MOTEL User Manual
Version 0.8.8 (April 1995)

Ullrich Hustadt
Andreas Nonnengart
Renate Schmidt
Jan Timm

MPI-I-92-236 September 1994
Author’s Address

Ullrich Hustadt
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
E-mail: Ullrich.Hustadt@mpi-sb.mpg.de

Andreas Nonnengart
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
E-mail: Andreas.Nonnengart@mpi-sb.mpg.de

Renate Schmidt
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 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.

We’d like to thank Ralph Schäfer and Ioan Alfred Letia for pointing out errors in previous version of this report.
Abstract

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

Whereas \textit{ALCVR} is a single-agent knowledge representation system, i.e. \textit{ALCVR} 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 reasoning 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
Contents

1 Reading the User Manual 3
   1.1 Predicate Descriptions .......................................................... 3
       1.1.1 Call Arguments .............................................................. 3
   1.2 Argument Types ................................................................. 4

2 Environments 5

3 Knowledge Representation 8
   3.1 Concept and Role Formation .................................................. 8
   3.2 Modal Terminological Axioms ................................................ 9
   3.3 Modal Assertional Axioms .................................................... 10
   3.4 Knowledge Revision ............................................................ 10
   3.5 Semantics ......................................................................... 11
   3.6 Modal Axioms ................................................................. 14
   3.7 Knowledge Bases .............................................................. 14

4 Classification 16
   4.1 Building the Semantic Network .............................................. 16
   4.2 Retrieval commands for concepts ........................................... 16
   4.3 Retrieval commands for roles .............................................. 19

5 Realization and Retrieval of objects 21

6 (In)consistency 24

7 Functional Dependencies 25
   7.1 Definition and Revision of Functional Dependencies .................... 25
   7.2 Deduction ................................................................. 26
   7.3 Abduction ................................................................. 28

8 Examples 30
   8.1 Modal Operators ............................................................. 30
   8.2 Role closure .............................................................. 30
   8.3 Abduction .............................................................. 31
   8.4 Defaults .............................................................. 32
Chapter 1

Reading the User Manual

1.1 Predicate Descriptions

Predicates are described according to the following grammar:

\[
\begin{align*}
<\text{Predicate Description}> & ::= <\text{CallPattern}> \\
\quad & \quad \text{Arguments}: <\text{ArgumentTypes}> \\
\quad & \quad <\text{Informal Description}> \\
<\text{CallPattern}> & ::= <\text{Predicate Name}> <\text{CallArguments}> \\
<\text{CallArguments}> & ::= \quad | ((<\text{CallArgument}>^+) ) \\
\quad & \quad \quad | ( <\text{CallArgument}>^+ ) \\
<\text{CallArgument}> & ::= [ <\text{Mode Annotation}> <\text{Meta Variable}> ], \\
\quad & \quad | [ <\text{Mode Annotation}> <\text{Meta Variable}> ] \\
\quad & \quad \quad | <\text{Mode Annotation}> <\text{Meta Variable}> \\
<\text{ArgumentTypes}> & ::= \quad | <\text{Meta Variable}> : <\text{Type}> \\
\quad & \quad \quad | <\text{ArgumentTypes}> \\
<\text{Predicate Name}> & ::= <\text{Identifier}> \\
<\text{Meta Variable}> & ::= <\text{Identifier}> \\
<\text{Type}> & ::= <\text{Informal Description}> \\
\end{align*}
\]

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

\[ ( <\text{CallArgument}>^+ ) \]

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

\[ ( <\text{CallArgument}>^+ ) \]
is used if the predicate has at least 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

\[ \langle \text{Mode Annotation} \rangle \langle \text{Meta Variable} \rangle, \]

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

\[ \langle \text{Mode Annotation} \rangle \langle \text{Meta Variable} \rangle \]

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

\[ \langle \text{Mode Annotation} \rangle \langle \text{Meta Variable} \rangle \]

denoting a non-optional argument.

The mode annotations are useful to tell whether 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:

\texttt{clearEnvironment/(\textit{EnvName})} \\
Argumen ts: \textit{EnvName} environment name  \\
removes the knowledge base in environment \textit{EnvName}. Without \textit{EnvName}\ the current environment is removed.

\texttt{compileEnvironment(\textit{FileName, EnvName})} \\
Argumen ts: \textit{FileName} file name \textit{EnvName} environment name  \\
loads the internal representation of an environment \textit{EnvName} in compiled form from a file named \textit{FileName}. If no \textit{EnvName} is given, the environment name stored in the file \textit{FileName} will be taken. If there already exists an environment \textit{EnvName}, it will be removed.

\texttt{copyEnvironment/(\textit{EnvName1, EnvName2})} \\
Argumen ts: \textit{EnvName1} environment name \textit{EnvName2} environment name  \\
creates a new environment \textit{EnvName2}\ and copies the knowledge base in \textit{EnvName1} to \textit{EnvName2}.

\texttt{environment/(\textit{EnvName, EnvId, Comment})} \\
Argumen ts: \textit{EnvName} environment name \textit{EnvId} internal environment name \textit{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 initialise. 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 environment 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.

$\text{switchToEnvironment}(+EnvName)$

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

\[
\begin{align*}
\text{and}([C_1, \ldots, C_k]) & \quad \text{(conjunction)} \\
\text{or}([C_1, \ldots, C_k]) & \quad \text{(disjunction)} \\
\text{not}(C) & \quad \text{(negation)} \\
\text{naf}(C) & \quad \text{(negation as failure)} \\
\text{all}(R,C) & \quad \text{(value restriction)} \\
\text{some}(R,C) & \quad \text{(exists restriction)} \\
\text{atleast}(n,R) & \quad \text{(number restriction)} \\
\text{atmost}(n,R) & \\
\text{b}(O,a,C) & \quad \text{(box agent introduction)} \\
\text{d}(O,a,C) & \quad \text{(diamond agent introduction)} \\
\text{bc}(O,C_1,C_2) & \quad \text{(box concept introduction)} \\
\text{dc}(O,C_1,C_2) & \quad \text{(diamond concept introduction)}
\end{align*}
\]

are modal concept terms and

\[
\begin{align*}
\text{and}([R_1, \ldots, R_l]) & \quad \text{(role conjunction)} \\
\text{inverse}(R) & \quad \text{(role inversion)} \\
\text{restr}(R,C) & \quad \text{(role restriction)}
\end{align*}
\]

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 \( MC \).

