```
%This specification is based on:
%Parsons, S., McBurney, P, and Wooldridge, M. The mechanics of some formal inter-agent dialogues, 2004 (Mechanics
%Parsons, S., McBurney, P., and Wooldridge, M. Some preliminary steps towards a meta-theory for formal inter-agent
dialogues, 2004 (Meta)
%-----------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------
%The games use the following external conditions:
%
% (Arg/!Arg, Stores, Content) checks whether an argument can or cannot be constructed for the Content from the Stores
% (AcceptanceAllowed, Player, Stores, Content) checks whether Player can accept Content based on his acceptance
attitude and the content of Stores
% (AssertionAllowed, Player, Stores, Content) checks whether Player can assert Content based on his assertion attitude
and the content of Stores
% (Support, {P, q}) checks whether the set P directly supports q
% (Implication p) checks whether p is an implication (i.e. a -> b)
% (ImplicationConsequent, p, q) checks whether q is the consequent of the implication formula p.
% (ImplicationAntecedent, p, q) checks whether q is an antecedent of the implication q.
% (Negation, p, q) checks whether q is the negation of p.
%-----------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------

IS{  %IS is information seeking, where B knows something A doesn't
  { turns, magnitude:multiple, ordering:strict} ;

  { roles, speaker, listener, winner } ;

  { players, min:2, max:2 } ;
  { player, id:A } ;
  { player, id:B } ;

  % agents have private knowledge bases and public commitment stores. The knowledge bases are essentially external
to the dialogue: they can be changed by processes other then the dialogue and should be administred externally.
However, here the knowledge bases are included as stores, so that it is possible to play these games without any
external processes that determine their contents.
  { store, id:CS, owner:A, structure:set, visibility:public } ;
  { store, id:CS, owner:B, structure:set, visibility:public } ;
  { store, id:KB, owner:A, structure:set, visibility:private, contents:$userDefined$ } ;
  { store, id:KB, owner:B, structure:set, visibility:private, contents:$userDefined$ } ;

  { backtrack, on };

  { rule, StartingRule, scope:initial,
        { assign(A, speaker) & move(add, next, question, {p}, A, {extCondition(!Arg, {{KB, A},{CS, A},{CS, B}}, {p}) &
```

extCondition(!Arg, {{KB, A},{CS, A},{CS, B}}, {q}) & extCondition(Negation, {{p},{q}})} } ; %An information seeking dialogue can only start if the initial speaker does not already have an arg for p or ~p (Mechanics, 335).

{ rule, Acceptance1, scope:movewise,
{ if { extCondition(AcceptanceAllowed, {speaker, {{KB, speaker},{CS, A},{CS, B}}, {p}}) & inspect(!in, {p}, CS, speaker)}
then { move(add, next, accept, {p}, speaker } }}; %This checks after each move whether the speaker can accept a point he has not already accepted (because, for example, he's accepted every s \in S that is the support of some p, i.e. he's accepted all the premises of an argument).

{ rule, Acceptance2, scope:movewise,
{ if { event(past, assert, {S}) & inspect(in, S, CS, speaker) }
then { move(add, next, accept, {S}, speaker } }}; %This checks after each move whether the speaker has accepted the full set {S} previously asserted by the other party (by accepting every s \in S separately), which then allows speaker to move the accept {S} move. Note that inspect(in, S, CS, speaker) means that all elements of S are in CS.

{ rule, Termination, scope:movewise,
{ if {size(LegalMoves, speaker, empty) & size(LegalMoves, listener, empty)}
then {status(terminate, IS)} } }; % Game terminates when there are no more possible moves . Note that this rule should fire movewise but after the above Acceptance rules, as these rules can concievably add new moves to Legalmoves.

{ interaction, question, {p}, questioning, {p},
{ if { extCondition(AssertionAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {p}}) & event(!past, assert, {p}, listener)}
then { move(add, next, assert, {p}, listener) & assign(speaker, listener) & assign(listener, speaker)}
elseif { extCondition(AssertionAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {q}}) & extCondition(Negation, {{p},{q}}) & & event(!past, assert, {q}, listener) }
then { move(add, next, assert, {q}, listener) & assign(speaker, listener) & assign(listener, speaker)}
else { move(add, next, no-answer, listener) & assign(speaker, listener) & assign(listener, speaker)}}}; %A asks question and B asserts p, ~p or U depending on what it can assert (Meta, section 4.1, 2) and whether p or ~p was asserted previously. Note that the same locution cannot be uttered twice (by any player).

