```
%% This is an experimental version of a pure Prolog counterpart of verifier.tlp.
%%
%% The approach is to visit each node at most once, and rewrite the expression
%% to take into account the valuation of propositions in that node.
%% The cost is O( number of nodes ) * O( length of formula ).
%% We don't yet have a proof of correctness, but all the examples work.
%%
%% Written by Feliks Kluzniak at UTD (March 2009).
%% Last update: 24 April 2009.


:- [ 'operators.pl' ].
:- [ 'normalize.pl' ].
:- [ 'looping_prefix.pl' ].
:- [ 'consistency_checker.pl' ].
:- [ '../../../general/higher_order.pl' ].

:- ensure_loaded( library( lists ) ).  % Sicstus, reverse/2.
```

```
%% Check whether the state satisfies the formula.
%% This is done by checking that it does not satisfy the formula’s negation.
%% (We have to apply the conditional, because our tabling interpreter does not
%%  support the cut, and we don’t yet support negation for coinduction.)

check( State, Formula ) :-
        check_consistency,
        (
            state( State )
        ->
            true
        ;
            write( ’\"’ ),
            write( State ),
            write( ’\" is not a state’ ),
            nl,
            fail
        ),
        write( ’Query for state ’ ),
        write( State ),
        write( ’: ’ ),
        write( Formula ),
        nl,
        once( normalize( ˜ Formula, NormalizedNegationOfFormula ) ),
        write( ’(Negated and normalized: ’ ),
        write( NormalizedNegationOfFormula ),
        write( ’)’ ),
        nl,
        (
            once( verify( State, NormalizedNegationOfFormula, [] ) )
        ->
            fail
        ;
            true
        ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% verify( + state, + formula, + path ) :
%% Verify whether the formula holds for this state (which we reached by this
%% path).
%% (The formula is our negated thesis, so we are looking for one path.)

verify( S, F, Path ) :-
        rewrite( F, S, NF ),
        (
            NF = true
        ->
            show_path( Path )
        ;
            NF = false
        ->
            fail
        ;
            (
                member( pair( S, F ), Path )
            ->
                (
                    disjunct( g _, F )
                ->
                    show_path( Path )
                ;
                    fail
                )
            ;
                once( strip_off_x( NF, NNF ) ),
                trans( S, NS ),
                verify( NS, NNF, [ pair( S, F ) | Path ] )
            )
        ).


%
show_path( Path ) :-
        write( 'COUNTEREXAMPLE: ' ),
        reverse( Path, RevPath ),
        map( first, RevPath, TruePath ),
        write( TruePath ),
        nl.

%
first( pair( State, _ ), State ).


%% disjunct( +- disjunct, + formula ):
%% Like member, only of an outermost disjunction rather than a list.

disjunct( A, A v _ ).
disjunct( A, _ v B ) :-  disjunct( A, B ).
disjunct( A, A     ).
```

```
%% strip_off_x( + formula, - formula ):
%% Strip off the "x" operator from every disjunct, raise an alarm and abort if
%% there are disjuncts that are not so wrapped.

strip_off_x( x A v B, A v NB ) :-  strip_off_x( B, NB ).

strip_off_x( x F, F ).

strip_off_x( F, _ ) :-
        F \= x F,
        write( 'Formula not in X : ' ),
        write( F ),
        nl,
        abort.
```

```
%% rewrite( + formula, + state, - new formula ):
%% The formula has been normalized, so that negations are applied only to
%% propositions.

rewrite( F, S, NF ) :-
        once( r( F, S, NF ) ).

%
r( A v B, S, NF  ) :-  r( A, S, NA ),  r( B, S, NB ),
                       simplify( NA v NB, NF ).

r( A ^ B, S, NF  ) :-  r( A, S, NA ),  r( B, S, NB ),
                       simplify( NA ^ NB, NF ).

r( x A  , _, x A ).

r( f A  , S, NF  ) :-  r( A, S, NA ),
                       simplify( NA v x f A, NF ).

r( g A  , S, NF  ) :-  r( A, S, NA ),
                       simplify( NA ^ x g A, NF ).

r( A u B, S, NF  ) :-  r( A, S, NA ),  r( B, S, NB ),
                       simplify( NA ^ x (A u B), Conj ),
                       simplify( NB v Conj, NF ).

r( A r B, S, NF  ) :-  r( A, S, NA ),  r( B, S, NB ),
                       simplify( NB ^ NA, Conj ),
                       simplify( Conj v x (A r B), NF ).

r( ~ P  , S, NF  ) :-  proposition( P ),
                       ( holds( S, P ) -> NF = false
                       ;                  NF = true
                       ).

r( P    , S, NF  ) :-  proposition( P ),
                       ( holds( S, P ) -> NF = true
                       ;                  NF = false
                       ).
```

```
%% simplify( + formula, + state,- new formula ):
%% The formula has now been rewritten: simplify the result.

simplify( F, NF ) :-
        once( s( F, NF ) ).

%
s( true    v _    , true  ).
s( _       v true , true  ).
s( false   v A    , NA    ) :-  s( A        , NA ).
s( A       v false, NA    ) :-  s( A        , NA ).
s(  A v A  v B    , NF    ) :-  s( A v B    , NF ).
s( (A v B) v C    , NF    ) :-  s( A v B v C, NF ).

s( false   ^ _    , false ).
s( _       ^ false, false ).
s( true    ^ A    , NA    ) :-  s( A        , NA ).
s( A       ^ true , NA    ) :-  s( A        , NA ).
s(  A ^ A  ^ B    , NF    ) :-  s( A ^ B    , NF ).
s( (A ^ B) ^ C    , NF    ) :-  s( A ^ B ^ C, NF ).
s( x A     ^ x B  , x NF  ) :-  s( A ^ B    , NF ).

s( F              , F     ).

%--------------------------------------------------------------------------------
```