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

\[
\text{defprimconcept}(\llbracket EnvName, \llbracket M, \rrbracket + A) \\
\text{Arguments:} \quad EnvName \quad \text{environment name} \\
M \quad \text{modal context} \\
A \quad \text{concept name}
\]

\[
\text{defprimrole}(\llbracket EnvName, \llbracket M, \rrbracket + P) \\
\text{Arguments:} \quad EnvName \quad \text{environment name} \\
M \quad \text{modal context} \\
P \quad \text{role name}
\]

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

\[
\text{defprimconcept}(\llbracket EnvName, \llbracket M, \rrbracket + A, + C) \\
\text{Arguments:} \quad EnvName \quad \text{environment name} \\
M \quad \text{modal context} \\
A \quad \text{concept name} \\
C \quad \text{concept term}
\]

\[
\text{defprimrole}(\llbracket EnvName, \llbracket M, \rrbracket + P, + R) \\
\text{Arguments:} \quad EnvName \quad \text{environment name} \\
M \quad \text{modal context} \\
P \quad \text{role name} \\
R \quad \text{role term}
\]

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

\[
\text{defconcept}(\llbracket EnvName, \llbracket M, \rrbracket + A, + C) \\
\text{Arguments:} \quad EnvName \quad \text{environment name} \\
M \quad \text{modal context} \\
A \quad \text{concept name} \\
C \quad \text{concept term}
\]
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

def disjoint(EnvName, M, CL)

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

which declares the mutual 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.

undef concept(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**(lett EnvName, lett M, lett 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** ($\Sigma$-Structures)

As usual we define a $\Sigma$-structure as a pair $(D, I)$ which consists of a domain $D$ and an interpretation function $I$ which maps the individual objects to elements of $D$, primitive concepts to subsets of $D$ and the primitive roles to subsets of $D \times D$.

**Definition 2** (Frames and Interpretations)

By a frame $\mathcal{F}$ we understand any pair $(W, \mathcal{R})$ where
• $\mathcal{W}$ is a non-empty set (of worlds).
• $\mathcal{R} = \bigcup_{D \in \mathcal{M}, a \in A} \mathcal{R}_D^a$ where the $\mathcal{R}_D^a$'s are binary relation on $\mathcal{W}$, the so-called accessibility relations between worlds.

By a $\Sigma$-interpretation $\mathcal{I}$ based on $\mathcal{F}$ we understand any tuple $(\mathcal{D}, \mathcal{F}, \mathcal{I})$ where

• $\mathcal{D}$ denotes the common domain of all $\Sigma$-structures in the range of $\mathcal{I}_{\text{loc}}$.
• $\mathcal{F}$ is a frame
• $\mathcal{I}_{\text{loc}}$ maps worlds to $\Sigma$-structures with common domain $\mathcal{D}$ which interpret agents' names equally.

△

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

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

where $\mathcal{I}[\chi] = (\mathcal{D}, \mathcal{F}, \mathcal{I}_{\text{loc}}, \chi)$

△

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

\[
\begin{align*}
\mathcal{I} &\models \text{defprimconcept}(C_1, C_2) \quad \text{iff} \quad \mathcal{I}(C_1) = \mathcal{I}(C_2) \\
\mathcal{I} &\models \text{defprimconcept}(\text{b}(O,a) \mid M], C_1, C_2) \quad \text{iff} \quad \mathcal{I}[\chi] \models \text{defprimconcept}(M, C_1, C_2) \\
&\quad \text{for every } \chi \text{ with } \mathcal{R}_D^O(\epsilon, \chi) \\
\mathcal{I} &\models \text{defprimconcept}(\text{d}(O,a) \mid M], C_1, C_2) \quad \text{iff} \quad \mathcal{I}[\chi] \models \text{defprimconcept}(M, C_1, C_2) \\
&\quad \text{for every } \chi \text{ with } \mathcal{R}_D^O(\epsilon, \chi) \\
\mathcal{I} &\models \text{defprimconcept}(\text{d}(O, a) \mid M], C_1, C_2) \quad \text{iff} \quad \mathcal{I}[\chi] \models \text{defprimconcept}(M, C_1, C_2) \\
&\quad \text{for every } \chi \text{ with } \mathcal{R}_D^O(\epsilon, \chi) \\
\mathcal{I} &\models \text{defprimconcept}(\text{d}(O, a) \mid M], C_1, C_2) \quad \text{iff} \quad \mathcal{I}[\chi] \models \text{defprimconcept}(M, C_1, C_2) \\
&\quad \text{for every } \chi \text{ with } \mathcal{R}_D^O(\epsilon, \chi) \\
\end{align*}
\]

12
\[ \models \text{defconcept}(M, C_1, C_2) \quad \text{iff} \quad \models \text{defprimconcept}(M, C_1, C_2) \]

\[ \models \text{defprimrole}(R_1, R_2) \quad \text{iff} \quad \models (R_1) = \models (R_2) \]

\[ \models \text{defprimrole}([b(O,a) \mid M], R_1, R_2) \quad \text{iff} \quad \models [\chi] \models \text{defprimrole}(M, R_1, R_2) \]

\[ \text{for every } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{defprimrole}([bc(O,A) \mid M], R_1, R_2) \quad \text{iff} \quad \models [\chi] \models \text{defprimrole}(M, R_1, R_2) \]

\[ \text{for every } a \text{ with } \models \models a \in A, \]

\[ \text{for every } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{defrole}(M, R_1, R_2) \quad \text{iff} \quad \models \text{defprimrole}(M, R_1, R_2) \]

\[ \text{and} \models \text{defprimrole}(M, R_2, R_1) \]

\[ \models \text{assert}_{\text{ind}}(X, A) \quad \text{iff} \quad \models (X) \models \text{assert}_{\text{ind}}(A) \]

\[ \models \text{assert}_{\text{ind}}([b(O,a) \mid M], X, A) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, A) \]

\[ \text{for every } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{assert}_{\text{ind}}([bc(O,A) \mid M], X, A) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, A) \]

\[ \text{for every } a \text{ with } \models \models a \in A, \]

\[ \text{for every } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{assert}_{\text{ind}}([d(O,a) \mid M], X, A) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, A) \]

\[ \text{for some } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{assert}_{\text{ind}}([dc(O,A) \mid M], X, A) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, A) \]

