Implementation of a cut-free sequent calculus for logics with adjoint modalities

Jael Kriener, Mehrnoosh Sadrzadeh & Roy Dyckhoff

Abstract

Sadrzadeh and Dyckhoff describe in [1] a cut-free sequent calculus for logics with adjoint pairs of modal operators. We give here a Prolog implementation of a decision procedure for this calculus and describe the simple mechanism for loop checking used to guarantee termination, which requires slight modification of some of the inference rules of the calculus.

1 Introduction

The second and third author introduce in [1] a cut-free sequent calculus for positive logic with adjoint mnodalities. It describes a minimal modal logic which is useful for reasoning about information in interactive multi-agent systems. We present here a Prolog implementation of an automated decision procedure for this calculus.

In section 2 we give as background a brief description of the inference rules of the calculus introduced in [1] and outline the implementation difficulty posed by the fact that it is non-terminating. We then describe in section 3 our strategy of loop prevention, which requires slight modification of some of the original inference rules. The new rules are presented and shown to be sound and complete with respect to the original calculus.

A Prolog implementation of the modified calculus is presented in section 4 and evaluated in section 5.

2 Sequent calculus for lattices with adjoint modalities

2.1 Description of the calculus

In the following we will outline roughly the calculus introduced by [1]. The notions of an *item* I and a *context* Γ are defined as follows:

$$\begin{aligned} I &::= m \mid \Gamma^A \\ \Gamma &::= I\, multiset, \end{aligned}$$

where A ranges over a set of *agents*, and *formulae* m are generated by the following grammar:

$$m ::= \bot \mid \top \mid p \mid m \wedge m \mid m \vee m \mid \Box_A\, m \mid \blacklozenge_A(m)$$

Put in intuitive language: An item is either a formula or a context labelled with an agent, while a context is a multi-set of items.

Further required for a description of the calculus are the notions of an *item-with-a-hole* J, of a *context-with-a-hole* Δ and of their application. The first two notions are defined as follows:

$$\begin{array}{rcl} J & ::= & [] \mid \Delta^A \\ \Delta & ::= & \Gamma, J \end{array}$$

Given a context-with-a-hole Δ and a context Γ, the result $\Delta[\Gamma]$ of applying the first to the second, i.e. replacing the hole [] in Δ by Γ, is a context, defined recursively (together with the application of an item-with-a-hole to a context) as follows:

$$\begin{array}{rcl} (\Gamma', J)[\Gamma] & = & \Gamma', J[\Gamma] \\ [][\Gamma] & = & \Gamma \\ \Delta^A[\Gamma] & = & \Delta[\Gamma]^A \end{array}$$

Given the notions of an item I, a context Γ, an item-with-a-hole J, a context-with-a-hole Δ and its application $\Delta[\Gamma]$, our calculus is described by the three forms of initial sequent and the nine inference rules shown below:

$$\frac{}{\Delta[\bot] \vdash m}\ \bot L \qquad \frac{}{\Gamma \vdash \top}\ \top R \qquad \frac{}{\Gamma, p \vdash p}\ Id \quad \textit{(p atomic)}$$

$$\frac{\Delta[m_1, m_2] \vdash m}{\Delta[m_1 \wedge m_2] \vdash m}\ \wedge L \qquad \frac{\Gamma \vdash m_1 \quad \Gamma \vdash m_2}{\Gamma \vdash m_1 \wedge m_2}\ \wedge R$$

$$\frac{\Delta[m_1] \vdash m \quad \Delta[m_2] \vdash m}{\Delta[m_1 \vee m_2] \vdash m}\ \vee L \qquad \frac{\Gamma \vdash m_1}{\Gamma \vdash m_1 \vee m_2}\ \vee R_1 \qquad \frac{\Gamma \vdash m_2}{\Gamma \vdash m_1 \vee m_2}\ \vee R_2$$

$$\frac{\Delta[m^A] \vdash m'}{\Delta[\blacklozenge_A(m)] \vdash m'}\ \blacklozenge_A L \qquad \frac{\Gamma \vdash m}{\Gamma', \Gamma^A \vdash \blacklozenge_A(m)}\ \blacklozenge_A R$$

$$\frac{\Delta[(\Box_A m, \Gamma)^A, m] \vdash m'}{\Delta[(\Box_A m, \Gamma)^A] \vdash m'}\ \Box_A L \qquad \frac{\Gamma^A \vdash m}{\Gamma \vdash \Box_A m}\ \Box_A R$$

