutsche Forschungsgemeinschaft, SFB 314 (TICS). The responsibility for the contents of this publication lies with the authors.

Abstract

MOTEL is a logic-based knowledge representation languages of the KL-ONE family. It contains as a kernel the $\mathcal{KRIS}$ language which is a decidable sublanguage of first-order predicate logic (see Baader and Hollunder (1990)).

Whereas $\mathcal{KRIS}$ is a single-agent knowledge representation system, i.e. $\mathcal{KRIS}$ is only able to represent general world knowledge or the knowledge of one agent about the world, MOTEL is a multi-agent knowledge representation system. The MOTEL language allows modal contexts and modal concept forming operators which allow to represent and reason about the believes and wishes of multiple agents. Furthermore it is possible to represent defaults and stereotypes.

Beside the basic resoning facilities for consistency checking, classification, and realization, MOTEL provides an abductive inference mechanism. Furthermore it is able to give explanations for its inferences.

Keywords

abduction, belief revision, default logic, modal logic, terminological logic, multi-agent knowledge representation, functional dependencies

Chapter 1

Reading the User Manual

1.1 Predicate Desciptions

Prediates are described according to the following grammar:

<Predicate Description> ::= <CallPattern>
`Arguments:` <ArgumentTypes>
<Informal Description>
<CallPattern> ::= <Predicate Name><CallArguments>
<CallArguments> ::=
| [(<CallArgument>$^+$)]
| (<CallArgument>$^+$)
<CallArgument> ::= [<Mode Annotation><Meta Variable>,]
| [<Mode Annotation><Meta Variable>]
| <Mode Annotation><Meta Variable>
<ArgumentTypes> ::=
| <Meta Variable> : <Type>
<ArgumentTypes>
<Predicate Name> ::= <Identifier>
<Meta Variable> ::= <Identifier>
<Type> ::= <Informal Description>

In the following subsections, we give further explanations for the parts of a predicate description.

1.1.1 Call Arguments

A predicate can have a varying number of arguments. If we use

[(<CallArgument>$^+$)]

we want to describe the situation that the predicate has either no arguments or atleast one arguments which must be enclosed in round brackets. The notation

(<CallArgument>$^+$)

is used if the predicate has atleast one argument which has to be enclosed in round brackets. If the predicate has no arguments, we simply give no call arguments.

If a call arguments takes the form

[<Mode Annotation><Meta Variable>,]

then it is an optional argument, i.e. it may be omitted, and it is followed be a comma unless it is the last argument, i.e. the last one before the closing round bracket. If we use

[<Mode Annotation><Meta Variable>]

then it is an optional argument which is never followed by a comma. The last form is

<Mode Annotation><Meta Variable>

denoting a non-optional argument.

The mode annotations are useful to tell wether an argument is input or output or both. They also describe formally the instantiation pattern to the call.

Following is a complete description of the mode annotations you will find in this user manual:

+ Input argument. This argument will be inspected by the predicate, and affects the behaviour of the predicate, but will not be further instantiated by the predicate.

− Deterministic output argument. This argument is unified with the output value of the predicate. Given the input arguments, the value of a deterministic output argument is uniquely defined.

* Nondeterministic output argument. This argument is unified with the output value of the predicate. The predicate might be resatisfiable, and might through backtracking generate more than one output value for this argument.

+− An input argument that deterministically might be further instantiated by the predicate.

+* An input argument that might be further instantiated by the predicate. The predicate might be resatisfiable, and might through backtracking generate more than one instantiation pattern for this argument.

All predicates of arity zero are determinate.

1.2 Argument Types

After the call pattern, we declare the types of the arguments occurring in the call pattern. For each meta variable in the call pattern the corresponding type is given. Types are not formally defined.

Chapter 2

Environments

An environment is a container for a knowledge base. Each environment has some user provided environment name, some system generated internal environment name, and a user provided comment. Although it is possible to have two different environments with the same environment name, the one generated later will be not accessible by the user. So the user should carefully choose the names for the environments. The internal environment name is unique and does not depend on the environment name. The comment can be used for any purpose, e.g. to remind the user what the knowledge base is about.

There is always a *current environment*. Whenever a predicate has an environment name as optional argument and the argument is not provided in a call to the predicate, the system will refer to the current environment. At the beginning, there exists an empty environment named `initial`.

We provide the following predicates for handling environments:

`clearEnvironment`*[*`(`+**EnvName*`)`*]*

Arguments: *EnvName* environment name

removes the knowledge base in environment *EnvName*. Without *EnvName* the current environment is removed.

`compileEnvironment(`+*FileName[*`,` *EnvName]*`)`

Arguments: *FileName* file name
EnvName environment name

loads the internal representation of an environment *EnvName* in compiled form from a file named *FileName*. If no *EnvName* is given, the enviroment name stored in the file *FileName* will be taken. If there already exists an environment *EnvName*, it will be removed.

`copyEnvironment(`*[*+*EnvName1*`,`*]*+*EnvName2*`)`

Arguments: *EnvName1* environment name
EnvName2 environment name

creates a new environment *EnvName2* and copies the knowledge base in *EnvName1* to *EnvName2*.

`environment(`+−*EnvName*`,` **EnvId*`,` **Comment*`)`

Arguments: *EnvName* environment name
EnvId internal environment name
Comment string

retrieves the internal environment identifier *EnvId* and the associated comment *Comment* for a given environment name *EnvName*.

`getCurrentEnvironment(`−*EnvName*`)`

Arguments: *EnvName* environment name

instantiates *EnvName* with the identifier for the current environment.

`initEnvironment`*[(+EnvName)]*

Arguments: *EnvName* environment name

provides the environment *EnvName* with the initial data structures. The current environment is initialized if no *EnvName* is given.

`initialise`

removes all environments, initialises the empty environment `initial`, and makes `initial` the current environment.

`initialize`

Identical to `initialize`. For those of us who prefer the alternative spelling.

`loadEnvironment(`*+FileName[, EnvName]*`)`

Arguments: *FileName* file name
EnvName environment name

loads the internal representation of an environment *EnvName* from a file named *FileName*. If no *EnvName* is given, the enviroment name stored in the file *FileName* will be taken. If there already exists an environment *EnvName*, it will be removed.

`makeEnvironment(`*+EnvName*`,` *+Comment*`)`

Arguments: *EnvName* environment name
Comment string

creates a new environment with identifier *EnvName* and associated comment *Comment*. This new environment becomes the current environment.

`removeEnvironment(`*+−EnvName*`)`

Arguments: *EnvName* environment name

removes the knowledge base and the environment *EnvName*. If *EnvName* was the current environment then initial environment becomes the current environment.

`renameEnvironment(`*+EnvName1*`,` *+EnvName2*`)`

Arguments: *EnvName* environment name

renames environment *+EnvName1* to *+EnvName2*.

`saveEnvironment(`*[+EnvName,] FileName*`)`

Arguments: *EnvName* environment name
FileName file name

saves the internal representation of environment *EnvName* into a file named *FileName*.

`showEnvironment`*[(+EnvName)]*

Arguments: *EnvName* environment name

displays the knowledge base in environment *EnvName*, i.e. the terminological axioms, the assertional axioms, and the modal axioms.

`switchToEnvironment(`*+EnvName*`)`

Arguments: *EnvName* environment name

makes *EnvName* the current environment (if an environment with this identifier exists).

Chapter 3

Knowledge Representation

3.1 Concept and Role Formation

Assume that we have four disjoint alphabets of symbols, called *concept names* $\mathcal{C}$, *role names* $\mathcal{R}$, *modal operators* $\mathcal{M}$, and *object names* $\mathcal{O}$. A distinguished subset $\mathcal{A}$ of $\mathcal{O}$ is the set of all *agent names*. There is a special agent name `all` and a special concept name `top` called *top concept*. The tuple $\Sigma := (\mathcal{C}, \mathcal{R}, \mathcal{M}, \mathcal{O})$ is a *knowledge signature*.

The sets of *modal concept terms* and *role terms* are inductively defined as follows. Every concept name is a modal concept term and every role name is a role term. Now let C, $C_1, \ldots, C_k$ be modal concept terms, R, $R_1, \ldots, R_l$ be role terms already defined, O be a modal operator, a some agent name, and let n be a nonnegative integer. Then

`and`([$C_1, \ldots, C_k$])	(conjunction)
`or`([$C_1, \ldots, C_k$])	(disjunction)
`not`(C)	(negation)
`naf`(C)	(negation as failure)
`all`(R,C)	(value restriction)
`some`(R,C)	(exists restriction)
`atleast`(n,R)	(number restriction)
`atmost`(n,R)	
`b`(O,a,C)	(box agent introduction)
`d`(O,a,C)	(diamond agent introduction)
`b`(O,C_1,C_2)	(box concept introduction)
`d`(O,C_1,C_2)	(diamond concept introduction)

are modal concept terms and

`and`([$R_1, \ldots, R_l$])	(role conjunction)
`inverse`(R)	(role inversion)
`restr`(R,C)	(role restriction)

are role terms.

3.2 Modal Terminological Axioms

A *modal context* is a (possibly empty) list of terms of the form `b`(O,a), `d`(O,a), `bc`(O,A) or `dc`(O,A) where O is a modal operator, a is an agent name and A is a concept name. The set of all modal contexts is denoted $\mathcal{MC}$.

So-called *modal terminological axioms* are used to introduce names for modal concept terms and role terms. A finite set of such axioms satisfying certain restrictions is called a *terminology* (TBox). There are three different ways of introducing new concepts (respectively roles) into a terminology.

By the *modal terminological axioms*

`defprimconcept(`[+*EnvName*`,`][+*M*`,`]+*A*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	A	concept name

`defprimrole(`[+*EnvName*`,`][+*M*`,`]+*P*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	P	role name

new concept and role names are introduced in environment *EnvName* and modal context *M* without restricting their interpretation. If no *EnvName* is given, the current environment will be taken. If no *M* is provided, the empty modal context will be used.