\[ \text{for every } a \text{ with } \models \models a \in A, \]

\[ \text{for some } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{assert}_{\text{ind}}(X, Y, P) \quad \text{iff} \quad \models (X), \models (Y) \models \text{assert}_{\text{ind}}(P) \]

\[ \models \text{assert}_{\text{ind}}([b(O,a) \mid M], X, Y, P) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, Y, P) \]

\[ \text{for every } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{assert}_{\text{ind}}([bc(O,A) \mid M], X, Y, P) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, Y, P) \]

\[ \text{for every } a \text{ with } \models \models a \in A, \]

\[ \text{for every } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{assert}_{\text{ind}}([d(O,a) \mid M], X, Y, P) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, Y, P) \]

\[ \text{for every } a \text{ with } \models \models a \in A, \]

\[ \text{for some } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \models \text{assert}_{\text{ind}}([dc(O,A) \mid M], X, Y, P) \quad \text{iff} \quad \models [\chi] \models \text{assert}_{\text{ind}}(M, X, Y, P) \]

\[ \text{for every } a \text{ with } \models \models a \in A, \]

\[ \text{for some } \chi \text{ with } \mathcal{R}^0_{\chi}(\epsilon, \chi) \]

\[ \triangle \]

**Definition 5**

Let \( \models \) be an interpretation and let \( \Phi \) be a modal terminological or modal assertional axiom with \( \models \models \Phi \). Then we call \( \Phi \) **satisfiable** and we call \( \models \) a **model** for \( \Phi \). If all interpretations are models...
for $\Phi$ then we call $\Phi$ a theorem. Any axiom for which no model exists is called unsatisfiable. Thus, $\Phi$ is a theorem iff its negation is unsatisfiable.

## 3.6 Modal Axioms

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Axiom Schema</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>$b(O,a,C) \subseteq d(O,a,C)$</td>
<td>$\forall x \exists y \mathcal{R}_O^a(x,y)$</td>
</tr>
<tr>
<td>$t$</td>
<td>$b(O,a,C) \subseteq C$</td>
<td>$\forall x \mathcal{R}_O^a(x,x)$</td>
</tr>
<tr>
<td>$b$</td>
<td>$C \subseteq b(O,a,d(O,a,C))$</td>
<td>$\forall x, y \mathcal{R}_O^a(x,y) \Rightarrow \mathcal{R}_O^a(y,x)$</td>
</tr>
<tr>
<td>$4$</td>
<td>$b(O,a,C) \subseteq b(O,a,b(O,a,C))$</td>
<td>$\forall x, y, z \mathcal{R}_O^a(x,y) \land \mathcal{R}_O^a(y,z) \Rightarrow \mathcal{R}_O^a(x,z)$</td>
</tr>
<tr>
<td>$5$</td>
<td>$d(O,a,C) \subseteq b(O,a,d(O,a,C))$</td>
<td>$\forall x, y, z \mathcal{R}_O^a(x,y) \land \mathcal{R}_O^a(x,z) \Rightarrow \mathcal{R}_O^a(y,z)$</td>
</tr>
</tbody>
</table>

The user specifies the properties of the accessibility relation using the predicate modalAxioms. The properties of the accessibility relations have to be specified before modal operators are used in terminological axioms, assertional axioms, or queries. 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

14
saves the terminological, assertional, and modal axioms of the knowledge base in environment EnvName into the file FileName.

\texttt{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.

\texttt{getKB}(\texttt{\texttt{+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 $\text{subsumes}(\{\text{EnvName}, \ldots, C, D\})$

Arguments:  
- EnvName: environment name
- M: modal context
- C: concept name
- D: concept name

succeeds if $C$ and $D$ are known concepts in environment $\text{EnvName}$ 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 $\text{classify}(\{\text{EnvName}, \ldots, M\})$

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 semantic network:

$\text{showHierarchy}(\text{EnvName}, M, \text{Type})$

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 $\text{Type}$ is concepts and the role hierarchy in the modal context $M$ if $\text{Type}$ 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  
M  modal context  
Concept  concept name  
CL  list of concept names

which are 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

$Role$ 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.

\[
\text{deduce}(+-EnvName, +-M, \text{elementOf}(+-X, +-CT), +-Exp)
\]

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 either 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 \( M \).

\[
\text{realize}(+EnvName, +M, +X, -CL)
\]

Arguments:

- \( EnvName \): environment name
- \( M \): modal context
- \( X \): object name
- \( CL \): list of concept names
- \( Exp \): explanation

try it.

\[
\text{getAllObjects}(+EnvName, +M, -OL)
\]

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 predicates
deduce(\(-\)EnvName, \(-\)M, roleFiller(\(+\)X, \(+\)R, \(-\)L, \(-\)N), \(-\)Exp)
Arguments:   EnvName  environment name
             M         modal context
             X         object name
             R         role term
             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.

deduce(\(-\)EnvName, \(-\)M, roleConstraints(+X, \(+\)R, [\(\geq\), \(-\)A, \(\leq\), \(-\)B, \(-\)L, \(-\)N]), \(-\)Exp)
Arguments:   EnvName  environment name
             M         modal context
             X         object name
             R         role term
             A         number restriction
             B         number restriction
             L         list of object names
             N         number
             Exp       explanation

A is the minimal number of objects in the range of role R for argument X. If no restriction on the minimal number can be derived, A is instantiated with noMinRestriction. B is the maximal number of objects in the range of role R for argument X. If no restriction on the minimal number can be derived, B is instantiated with noMaxRestriction. L is the list of all objects in the range of role R for argument X and N is the number of elements in this list.

It is possible to use abduction to find a set of hypotheses, i.e. terminological axioms, such that some object X is an element of a concept C if these hypotheses 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 hypotheses
             Exp       explanation

For a given object name X all concepts C such that X is an instance of C using the additional set of hypotheses 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)`
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 \(\text{not}(\emptyset A)\). Otherwise the knowledge base is consistent.

\[
\text{consistent}(K \vdash EnvName, \vdash M) \]

Arguments: \(EnvName\) environment name
\(M\) modal context
succeeds if the environment \(EnvName\) and modal context \(M\) is consistent.

\[
\text{inconsistent}(K \vdash EnvName, \vdash M) \]

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 Functional 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(\(+X\))`

Where \(X\) 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.

\[
\text{def}(\{^\star \text{EnvName}\}, \{^\star \text{MS}\}, +\text{Fact})
\]

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.

\[
\text{undef}(\{^\star \text{EnvName}\}, \{^\star \text{MS}\}, +\text{Fact})
\]

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.