2.2 The problem of termination

The calculus outlined above is non-terminating. That is, a root-first proof search applying the rules shown above is not guaranteed to terminate, but may run into loops. The reason for this is the $\Box_A L$-rule: it duplicates its principal item ($(\Box_A m, \Gamma)^A$) into the premiss unchanged and may therefore be applied several times on the same item at different stages during the proof search. These successive applications of the rule will expand the antecedent vacuously, that is increase its size while not adding new content, and thus produce a loop, that is lead to a situation where the new sequent to be proved has effectively occurred already as a conclusion on the same branch in the proof.

Depending on the order of application of the rules in a proof search, these loops may or may not be easily detected. However, the possibility of their occurrence entails that an automated decision procedure may not terminate. Some general way of spotting a loop in a proof is therefore required for the implementation of an automated decision procedure for this calculus.

3 Loop checking

3.1 Loop checking by accumulation in the context

Considering that the possibility of a vacuous expansion of the antecedent is what leads to the occurrence of loops within proof searches in this calculus, one can see that there is a simple mechanism for detecting these loops very early and preventing the procedure from following a path that will lead to a circle. One simply has to make sure that the antecedent is never extended vacuously, i.e. one has to check, whenever the application of a rule in the proof search will cause some item I to be added to the antecedent Γ to produce a new antecedent Γ', that I is not subsumed by Γ, and if that is the case go back and try to apply some different rule at the last stage.[1]

In order to do this test for subsumption effectively it is convenient to modify some of the inference rules of the calculus in such a way as to ensure that, in cases where some item I' is extracted from the antecedent Γ and then some different (but related) item I added to Γ, I' is not 'thrown away', but rather kept in Γ. That means that, in effect, all rules that expand the antecedent in some way will have to be made *cumulative* in the way that the $\Box_A L$-rule given above already is. It will then be easy to test whether I is already contained in Γ at the appropriate level, in which case it is certainly subsumed by Γ.

3.2 Modification of the original calculus

Of the rules shown above, the L-rules change the antecedent by adding new items to it and therefore need to be modified in the way described above. The new set of L-rules will be:

$$\frac{\Delta[m_1, m_2, m_1 \wedge m_2] \vdash m}{\Delta[m_1 \wedge m_2] \vdash m}\ \wedge L \qquad \frac{\Delta[m_1, m_1 \vee m_2] \vdash m \quad \Delta[m_2, m_1 \vee m_2] \vdash m}{\Delta[m_1 \vee m_2] \vdash m}\ \vee L$$

$$\frac{\Delta[m^A, \blacklozenge_A(m)] \vdash m'}{\Delta[\blacklozenge_A(m)] \vdash m'}\ \blacklozenge_A L \qquad \frac{\Delta[m, (\Box_A m, \Gamma)^A] \vdash m'}{\Delta[(\Box_A m, \Gamma)^A] \vdash m'}\ \Box_A L$$

Note that the $\Box_A L$-rule has not changed, as it already is cumulative in the required way.

3.3 Soundness and completeness of new rules

This new calculus is sound and complete with respect to the original one described in [1]. Soundness can be shown straightforwardly: every instance of one of the three modified rules can be translated into a combination of the corresponding original rule together with an instance of $Contraction$ (which is shown to by admissible in the old calculus in [1]). We will show the translation of the new $\wedge L$ rule as an example; other cases are parallel:

$$\cfrac{\cfrac{\Delta[m_1, m_2, m_1 \wedge m_2] \vdash m}{\Delta[m_1 \wedge m_2, m_1 \wedge m_2] \vdash m}\ \wedge L_{old}}{\Delta[m_1 \wedge m_2] \vdash m}\ Contr$$

[1]Strictly speaking this mechanism is hardly worth being called a method for loop checking; it is "loop-prevention" rather than "loop-detection".