{ interaction, no-answer,
{status(terminate,IS)}}; %The dialogue terminates after an assert(U) (Meta, section 4.1, 3)

{ interaction, assert, {p}, asserting, {p},
{ if { extCondition(AcceptanceAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {p}}) & event(!past, accept, {p}, listener) }
then { store(add, {p}, CS, speaker) & move(add, next, accept, {p}, listener) & assign(speaker, listener) & assign(listener, speaker) }
elseif { event(!past, challenge, {p}, listener) }
then { store(add, {p}, CS, speaker) & move(add, next, challenge, {p}, listener) & assign(speaker, listener) & assign(listener, speaker) }

```
      else {status(terminate, IS)}}} ; %after an assert the player either accepts or challenges, otherwise (in the event something was already challenged but can still not be accepted) the dialogue terminates (Meta, section 4.1, 3)

 { interaction, assert, {S}, asserting, S,
    { if { extCondition(AcceptanceAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, s}) & event(!past, accept, s, listener) }
      then { store(add, s, CS, speaker) & move(add, future, accept, s, listener) & assign(speaker, listener) & assign(listener, speaker) }
           elseif { event(!past, challenge, s, listener) }
           then { store(add, s, CS, speaker) & move(add, future, challenge, s, listener) & assign(speaker, listener) & assign(listener, speaker) }}} ;   %the player either accepts or challenges each element of some  set {S} asserted by B (section 4.1, 3). This is basically the same as for assert, {p} for each s \in S.

  { interaction, accept, {p}, asserting, {p},
           { store(add, {p}, CS, speaker)}}; %accepting something adds it to the speaker's commitments. Note that it does not give the turn to the other player: a player can accept multiple propositions after another in a turn (see also the Acceptance1 and2 rules).

   { interaction, challenge, {p}, challenging, {p},
           { if { extCondition(Support, {{S}, {p}}) & extCondition(AssertionAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, S}) & event(!past, assert, {S}, listener) }
      then { move(add, next, assert, {S}, listener) & assign(speaker, listener) & assign(listener, speaker) }
           else { move(add, next, no-answer, listener) & assign(speaker, listener) & assign(listener, speaker)} }}; %after a challenge the other player asserts one or more supporting elements.
}

%-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
%-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
%-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

I{  %Inquiry, where A and B try to find a proof for p
  { turns, magnitude:multiple, ordering:strict} ;

  { roles, speaker, listener, winner } ;

  { players, min:2, max:2 } ;
  { player, id:A } ;
  { player, id:B } ;

  { store, id:CS, owner:A, structure:set, visibility:public } ;
  { store, id:CS, owner:B, structure:set, visibility:public } ;
```

```
{ store, id:KB, owner:A, structure:set, visibility:private, contents:$userDefined$ } ;
{ store, id:KB, owner:B, structure:set, visibility:private, contents:$userDefined$ } ;
{ store, id:IP, structure:set, visibility:public } ;

{ backtrack, on };

{ rule, StartingRule, scope:initial,
      { if { extCondition(!Arg, {{KB, A},{CS, A},{CS, B}}, {p}) & extCondition(!Arg, {{KB, B},{CS, A},{CS, B}}, {p}) }
   then { assign(B, speaker) & move(add, next, prove, {p}, B) & store(add, {p}, IP)} } ; % the players try to answer a question whose answer is not known to either (the initial point) (Mechanics, sec 4.2)

{ rule, AcceptanceIP, scope:movewise,
      { if { inspect(in, {p}, IP) & inspect(in, {p}, CS, speaker) & inspect(in, {p}, CS, listener)}
   then { status(terminate, I) }
   elseif { inspect(in, {p}, IP) & extCondition(AcceptanceAllowed, {speaker, {{CS, A},{CS, B}}, {p}}) & extCondition(AcceptanceAllowed, {listener, {{CS, A},{CS, B}}, {p}}) & inspect(!in, {p}, CS, speaker) & inspect(in, {p}, CS, listener)}
   then { move(add, next, accept, {p}, speaker) }
