Game Prolog

Mario R. F. Benevides and Ricardo F. Ribeiro
E-mail: {mario,ribeiro}@cos.ufrj.br

Federal University of Rio de Janeiro, Brazil

Abstract. This work proposes a Game Prolog (Modal Action language), which aims to represent and reasoning about extensive games. A sound SLD resolution method is present. The language is extended with choice, sequential composition, converse and strategy operators. The use of the converse operator is discussed when representing and reasoning about actions that were performed in the past.

1 Introduction

Extensive Games play an important role in Game Theory, they model the interaction between players in turns and the possible outcomes of the game. They are represented as trees. In the figure 1, we present an extensive game with players 1 and 2, and the possible outcomes are propositions p and q being true or false in the final states. The same example is presented in Game Prolog clauses in the table 1.

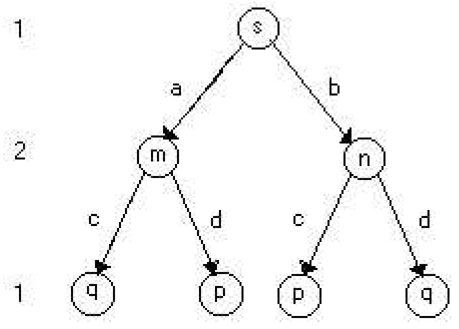

Fig. 1. A simple example

The use of Propositional Dynamic Logic to model extensive games has been investigate in some works [18], [5], [11] and [8]. In this work, we use a subset of Dynamic Language, as presented in [6], [14], to reasoning about extensive games. We define a modal Horn clause and present a resolution method to perform SLD(f) refutations for these clauses. This method works backwards.

Table 1. A simple example

Clause	Rule	Action sequence
1. $s \leftarrow$		
2. $[1:a]m \leftarrow s$		
3. $[1:b]n \leftarrow s$		
4. $[2:c]q \leftarrow m$		
5. $[2:c]p \leftarrow n$		
6. $[2:d]p \leftarrow m$		
7. $[2:d]q \leftarrow n$		
8. $\leftarrow p$		$[[s]]_0 = \epsilon$
9. $\leftarrow m$	EXT(4)	$[[s]]_1 = [2:c]$
10. $\leftarrow s$	EXT(2)	$[[s]]_2 = [1:a][2:c]$
11. $\leftarrow$	RED(1)	$[[s]]_3 = [1:a][2:c]$

Associating p to the victory for the 1^{st} player, and q to the victory for the 2^{nd} player, we can find a path, which is $[1:a][2:d]$, that leads player 1 to the victory. However, there's no winning strategy for the player 1, because after the action a, the player 2 can reach q by performing the action c. Even in the path $[1:b][2:c]$, that would bring the victory for player 1, the player 2 would reach q by doing the action d.

The winning strategy operator, in the propositional mode, is similar to ATL [1], which is briefly presented in section 2 and ATEL [12], [13]. In first-order mode, it is similar to the Negation as Failure of SLD Resolution [15].

Game Prolog is inspired in Modal Action Prolog, as conceived in [22], [4]. A presentation to some Game Theory concepts is done in section 3. Section 4 presents the language and the SLD resolution method for modal Horn clauses. In the section 5, it is presented extensions to Modal Prolog. The Winning Strategy operator is discussed separately, in section 6. An example of using Game Prolog is shown in section 7. Finally, in section 8, the conclusions of this work are presented.

2 An ATL summary

In this section, a brief description of ATL [1] is presented. This logic is a tool developped to check whether a set of agents Γ can make a given proposition β true in the next state ($\ll \Gamma \gg \bigcirc\beta$), or in some future state ($\ll \Gamma \gg \Diamond\beta$), or in every state inside a given path ($\ll \Gamma \gg \Box\beta$).

The second operator has a similar behavior to the winning strategy operator, as can be seen in subsections 6.1 and 6.3.

ATEL logic [12], [13] is the ATL logic extended with epistemic operators.

2.1 Alternating Transition Systems

An *Alternating Transition System* (ATS) is a tuple $S =< \Pi, \Sigma, Q, \pi, \delta >$, where:

Π is a finite, non-empty set of *atomic propositions.*

$\Sigma = \{a_1, ..., a_n\}$ is a finite, non-empty set of *agents.*

Q is a finite, non-empty set of *states*

$\pi : Q \rightarrow 2^{\Pi}$ gives the set of propositions satisfied in each state.