Completeness depends on the admissibility of the following $Weakening$-rule in the new calculus:

$$\frac{\Delta[\Gamma] \vdash m}{\Delta[\Gamma, \Gamma'] \vdash m}\ Wk$$

The proof of this is a routine induction on the depth of the derivation with a case analysis of the rule used in the last step: in the cases where the last step was an instance of $\wedge R$, $\vee R1$, $\vee R2$, $\Box_A R$, $\wedge L$, $\vee L$, $\blacklozenge_A L$ and $\Box_A L$ the rule is straightforwardly admissible. The only interesting case is $\blacklozenge_A R$. Suppose the last step is by $\blacklozenge_A R$:

$$\frac{\Gamma^* \vdash m'}{\Gamma'', \Gamma^{*A} \vdash \blacklozenge_A(m')}\ \blacklozenge_A R$$

and $\Delta[\Gamma] = \Gamma'', \Gamma^{*A}$. To obtain $\Delta[\Gamma, \Gamma']$ from this there are two possibilities. In the first case, Γ occurs inside Γ^{*A}, and we make a routine use of the inductive hypothesis and reapply $\blacklozenge_A R$ with Γ'' unchanged. The second possibility is that Γ occurs inside Γ'', in which case we change Γ'' to an appropriate Γ''' and use the $\blacklozenge_A R$-rule with $\Gamma''', \Gamma^{*A} \vdash \blacklozenge_A(m')$ as the conclusion.

To show completeness of the new rules one can now give a translation of the original L-rules into a combination of the corresponding modified rule together with an instance of $Weakening$. We will again give the $\wedge L$ rule as an example:

$$\frac{\dfrac{\Delta[m_1, m_2] \vdash m}{\Delta[m_1, m_2, m_1 \wedge m_2] \vdash m}\ Wk}{\Delta[m_1 \wedge m_2] \vdash m}\ \wedge L_{new}$$

Other cases are similar.

4 Implementation in Prolog

4.1 Introduction

The calculus described above with the modified rules shown in section 3.2 is implemented using gprolog in the XGP - programming environment. We will present the code including comments and add further explanations where the code is not self-explanatory.

4.2 Syntax

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  SYNTAX
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op( 450, yfx, v). % disjunction
:- op( 450, yfx, &). % conjunction
:- op( 500, xfx, ^). % label

/*
% Formulae are of the form: atom(p), atom(q), true, false,
m v n, m & n, box( anne, m), bdia( paul, n), where m, n are formulae.
*/
formula( M v N , M1 v N1 ) :-
    !,
    formula( M, M1 ),
    formula( N, N1 ).
formula( M & N, M1 & N1 ) :-
    !,
    formula( M, M1 ),
    formula( N, N1 ).
formula( box(A, M ), box( A, M1 )) :-
    !,
    formula( M, M1 ).
formula( bdia( A, M ), bdia( A, M1 ) ) :-
    !,
    formula( M, M1 ).
formula( true, true ) :- !.
formula( false, false ) :- !.
formula( M, atom(M) ).
```

4.3 List Utilities

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  LIST UTILITIES - RD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% myselect(List, Element, Remainder)
myselect( [X | L], X,  L).
myselect( [Y | L], X,  [Y | M]) :-
    myselect(L,X, M).

on( A, [ A | _ ] ).
on( A, [ _ | Rest] ) :-
    on(A, Rest).

/*
append( [], L, L ).
append( [ H | T], L, [ H | TL] ) :-
    append(T, L, TL).

*/
```

These utilites are taken from [2]. Note that some versions of Prolog, including the one we are using, have the *append* predicate built in, while others do not. For that reason we have added it in a comment.

4.4 Type Definitions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   TYPE DEFINITIONS - ( item, context, iwh, cwh )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*  An item is either a formula (first clause - 1 parameter),
    or a context, labelled with an agent name (second clause - 2 parameters)
*/
item( F ) :-
    formula( F ).
item( Gamma^ _Agent ) :-
    context( Gamma ).