\texttt{showFD(}^{+ -}Env\texttt{)}

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

\texttt{showInf1(}^{+ -}Env\texttt{)}

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

\texttt{showChange(}^{+ -}Env\texttt{)}

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

\texttt{showFD(}^{+ -}Env\texttt{)}

Arguments: Env environment name (internal representation)
displays the user defined functional dependencies in the knowledge base. Similar to \texttt{showFDW}, but the default representation is chosen.

\section{7.2 Deduction}

\texttt{deduce(}^{+ EnvName}, [+ MS], + -Inf0, [- E])

Arguments: EnvName environment name
MS modal context
Inf0 a literal of the appropriate kind, see description below
E explanations (not as yet implemented)

Succeeds if Inf0 can be inferred by deduction. Here is a short description of Inf0 that can be inferred.

\texttt{inf1(}^{+ -X, + -Y, + -W}\texttt{)}

\texttt{X} attribute/role name
\texttt{Y} attribute/role name
\texttt{W} list of weights weight, a value
computes the cumulative weight \texttt{W} of all the influence links between the attributes \texttt{X} and \texttt{Y}.

\texttt{simultInf1(}^{+ -Xs, + -Y, + -W}\texttt{)}

\texttt{Xs} list of attributes/role names
\texttt{Y} attribute/role name
\texttt{W} list of weights weight, a value
checks if the list \texttt{Xs} is well-defined (that is, \texttt{Xs} a SET of independent attributes) and computes the total weight \texttt{W} of the attributes in the list \texttt{Xs} simultaneously influencing attribute \texttt{Y}.

\texttt{leastInf1(}^{+ -X, + -Y}\texttt{)}

\texttt{X} attribute/role name
\texttt{Y} attribute/role name
succeeds if \( X \) is a least attribute influencing \( Y \).

\[
\text{leastInfls}(+Xs, +Y)
\]

\( Xs \) list of attributes/role names
\( Y \) attribute/role name

collects the least attributes influencing \( Y \) in \( Xs \).

\[
\text{greatestInfl}(+X, +Y)
\]

\( X \) attribute/role name
\( Y \) attribute/role name

succeeds if \( Y \) is a greatest attribute influenced by \( X \).

\[
\text{greatestInfls}(+Xs, +Y)
\]

\( X \) attribute/role name
\( Ys \) list of attributes/role names

collects the greatest attributes influenced by \( X \) in \( Ys \).

\[
\text{maxPosInfl}(+-X, +-Y, +Wmax)
\]

\( X \) attribute/role name
\( Y \) attribute/role name
\( Wmax \) weight, a value

succeeds if \( Wmax \) is the greatest weight with which \( X \) influences \( Y \) positively.

\[
\text{maxNegInfl}(+-X, +-Y, +WMin)
\]

\( X \) attribute/role name
\( Y \) attribute/role name
\( WMin \) a value

succeeds if \( WMin \) is the greatest weight with which \( X \) influences \( Y \) negatively.

\[
\text{change}(+-Y, +-W)
\]

\( Y \) attribute/role name
\( W_y \) weight of change of \( Y \)

determines the change in \( Y \).

\[
\text{posInfl}(+-X, +-Y)
\]

\( X \) attribute/role name
\( Y \) attribute/role name

succeeds if attribute \( X \) influences attribute \( Y \) positively.

\[
\text{negInfl}(+-X, +-Y)
\]

\( X \) attribute/role name
\( Y \) attribute/role name

succeeds if attribute \( X \) influences attribute \( Y \) negatively.

\[
\text{noInfl}(+-X, +-Y)
\]

\( X \) attribute/role name
\( Y \) attribute/role name

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

\[
\text{simultPosInfl}(+Xs, +-Y)
\]

\( Xs \) list of attributes/role names
\( Y \) attribute/role name

27
succeeds if the simultaneous influence of the attributes in the list \( Xs \) on the attribute \( Y \) is positive.

\[ \text{simultNegInfl}(+-Xs, +-Y) \]
\[ Xs \] 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.

\[ \text{simultNoInfl}(+-Xs, +-Y) \]
\[ Xs \] 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.

\[ \text{increase}(+-X) \]
\[ Y \] attribute/role name
succeeds if attribute \( Y \) increases.

\[ \text{decrease}(+-X) \]
\[ Y \] attribute/role name
succeeds if attribute \( Y \) decreases.

\[ \text{noChange}(+-X) \]
\[ Y \] 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

\[ \text{abduce}(\{+\text{EnvName}\}, \{+\text{MS}\}, +-H, +-C, E) \]

where \( \text{EnvName} \) denotes an environment name, \( \text{MS} \) a modal context and \( E \) a list of explanations. \( H \) and \( C \) respectively 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.

\[ \text{abduce}(\{+\text{EnvName}\}, \{+\text{MS}\}, +-\text{change}(+-X, +-Wx), +-\text{change}(+-Y, +-Wy), []) \]

**Arguments:**

- \( \text{EnvName} \) environment name
- \( \text{MS} \) 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 \( \text{change}(+-X, +-Wx), \text{change}(+-Y, +-Wy) \) follows.
\textbf{abduce}(\texttt{EnvName}, \texttt{MS}, +Hypothesis, +Consequent, [])

Arguments:  
\begin{itemize}
\item \texttt{EnvName} environment name
\item \texttt{MS} modal context
\item \texttt{Hypothesis} a literal of appropriate kind
\item \texttt{Consequent} a literal of appropriate kind
\end{itemize}

Succeeds if \texttt{Consequent} follows under the hypothesis \texttt{Hypothesis}. \texttt{Hypothesis} and \texttt{Consequent} are of the form:

- \texttt{increase}($+X$)
- \texttt{decrease}($+X$)
- \texttt{noChange}($+X$)

\textbf{abduce}(\texttt{EnvName}, \texttt{MS}, +Changes, +\texttt{change}($+Y$, +WY), [])

Arguments:  
\begin{itemize}
\item \texttt{EnvName} environment name
\item \texttt{MS} modal context
\item \texttt{Changes} a list of literals of the form
\begin{itemize}
\item \texttt{change}($+X$, +W)
\end{itemize}
\item \texttt{Y} attribute/role name
\item \texttt{WY} weight of change of \texttt{Y}
\end{itemize}

Succeeds if \texttt{change}($+Y$, +W) follows under the hypotheses of \texttt{Changes}.