$\delta : Q \times \Sigma \rightarrow 2^{2^Q}$ is the system transition function, which maps states and agents to the choices available to agent a when the system is in state q. It's required that this function satisfy the constraint that the system is completely controlled by its component agents: for every state $q \in Q$, and every set $Q_1, ..., Q_n$ of choices $Q_a \in \delta(q, a)$, the intersection $Q_1 \cap ... \cap Q_n$ is a singleton. In other words, if every agent has made his choice, the system is completely determined.

2.2 Language

An ATL formula, formed with respect to an ATS $S =< \Pi, \Sigma, Q, \pi, \delta >$, is one of the following:

$\top$

p, where $p \in \Pi$ is a primitive proposition.

$\neg\varphi$ or $\varphi \vee \psi$, where φ and ψ are formulae of ATL.

$\ll \Gamma \gg \bigcirc\varphi$, $\ll \Gamma \gg \Box\varphi$ or $\ll \Gamma \gg \varphi\mathcal{U}\psi$, where $\Gamma \in \Sigma$ and φ and ψ are formulae of ATL.

2.3 Semantics

The semantics of ATL requires the notion of *strategy*, which is a mapping $f_a : Q^+ \rightarrow 2^Q$, where a is an agent, Q^+ is a sequence of states and $f_a(\lambda.q) \in \delta(q, a)$. This set is the computations that can be enforced by agent a.

From the definition of strategy, for a state q, a set Γ of agents, and a set $F_\Gamma = \{f_a | a \in \Gamma\}$, it can be defined the notion of *outcomes* of F_Γ from q as the set $out(q, F_\Gamma)$ of the computations started in q that can be enforced by the agents in Γ when they cooperate and follow the strategies in F_Γ.

If S is an ATS, q is a state in S, and φ is a formula of ATL over S, then it's written $S, q \models \varphi$ to mean that φ is satisfied at state q in system S. The rules defining the satisfaction relation $\models$ are as follows:

$S, q \models \top$

$S, q \models p$ iff $p \in \pi(q)$, where $p \in \Pi$.

$S, q \models \neg\varphi$ iff $S, q \not\models \varphi$.

$S, q \models \varphi \vee \psi$ iff $S, q \models \varphi$ or $S, q \models \psi$.

$S, q \models \ll \Gamma \gg \bigcirc\varphi$ iff there exists a set F_Γ of strategies, one for each $a \in \Gamma$, such that for all computations $\lambda \in out(q, F_\Gamma)$ we have $\lambda[1] \models \varphi$.

$S, q \models \ll \Gamma \gg \Box\varphi$ iff there exists a set F_Γ of strategies, one for each $a \in \Gamma$, such that for all computations $\lambda \in out(q, F_\Gamma)$ we have $\lambda[u] \models \varphi$, for all $u \geq 0$.

$S, q \models \ll \Gamma \gg \varphi\mathcal{U}\psi$ iff there exists a set F_Γ of strategies, one for each $a \in \Gamma$, such that for all computations $\lambda \in out(q, F_\Gamma)$ there exists a position $u \geq 0$ such that $\lambda[u] \models \psi$ and, for all positions $0 \leq v < u$, we have $\lambda[v] \models \varphi$.

3 Game Theory concepts

In this section, a brief overview of Game Theory concepts are present. For further information, see [21] and [17].

3.1 Strategic Games

Definition The *strategic game* is the most simple model of game, which is a tuple

$G = \{N, \{A_i\}_{i \in N}, \{\succeq_i\}_{i \in N}\}$

where:

– $N = \{1, ..., n\}$ is a finite set of players
– $\{A_i\}_{i \in N}$ associates to each player i a nonempty set of actions or strategies.
– $\{\succeq_i\}_{i \in N}$ is the preference relation of player i: Let $A = A_1 \times ... \times A_n$ be the set of action profiles. Then, for every player $i \in N$, $\succeq_i$ is a complete transitive relation for A.