/*  A context is a multiset of items, ie:
    either a list containing only a single item (first clause),
    or a list, the head of which is an item and the tail a context.
```

```
*/
context( [] ).
context( [I | Gamma] ) :-
    item( I ),
    context( Gamma ).

/*  An item-with-a-hole (iwh) is either a hole (first clause),
    or a cwh, labelled with an agent name (second clause)
*/
iwh( hole ).
iwh( Delta^ _Agent ) :-
    cwh( Delta ).

/*  A context-with-a-hole (cwh) is a multiset of items, one of which is an iwh, ie:
    or a list, the head of which is an iwh, the tail is a context
    or a list, the head of which is an item, and the tail a cwh
*/
cwh( [J | Gamma] ) :-
    iwh( J ),
    context( Gamma ).
cwh( [ I | Delta] ) :-
    item( I ),
    cwh( Delta ).
```

4.5 Match

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  MATCH - (matchI, matchC)
%  an item is represented as Context^ Agent, a context is represented as a list
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* matchI( Item I, Item Pattern IP,
           Item-with-a-hole J,
           Context Acc1, Context Acc2,
           Context Gamma_Rest )
    given Item I, Item Pattern IP,
    returns Item-with-a-hole J, such that J[ IP ] = I,
    and Context Gamma_Rest, such that Gamma_Rest contains all those
    formulae that can be found in Delta on the same level (ie in the
    same list) as the hole
    Acc1 and Acc2 are accumulators
*/
matchI( I, I, hole, Acc1, Acc2, Gamma_Rest) :-
     % I matches any kind of item
     !,
     % two accumulators: Acc1 is items already seen and Acc2 is those yet to come
     append(Acc1, Acc2, Gamma_Rest).

matchI( Gamma^ Agent, IP, Delta^ Agent, _, _, Gamma_Rest ) :-
     % discard accumulators; we need to go inside Gamma
     matchC( Gamma, IP, Delta, Gamma_Rest ).


/* matchC( Context, Item Pattern, Context-with-a-hole, Context )
    given Context Gamma, Item Pattern IP,
    returns Context-with-a-hole Delta, such that Delta[ [IP] ] = Gamma,
    and Context Gamma_Rest, such that Gamma_Rest contains all and only the
    formulae contained in the item which immediately contains the hole
```

```
*/
matchC(Gamma, IP, Delta, Rest) :-
matchC_acc(Gamma, IP, Delta,  [], Rest).

matchC_acc( [I | Gamma], IP, [J | Gamma], Acc, Gamma_Rest ) :-
matchI( I,      IP,   J,   Acc, Gamma, Gamma_Rest ).

matchC_acc( [I | Gamma], IP, [I | Delta], Acc, Gamma_Rest ) :-
matchC_acc( Gamma,  IP, Delta,  [I | Acc],  Gamma_Rest ).
```

Gamma_Rest is required for the loop detection. An example will show which formulae it consists of:
Suppose $matchC(Gamma, M_1\&M_2, Delta, Gamma\_Rest)$ is called with

$$Gamma = [a_1, a_2, a_3, [b_1, b_2, b_3, [c_1, c_2\&c_3, c_4, c_5]^{catherine}, b_4, b_5]^{bertrand}, a_4, a_5],$$

where a_n, b_n, c_n are formulae with main connectives other than '&', and the *matchC* predicate is used to extract $c_2\&c_3$, returning

$$Delta = [a_1, a_2, a_3, [b_1, b_2, b_3, [c_1, hole, c_4, c_5]^{catherine}, b_4, b_5]^{betrand}, a_4, a_5],$$

then *Gamma_Rest* should consist of the remaining formulae within the context labelled with *'catherine'*, that is

$$Gamma\_Rest = [c_1, c_4, c_5].$$

This is needed in order to prevent the generation of loops, because *Gamma_Rest* is the context that new items will be added to in the application of a L-rule, and this expansion will generate a loop if *Gamma_Rest* already contains all of these items (see below, section 4.7, for details.)