\textbf{abduce}(\texttt{EnvName}, \texttt{MS}, +Hypotheses, +Consequent, [])

Arguments:  
\begin{itemize}
\item \texttt{EnvName} environment name
\item \texttt{MS} modal context
\item \texttt{Hypotheses} a list of literals of the appropriate kind
\item \texttt{Consequent} a literal of the appropriate kind
\end{itemize}

Succeeds if \texttt{Consequent} follows under the hypotheses \texttt{Hypotheses}. \texttt{Hypotheses} is a list of

- \texttt{increase}($+X$)
- \texttt{decrease}($+X$)
- \texttt{noChange}($+X$)

\texttt{Consequent} is one of these literals.

\textbf{abduce}(\texttt{EnvName}, \texttt{MS}, +Change, +Changes, [])

Arguments:  
\begin{itemize}
\item \texttt{EnvName} environment name
\item \texttt{MS} modal context
\item \texttt{Change} a literal of the form
\begin{itemize}
\item \texttt{change}($+X$, +W)
\end{itemize}
\item \texttt{Changes} a list of literals of the form
\begin{itemize}
\item \texttt{change}($+X$, +W)
\end{itemize}
\end{itemize}

Succeeds if \texttt{Changes} hold under the hypothesis that \texttt{Change} holds.

\textbf{abduce}(\texttt{EnvName}, \texttt{MS}, +Hypothesis, +Consequents, [])

Arguments:  
\begin{itemize}
\item \texttt{EnvName} environment name
\item \texttt{MS} modal context
\item \texttt{Hypothesis} a literal of the form
\begin{itemize}
\item \texttt{increase}($+X$)
\item \texttt{decrease}($+X$)
\item \texttt{noChange}($+X$)
\end{itemize}
\end{itemize}

\texttt{Consequents} a list of literals of this form

Succeeds if \texttt{Consequents} follow under the hypothesis \texttt{Hypothesis}.
Chapter 8

Examples

8.1 Modal Operators

Let’s suppose that we have some agent $a_1$ in our world. We can form the concept containing everything that $a_1$ believes to be a car using the terminological axiom (2) in the following knowledge base. We call this concept $c_1$. Furthermore we specify that $a_1$ believes that $c_1$ is the concept containing everything he believes to be a car using axiom (4). And we assert that provability for the believe of $a_1$ is like the modal logic $kd45$.

That implies that $a_1$ is able to perform positive introspection, i.e. he believes what he believes. Suppose $audi$ is an element of $c_1$ (axiom (6)). If $c_3$ is the concept containing everything that $a_1$ believes to be an element of $c_1$ (axiom (3)) and $a_1$ believes that this equivalence is true, then $audi$ must be an element of $c_3$.

(1) modalAxioms($kd45$, believe, $a_1$).
(2) defconcept($c_1$, b(believe, $a_1$, auto)).
(3) defconcept($c_3$, b(believe, $a_1$, $c_1$)).
(4) defconcept([b(believe, $a_1$)], $c_1$, b(believe, $a_1$, auto)).
(5) defconcept([b(believe, $a_1$)], $c_3$, b(believe, $a_1$, $c_1$)).
(6) assert_ind($audi$, $c_1$).

We can check this using the query

```
| ?- deduce(elementOf($audi$, $c_3$)).
   yes
```

So the believes of $a_1$ 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 $tom$. 
def concept(onlyMaleChildren, all(child, male)).
assert ind(tom, peter, child).
assert ind(tom, chris, child).
assert ind(tom, tim, child).
assert ind(peter, male).
assert ind(chris, male).
assert ind(tim, male).
def closed(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).

assert ind(tom, betty, child)
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:

def prim concept(and([quaker, normalQuaker]), dove).
def prim concept(and([republican, normalRepublican]), hawk).
assert ind(nixon, quaker).
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 infer that *nixon* is a dove.

```
| ?- abduction(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(4),rn(J,G,F,E),modal(□),
                                 normalQuaker,nixon,hyp(J),ab(D),call(L),
                                 proved(in(□,normalQuaker,nixon),
                                 hyp(B),basedOn(A))))),
                H = [in(env(4),rn(P,O,N,M),modal(□),normalRepublican,nixon,hyp(J),ab(L),
                     call(K),proved(in(□,normalRepublican,nixon),hyp(J),basedOn(□)))]).
```

The PROLOG variable *H* is instantiated with the set hypotheses that we need to infer that *nixon* is a dove. Here we needed only one hypothesis, namely 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:

```
| ?- abduction(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(4),rn(J,G,F,E),modal(□),
                                 normalRepublican,nixon,hyp(J),ab(D),call(L),
                                 proved(in(□,normalRepublican,nixon),
                                 hyp(B),basedOn(A))))),
                H = [in(env(4),rn(P,O,N,M),modal(□),normalRepublican,nixon,hyp(J),ab(L),
                     call(K),proved(in(□,normalRepublican,nixon),hyp(J),basedOn(□)))]).
```

### 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))`.  

32
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,not(some(hasCol,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 whether 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

\texttt{ask(EnvName,M,elementOf(X,C,Exp))}

Arguments: \begin{itemize}
  \item \texttt{EnvName}: environment name
  \item \texttt{M}: modal context
  \item \texttt{X}: object name
  \item \texttt{C}: concept name
  \item \texttt{Exp}: explanation
\end{itemize}

A synonym for the \texttt{deduce} predicate described in chapter 5.

\texttt{ask(EnvName,M,roleFiller(X,R,L,N,Exp))}

Arguments: \begin{itemize}
  \item \texttt{EnvName}: environment name
  \item \texttt{M}: modal context
  \item \texttt{X}: object name
  \item \texttt{R}: role name
  \item \texttt{L}: list of object names
  \item \texttt{N}: number
  \item \texttt{Exp}: explanation
\end{itemize}

A synonym for the \texttt{deduce} predicate described in chapter 5.