The modal terminological axioms
`defprimconcept(`[+*EnvName*`,`][+*M*`,`]+*A*`,` +*C*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	A	concept name
	C	concept term

`defprimrole(`[+*EnvName*`,`][+*M*]+*P*`,` +*R*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	P	role name
	R	role term

impose necessary conditions on the interpretation of the introduced concept and role names in environment *EnvName* and modal context *M*.

Finally, one can impose necessary and sufficient conditions by the modal terminological axioms
`defconcept(`[+*EnvName*`,`][+*M*`,`]+*A*`,` +*C*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	A	concept name
	C	concept term

`defrole(`[+*EnvName*`,`][+*M*`,`]+*P*`,` +*R*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	P	role name
	R	role term

One can impose an additional restriction on the interpretation of already introduced concept names by the terminological axiom
`defdisjoint(`[+*EnvName*`,`][+*M*`,`]+*CL*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	CL	list of concept names

which declares the mutal disjointness of all concepts in the given list of concept names.

3.3 Modal Assertional Axioms

Assertional axioms have the form
`assert_ind(`[+*EnvName*`,`][+*M*`,`]+*X*`,` +*A*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	A	concept name

`assert_ind(`[+*EnvName*`,`][+*M*`,`]+*X*`,` +*Y*`,` +*P*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	Y	object name
	P	role name

The first one defines X to be an element of concept A in environment *EnvName* and modal context *M*. The second one defines the pair (X, Y) to be an element of the role P.

A finite set set of such axioms is called *world description*.

3.4 Knowledge Revision

MOTEL has predicates for revising the terminology and the world description of a knowledge base. The following predicates allows to delete a concept, i.e. after deleting the concept A it is no longer possible to prove that some object a is an element of A unless it is explicitly stated in the world description.
`undefconcept(`+*EnvName*`,` *M*`,` *A*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	A	concept name

deletes concept A in environment *EnvName* and modal context *M*.

The following predicates delete the relationship between a concept name and a concept term previously defined by some terminological axiom.
`undefconcept(`+*EnvName*`,` +*M*`,` +*A*`,` +*CT*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	A	concept name
	CT	concept term

deletes the axiom defining the equivalence of A and CT in environment *EnvName* and modal context *M*.

`undefprimconcept(`+*EnvName*`,` *M*`,` *A*`,` *CT*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	A	concept name
	CT	concept term

deletes the axiom defining the inclusion of A in CT in environment *EnvName* and modal context *M*.

To revise the world description one can either delete the membership of some object a in a concept A or the membership of a pair (a, b) in the role P.

`delete_ind(`$+EnvName$`,` $+M$`,` $+X$`,` A`)`

Arguments:	$EnvName$	environment name
	M	modal context
	X	object name
	A	concept name

deletes the assertional axiom defining the membership of X in A.

`delete_ind(`$[+EnvName,][+M,]+X$`,` $+Y$`,` $+P$`)`

Arguments:	$EnvName$	environment name
	M	modal context
	X	object name
	Y	object name
	P	role name

deletes the assertional axiom defining the membership of the pair (X,Y) in role P.

3.5 Semantics

Suppose $\Sigma = (\mathcal{C}, \mathcal{R}, \mathcal{M}, \mathcal{O})$ is a knowledge signature.

Definition 1 (Σ-Structures)
As usual we define a Σ-*structure* as a pair $(\mathcal{D}, \mathcal{I})$ which consists of a domain $\mathcal{D}$ and an interpretation function $\mathcal{I}$ which maps the individual objects to elements of $\mathcal{D}$, primitive concepts to subsets of $\mathcal{D}$ and the primitive roles to subsets of $\mathcal{D} \times \mathcal{D}$. $\triangle$

Definition 2 (Frames and Interpretations)
By a frame $\mathcal{F}$ we understand any pair $(\mathcal{W}, \Re)$ where

- $\mathcal{W}$ is a non-empty set (of worlds).
- $\Re = \biguplus_{O \in \mathcal{M}, a \in \mathcal{A}} \Re^a_O$ where the $\Re^a_O$'s are binary relation on $\mathcal{W}$, the so-called *accessibility relations* between worlds.

By a Σ-interpretation $\Im$ based on $\mathcal{F}$ we understand any tuple $(\mathcal{D}, \mathcal{F}, \Im_{\text{loc}}, \epsilon)$ where

- $\mathcal{D}$ denotes the common domain of all Σ-structures in the range of $\Im_{\text{loc}}$.
- ϵ denotes the actual world (the current situation)
- $\mathcal{F}$ is a frame
- $\Im_{\text{loc}}$ maps worlds to Σ-structures with common domain $\mathcal{D}$ which interpret agents' names equally.

$\triangle$

Definition 3 (Interpretation of Terms)
Let $\Im = (\mathcal{D}, \mathcal{F}, \Im_{\text{loc}}, \epsilon)$ be a Σ-interpretation and let $\Im_{\text{loc}}(\epsilon) = (\mathcal{D}, \mathcal{I})$. We define the interpretation of terms inductively over their structure:

$$
\begin{array}{lcl}
\Im(A) & = & \mathcal{I}(A) \text{ if } A \text{ is a concept name} \\
\Im(P) & = & \mathcal{I}(P) \text{ if } P \text{ is a role name} \\
\Im(\texttt{and([}C_1, \ldots, C_n\texttt{])}) & = & \Im(C_1) \cap \ldots \cap \Im(C_n) \\
\Im(\texttt{or([}C_1, \ldots, C_n\texttt{])}) & = & \Im(C_1) \cup \ldots \cup \Im(C_n) \\
\Im(\texttt{not(}C\texttt{)}) & = & \mathcal{D} \setminus \Im(C) \\
\Im(\texttt{all(}R\texttt{,}C\texttt{)}) & = & \{d \in \mathcal{D} \mid e \in \Im(C) \text{ for all } e \text{ with } (d,e) \in \Im(R)\} \\
\Im(\texttt{some(}R\texttt{,}C\texttt{)}) & = & \{d \in \mathcal{D} \mid e \in \Im(C) \text{ for some } e \text{ with } (d,e) \in \Im(R)\} \\
\Im(\texttt{b(}O\texttt{,}a\texttt{,}C\texttt{)}) & = & \{d \in \mathcal{D} \mid d \in \Im[\chi](C) \text{ for all } \chi \text{ with } \Re^a_O(\epsilon, \chi)\} \\
\Im(\texttt{d(}O\texttt{,}a\texttt{,}C\texttt{)}) & = & \{d \in \mathcal{D} \mid d \in \Im[\chi](C) \text{ for some } \chi \text{ with } \Re^a_O(\epsilon, \chi)\} \\
\Im(\texttt{and([}R_1, \ldots, R_n\texttt{])}) & = & \Im(R_1) \cap \ldots \cap \Im(R_n) \\
\Im(\texttt{inverse(}R\texttt{)}) & = & \{(x,y) \in \mathcal{D} \times \mathcal{D} \mid (y,x) \in \Im(R)\} \\
\Im(\texttt{restr(}R\texttt{,}C\texttt{)}) & = & \{(x,y) \in \Im(R) \mid y \in \Im(C)\}
\end{array}
$$

where $\Im[\chi] = (\mathcal{D}, \mathcal{F}, \Im_{\text{loc}}, \chi)$ $\triangle$

Definition 4 (Satisfiability)
Let $\Im = (\mathcal{D}, \mathcal{F}, \Im_{\text{loc}}, \epsilon)$ be a Σ-interpretation. We define the satisfiability relation $\models$ inductively over the structure of modal terminological and modal assertional axioms:

$\Im \models$ `defprimconcept(`C_1`,` C_2`)`	iff	$\Im(C_1) = \Im(C_2)$
$\Im \models$ `defprimconcept([b(`O`,`a`)|`M`],` C_1`,` C_2`)`	iff	$\Im[\chi] \models$ `defprimconcept(`M`,` C_1`,` C_2`)` for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defprimconcept([bc(`O`,`A`)|`M`],` C_1`,` C_2`)`	iff	$\Im[\chi] \models$ `defprimconcept(`M`,` C_1`,` C_2`)` for every a with $\Im \models a \in A$, for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defprimconcept([d(`O`,`a`)|`M`],` C_1`,` C_2`)`	iff	$\Im[\chi] \models$ `defprimconcept(`M`,` C_1`,` C_2`)` for some χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defprimconcept([dc(`O`,`A`)|`M`],` C_1`,` C_2`)`	iff	$\Im[\chi] \models$ `defprimconcept(`M`,` C_1`,` C_2`)` for every a with $\Im \models a \in A$, for some χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defconcept(`M`,` C_1`,` C_2`)`	iff	$\Im \models$ `defprimconcept(`M`,` C_1`,` C_2`)`and $\Im \models$ `defprimconcept(`M`,` C_2`,` C_1`)`
$\Im \models$ `defprimrole(`R_1`,` R_2`)`	iff	$\Im(R_1) = \Im(R_2)$
$\Im \models$ `defprimrole([b(`O`,`a`)|`M`],` R_1`,` R_2`)`	iff	$\Im[\chi] \models$ `defprimrole(`M`,` R_1`,` R_2`)` for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defprimrole([bc(`O`,`A`)|`M`],` R_1`,` R_2`)`	iff	$\Im[\chi] \models$ `defprimrole(`M`,` R_1`,` R_2`)` for every a with $\Im \models a \in A$, for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defprimrole([d(`O`,`a`)|`M`],` R_1`,` R_2`)`	iff	$\Im[\chi] \models$ `defprimrole(`M`,` R_1`,` R_2`)` for some χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defprimrole([dc(`O`,`A`)|`M`],` R_1`,` R_2`)`	iff	$\Im[\chi] \models$ `defprimrole(`M`,` R_1`,` R_2`)` for every a with $\Im \models a \in A$, for some χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `defrole(`M`,` R_1`,` R_2`)`	iff	$\Im \models$ `defprimrole(`M`,` R_1`,` R_2`)`and $\Im \models$ `defprimrole(`M`,` R_2`,` R_1`)`
$\Im \models$ `assert_ind(`X`,` A`)`	iff	$\Im(X) \in \Im(A)$
$\Im \models$ `assert_ind([b(`O`,`a`)|`M`],` X`,` A`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` A`)` for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `assert_ind([bc(`O`,`A`)|`M`],` X`,` A`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` A`)` for every a with $\Im \models a \in A$, for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `assert_ind([d(`O`,`a`)|`M`],` X`,` A`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` A`)` for some χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `assert_ind([dc(`O`,`A`)|`M`],` X`,` A`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` A`)` for every a with $\Im \models a \in A$, for some χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `assert_ind(`X`,` Y`,` P`)`	iff	$(\Im(X), \Im(Y)) \in \Im(P)$
$\Im \models$ `assert_ind([b(`O`,`a`)|`M`],` X`,` Y`,` P`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` Y`,` P`)` for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `assert_ind([bc(`O`,`A`)|`M`],` X`,` Y`,` P`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` Y`,` P`)` for every a with $\Im \models a \in A$, for every χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `assert_ind([d(`O`,`a`)|`M`],` X`,` Y`,` P`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` Y`,` P`)` for some χ with $\Re^a_O(\epsilon, \chi)$
$\Im \models$ `assert_ind([dc(`O`,`A`)|`M`],` X`,` Y`,` P`)`	iff	$\Im[\chi] \models$ `assert_ind(`M`,` X`,` Y`,` P`)` for every a with $\Im \models a \in A$, for some χ with $\Re^a_O(\epsilon, \chi)$

$\triangle$

Definition 5
Let $\Im$ be an interpretation and let Φ be a modal terminological or modal assertional axiom with $\Im \models \Phi$.

Then we call Φ *satisfiable* and we call $\Im$ a *model* for Φ. If all interpretations are models for Φ then we call Φ a *theorem*. Any axiom for which no model exists is called *unsatisfiable*. Thus, Φ is a theorem iff its negation is unsatisfiable. $\triangle$

3.6 Modal Axioms

For any modal operator O and any agent a one has to specify the properties of the accessibility relation $\Re^a_O$. On the other hand, these properties correspond to subsetrelationships on modal concepts. Some of these correspondences are listed below. For further details see Nonnengart (1992).

Name	Axiom Schema	Property
d	$\mathtt{b}(O,a,C) \subseteq \mathtt{d}(O,a,C)$	$\forall x \exists y\ \Re^a_O(x,y)$
t	$\mathtt{b}(O,a,C) \subseteq C$	$\forall x\ \Re^a_O(x,x)$
b	$C \subseteq \mathtt{b}(O,a,\mathtt{d}(O,a,C))$	$\forall x,y\ \Re^a_O(x,y) \Rightarrow \Re^a_O(y,x)$
4	$\mathtt{b}(O,a,C) \subseteq \mathtt{b}(O,a,\mathtt{b}(O,a,C))$	$\forall x,y,z\ \Re^a_O(x,y) \wedge \Re^a_O(y,z) \Rightarrow \Re^a_O(x,z)$
5	$\mathtt{d}(O,a,C) \subseteq \mathtt{b}(O,a,\mathtt{d}(O,a,C))$	$\forall x,y,z\ \Re^a_O(x,y) \wedge \Re^a_O(x,z) \Rightarrow \Re^a_O(y,z)$

The user specifies the properties of the accessibility relation using the predicate `modalAxioms`. At the moment, the conjunctions `d45`, `d4`, `d5`, and `t` are allowed. The identifiers `kd45`, `kd4`, `kd5`, and `kt` together form the argument type of *Kripke classes*.

`modalAxioms(`*[+EnvName,]* *+Class*`,` *+O*`,` *+a*`)`

Arguments:	*EnvName*	environment name
	Class	Kripke class
	O	modal operator
	a	agent name

asserts the internal representation of the properties defined by the given Kripke class *Class* for the accessibility relation of the modal operator O and agent a.

`modalAxioms(`*[+EnvName,]* *+Class*`,` *+O*`, concept(`*+A*`))`

Arguments:	*EnvName*	environment name
	Class	Kripke class
	O	modal operator
	A	concept name

asserts the internal representation of the properties defined by the given Kripke class *Class* for the accessibility relation of the modal operator O for every agent in concept A.

3.7 Knowledge Bases

A triple consisting of a terminology, a world description, and modal axioms is a *knowledge base*. It is possible to load and to save knowledge bases using the following predicates.

`saveKB(`*[+EnvName,]+FileName*`)`

Arguments:	*EnvName*	environment name
	FileName	file name

saves the terminological, assertional, and modal axioms of the knowledge base in environment *EnvName* into the file *FileName*.

`loadKB(`*+FileName*`,` *-EnvName*`)`

Arguments:	*FileName*	file name
	EnvName	environment name

loads the terminological, assertional, and modal axioms from file *FileName*, turns them into their internal representation in environment *EnvName*.

`getKB(`*[+EnvName]*`,` *−Axioms*`)`

Arguments:	*EnvName*	environment name
	Axioms	list of axioms

Axioms is instantiated with a list of all terminological, assertional, and modal axioms in environment *EnvName*.

Chapter 4

Classification

4.1 Building the Semantic Network

Suppose C and D are concepts in a modal context M. Then C subsumes D if we can prove from the assumption that a skolem constant a is an element of D that is also an element of C. The predicate for doing this in MOTEL is
`subsumes(`*[+EnvName,][+M,]+C, +D*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	C	concept name
	D	cocept name

succeeds if *C* and *D* are known concepts in environment *EnvName* and modal context *M* and *C* subsumes *D.*

Let $C(E, M)$ be the set of all concepts in environment E and modal context M. We can compute the subsumption relation on $C(M)$, called *semantic network of* M, using the predicate
`classify`*[*`(`*[+EnvName,][+M]*`)`*]*

Arguments:	*EnvName*	environment name
	M	modal context

computes the semantic network in modal context *M.*

4.2 Retrieval commands for concepts

After the classification is done, one can use the following commands to retrieve informations about the semantik network:
`showHierarchy(`*+EnvName*`,` *+M*`,` *+Type*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	Type	either `concepts` or `roles`

displays the concept hierarchy, i.e. the semantic network in the modal context *M* if *Type* is `concepts` and the role hierarchy in the modal context *M* if *Type* is `roles`.

`getHierarchy(`*+EnvName*`,` *+M*`,` *+Type*`,` *−H*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	Type	either `concepts` or `roles`
	H	internal representation of the subsumption hierarchy

instantiates *H* with the internal representation of the concept hierarchy, i.e. the semantic network in the modal context *M* if *Type* is `concepts` and with the internal representation of the role hierarchy in the modal context *M* if *Type* is `roles`.

`getDirectSuperConcepts(`+*EnvName*, +*M*, +*Concept*, −*CL*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	Concept	concept name
	CL	list of concept names

CL is the list of all concept names which are direct super concepts of *Concept*.

`getAllSuperConcepts(`+*EnvName*, +*M*, +*Concept*, −*CL*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	Concept	concept name
	CL	list of concept names

CL is the list of all concept names which are super concepts of *Concept*.

`getDirectSubConcepts(`+*EnvName*, +*M*, +*Concept*, −*CL*`)`

Arguments:	*EnvName*	environment name	*CL* is the list of all concept names which are
	M	modal context	
	Concept	concept name	
	CL	list of concept names	

direct sub concepts of *Concept*.

`getAllSubConcepts(`+*EnvName*, +*M*, +*Concept*, −*CL*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	Concept	concept name
	CL	list of concept names

CL is the list of all concept names which are sub concepts of *Concept*.

`getConcepts(`+*EnvName*, +*M*, −*CL*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	CL	list of concept names

CL is the list of all concept names in the subsumption hierarchy.

`testDirectSuperConcept(`+*EnvName*, +*M*, +*Concept1*, +*Concept2*, −*Concept*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	Concept1	concept name
	Concept2	concept name
	Concept	concept name

Concept is *Concept1* iff *Concept1* is a direct super concept of *Concept2* or *Concept* is *Concept2* iff *Concept2* is a direct super concept of *Concept1* otherwise the predicate fails.

`testDirectSubConcept(`+*EnvName*, +*M*, +*Concept1*, +*Concept2*, −*Concept*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	Concept1	concept name
	Concept2	concept name
	Concept	concept name

Concept is *Concept1* iff *Concept1* is a direct sub concept of *Concept2* or *Concept* is *Concept2* iff *Concept2* is a direct sub concept of *Concept1* otherwise the predicate fails.

`testSuperConcept`(+*EnvName*, +*M*, +*Concept1*, +*Concept2*, −*Concept*)

Arguments:	*EnvName*	environment name
	M	modal context
	Concept1	concept name
	Concept2	concept name
	Concept	concept name

Concept is *Concept1* iff *Concept1* is a direct super concept of *Concept2* or *Concept* is *Concept2* iff *Concept2* is a direct super concept of *Concept1* otherwise the predicate fails.

`testSubConcept`(+*EnvName*, +*M*, +*Concept1*, +*Concept2*, −*Concept*)

Arguments:	*EnvName*	environment name
	M	modal context
	Concept1	concept name
	Concept2	concept name
	Concept	concept name

Concept is *Concept1* iff *Concept1* is a direct super concept of *Concept2* or *Concept* is *Concept2* iff *Concept2* is a direct super concept of *Concept1* otherwise the predicate fails.