   elseif { inspect(in, {p}, IP) & extCondition(AcceptanceAllowed, {speaker, {{CS, A},{CS, B}}, {p}}) & extCondition(AcceptanceAllowed, {listener, {{CS, A},{CS, B}}, {p}}) & inspect(in, {p}, CS, speaker) & inspect(!in, {p}, CS, listener)}
   then { move(add, next, accept, {p}, listener) & assign(speaker, listener) } }}; %If at any time both players have an acceptable arg for the initial point, then they both accept.

{ rule, Acceptance1, scope:movewise,
      { if { extCondition(AcceptanceAllowed, {speaker, {{KB, speaker},{CS, A},{CS, B}}, {p}}) & inspect(!in, {p}, CS, speaker)}
   then { move(add, next, accept, {p}, speaker) } }};

{ rule, Acceptance2, scope:movewise,
      { if { event(past, assert, {S}) & inspect(in, S, CS, speaker) }
   then { move(add, next, accept, {S}, speaker } }};

{rule, Switch, scope:movewise,
      { if { size(LegalMoves, speaker, empty) & size(LegalMoves, listener, empty) & event(last, accept, {p}, {extCondition(implication, {p})}}
   then { move(add, next, prove, {q}, listener, {extCondition(ImplicationAntecedent, {p}, {q})}) & assign(listener, speaker)
   else { status(terminate, I) } ; %whenever one player accepts an implication the other player asks him to prove the antecedent (Mechanics p. 334).

{ rule, Termination, scope:movewise,
      { if {size(LegalMoves, speaker, empty) & size(LegalMoves, listener, empty)}
      then {status(terminate, IS)} } };
```

```
 {interaction, prove, {p}, questioningGrounds, {p},
   { if { extCondition(AssertionAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {p}}) & event(!past, assert, {p}, listener) }
     then { move(add, next, assert, {p}, listener) & assign(speaker, listener) & assign(listener, speaker) }
          elseif { extCondition(ImplicationConsequent, {p}, {q}) & extCondition(AssertionAllowed, {listener, {{KB,
listener},{CS, A},{CS, B}}, {q}}) & event(!past, assert, {q}, listener) }
     then { move(add, next, assert, {q}, listener) & assign(speaker, listener) & assign(listener, speaker) }
     else { move(add, next, no-answer, listener) & assign(speaker, listener) & assign(listener, speaker) } }}; %player A
asks for proof, player B replies with either the proposition p itself or an implication q: r->p.

 {interaction, no-answer,
   {status(terminate,IS)}};

{ interaction, assert, {p}, asserting, {p},
   { if { extCondition(AcceptanceAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {p}}) & event(!past, accept, {p},
listener) }
     then { store(add, {p}, CS, speaker) & move(add, next, accept, {p}, listener) & assign(speaker, listener) &
assign(listener, speaker) }
          elseif { event(!past, challenge, {p}, listener) }
          then { store(add, {p}, CS, speaker) & move(add, next, challenge, {p}, listener) & assign(speaker, listener) &
assign(listener, speaker) }
     else {status(terminate, IS)}}} ;  %player A either accepts or challenges, otherwise the dialogue terminates.

 { interaction, assert, {S}, asserting, S,
   { if { extCondition(AcceptanceAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, s}) & event(!past, accept, s, listener) }
     then { store(add, s, CS, speaker) & move(add, future, accept, s, listener) & assign(speaker, listener) &
assign(listener, speaker) }
          elseif { event(!past, challenge, s, listener) }
          then { store(add, s, CS, speaker) & move(add, future, challenge, s, listener) & assign(speaker, listener) &
assign(listener, speaker) }}} ;  %player A either accepts or challenges each element of some  set {S} asserted by
B(section 4.1, 3)

 { interaction, challenge, {p}, challenging, {p},
          { if { extCondition(Support, {{S}, {p}}) & extCondition(AssertionAllowed, {listener, {{KB, listener},{CS, A},{CS, B}},
S}) & event(!past, assert, {S}, listener) }
     then { move(add, next, assert, {S}, listener) & assign(speaker, listener) & assign(listener, speaker) }
          else { move(add, next, no-answer, listener) & assign(speaker, listener) & assign(listener, speaker)} }};


 { interaction, accept, {p}, asserting, {p},
          { store(add, {p}, CS, speaker)}};

}
```