\texttt{saveMOTEL(File Name)}

Arguments: \begin{itemize}
  \item \texttt{File Name}: file name
\end{itemize}

Saves the whole program state, containing all user defined predicates. The file \texttt{FileName} becomes an executable file.
Appendix B

SICStus 2.1 Specific Predicates

\text{ask}(+EnvName, +M, \text{elementOf}(+X, +C), +Exp)

Arguments: \begin{align*}
\text{EnvName} & \quad \text{environment name} \\
M & \quad \text{modal context} \\
X & \quad \text{object name} \\
C & \quad \text{concept name} \\
Exp & \quad \text{explanation}
\end{align*}

A synonym for the \text{deduce} predicate described in chapter 5.

\text{ask}(+EnvName, +M, \text{roleFiller}(+X, +R, -L, -N), -Exp)

Arguments: \begin{align*}
\text{EnvName} & \quad \text{environment name} \\
M & \quad \text{modal context} \\
X & \quad \text{object name} \\
R & \quad \text{role name} \\
L & \quad \text{list of object names} \\
N & \quad \text{number} \\
Exp & \quad \text{explanation}
\end{align*}

A synonym for the \text{deduce} predicate described in chapter 5.

\text{saveMOTEL}(+File\ Name)

Arguments: \begin{align*}
\text{File\ Name} & \quad \text{file name}
\end{align*}

Saves the whole program state, containing all user defined predicates. The file \text{File\ Name} becomes an executable file.
Appendix C

SB-LITTERS Interface

\texttt{sb\_defenv(+EnvName,+Comment)}

(\texttt{SB\_DEFENV EnvName COMMENT})

Arguments: \texttt{EnvName} environment name
\texttt{Comment} string

creates a new environment with identifier \texttt{EnvName} and associated comment \texttt{Comment}.

\texttt{sb\_initenv(+EnvName)}

(\texttt{SB\_INITENV EnvName})

Arguments: \texttt{EnvName} environment name
initializes environment \texttt{EnvName} or the current environment if no argument is given.

\texttt{sb\_primconcept(+EnvName,\{M\},+CName1,+SpecList)}

(\texttt{SB\_PRIMCONCEPT [ENVNAME] [:LIST \{B O A\} \{D O A\} \{BC O A\} \{DC O A\}]\*)

\texttt{CNAME1 SpecList}}

Arguments: \texttt{EnvName} environment name
\texttt{M} modal context
\texttt{CName1} concept name
\texttt{SpecList} SB-ONE concept specification

impose necessary conditions on the interpretation of \texttt{CName1} in environment \texttt{EnvName}
and modal context \texttt{M}. The conditions are specified by \texttt{SpecList}.

\texttt{sb\_defconcept(+EnvName,\{M\},+CName1,+SpecList)}

(\texttt{SB\_PRIMCONCEPT [ENVNAME] [:LIST \{B O A\} \{D O A\} \{BC O A\} \{DC O A\}]\*)

\texttt{CNAME1 SpecList}}

Arguments: \texttt{EnvName} environment name
\texttt{M} modal context
\texttt{CName1} concept name
\texttt{SpecList} SB-ONE concept specification

impose necessary and sufficient conditions on the interpretation of \texttt{CName1} in environment
\texttt{EnvName} and modal context \texttt{M}. The conditions are specified by \texttt{SpecList}.

\texttt{SpecList} is a list of SB-ONE concept specification elements having the following form:

- \texttt{supers([+C1,\ldots,+Cn])}

  (\texttt{SUPERS [:LIST C1 C2 \ldots Cn]})
spe/ies a concept whic h is the conjunction of \( C_1, \ldots, C_n \).

- **restrict_inh\((+RName_1, restricts(+RName_2, range(+CName_2, +CNameDef)))\)**

  (RESTRICT_INH RNAME\(_1\) (RESTRICTS RNAME\(_2\) (RANGE CNAME\(_2\) CNAMEDEF)))

  specifies a concept which is the domain of \( RName_1 \). \( RName_1 \) is the restriction of \( RName_2 \) to the range \( CName_2 \) and to the default range \( CNameDef \).

- **nr\((+RName_1, MinNr, +MaxNr, +DefNr)\)**

  (NR RNAME\(_1\) 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 \( RName_1 \).

sb_primelemrole\(([\text{EnvName},[+\text{MS},]+RName_1, +PrimRSpec})\)

(SB_PRIMELEMROLE [ENVNAME] [MS] RNAME\(_1\) (DOMAIN-RANGE CNAME\(_1\) CNAME\(_2\) CNAMEDEF))

Arguments:  
EnvName  environment name  
M  modal context  
RName\(_1\)  role name  
PrimRSpec  SB-ONE primitive role specification

impose necessary conditions on the interpretation of \( RName_1 \) in environment \( EnvName \) and modal context \( M \). The conditions are specified by \( PrimRSpec \). \( PrimRSpec \) takes the following form: \text{domain-range}(+CName\(_1\), +CName\(_2\), +CNameDef). This defines \( RName_1 \) to be a role with domain \( CName_1 \), range \( CName_2 \) and default range \( CNameDef \) in environment \( EnvName \) and modal context \( M \).

sb_defelemrole\(([\text{EnvName},][+\text{M},]+RName_1, +RSpec})\)

(SB_DEFELEMROLE [ENVNAME] [M] RNAME\(_1\) (RESTRICTS RNAME\(_2\) (RANGE CNAME\(_1\) CNAMEDEF)))

Arguments:  
EnvName  environment name  
M  modal context  
RName\(_1\)  role name  
RSpec  SB-ONE role specification

impose necessary and sufficient conditions on the interpretation of \( RName_1 \) in environment \( EnvName \) and modal context \( M \). The conditions are specified by \( RSpec \) which takes the form \text{restricts}(+RName\(_2\), range(+CName\(_1\), +CNameDef)). \( RName_1 \) is a maximal subset of the role \( RName_2 \) such that each role filler of \( RName_1 \) is in \( CName_1 \).

sb_disjoint\(([\text{EnvName},][+\text{M},]+CName_1, +CName_2})\)

(SB_DISJOINT [ENVNAME] [M] CNAME\(_1\) CNAME\(_2\))

Arguments:  
EnvName  environment name  
M  modal context  
CName\(_1\)  concept name  
CName\(_2\)  concept name

declares the concepts \( CName_1 \) and \( CName_2 \) to be disjoint.

sb_defelem\(([\text{EnvName},][+\text{M},]+ICName_1, +ISpecList})\)

specifies a concept which is the conjunction of \( C_1, \ldots, C_n \).
(SB_DEFLELEM [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