`getCommonSuperConcepts`(+*EnvName*, +*M*, +*CL1*, −*CL2*)

Arguments:	*EnvName*	environment name
	M	modal context
	CL1	list of concept names
	CL2	list of concept names

CL2 is the list of all concept names subsuming all concepts in *CL1*.

`getCommonSubConcepts`(+*EnvName*, +*M*, +*CL1*, −*CL2*)

Arguments:	*EnvName*	environment name
	M	modal context
	CL1	list of concept names
	CL2	list of concept names

CL2 is the list of all concept names which are subsumed by all concepts in *CL1*.

4.3 Retrieval commands for roles

`getDirectFatherRoles`(+*EnvName*, +*M*, +*Role*, −*RL*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role	role name
	RL	list of role names

RL is the list of all role names which are direct father roles of *Role*.

`getAllFatherRoles`(+*EnvName*, +*M*, +*Role*, −*RL*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role	role name
	RL	list of role names

RL is the list of all role names which are father roles of *Role*.

`getDirectSonRoles`(+*EnvName*, +*M*, +*Role*, −*RL*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role	role name
	RL	list of role names

RL is the list of all role names which are direct son roles of *Role*.

`getAllSonRoles`(+*EnvName*, +*M*, +*Role*, −*RL*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role	role name
	RL	list of role names

RL is the list of all role names which are son roles of *Role*.

`getRoles`(+*EnvName*, +*M*, −*RL*)

Arguments:	*EnvName*	environment name
	M	modal context
	RL	list of role names

RL is the list of all role names in the subsumption hierarchy.

`testDirectFatherRole`(+*EnvName*, +*M*, +*Role1*, +*Role2*, −*Role*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role1	role name
	Role2	role name
	Role	role name

Role is *Role1* iff *Role1* is a direct father role of *Role2* or *Role* is *Role2* iff *Role2* is a direct father role of *Role1* otherwise the predicate fails

`testDirectSonRole`(+*EnvName*, +*M*, +*Role1*, +*Role2*, −*Role*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role1	role name
	Role2	role name
	Role	role name

Role is *Role1* iff *Role1* is a direct son role of *Role2* or *Role* is *Role2* iff *Role2* is a direct son role of *Role1* otherwise the predicate fails

`testFatherRole`(+*EnvName*, +*M*, +*Role1*, +*Role2*, −*Role*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role1	role name
	Role2	role name
	Role	role name

Role is *Role1* iff *Role1* is a direct father role of *Role2* or *Role* is *Role2* iff *Role2* is a direct father role of *Role1* otherwise the predicate fails

`testSonRole`(+*EnvName*, +*M*, +*Role1*, +*Role2*, −*Role*)

Arguments:	*EnvName*	environment name
	M	modal context
	Role1	role name
	Role2	role name
	Role	role name

Role is *Role1* iff *Role1* is a direct son role of *Role2* or *Role* is *Role2* iff *Role2* is a direct son role of *Role1* otherwise the predicate fails `getCommonFatherRoles(`+*EnvName*`,` +*M*`,` +*RL1*`,` −*RL2*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	RL1	list of role names
	RL2	list of role names

RL2 is the list of all role names subsuming all roles in *RL1*.

`getCommonSonRoles(`+*EnvName*`,` +*M*`,` +*RL1*`,` −*RL2*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	RL1	list of role names
	RL2	list of role names

RL2 is the list of all role names which are subsumed by all roles in *RL1*.

Chapter 5

Realization and Retrieval of objects

The *realization problem* is to find for an object a all concepts C such that a is an instance of C. The *retrieval problem* is to find for a concept C all objects a such that a is an instance of C. In MOTEL both problems are solved using the `deduce`-command.
`deduce(`+−*EnvName*`,` +−*M*`,` `elementOf(`+−*X*`,` +−*CT*`)`, +−*Exp*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	CT	concept term
	Exp	explanation

For a given object name X all concept names CT such that X is an instance of CT will be enumerated. *Exp* provides some explanation why this is true. For a given concept term CT all object names X such that X is an instance of CT will be enumerated. The concept term CT can be eiter a variable or a concept term containing role names but not general role terms only. Again *Exp* provides some explanation why this is true. If M is not instantiated, it will enumerate all modal contexts such that X is an instance of C. Finally, if *EnvName* is a variable, it will be instantiated with an environment such that X is an instance of C in modal context M.

`realize(`+*EnvName*`,` +*M*`,` +*X*`,` −*CL*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	CL	list of concept names
	Exp	explanation

try it.

`getAllObjects(`+*EnvName*`,` +*M*`,` −*OL*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	OL	list of object names

OL is the list of names of all objects known to exist in environment *EnvName* and modal context *M*.

To get information about roles we have the predicate `deduce(`+−*EnvName*`,` +−*M*`,` `roleFiller(`+−*X*`,` +−*R*`,` −*L*`,` −*N*`),` -*Exp*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	R	role name
	L	list of object names
	N	number
	Exp	explanation

gets all objects in the range of role *R* for argument *X* in environment *EnvName* and modal context *M*. *L* is instantiated with the list of all these objects and *N* is the number of elements in this list.

It is possible to use abduction to find a set of hypothesises, i.e. terminological axioms, such that some object *X* is an element of a concept *C* if these hypothesises are true.
`abduce(`+−*EnvName*`,` +−*M*`,` **H*`,` `elementOf(`+−*X*`,` +−*C*`),` +−*Exp*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	C	concept name
	**H*	set of hypothesises
	Exp	explanation

For a given object name *X* all concepts *C* such that *X* is an instance of *C* using the additional set of hypothesises will be enumerated. *Exp* provides some explanation why this is true. For a given concept name *C* all object names *X* such that *X* is an instance of *C* will be enumerated. Again *Exp* provides some explanation why this is true. If *M* is not instantiated, it will enumerate all modal contexts such that *X* is an instance of *C*. Finally, if *EnvName* is a variable, it will be instantiated with an environment such that *X* is an instance of *C* in modal context *M*.

Usually, MOTEL does not compute all possible explanations. However, this can be changed using `setOption(allProofs,yes)`

Chapter 6

(In)consistency

We call a knowledge base inconsistent, if we can prove form some object name X and some concept name A that X is an element of A and of `not(()`A`)`. Otherwise the knowledge base is consistent.

`consistent`*[([+EnvName,][+M])]*

Arguments:	*EnvName*	environment name
	M	modal context

succeeds if the environment *EnvName* and modal context *M* is consistent.

`inconsistent`*[([+EnvName,][+M])]*

Arguments:	*EnvName*	environment name
	M	modal context

succeeds if the environment *EnvName* and modal context *M* is inconsistent.

Chapter 7

Functional Dependencies

In this chapter we describe the component of MOTEL for specifying and reasoning about functional dependencies among roles.

7.1 Definition and Revision of Funcional Dependencies

Functional dependencies are described using functional dependency literals of the following form

`infl`$(+X, +Y, +W)$
`posInfl`$(+X, +Y)$
`negInfl`$(+X, +Y)$
`noInfl`$(+X, +Y)$
`change`$(+X, +W)$
`increase`$(+X)$
`decrease`$(+X)$

X and Y denote roles/attributes and W denotes the weight of X influencing Y or W denotes the weight of change of an attribute. `posInfl` is assigned the weight 1.0, `negInfl` the weight -1.0 and `noInfl` the weight 0.0. The weights for `increase`, `decrease` and `noChange` are 1.0, -1.0 and 0.0, respectively.

The command `def` can be used to define a functional dependency, the command `undef` can be used to remove it.

`def(`*[+EnvName]*, *[+MS]*, *+Fact*`)`

Arguments:	*EnvName*	environment name
	MS	modal context
	Fact	functional dependency literal

This predicate is used to update the knowledge base of information about the functional dependencies. The definition of multiple influences between attributes and multiple changes on an attribute are prevented.

`undef(`*[+EnvName]*, *[+MS]*, *+−Fact*`)`

Arguments:	*EnvName*	environment name
	MS	modal context
	Fact	functional dependency literal

retracts all facts matching Fact.

With the following predicates it is possible to display information about the functional dependencies which are currently defined.

`showFDW(`*[+−Env]*`)`

Arguments:	*Env*	environment name (internal representation)

displays the user defined functional dependencies in the knowledge base.

`showInfl`(+−*Env*)

Arguments: *Env* environment name (internal representation)
displays the user defined influence relationships in the knowledge base.

`showChange`(+−*Env*)

Arguments: *Env* environment name (internal representation)
displays the user defined changes in the knowledge base.

`showFD`(*[+−Env]*)

Arguments: *Env* environment name (internal representation)
displays the user defined functional dependencies in the knowledge base. Similar to `showFDW`, but the default reprsentation is chosen.

7.2 Deduction

`deduce`(*[+EnvName]*, *[+MS]*, +−*Info*, *[−E]*)

Arguments:

	EnvName	environment name
	MS	modal context
	Info	a literal of the appropriate kind, see description below
	E	explanations (not as yet implemented)

Succeeds if *Info* can be inferred by deduction. Here is a short description of *Info* that can be inferred.

`infl`(+−X, +−Y, +−W)

- X attribute/role name
- Y attribute/role name
- W list of weights weight, a value

computes the cumulative weight W of all the influence links between the attributes X and Y.

`simultInfl`(+−Xs, +−Y, +−W)

- Xs list of attributes/role names
- Y attribute/role name
- W list of weights weight, a value

checks if the list Xs is well-defined (that is, is Xs a SET of independent attributes) and computes the total weight W of the attributes in the list Xs simultaneously influencing attribute Y.

