```
%% Gopal Gupta's LTL interpreter (modified by F.K.).
%%
%% This is a version of verifier.tlp that has been extended to produce a
%% counterexample.
%%
%% NOTE: This version loops for certain queries, because the treatment of paths
%%       is too naive. (Discovered by Brian W. DeVries.)
%%
%% Include with the definition of an automaton, which should specify
%% the following predicates:
%%      proposition/1   - succeeds only if the argument is a proposition,
%%                        can be used to enumerate all the symbols that denote
%%                        propositions.
%%      state/1         - succeeds only if the argument is a state,
%%                        can be used to enumerate all the symbols that denote
%%                        states.
%%      trans/2         - given the first argument (which should represent a
%%                        state S) nondeterministically produces the symbols
%%                        that represent all the states that can be reached from
%%                        S.
%%      holds/2         - succeeds only if the first argument represents a
%%                        state, and the second represents a proposition that
%%                        holds in that state.
%%
%% Invoke through
%%
%%    check( state, formula ).
%%
%% The formula will be normalized and negated by the program.
```

```
:- [ 'operators.pl' ].
:- [ 'normalize.pl' ].
:- [ 'looping_prefix.pl' ].
:- [ 'consistency_checker.pl' ].


:- top check/2.  % The "entry point"


%% 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, Path ) )
        ->
            write( 'COUNTEREXAMPLE: ' ),
            looping_prefix( Path, Prefix ),
            write( Prefix ),
            nl,
            fail
        ;
            true
        ).
```

```
%---------------  The verifier proper  ---------------

%--- The formula is normalized: only propositions can be negated.


% NOTE:  The rule for conjunction imposes restrictions on paths,
%        so results might be different than for the version without paths.
%        The restriction is that the path for one conjunct must be a prefix
%        of the path for the other.

verify( S, g A,   Path      ) :-  once( coverify( S, g A,   Path ) ).
verify( S, A r B, Path      ) :-  once( coverify( S, A r B, Path ) ).
verify( S, f A,   Path      ) :-  tverify( S, f A,   Path ).
verify( S, A u B, Path      ) :-  tverify( S, A u B, Path ).

verify( S, A,     [ S ]     ) :-  proposition( A ),    holds( S, A ).
verify( S, ~ A,   [ S ]     ) :-  proposition( A ), \+ holds( S, A ).

verify( S, A ^ B, Path      ) :-  verify( S, A, PathA ), verify( S, B, PathB ),
                                  (
                                      append( PathA, _, PathB )       % prefix?
                                  ->
                                      Path = PathB
                                  ;
                                      append( PathB, _, PathA )       % prefix?
                                  ->
                                      Path = PathA
                                  % otherwise fail!
                                  ).

verify( S, A v B, Path      ) :-  verify( S, A, Path ) ; verify( S, B, Path ).

verify( S, x A,   [ S | P ] ) :-  trans( S, S2 )  , verify( S2, A, P ).

                            % The last clause is correct only because the query is
                            % always negated, so for a successful query we will
                            % try out all the relevant clauses of trans/2 through
                            % backtracking.


:- tabled tverify/3.

tverify( S, f A,   Path ) :-  verify( S, A, Path )
                              ;
                              verify( S, x f A, Path ).

tverify( S, A u B, Path ) :-  verify( S, B, Path )
                              ;
                              verify( S, A ^ x( A u B), Path ).


:- coinductive coverify/3.

coverify( S, g A,   Path ) :-  verify( S, A ^ x g A,       Path ).

coverify( S, A r B, Path ) :-  verify( S, A ^ B,           Path ).

coverify( S, A r B, Path ) :-  verify( S, B ^ x( A r B ),  Path ).

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