Strategic games are usually represented by n-dimensional matrices, where the avaliable actions for each player are put on the respective dimension, and for each set of actions it is assigned a result, which is a tuple contaning the payoff of each player. The players perform their actions in the same moment.

Example: Prisoners' Dilemma Two prisoners are put in separate cells and interrogated. If both confess, they receive 6 years in prison. If only one confess, he is released and used as a witness against the other, who will receive 10 years. If no one confess, they will be accused of a minor offense and receive 2 years each one. The matrix representation of this game is shown in table 2.

Table 2. The Prisoners' Dilemma

	Don't confess	Confess
Don't confess	(-2, -2)	(-10, 0)
Confess	(0, -10)	(-6, -6)

Nash Equilibrium Given an action profile $a \in A$ and an action $x \in A_i$, let $a[i : x]$ denote the profile which is similar to a, with player i taking action x. Given a strategic game $G = \{N, \{A_i\}_{i \in N}, \{\succeq_i\}_{i \in N}\}$, an action profile is a *Nash equilibrium* if and only if:

$\forall i \in N \forall x \in A_i : a \succeq_i a[i : x]$

In other words, this is the profile where each player acts optimally given the actions of the other players. The Nash equilibrium can be seen as a steady state of the game: once it was reached, changing the action is an irracional action for every player.

There are games without Nash equilibrium, and others which have more than one equilibrium. In the Prisoners' Dilemma example, the profile (Confess, Confess) is the Nash equilibrium, since not confessing will lead a worst payoff.

3.2 Extensive Games

Definition The Extensive Game model allows a detailed representation of a game, with intermediate states as the actions are performed, instead of the instantaneous strategic game.

Given a sequence of actions $h = (a_1, a_2, ...)$, let $h|k = (a_1, a_2, ..., a_k)$ denote the initial subsequence of h. Then, an *Extensive Game with Perfect Information* G can be defined as a tuple:

$G = \{N, H, P, \{\succeq_i\}_{i \in N}\}$

where:

- $N = \{1, ..., n\}$ is a finite set of players
- H is a set of sequences over a set A of actions. H must satisfy these requirements:
  1. The empty sequence $() \in H$
  2. H is closed under initial subsequences. In other words, if $h \in H$ has length l, then for all $k < l$, $h|k \in H$, for all k.
  3. Given an infinite sequence h such that for all k, $h|k \in H$, then $h \in H$.

  Let $Z \subseteq H$ be the set of terminal histories, i.e., $h \in Z$ iff for all $h' \in H$ such that $h'|k = h$, then $h' = h$.
- $P : H \setminus Z \to N$ is the player function that assigns to each nonterminal history the player whose turn it is to move.
- $\succeq_i$ is a relation on Z, called preference relation.

The extensive games are usually represented as trees, with its terminal states containing the payoffs.

In an *extensive game with imperfect information*, the tuple G, has an extra component, $\{\mathcal{I}_i\}_{i \in N}$, which are *information sets for each player.* $\mathcal{I}_i$ is a partition of the set of histories where i has to move. If player i has to make a decision in a game at history $h \in \mathcal{I}_r \in \mathcal{I}_i$, (s)he does not know which of the histories in $\mathcal{I}_r$ is the real history. Then, his/her strategies ought to be uniform within every information set.

If all information sets are singletons, the game is of perfect information.

Given a history $h = (a_1, ..., a_n)$ and an action $b \in A$, let $(h, b) = (a_1, ..., a_n, b)$. Furthermore, let $A(h) = \{x \in A | (h, x) \in H\}$ be the set of actions possible after h. A strategy for player i can be defined as a function $s_i : \{h \in H \backslash Z | P(h) = i\} \to A$ such that $s_i(h) = A(h)$.

Given a strategy profile $s = \{s_i\}_{i \in N}$, let $O(s) \in H$ be the history which results when the players use their respective strategies.

Subgame-perfect Equilibrium Let $G = \{N, H, P, \{\succeq_i\}_{i\in N}\}$ be an extensive game and let (h, h') the concatenation of h and h'. For each history h of G, a *subgame* started after h can be defined as a tuple:

$G(h) = \{N, H|_h, P|_h, \{\succeq_i |_h\}_{i\in N}\}$

where:

- $H|_h = \{h'|(h, h') \in H\}$
- $P|_h(h') = P(h, h')$ for each $h' \in H|_h$
- $h' \succeq_i |_h h''$ iff $(h, h') \succeq_i (h, h'')$

Similarly, strategies s_i and strategy profiles s can also be restricted to subgames by setting $s_i|_h(h') = s_i(h, h')$.