4.6 Apply

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  APPLY - (applyI, applyC)
%  an item is represented as Context^ Agent, a context is represented as a list
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* applyI( Item-with-a-hole, Context, Context ),
    replaces the hole within #1 by #2
    returns the resulting context (#3)
*/
applyI( hole, Gamma, Gamma ) :-
    !.
applyI( Delta^ A, Gamma, [ Gamma1^ A ] ) :-
    applyC( Delta, Gamma, Gamma1 ).

/* applyC( Context-with-a-hole, Context, Context )
    replaces the hole within #1 by #2,
    returns the resulting context (#3)
*/
applyC( [J | Rest], Gamma, Gamma2 ) :-
    iwh( J ),
    !,
    applyI( J,  Gamma, Gamma1 ),
    append( Gamma1, Rest, Gamma2).
applyC( [I | Delta], Gamma, [I | Gamma1] ) :-
    % not iwh( I ) **!**
    applyC( Delta, Gamma, Gamma1 ).
```

4.7 Loop

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LOOP - ( Case, Item, Context )
% succeeds iff the application of the left rule corresponding to Case
% will produce a loop in the proof search
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

loop( conjunction, M1 & M2, Gamma ) :-
    on( M1, Gamma ),
    on( M2, Gamma ),
    !.

loop( disjunction, M1 v M2, Gamma ) :-
    (    on( M1, Gamma )
    ;
         on( M2, Gamma )
    ),
    !.

loop( diamond, bdia(A, M ), Gamma ) :-
    on( [ M ] ^ A, Gamma ),
    !.

loop( box, M, Gamma ) :-
    on( M, Gamma ),
    !.
```

There are four clauses of the loop-predicate corresponding to the four ways in which the antecedent can be vacuously expanded. In each case, what is checked is whether the item that will be newly added to the antecedent if the rule is applied is already in the context that it will be added to. That context is $Gamma$ and it is contained in some item on some level within the antecedent:

Application of the $\wedge L$-rule will generate a loop if both conjuncts ($M1$ and $M2$) are already in $Gamma$.

Application of the $\vee L$-rule will generate a loop if either one or both disjuncts ($M1$ or $M2$) are already in $Gamma$.

Application of the $\blacklozenge_A L$-rule will generate a loop if $[M]^A$ already is in $Gamma$.

Application of the $\Box_A L$-rule will generate a loop if M is already in $Gamma$.

4.8 Derivable

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DERIVABLE - derivable( context, formula )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initial Sequents  %
% 'true' is derivable from anything
derivable( _, true ).

% EFQ
derivable( Gamma, _ ) :-
    matchC( Gamma, false, _, _ ).

derivable( Gamma, M ) :-
    on( M, Gamma).


% box_A R - invertible, non-branching
derivable( Gamma, box(A, M )) :-
    !,  % invertible
    derivable( [ Gamma ^ A ], M ).

% &L - invertible, non-branching
derivable( Gamma, M ) :-
    matchC( Gamma, M1 & M2, Delta, Gamma_Rest ),
    \+ loop( conjunction, M1 & M2, Gamma_Rest ),
    !,  % invertible
    (   on( M1, Gamma_Rest ),
        applyC( Delta, [M2, M1 & M2], Gamma1 )
    ;   % M1 not on Gamma
    (   on( M2, Gamma_Rest ),
            applyC( Delta, [M1, M1 & M2], Gamma1 )
        ;
            applyC( Delta, [M1, M2, M1 & M2], Gamma1 )
        )
    ),
    derivable( Gamma1, M ).

% bdia_A L - invertible, non-branching
derivable( Gamma, M1 ) :-
    matchC( Gamma, bdia(A, M ), Delta, Gamma_Rest ),
    \+ loop( diamond, bdia(A, M ), Gamma_Rest ),
    !,   % invertible
    applyC( Delta, [[ M ]^ A,  bdia(A, M )], Gamma1 ),
    derivable( Gamma1, M1 ).