`leastInfl`(+−X, +−Y)

- X attribute/role name
- Y attribute/role name

succeeds if X is a least attribute influencing Y.

`leastInfls`(+−Xs, +−Y)

- Xs list of attributes/role names
- Y attribute/role name

collects the least attributes influencing Y in Xs.

`greatestInfl`(+−X, +−Y)

- X attribute/role name
- Y attribute/role name

succeeds if Y is a greatest attribute influenced by X.

`greatestInfls`(+−Xs, +−Y)

- X attribute/role name
- Ys list of attributes/role names

collects the greatest attributes influenced by X in Ys.

`maxPosInfl(`$+-X$`,` $+-Y$`,` $+-Wmax$`)`

X	attribute/role name
Y	attribute/role name
$Wmax$	weight, a value

succeeds if $Wmax$ is the greatest weight with which X influences Y positively.

`maxNegInfl(`$+-X$`,` $+-Y$`,` $+-WMin$`)`

X	attribute/role name
Y	attribute/role name
$WMin$	a value

succeeds if $WMin$ is the greatest weight with which X influences Y negatively.

`change(`$+-Y$`,` $+-W$`)`

Y	attribute/role name
Wy	weight of change of Y

determines the change in Y.

`posInfl(`$+-X$`,` $+-Y$`)`

X	attribute/role name
Y	attribute/role name

succeeds if attribute X influences attribute Y positively.

`negInfl(`$+-X$`,` $+-Y$`)`

X	attribute/role name
Y	attribute/role name

succeeds if attribute X influences attribute Y negatively.

`noInfl(`$+-X$`,` $+-Y$`)`

X	attribute/role name
Y	attribute/role name

succeeds if the cumulative influence between the attributes X and Y is 0.0.

`simultPosInfl(`$+-Xs$`,` $+-Y$`)`

Xs	list of attributes/role names
Y	attribute/role name

succeeds if the simultaneous influence of the attributes in the list Xs on the attribute Y is positive.

`simultNegInfl(`$+-Xs$`,` $+-Y$`)`

Xs	list of attributes/role names
Y	attribute/role name

succeeds if the simultaneous influence of the attributes in the list Xs on the attribute Y is positive.

`simultNoInfl(`$+-Xs$`,` $+-Y$`)`

Xs	list of attributes/role names
Y	attribute/role name

succeeds if the simultaneous influence of the attributes in the list Xs on the attribute Y is positive.

`increase(`$+-X$`)`

Y	attribute/role name

succeeds if attribute Y increases.

`decrease(`$+-X$`)`

Y	attribute/role name

succeeds if attribute Y decreases.

`noChange(`$+-X$`)`

Y	attribute/role name

succeeds if attribute Y does not change (i.e. there is neither an increase nor a decrease).

7.3 Abduction

The standard query for abduction is

`abduce(`*[+EnvName]*, *[+MS]*, +−*H*, +−*C*, E`)`

where *EnvName* denotes an environment name, *MS* a modal context and *E* a list of explanations. *H* and *C* repectively denote a hypothesis and its consequent. In this component of MOTEL *H* and *C* can also be lists of hypotheses, respectively, consequents. The different possibilities are listed below. Explanations are not as yet generated for inference with functional dependencies. Provision was made for future implementation.

`abduce(`*[+EnvName]*, *[+MS]*, +−`change(`+−*X*, +−*Wx*`)`, +−`change(`+−*Y*, +−*Wy*`)`, []`)`

Arguments:		
	EnvName	environment name
	MS	modal context
	X	attribute/role name
	Wx	weight of change of *X*
	Y	attribute/role name
	Wy	weight of change of *Y*

Succeeds if, under the hypothesis of `change(`+−*X*, +−*Wx*`)`, `change(`+−*Y*, +−*Wy*`)` follows.

`abduce(`*[+EnvName]*, *[+MS]*, +−*Hypothesis*, +−*Consequent*, []`)`

Arguments:		
	EnvName	environment name
	MS	modal context
	Hypothesis	a literal of appropriate kind
	Consequent	a literal of appropriate kind

Succeeds if *Consequent* follows under the hypothesis *Hypothesis*. *Hypothesis* and *Consequent* are of the form:

`increase(`+−*X*`)`, `decrease(`+−*X*`)`, `noChange(`+−*X*`)`.

`abduce(`*[+EnvName]*, *[+MS]*, +*Changes*, +−`change(`+−*Y*, +−*Wy*`)`, []`)`

Arguments:		
	EnvName	environment name
	MS	modal context
	Changes	a list of literals of the form `change(`+−*X*, +*W*`)`
	Y	attribute/role name
	Wy	weight of change of *Y*

Succeeds if `change(`+−*Y*, +−*W*`)` follows under the hypotheses of *Changes*.

`abduce(`*[+EnvName]*, *[+MS]*, +−*Hypotheses*, +−*Consequent*, []`)`

Arguments:		
	EnvName	environment name
	MS	modal context
	Hypotheses	a list of literals of the appropriate kind
	Consequent	a literal of the appropriate kind

Succeeds if *Consequent* follows under the hypotheses *Hypotheses*. *Hypotheses* is a list of

`increase(`+−*X*`)`, `decrease(`+−*X*`)`, `noChange(`+−*X*`)`

literals and *Consequent* is one of these literals.

`abduce(`*[+EnvName]*, *[+MS]*, +−*Change*, +−*Changes*, []`)`

Arguments:		
	EnvName	environment name
	MS	modal context
	Change	a literal of the form `change(`+−*X*, +−*W*`)`
	Changes	a list of literals of the form `change(`+−*X*, +−*W*`)`

Succeeds if *Changes* hold under the hypothesis that *Change* holds.

`abduce(`*[+EnvName]*, *[+MS]*, +−*Hypothesis*, +−*Consequents*, [] `)`

Arguments:	*EnvName*	environment name
	MS	modal context
	Hypothesis	a literal of the form
		`increase(`+−X`)`
		`decrease(`+−X`)`
		`noChange(`+−X`)`
	Consequents	a list of literals of this form

Succeeds if *Consequents* follow under the hypothesis *Hypothesis*.

Chapter 8

Examples

8.1 Modal Operators

Let's suppose that we have some agent a1 in our world. We can form the concept containing everything that a1 believes to be a car using the terminological axiom (2) in the following knowledge base. We call this concept c1. Furthermore we specify that a1 believes that c1 is the concept containing everything he believes to be a car using axiom (4). And we assert that provability for the believe of a1 is like the modal logic kd45.

That implies that a1 is able to perform positive introspection, i.e. he believes what he believes. Suppose audi is an element of c1 (axiom (6)). If c3 is the concept containing everything that a1 believes to be an element of c1 (axiom (3)) and a1 believes that this equivalence is true, then audi must be an element of c3.

```
(1) modalAxioms(kd45,believe,a1).
(2) defconcept(c1,b(believe,a1,auto)).
(3) defconcept(c3,b(believe,a1,c1)).
(4) defconcept([b(believe,a1)],c1,b(believe,a1,auto)).
(5) defconcept([b(believe,a1)],c3,b(believe,a1,c1)).
(6) assert_ind(audi,c1).
```

We can check this using the query

```
| ?- deduce(elementOf(audi,c3)).

yes
```

So the believes of a1 act like we expect them to do.

8.2 Role closure

Suppose we define a concept onlyMaleChildren using the terminological axiom (1) in the following knowledge base. Then given the assertional axioms (2)–(7) we cannot prove that tom is an element of onlyMaleChildren because there might exists children of tom which are not male.

But using the axiom (8) we state that at any point of time we know all objects which are role fillers of the child role for tom.

```
(1) defconcept(onlyMaleChildren,all(child,male)).
(2) assert_ind(tom,peter,child).
(3) assert_ind(tom,chris,child).
(4) assert_ind(tom,tim,child).
(5) assert_ind(peter,male).
(6) assert_ind(chris,male).
(7) assert_ind(tim,male).
(8) defclosed(tom,Y,child).
```

So we can actually prove that tom is an element of onlyMaleChildren.

```
| ?- deduce(elementOf(tom,onlyMaleChildren)).

yes
```

If we get to know a new child of `tom`, say `betty`, which is not `male`, we just add the assertional axioms `(9)` and `(10)`.

`(10) assert_ind(tom,betty,child)`
`(11) assert_ind(betty,not(male))`

Now we are no longer able to deduce that `tom` is an element of `onlyMaleChildren`, but we are still consistent.

```
| ?- deduce(elementOf(tom,onlyMaleChildren)).

no
| ?- consistent([]).

yes
```

8.3 Abduction

Here we consider the famous nixon-diamond. Suppose we specify that somebody who is a `quaker` and a `normalQuaker` is a `dove`. And somebody who is a `republican` and a `normalRepublican` is a `hawk`. The agent `nixon` is a `quaker` and a `republican`. This can be done using the following axioms:

`(1) defprimconcept(and([quaker,normalQuaker]),dove).`
`(2) defprimconcept(and([republican,normalRepublican]),hawk).`
`(3) assert_ind(nixon,quaker).`
`(4) assert_ind(nixon,republican).`

Now we are neither able to deduce that `nixon` is a `dove` nor that he is a `hawk`.

```
| ?- deduce(elementOf(nixon,dove)).

no
| ?- deduce(elementOf(nixon,hawk)).

no
```

But we can use the abductive inference mechanism to get information about the additional knowledge we need to infere that `nixon` is a `dove`.

```
| ?- abdeduce(elementOf(nixon,dove),H,E).

E = proved(in([],dove,nixon),hyp([]),
      basedOn(and([proved(in([],quaker,nixon),hyp([]),basedOn(abox)),
                   proved(in([],normalQuaker,nixon),hyp([]),
                          basedOn(usingAbHyp(in(env(e4),rn(_H,_G,_F,_E),modal([]),
                                normalQuaker,nixon,hyp(_B),ab(_D),call(_C),
                                proved(in([],normalQuaker,nixon),
                                        hyp(_B),basedOn(_A))))))]))),
H = [in(env(e4),rn(_P,_O,_N,_M),modal([]),normalQuaker,nixon,hyp(_J),ab(_L),
        call(_K),proved(in([],normalQuaker,nixon),hyp(_J),basedOn(_I)))] ?

yes
```

The PROLOG variable `H` is instantiated with the set hypothesises that we need to infer that `nixon` is a `dove`. Here we needed only one hypothesis, namly that `nixon` is a `normalQuaker`. The PROLOG variable `E` is instantiated with the explanation why we were able to prove that `nixon` is a `dove`. The proof was based on the fact that `nixon` is a `quaker` and on the hypothesis that he is a `normalQuaker`.