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

where IRSpec\textsubscript{i} is

\[
\text{ироле}(\text{+RName}_i, \text{name}(\text{+IRName}_i), \text{+IRList}_i) \\
(IROLE RNAMEi (INAME IRNAME) IRLIST)
\]

and the argument IRList\textsubscript{i} is a list which is either empty or contains either
\[
\text{nr}(\text{MinNr}_i, \text{MaxNr}_i, \text{DefNr}_i) \text{ (NR MINNRi MAXNRi DEFNRi)}, \text{vr}(\text{ICName}_i) \text{ (VR ICNAMEi)}, \text{or both.}
\]

The result of SB_DEFLELEM is the introduction of an object ICName\textsubscript{1} which is a member of CName and pairs (ICName\textsubscript{1}, ICName\textsubscript{i}) which are elements of IRName\textsubscript{i}. The role IRName\textsubscript{i} is a subset of RName\textsubscript{i} and has at least MinNr\textsubscript{i} role fillers and at most MaxNr\textsubscript{i} role fillers. The default number of role fillers is DefNr\textsubscript{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 either a
variable or a concept name. Again Exp provides some explanation why this is true.

SB_FACT(\[\# EnvName, \# M, \# irole(\# RName, \# IName1, \# IName2)\])

(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([\textit{EnvName}, \textit{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([\textit{EnvName}, \textit{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([\textit{EnvName}, \textit{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([\textit{EnvName}, \textit{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:
  \texttt{atomic(1)} gets \texttt{(atomic 1)}.

- Function arguments are separated by spaces:
  \texttt{defprimconcept(female, not(male))}
  gets \texttt{(defprimconcept female (not male))}.

- PROLOG variables have a '?' as first character, e.g. \texttt{?a} or \texttt{?x}.

- PROLOG lists get lisp lists with the keyword \texttt{:list} as the first element:
  \texttt{[male, female]} gets \texttt{(:list male female)}.

- An open PROLOG list is written as follows:
  \texttt{[a, b, c, d, e | \_]} gets \texttt{(:openlist (a b c d e) \_)}.

- To conserve PROLOG symbols with capital letter, they are escaped with `'` in lisp:
  - \texttt{makeEnvironment} gets \texttt{make\_environment},
  - \texttt{assert\_ind} gets \texttt{assert\_ind},
  - \texttt{make\_Env} gets \texttt{make\_env}.

- The existential quantifier is used as follows:
  - \texttt{E^ expression} gets \texttt{(\(?e\) expression)},
  - \texttt{D^ E^ expression} gets \texttt{(\(?d ?e\) 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 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.
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:

```lisp
(setq *consult-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 your 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:

```lisp
(load"motel.lisp")
```

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


# Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>abduce</td>
<td>20, 25, 26</td>
</tr>
<tr>
<td>Agent names</td>
<td>8</td>
</tr>
<tr>
<td>ask</td>
<td>31, 32</td>
</tr>
<tr>
<td>assert_ind</td>
<td>10</td>
</tr>
<tr>
<td>change</td>
<td>22, 24, 25</td>
</tr>
<tr>
<td>classify</td>
<td>14</td>
</tr>
<tr>
<td>clearEnvironment</td>
<td>6</td>
</tr>
<tr>
<td>compileEnvironment</td>
<td>6</td>
</tr>
<tr>
<td>Concept</td>
<td></td>
</tr>
<tr>
<td>top</td>
<td>8</td>
</tr>
<tr>
<td>Concept names</td>
<td>8</td>
</tr>
<tr>
<td>Conjunction</td>
<td>8</td>
</tr>
<tr>
<td>role</td>
<td>8</td>
</tr>
<tr>
<td>consistent</td>
<td>21</td>
</tr>
<tr>
<td>copyEnvironment</td>
<td>6</td>
</tr>
<tr>
<td>decrease</td>
<td>22, 24–26</td>
</tr>
<tr>
<td>deduce</td>
<td>19, 20, 23</td>
</tr>
<tr>
<td>def</td>
<td>22</td>
</tr>
<tr>
<td>defconcept</td>
<td>9</td>
</tr>
<tr>
<td>defdisjoint</td>
<td>9</td>
</tr>
<tr>
<td>defprimconcept</td>
<td>9</td>
</tr>
<tr>
<td>defprimrole</td>
<td>9</td>
</tr>
<tr>
<td>defrole</td>
<td>9</td>
</tr>
<tr>
<td>delete_ind</td>
<td>11</td>
</tr>
<tr>
<td>Disjunction</td>
<td>8</td>
</tr>
<tr>
<td>environment</td>
<td>6</td>
</tr>
<tr>
<td>Exists restriction</td>
<td>8</td>
</tr>
<tr>
<td>getAllFatherRoles</td>
<td>16</td>
</tr>
<tr>
<td>getAllObjects</td>
<td>19</td>
</tr>
<tr>
<td>getAllSonRoles</td>
<td>17</td>
</tr>
<tr>
<td>getAllSubConcepts</td>
<td>15</td>
</tr>
<tr>
<td>getAllSuperConcepts</td>
<td>15</td>
</tr>
<tr>
<td>getCommonFatherRoles</td>
<td>18</td>
</tr>
<tr>
<td>getCommonSonRoles</td>
<td>18</td>
</tr>
<tr>
<td>getCommonSubConcepts</td>
<td>16</td>
</tr>
<tr>
<td>getCommonSuperConcepts</td>
<td>16</td>
</tr>
<tr>
<td>getConcepts</td>
<td>15</td>
</tr>
<tr>
<td>getCurrentEnvironment</td>
<td>6</td>
</tr>
<tr>
<td>getDirectFatherRoles</td>
<td>16</td>
</tr>
<tr>
<td>getDirectSonRoles</td>
<td>17</td>
</tr>
<tr>
<td>getDirectSubConcepts</td>
<td>15</td>
</tr>
<tr>
<td>getDirectSuperConcepts</td>
<td>15</td>
</tr>
<tr>
<td>getHierarchy</td>
<td>14</td>
</tr>
<tr>
<td>getKB</td>
<td>13</td>
</tr>
<tr>
<td>getRoles</td>
<td>17</td>
</tr>
<tr>
<td>greatestInfl</td>
<td>23</td>
</tr>
<tr>
<td>greatestInfls</td>
<td>23</td>
</tr>
<tr>
<td>inconsistent</td>
<td>21</td>
</tr>
<tr>
<td>increase</td>
<td>22, 24–26</td>
</tr>
<tr>
<td>infl</td>
<td>22, 23</td>
</tr>
<tr>
<td>initEnvironment</td>
<td>7</td>
</tr>
<tr>
<td>Inversion</td>
<td></td>
</tr>
<tr>
<td>role</td>
<td>8</td>
</tr>
<tr>
<td>Knowledge signature</td>
<td>8</td>
</tr>
<tr>
<td>Kripke class</td>
<td>13</td>
</tr>
<tr>
<td>leastInfl</td>
<td>23</td>
</tr>
<tr>
<td>leastInfls</td>
<td>23</td>
</tr>
<tr>
<td>loadEnvironment</td>
<td>7</td>
</tr>
<tr>
<td>loadKB</td>
<td>13</td>
</tr>
<tr>
<td>makeEnvironment</td>
<td>7</td>
</tr>
<tr>
<td>maxNegInfl</td>
<td>24</td>
</tr>
<tr>
<td>maxPosInfl</td>
<td>24</td>
</tr>
<tr>
<td>Modal concept terms</td>
<td>8</td>
</tr>
<tr>
<td>Modal context</td>
<td>8</td>
</tr>
<tr>
<td>Modal operators</td>
<td>8</td>
</tr>
<tr>
<td>modalAxioms</td>
<td>13</td>
</tr>
<tr>
<td>Negation</td>
<td>8</td>
</tr>
</tbody>
</table>
Negation as failure, 8
negInfl, 22, 24
noChange, 24-26
noInfl, 22, 24
Number restriction, 8
Object names, 8
posInfl, 22, 24
realize, 19
removeEnvironment, 7
renameEnvironment, 7
Role names, 8
Role restriction, 8
Role terms, 8
saveEnvironment, 7
saveKB, 13
saveMOTEL, 31, 32
sb_ask, 35-38
sb_attributes, 35
sb_defconcept, 33
sb_defelem, 34
sb_defelemrole, 34
sb_defenv, 33
sb_disjoint, 34
sb_primconcept, 33
sb_primelemrole, 34
Semantic network, 14
setOption, 20
showChange, 23
showEnvironment, 7
showFD, 23
showFDW, 22
showHierarchy, 14
showInfl, 23
simultInfl, 23
simultNegInfl, 24
simultNoInfl, 24
simultPosInfl, 24
subsumes, 14
switchToEnvironment, 7
Terminological axioms, 8
modal, 8
testDirectFatherRole, 17
testDirectSonRole, 17
testDirectSubConcept, 15
testDirectSuperConcept, 15
testFatherRole, 17
testSonRole, 17
testSubConcept, 16
testSuperConcept, 16
Top concept, 8
undef, 22
undefconcept, 10
undefprimconcept, 10
Value restriction, 8
Below you find a list of the most recent technical reports of the research group *Logic of Programming* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server ftp.mpi-sb.mpg.de under the directory pub/papers/reports. Most of the reports are also accessible via WWW using the URL http://www.mpi-sb.mpg.de. If you have any questions concerning ftp or WWW access, please contact reports@mpi-sb.mpg.de. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik  
Library  
attn. Regina Kraemer  
Im Stadtwald  
D-66123 Saarbrücken  
GERMANY  
e-mail: kraemer@mpi-sb.mpg.de

<table>
<thead>
<tr>
<th>Report ID</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI-I-95-2-007</td>
<td>A. Nommenart, A. Szalas</td>
<td>A Fixpoint Approach to Second-Order Quantifier Elimination with Applications to Correspondence Theory</td>
</tr>
<tr>
<td>MPI-I-95-2-005</td>
<td>F. Baader, H. J. Ohlbach</td>
<td>A Multi-Dimensional Terminological Knowledge Representation Language</td>
</tr>
<tr>
<td>MPI-I-95-2-003</td>
<td>P. Barth</td>
<td>A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization</td>
</tr>
<tr>
<td>MPI-I-95-2-001</td>
<td>S. Vorobyov</td>
<td>Proof normalization and subject reduction in extensions of Fsub</td>
</tr>
<tr>
<td>MPI-I-94-261</td>
<td>P. Barth, A. Bockmayr</td>
<td>Finite Domain and Cutting Plane Techniques in CLP((\mathcal{PB}))</td>
</tr>
<tr>
<td>MPI-I-94-257</td>
<td>S. Vorobyov</td>
<td>Structural Decidable Extensions of Bounded Quantification</td>
</tr>
<tr>
<td>MPI-I-94-254</td>
<td>P. Madden</td>
<td>Report and abstract not published</td>
</tr>
<tr>
<td>MPI-I-94-252</td>
<td>P. Madden</td>
<td>A Survey of Program Transformation With Special Reference to <em>Unfold/Fold</em> Style Program Development</td>
</tr>
<tr>
<td>MPI-I-94-251</td>
<td>P. Graf</td>
<td>Substitution Tree Indexing</td>
</tr>
<tr>
<td>MPI-I-94-246</td>
<td>M. Hanus</td>
<td>On Extra Variables in <em>(Equational)</em> Logic Programming</td>
</tr>
<tr>
<td>MPI-I-94-240</td>
<td>P. Madden</td>
<td>Recursive Program Optimization Through Inductive Synthesis Proof Transformation</td>
</tr>
<tr>
<td>MPI-I-94-239</td>
<td>P. Madden, I. Green</td>
<td>A General Technique for Automatically Optimizing Programs Through the Use of Proof Plans</td>
</tr>
<tr>
<td>MPI-I-94-238</td>
<td>P. Madden</td>
<td>Formal Methods for Automated Program Improvement</td>
</tr>
<tr>
<td>MPI-I-94-235</td>
<td>D. A. Plaisted</td>
<td>Ordered Semantic Hyper-Linking</td>
</tr>
<tr>
<td>MPI-I-94-234</td>
<td>S. Matthews, A. K. Simpson</td>
<td>Reflection using the derivability conditions</td>
</tr>
<tr>
<td>MPI-I-94-233</td>
<td>D. A. Plaisted</td>
<td>The Search Efficiency of Theorem Proving Strategies: An Analytical Comparison</td>
</tr>
</tbody>
</table>