% box_A L - invertible, non-branching
derivable( Gamma, M1 ) :-
    matchC( Gamma, GammaX ^ A, Delta, Gamma_Rest ),
    on( box(A, M ), GammaX ),
    \+ loop( box, M, Gamma_Rest ),
    !,  % invertible
    applyC( Delta, [M, GammaX ^ A], Gamma1 ),
    derivable( Gamma1, M1 ).
```

```
% vL - invertible, branching
derivable( Gamma, M ) :-
    matchC( Gamma, M1 v M2, Delta, Gamma_Rest ),
    \+ loop( disjunction, M1 v M2, Gamma_Rest ),
    !,  % invertible
    applyC( Delta, [M1, M1 v M2], Gamma1 ),
    derivable( Gamma1, M ),
    !,
    applyC( Delta, [M2, M1 v M2], Gamma2 ),
    derivable( Gamma2, M ).

% &R - invertible, branching
derivable( Gamma, M1 & M2 ) :-
    !,  % invertible
    derivable( Gamma, M1 ),
    derivable( Gamma, M2).

% vR (1) - non-invertible, non-branching
derivable( Gamma, M v _ )  :-
    derivable( Gamma, M ).

% vR (2) - non- invertible, non-branching
derivable( Gamma, _ v M )  :-
    derivable( Gamma, M ).

% bdia_A R - non-invertible, non-branching
derivable( Gamma1, bdia(A, M ) ) :-
    myselect( Gamma1, Gamma ^ A, _ ),
    derivable( Gamma, M ).
```

5 Inefficiencies and possible improvements

The loop-predicate is supposed to be testing whether certain items are subsumed by a context Γ. The way it does that in this implementation is simply to check whether the relevant item can be found in Γ. While this method effectively prevents loops, it is not an efficient way to test for subsumption; $[m_1 \wedge m_2]$ subsumes m_1, but the loop-predicate will not detect that, because the item $m_1 \wedge m_2$ is different from the item m_1. A more efficient version of this predicate should implement a test for actual subsumption rather than just occurrence.

6 Extensions

[1] (for which see details) extends the above framework with a mechanism for adding assumptions, allowing facts about particular scenarios (like the muddy children scenario) to be assumed. The above code is extended at appropriate places with

```
loop( assumption, RHS, Gamma ) :-
    on( RHS, Gamma ),
    !.


% Assn - invertible, non-branching
derivable( Gamma, M ) :-
```

```
    assumption( bdia(Agent, P), RHS),
    matchC( Gamma, Gamma1 ^ Agent, Delta, Gamma_Rest ),
    on(P, Gamma1),
    \+ loop( assumption, RHS, Gamma_Rest),
    !,
    applyC( Delta,[RHS, Gamma1 ^ Agent], Gamma2),
    derivable( Gamma2, M ).


% Some sample assumptions
assumption( bdia(2, atom(s123)), atom(s123) ).
assumption( bdia(3, atom(s123)), atom(s123) ).
```

7 Usage

```
example(1, [M01], M1) :-
    M0 = bdia(a, box(a, (m v n))),
    M = (m & bdia(a, box(a, m v n))) v (n & bdia(a, box(a, m v n))),
    formula(M0,M01),
    formula( M, M1).

example(2, [M01], M1) :-
    M0 = bdia(a, box(a, (m v p))),
    M = (m & bdia(a, box(a, m v n))) v (n & bdia(a, box(a, m v n))),
    formula(M0,M01),
    formula( M, M1).

example(3, [M01], M1 ) :-
    formula( s123, M01),
    formula( box(2, s123), M1).

example(4, [M01], M1 ) :-
    formula( s123, M01),
    formula( box(2, box(3, s123)), M1).

example(5, [M01], M1 ) :-
    formula( s123, M01),
    formula( box(2, s123) & box(2, box(3, s123)), M1).

test(N) :-
    example(N, Gm, M), derivable(Gm, M).
```

Tests 1, 3 and 4 should succeed (given the sample assumptions above); test 2 should fail.

References

[1] M. Sadrzadeh and R. Dyckhoff, *Positive Logic with Adjoint Modalities: Proof Theory, Semantics and Reasoning about Information*, Proceedings of MFPS 2009, to appear.

[2] R. Dyckhoff, *Intuitionistic theorem prover*, 1991, http://www.cs.st-andrews.ac.uk/ rd/logic/nonmac/LJT.pl.html