Of course, we able to abduce that `nixon` is a `hawk`:

```
| ?- abduce(H,elementOf(nixon,hawk),H).

E = proved(in([],hawk,nixon),hyp([]),
      basedOn(and([proved(in([],republican,nixon),hyp([]),basedOn(abox)),
                   proved(in([],normalRepublican,nixon),hyp([]),
                            basedOn(usingAbHyp(in(env(e4),rn(_H,_G,_F,_E),modal([]),
                                 normalRepublican,nixon,hyp(_B),ab(_D),call(_C),
                                 proved(in([],normalRepublican,nixon),
                                           hyp(_B),basedOn(_A))))))]))),
H = [in(env(e4),rn(_P,_O,_N,_M),modal([]),normalRepublican,nixon,hyp(_J),ab(_L),
        call(_K),proved(in([],normalRepublican,nixon),hyp(_J),basedOn(_I)))] ?

yes
```

8.4 Defaults

In this example we want to specify that children of doctors are rich person by default. So we have some role hasChild and to talk about the children of doctors we need the role hasDoctorParent which is the restriction of the inverse of hasChild, i.e. the parent role, to doctor.

```
(1) defprimrole(hasChild).
(2) defrole(hasDoctorParent,restr(inverse(hasChild),doctor)).
```

So if somebody is in the domain of hasDoctorParent, i.e. is a child of doctor, and we cannot prove that he is an element of not(richPerson), then we expect him to be an element of richPerson. This is what axiom (3) says:

```
(3) defprimconcept(and([some(hasDoctorParent,top),
                        naf(not(richPerson))]),richPerson).
```

Let's add some assertional axioms:

```
(4) assert_ind(chris,doctor).
(5) assert_ind(chris,tom,hasChild).
```

Because tom is a child of a doctor he must be rich:

```
| ?- deduce(elementOf(tom,richPerson)).

yes
```

On the other hand, we can add to our knowledge that tom is not rich using the assertional axiom (6).

```
(6) assert_ind(tom,not(richPerson)).
```

Now we no longer able to deduce that tom is a richPerson and we are still consistent.

```
| ?- deduce(elementOf(tom,richPerson)).

no
| ?- consistent([]).

yes
```

8.5 Enumeration Types

Suppose we are talking about some bmw. We expect this car to be either yellow, red, or red. We can put this in our knowledge base using the axioms (1) and (2).

```
(1) defconcept(c1,and([car,some(hasCol,set([yellow,blue,red])),
                        all(hasCol,set([yellow,blue,red]))])).
(2) assert_ind(bmw,c1).
```

Now somebody tells us that the bmw is not yellow. Then we can add this knowledge by axioms (3) and (4).

```
(3) defconcept(c2,some(hasCol,not(set([yellow])))).
```

(4) `assert_ind(bmw,c2).`

Of course, we expect the `bmw` to be either `blue` or `red`. Therefore we build the following concept `c3`:

(5) `defconcept(c3,some(hasCol,set([blue,red]))).`

and ask wether `bmw` is an element of `c3`.

```
| ?- deduce(elementOf(bmw,c3)).

yes
```

We get the expected answer.

Appendix A

Quintus Prolog Release 3.1.1 Specific Predicates

`ask(`+−*EnvName*`,` +−*M*`, elementOf(`+−*X*`,` +−*C*`),` +−*Exp*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	C	concept name
	Exp	explanation

A synonym for the `deduce` predicate described in chapter 5.

`ask(`+−*EnvName*`,` +−*M*`, roleFiller(`+−*X*`,` +−*R*`,` −*L*`,` −*N*`),` -*Exp*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	R	role name
	L	list of object names
	N	number
	Exp	explanation

A synonym for the `deduce` predicate described in chapter 5.

`saveMOTEL(`+*FileName*`)`

Arguments:	*FileName*	file name

Saves the whole program state, containing all user defined predicates. The file *FileName* becomes an executable file.

Appendix B

SICStus 2.1 Specific Predicates

`ask(`+−*EnvName*`,` +−*M*`, elementOf(`+−*X*`,` +−*C*`),` +−*Exp*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	C	concept name
	Exp	explanation

A synonym for the `deduce` predicate described in chapter 5.

`ask(`+−*EnvName*`,` +−*M*`, roleFiller(`+−*X*`,` +−*R*`,` −*L*`,` −*N*`),` -*Exp*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	R	role name
	L	list of object names
	N	number
	Exp	explanation

A synonym for the `deduce` predicate described in chapter 5.

`saveMOTEL(`+*FileName*`)`

Arguments:	*FileName*	file name

Saves the whole program state, containing all user defined predicates. The file *FileName* becomes an executable file.

Appendix C

SB-LITTERS Interface

`sb_defenv`(+*EnvName*, +*Comment*)
(`SB__DEFENV` *ENVNAME COMMENT*)

Arguments: *EnvName* environment name
Comment string
creates a new environment with identifier *EnvName* and associated comment *Comment*.

`sb_initenv`*[*(+*EnvName*)*]*
(`SB__INITENV` *[ENVNAME]*)

Arguments: *EnvName* environment name
initializes environment *EnvName* or the current environment if no argument is given.

`sb_primconcept`(*[*+*EnvName*,*]* *[*+*M*,*]* +*CName1*, *[*+*CSpecList]*)
(`SB__PRIMCONCEPT` *[ENVNAME]* *[*(`:LIST` *[*(`B` *O A*) (`D` *O A*) (`BC` *O A*) (`DC` *O A*)*]* *) *CNAME1* *[*CSpecList*]*)

Arguments:		
	EnvName	environment name
	M	modal context
	CName1	concept name
	CSpecList	SB-ONE concept specification

impose necessary conditions on the interpretation of *CName1* in environment *EnvName* and modal context *M*. The conditions are specified by *CSpecList*.

`sb_defconcept`(*[*+*EnvName*,*]* *[*+*M*,*]* +*CName1*, +CSpecList)
(`SB__PRIMCONCEPT` *[ENVNAME]* *[*(`:LIST` *[*(`B` *O A*) (`D` *O A*) (`BC` *O A*) (`DC` *O A*)*]* *) *CNAME1* CSpecList)

Arguments:		
	EnvName	environment name
	M	modal context
	CName1	concept name
	CSpecList	SB-ONE concept specification

impose necessary and sufficient conditions on the interpretation of *CName1* in environment *EnvName* and modal context *M*. The conditions are specified by *CSpecList*.

CSpecList is a list of SB-ONE concept specification elements having the following form:

- `supers`([+C_1, ... , +C_n])
  (`SUPERS` (`:LIST` C_1 C_2 ... C_n))
  specifies a concept which is the conjunction of $C_1,\ldots,C_n$.
- `restrict_inh`(+*RName1*, `restricts`(+*RName2*, `range`(+*CName2*, +*CNameDef*)))
  (`RESTRICT__INH` *RNAME1* (`RESTRICTS` *RNAME2* (`RANGE` *CNAME2 CNAMEDEF*)))
  specifies a concept which is the domain of *RName1*. *RName1* is the restriction of *RName2* to the range *CName2* and to the default range *CNameDef*.

- `nr(`+*RName1*`,` *MinNr*`,` +*MaxNr*`,` +*DefNr*`)`
  `(NR` *RNAME1 MINNR MAXNR DEFNR*`)`
  specifies a concept which contains all object having at least *MinNr*, at most *MaxNr*, and by default *DefNr* role fillers for role *RName1*.

`sb_primelemrole(`*[+EnvName,][+MS,]+RName1*`,` +*PrimRSpec*`)`
`(SB__PRIMELEMROLE` *[ENVNAME] [MS] RNAME1* `(DOMAIN-RANGE` *CNAME1 CNAME2 CNAMEDEF*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	RName1	role name
	PrimRSpec	SB-ONE primitive role specification

impose necessary conditions on the interpretation of *RName1* in environment *EnvName1* and modal context *M*. The conditions are specified by *PrimRSpec*. *PrimRSpec* takes the following form: `domain-range(`+*CName1*`,` +*CName2*`,` +*CNameDef*`)`. This defines *RName1* to be a role with domain *CName1*, range *CName2* and default range *CNameDef* in environment *EnvName* and modal context *M*.

`sb_defelemrole(`*[+EnvName,][+M,]+RName1*`,` +*RSpec*`)`
`(SB__DEFELEMROLE` *[ENVNAME] [M] RNAME1* `(RESTRICTS` *RNAME2* `(RANGE` *CNAME1 CNAMEDEF*`)))`

Arguments:	*EnvName*	environment name
	M	modal context
	RName1	role name
	RSpec	SB-ONE role specification

impose necessary and sufficient conditions on the interpretation of *RName1* in environment *EnvName1* and modal context *M*. The conditions are specified by *RSpec* which takes the form `restricts(`+*RName2*`, range(`+*CName1*`,` +*CNameDef*`))`. *RName1* is a maximal subset of the role *RName2* such that each role filler of *RName1* is in *CName1*.