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

Per{  %P is persuasion
  { turns, magnitude:multiple, ordering:strict} ;

  { roles, speaker, listener, winner } ;

  { players, min:2, max:2 } ;
  { player, id:A } ;
  { player, id:B } ;

  { store, id:CS, owner:A, structure:set, visibility:public } ;
  { store, id:CS, owner:B, structure:set, visibility:public } ;
  { store, id:KB, owner:A, structure:set, visibility:private, contents:$userDefined$ } ;
  { store, id:KB, owner:B, structure:set, visibility:private, contents:$userDefined$ } ;
  { store, id:IP, structure:set, visibility:public } ;

  { backtrack, on };

  { rule, StartingRule, scope:initial,
          { if {extCondition(AcceptableArg, {{KB, A},{CS, A},{CS, B}}, {p}) & extCondition(!AcceptableArg, {{KB, B},{CS,
A},{CS, B}}, {p}) & extCondition(AcceptableArg, {{KB, B},{CS, A},{CS, B}}, {q}) & extCondition(Negation, {{p},{q}}) }
      then {assign(A, speaker) & move(add, next, assert, {p}, A) & store(add, {p}, IP)} } } ;

  { rule, Acceptance1, scope:movewise,
          { if { extCondition(AcceptanceAllowed, {speaker, {{KB, speaker},{CS, A},{CS, B}}, {p}}) & inspect(!in, {p}, CS,
speaker)}
      then { move(add, next, accept, {p}, speaker } }};

  { rule, Acceptance2, scope:movewise,
          { if { event(past, assert, {S}) & inspect(in, S, CS, speaker) }
      then { move(add, next, accept, {S}, speaker } }};

 { rule, Termination, scope:movewise,
          { if {size(LegalMoves, speaker, empty) & extCondition(AcceptableArg, {{KB, B},{CS, A},{CS, B}}, IP) }
          then {assign(A, winner) & status(terminate, Per)}
      else {assign(B, winner) & status(terminate, Per)}} };
```

```
  { interaction, assert, {p}, asserting, {p},
    { if { extCondition(AcceptanceAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {p}}) & event(!past, accept, {p},
listener) }
      then { store(add, {p}, CS, speaker) & move(add, next, accept, {p}, listener) & assign(speaker, listener) &
assign(listener, speaker) }
           elseif { extCondition(AssertionAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {q}}) & extCondition(Negation,
{{p},{q}}) & event(!past, assert, {q}, listener) }
           then { store(add, {p}, CS, speaker) & move(add, next, assert, {q}, listener) & assign(speaker, listener) &
assign(listener, speaker) }
      elseif { event(!past, challenge, {p}, listener) }
           then { store(add, {p}, CS, speaker) & move(add, future, challenge, {p}, listener) & assign(speaker, listener) &
assign(listener, speaker) }}} ;

 { interaction, assert, {S}, asserting, S,
    { if { extCondition(AcceptanceAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, s}) & event(!past, accept, s, listener) }
      then { store(add, s, CS, speaker) & move(add, future, accept, s, listener) & assign(speaker, listener) &
assign(listener, speaker) }
      elseif { extCondition(AssertionAllowed, {listener, {{KB, listener},{CS, A},{CS, B}}, {q}}) & extCondition(Negation, {s,{q}
& event(!past, assert, {q}, listener) }
           then { store(add, s, CS, speaker) & move(add, next, assert, {q}, listener) & assign(speaker, listener) &
assign(listener, speaker) }
           elseif { event(!past, challenge, s, listener) }
           then { store(add, s, CS, speaker) & move(add, future, challenge, s, listener) & assign(speaker, listener) &
assign(listener, speaker) }}} ;

  { interaction, accept, {p}, asserting, {p},
           { store(add, {p}, CS, speaker)}};
}
```