`sb_disjoint(`*[+EnvName,][+M,]+CName1*`,` +*CName2*`)`
`(SB__DISJOINT` *[ENVNAME] [M] CNAME1 CNAME2*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	CName1	concept name
	CName2	concept name

declares the concepts *CName1* and *CName2* to be disjoint.

`sb_defelem(`*[+EnvName,][+M,]+ICName1*`,` +*ISpecList*`)`
`(SB__DEFELEM` *[ENVNAME] [M] ICNAME1 ISPECLIST*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	ICName1	object name
	ISpecList	SB-ONE individual specification

introduces an object in environment *EnvName* and modal context *M* which obeys the restrictions given in *ISpecList*.

A SB-ONE individual specification takes the following form

$$\texttt{[isa(}+\mathit{CName}\texttt{),}+\mathit{IRSpec}_1\texttt{, \ldots, }+\mathit{IRSpec}_n\texttt{]}$$
$$\texttt{(:LIST (ISA } \mathit{CNAME}\texttt{) } \mathit{IRSPEC}_1 \ldots \mathit{IRSPEC}_n\texttt{)}$$

where IRSpec_i is

$$\texttt{irole(}+\mathit{RName}_i\texttt{, iname(}+\mathit{IRName}_i\texttt{), }+\mathit{IRList}_i\texttt{)}$$
$$\texttt{(IROLE } \mathit{RNAMEi} \texttt{ (INAME } \mathit{IRNAME}_i\texttt{) } \mathit{IRLIST}_i\texttt{)}$$

and the argument *IRList*$_i$ is a list which is either empty or contains either `nr(`*+MinNr*$_i$`,` *+MaxNr*$_i$`,` *+DefNr*$_i$`)` (`NR` *MINNRi MAXNRi DEFNRi*), `vr(`*+ICName*$_i$`)` (`VR` *ICNAMEi*), or both.

The result of `sb_defelem` is the introduction of an object *ICName1* which is a member of *CName* and pairs (*ICName1*,*ICName*$_i$) which are elements of *IRName*$_i$. The role *IRName*$_i$ is a subset of *RName*$_i$ and has atleast *MinNr*$_i$ role fillers and atmost *MaxNr*$_i$ role fillers. The default number of role fillers is *DefNr*$_i$.

`sb_attributes(`*[+EnvName,][+M,]+CN*`,` *+InfoList*`)`
`(SB_ATTRIBUTES` *[ENVNAME] [M] CN INFOLIST*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	CN	concept name
	InfoList	list of info nodes

attaches some attributive information to concept *CN* in environment *EnvName* and *M*. The information is taken from *InfoList* which is a list of info nodes of the form (*Attribute*, *Value*).
Lisp syntax for INFOLIST:
`(:LIST (:LIST` *ATTR1 VALUE1*`) ... (:LIST` *ATTRn VALUEn*`))`

`sb_attributes(`*[+EnvName,][+M,]+CN*`,` *+RN*`,` *+InfoList*`)`
`(SB_ATTRIBUTES` *[ENVNAME] [M] CN RN INFOLIST*`)`

Arguments:	*EnvName*	environment name
	M	modal context
	CN	concept name
	RN	role name
	InfoList	list of info nodes

attaches some attributive information to role *RN* at concept *CN* in environment *EnvName* and *M*. The information is taken from *InfoList* which is a list of info nodes of the form (*Attribute*, *Value*).
Lisp syntax for INFOLIST:
`(:LIST (:LIST` *ATTR1 VALUE1*`) ... (:LIST` *ATTRn VALUEn*`))`

`sb_fact(`*[+EnvName,][+M,]*`isa(`*+−X*`,` *+−CT*`))`
`(SB_FACT` *[ENVNAME] [M]* `(ISA` *X CT*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	X	object name
	CT	concept term

For a given object name *X* all concept names *CT* such that *X* is an instance of *CT* in the world description will be enumerated. *Exp* provides some explanation why this is true. For a given concept term*CT* all object names *X* such that *X* is an instance of *CT* in the world description will be enumerated. The concept term *CT* can be eiter a variable or a concept name. Again *Exp* provides some explanation why this is true.

`sb_fact(`*[+EnvName,][+M,]*`irole(`*+∗RName*`,` *+∗ICName1*`,` *+∗ICName2*`))`
`(SB_FACT` *[ENVNAME] [M]* `(IROLE` *RNAME ICNAME1 ICNAME2*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	RName	role name
	ICName1	object name
	ICName2	object name

succeeds if the pair (*ICName1*,*ICName2*) is an element of the role *RName* in the world description in environment *EnvName* and modal context *M*.

`sb_fact(`*[+EnvName,] [+M,]*`role(`+∗*RName*, +∗*CNameDom*, +∗*CNameRan*`))`
`(SB_FACT` *[ENVNAME] [M]* `(ROLE` *RNAME CNAMEDOM CNAMERAN*`))`

Arguments:		
	EnvName	environment name
	M	modal context
	RName	role name
	CNameDom	concept name
	CNameRan	concept name

succeeds if *RName* is a role with domain *CNameDom* and range *CNameRan* in the terminology.

`sb_fact(`*[+EnvName,] [+M,]*`attributes(`+∗*CN*, +∗*Attribute*, +∗*Value*`))`
`(SB_FACT` *[ENVNAME] [M]* `(ATTRIBUTES` *CN ATTRIBUTE VALUE*`))`

Arguments:		
	EnvName	environment name
	M	modal context
	CN	concept name
	Attribute	term
	Value	term

succeeds if the *Value* is the value of *Attribute* for concept *CN* in environment *EnvName* and modal context *M*.

`sb_fact(`*[+EnvName,] [+M,]*`attributes(`+∗*CN*, +∗*RN*, +∗*Attribute*, +∗*Value*`))`
`(SB_FACT` *[ENVNAME] [M]* `(ATTRIBUTES` *CN RN ATTRIBUTE VALUE*`))`

Arguments:		
	EnvName	environment name
	M	modal context
	CN	concept name
	RN	role name
	Attribute	term
	Value	term

succeeds if the *Value* is the value of *Attribute* for role *RN* at concept *CN* in environment *EnvName* and modal context *M*.

`sb_fact(`*[+EnvName,] [+M,]*`allRoles(`+*CName*, −*Info*`))`
`(SB_FACT` *[ENVNAME] [M]* `(ALL_ROLES` *CNAME INFO*`))`

Arguments:		
	EnvName	environment name
	M	modal context
	CName	concept name
	Info	list containing informations

Info is a list consisting of lists each containing the role name, the domain, the codomain, the minimal number of role fillers, the maximal number of role fillers, and the default number of role fillers of a role with domain *CName*.

Example:
```
?- sb_fact(initial,[],allRoles(golf,X))
   X = [[has_part,golf,windshield,1,1,1],[consumes,golf,gasoline]]
```

`sb_ask(`*[+EnvName,] [+M,]*`supers(`+∗*CName1*, +∗*CName2*`))`
`(SB_ASK` *[ENVNAME] [M]* `(SUPERS` *CNAME1 CNAME2*`))`

Arguments:		
	EnvName	environment name
	M	modal context
	CName1	concept name
	CName2	concept name

succeeds if *CName2* is a direct superconcept of *CName1* in the current subsumption hierarchy.

`sb_ask(`*[+EnvName,] [+M,]*`supers*(`*+∗CName1*`,` *+∗CName2*`))`
`(SB_ASK` *[ENVNAME] [M]* `(SUPERS*` *CNAME1 CNAME2*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	CName1	concept name
	CName2	concept name

succeeds if *CName2* is a superconcept of *CName2* in the current subsumption hierarchy.

`sb_ask(`*[+EnvName,] [+M,]*`role(`*+∗RName*`,` *+∗CNameDom*`,` *+∗CNameRan*`))`
`(SB_ASK` *[ENVNAME] [M]* `(ROLE` *RNAME CNAMEDOM CNAMERAN*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	CName1	concept name
	CName2	concept name

succeeds if *RName* is a role with domain *CNameDom* and range *CNameRan*.

`sb_ask(`*[+EnvName,] [+M,]*`roleDef(`*+∗RName*`,` *+∗CNameDef*`))`
`(SB_ASK` *[ENVNAME] [M]* `(ROLEDEF` *RNAME CNAMEDEF*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	RName	role name
	CNameDef	concept name

succeeds if *RName* is a role with default range *CNameDef*.

`sb_ask(`*[+EnvName,] [+M,]*`roleNr(`*+∗RName*`,` *+∗MinNr*`,` *+∗MaxNr*`))`
`(SB_ASK` *[ENVNAME] [M]* `(ROLENR` *RNAME MINNR MAXNR*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	RName	role name
	MinNr	number
	MaxNr	number

succeeds if *RName* is a role with at least *MinNr* and at most *MaxNr* role fillers.

`sb_ask(`*[+EnvName,] [+M,]*`roleDefNr(`*+∗RName*`,` *+∗DefNr*`))`
`(SB_ASK` *[ENVNAME] [M]* `(ROLEDEFNR` *RNAME DEFNR*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	RName	role name
	DefNr	number

succeeds if *RName* is a role with default number *DefNr* of role fillers.

`sb_ask(`*[+EnvName,] [+M,]*`isa(`*+∗ICName*`,` *+∗CName*`))`
`(SB_ASK` *[ENVNAME] [M]* `(ISA` *ICNAME CNAME*`))`

Arguments:	*EnvName*	environment name
	M	modal context
	ICName	object name
	CName	concept name

succeeds if *ICName* is an element of *CName* in environment *EnvName* and modal context *M*.

sb_ask(*[+EnvName,]* *[+M,]* irole(+**RName*, +**ICName1*, +**ICName2*))
(SB_ASK *[ENVNAME]* *[M]* (IROLE *RNAME ICNAME1 ICNAME2*))

Arguments:		
	EnvName	environment name
	M	modal context
	RName	role name
	ICName1	object name
	ICName2	object name

succeeds if the pair (*ICName1*,*ICName2*) is an element of the role *RName* in environment *EnvName* and modal context *M*.

sb_ask(*[+EnvName,]* *[+M,]* attributes(+**CN*, +**Attribute*, +**Value*))
(SB_ASK *[ENVNAME]* *[M]* (ATTRIBUTES *CN ATTRIBUTE VALUE*))

Arguments:		
	EnvName	environment name
	M	modal context
	CN	concept name
	Attribute	term
	Value	term

succeeds if the *Value* is the value of *Attribute* for concept *CN* in environment *EnvName* and modal context *M*.

sb_ask(*[+EnvName,]* *[+M,]* attributes(+**CN*, +**RN*, +**Attribute*, +**Value*))
(SB_ASK *[ENVNAME]* *[M]* (ATTRIBUTES *CN RN ATTRIBUTE VALUE*))

Arguments:		
	EnvName	environment name
	M	modal context
	CN	concept name
	RN	role name
	Attribute	term
	Value	term

succeeds if the *Value* is the value of *Attribute* for role *RN* at concept *CN* in environment *EnvName* and modal context *M*.

sb_ask(*[+EnvName,]* *[+M,]* allRoles(+*CName*, −*Info*))
(SB_ASK *[ENVNAME]* *[M]* (ALL_ROLES *CNAME INFO*))

Arguments:		
	EnvName	environment name
	M	modal context
	CName	concept name
	Info	list containing informations

Info is a list consisting of lists each containing the role name, the domain, the codomain, the minimal number of role fillers, the maximal number of role fillers, and the default number of role fillers of a role with domain *CName*.

Example:
```
?- sb_ask(initial,[],allRoles(golf,X))
   X = [[has_part,golf,windshield,1,1,1],[consumes,golf,gasoline]]
```

Appendix D

The Common Lisp to PROLOG interface

This interface provides functions to call a PROLOG goal from within lisp in a lisp–like syntax. The results produced by PROLOG are bound to the corresponding variables in lisp.

D.1 The syntax of a PROLOG goal in lisp

- Functions are notated in infix notation:
  `atomic(1)` gets `(atomic 1)`.
- Function arguments are separated by spaces:
  `defprimconcept(female, not(male))`
  gets `(defprimconcept female (not male))`.
- PROLOG variables have a '?' as first character, e.g. `?a` or `?x`.
- PROLOG lists get lisp lists with the keyword `:list` as the first element:
  `[male, female]` gets `(:list male female)`.
- An open PROLOG list is written as follows:
  `[a,b,c,d,e | V]` gets `(:openlist (a b c d e) ?v)`.
- To conserve PROLOG symbols with capital letter, the are escaped with '_' in lisp:
  - `makeEnvironment` gets `make_environment`,
  - `assert_ind` gets `assert__ind`,
  - `make_Env` gets `make___env`.
- The existential quantifier is used as follows:
  - `E^expression` gets `(^(?e) expression)`,
  - `D^E^expression` gets `(^(?d ?e) expression)` and so on.

D.2 The functions `(start-prolog)`, `(start-motel)`, `(reset-prolog)` and `(kill-prolog)`.

- `(start-prolog)` starts SICStus Prolog as a subprocess. This function must be called before using `(prolog-goal)` or `(do-prolog)`. It returns three values: The input/output-stream, the error-output-stream and the process-id of the PROLOG process. These values may be stored and used later as optional parameters of the other functions, if more than one PROLOG process is used.

- `(start-motel)` has the same effect as `(start-prolog)`, except that it immediately consults MOTEL. It returns the same three values as `(start-prolog)`.
- `(reset-motel &optional` *i e p*`)` resets and / or stops the PROLOG process. Of course this can be done only if `(prolog-goal)` was called using the multitasking features of LUCID LISP of if the lisp process was interrupted before.
- `(kill-prolog &optional` *i e p*`)` kills the last by `(start-prolog)` or `(start-motel)` invoked PROLOG process. If the optional parameters i, e, p (that are given from start-prolog or start-motel) are specified, the corresponding process is killed.

D.3 The function `(prolog-goal).`

`prolog-goal` ({*prolog-goal-expression*}* `&optional i e p )`

`prolog-goal` takes the given list of PROLOG goals (in lisp-like syntax as given above) and converts them into PROLOG syntax. These goals are send then to the PROLOG process (if the optional parameters are specified, then the corresponding process is used), seperated by commas. The first return value is a (possibly empty) string with the output from the PROLOG process, the second return value is on of `´last, nil or t`: When PROLOG returns `yes`, `prolog-goal` returns `´last`. When PROLOG returns `no`, `prolog-goal` returns `nil`. When PROLOG returns variable bindings, these bindings are converted to lisp syntax and bound to the appropriate lisp variable. In this case `t` is returned.

D.4 The function `(prolog-next).`

`prolog-next (& optional` *i e p*`)` gets the next answer (if there are more than one) from PROLOG, and treats the result as `prolog-goal` does. It returns `nil` if this was the last answer and PROLOG returned `no ´last`, if it was the last answer and PROLOG returned `yes` and `t` otherwise.

D.5 The macro `(do-prolog)`

`do-prolog` ({*prolog-goal-expression*}*)
({(*var* [*init* [*step*]])}*)
(*end-test* {*result*}*)
{*declaration*}* {*tag* | *statement*}*

This macro works in the same way as the lisp DO macro. The goals are given in a list as in `prolog-goal`, The variables are lisp symbols prefixed with ?. The rest works like the do macro: The macro calls `prolog-goal` and `prolog-next` in each loop and binds the variables accordingly.

D.6 The macro `(do-prolog-with-streams)`

In order to use the do-prolog macro (see above) with a PROLOG process different from the last recently created, you have to call `(do-prolog-with-stream` *i e p (do-prolog . . .)*`)`.

Appendix E

Installing MOTEL

E.1 Requirements

You need one of the following PROLOG systems to use MOTEL:

- Quintus Prolog 3.1.1
- SICStus Prolog 2.1 Patch level 5 – Patch level 7
- SWI-Prolog (Version 1.6.10)
- ECRC Common Logic Programming System (Version 3.2.2)

The interface between Lisp and Prolog is only available for Lucid Common Lisp and SICStus Prolog.

E.2 Installation

The MOTEL distribution contains one compressed tar file, which includes the MOTEL system. To install the system on a SUN-4 (SunOS 4.1.x) execute the following steps:

Uncompress the compressed tar file

`prompt(1)%` **uncompress motel.tar.Z**

Extract the source file and documentation file from the tar file

`prompt(2)%` **tar xvf motel.tar**

This results in the files `README`, `int.c`,`int.o`,`int.pl`, `motel.lisp`, `motel.pl`, `motel.dvi`, and `hn.dvi`. The file `README` gives a brief description how the system can be used, the file `motel.dvi` is the the user manual for the MOTEL, `hn.dvi` gives an introduction to modal terminological logics. The file `motel.pl` is the MOTEL source file, the files `motel.lisp`,`int.pl`, and `int.o` contain the code for the interface between Lucid Common Lisp and SICStus Prolog.

After starting your PROLOG system you have to consult the source file.

```
prompt(3)% sicstus
SICStus 2.1 #5 : Tue Jul 21 16:16:49 MET DST 1992
| ?- consult(motel).
{consulting motel.pl...}
{motel.pl consulted, 5600 msec 329168 bytes}
yes
| ?-
```

Now you can work with the MOTEL system as described in the previous chapters.

To use the interface between Lucid Common Lisp and SICStus Prolog, you have to modify the file `motel.lisp`. At the beginning it contains three `setq`-commands:

```
(setq *consult-motel-string* "['/usr/local/motel/motel.pl'].")
(setq *prolog-executable* "/usr/local/sicstus2.1/sicstus")
(setq *int_dot_pl* "/HG/hiwis/timm/lucid/int.pl")
```

You should replace `/usr/local/motel/motel.pl` with the filename of your installation of the `motel.pl` file. Furthermore you should replace `/usr/local/sicstus2.1/sicstus` with the filename of you PROLOG system. The variable `*int_dot_pl*` contains the location of the file `int.pl` included in the distribution.

Now you can load this file after you have started Lucid Common Lisp:

```
prompt(3)% lucid
;;; Lucid Common Lisp/SPARC
;;; Application Environment Version 4.0.0, 6 July 1990
;;; Copyright (C) 1985, 1986, 1987, 1988, 1989, 1990, 1991 by Lucid, Inc.
;;; All Rights Reserved
;;;
;;; This software product contains confidential and trade secret information
;;; belonging to Lucid, Inc.  It may not be copied for any reason other than
;;; for archival and backup purposes.
;;;
;;; Lucid and Lucid Common Lisp are trademarks of Lucid, Inc.  Other brand
;;; or product names are trademarks or registered trademarks of their
;;; respective holders.

 > (load "motel.lisp")
;;; Loading source file "motel.lisp"
;;; Warning: File "motel.lisp" does not begin with IN-PACKAGE.
    Loading into package "USER"
#P"/usr/local/motel/src/motel/motel.lisp"
 >
```

Then you are able to work with the interface between Lucid Lisp and SICStus Prolog as described in chapter D.

References

Franz Baader and Bernard Hollunder, 1990. *KRIS*: Knowledge Representation and Inference System — System Description —. Technical Memo DFKI-TM-90-03, Deutsches Forschunszentrum für Künstliche Intelligenz.

Andreas Nonnengart, 1992. First-Order Modal Logic Theorem Proving and Standard PROLOG. Internal report MPI-I-92-228, Max-Planck-Institute for Computer Science.

Index