A strategy profile s is a *subgame-perfect equilibrium* iff for all histories $h \in H$, $s|_h$ cannot be altered by any player without decreasing his/her payoff. In a game with perfect information, the subgame-perfect equilibrium can be found by backward induction, which is presented below.

Backward Induction In the backward induction algorithm, the tree of the extensive game is scanned from the terminal nodes to the root. When a terminal node is visited, there are no decisions to be made. In a node n, which is immediately before the terminals, the successor that gives the best result for the player given by $P(a_1, ..., a_k, n)$, and the payoffs are assigned to this node.

This process are repeated until the root is reached. In the figure 2 is presented an example of backward induction. The blue path is the subgame equilibrium. The purple paths are candidates to be the equilibrium in inner nodes, and were discarded as the algorithm get closer to the root. The pink and green payoffs highlight the choice done by players 1 and 2, respectively.

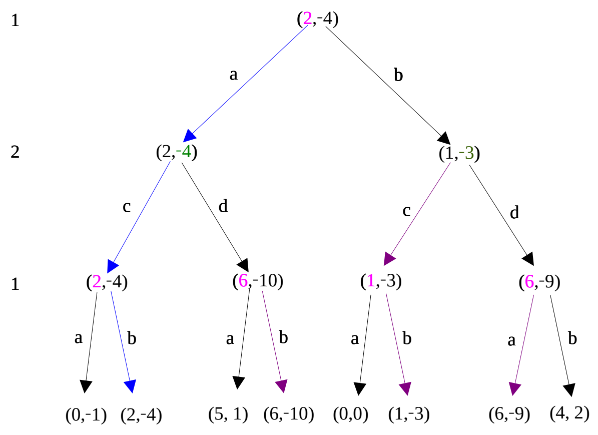

Fig. 2. An example of backward induction

4 Modal Prolog

4.1 Language

Alphabet The alphabet of modal Horn clauses consists of:

– a set of n-aries predicative symbols,
– a set of variables,
– a set of n-aries functional symbols,
– a set of constants,
– a finite set of actions Ac and
– logical symbols: $\leftarrow$, [,] and ,.

Language symbols The language is presented below.

– Term: A constant, or a variable, or $f(t_1, ..., t_n)$, where f is a n-ary functional symbol and $t_1, ..., t_n$ are terms.
– Positive literal: $p(t_1, ..., t_n)$ where p is a predicative symbol and $t_1, ..., t_n$ are terms.
– Pure atom: Positive literal.
– Player: A constant that identifies an agent.
– Action: If a^n is a name of a n-ary action, p is a player, and $t_1, ..., t_n$ are terms, then $[p : a(t_1, ..., t_n)]$ is an action.
– $[[a]]^n$: $[a_1]...[a_n]$, where the a_i's are actions performed serially.
– Modal atom: $[a_1]...[a_n]q$, where q is a pure atom, $n > 0$ and a_i belongs to Ac.
– Atom: Pure atom or modal atom.
– Facts: Pure atoms of the language.
– Non-modal definite clause: $q \leftarrow h_1, ..., h_m (m \geq 0)$.
– Modal definite clause: $[[a]]^n q \leftarrow h_1, ..., h_m (m \geq 0)$.
– Objective clause: $\leftarrow h_1, ..., h_m (m \geq 0)$

A clause like $q \leftarrow h_1, ..., h_m$ holds in any scenery. If there is a clause like $[[a]]^n \leftarrow h_1, ..., h_m$, and a scenery contains $h_1, ..., h_m$, the scenery obtained by $[[a]]^n$ contains q.

4.2 Rules

Modal Action Prolog uses two resolution rules: modal f-reduction and modal f-extension, as follows.

Modal f-reduction Let G be an objective clause, f a selection function, D an auxiliary clause and ψ a variable relabelling for D.
$G = \leftarrow g_1, ..., g_n$
$f(G) = g_i = [[b]]^j [[c]]^k q (j \geq 0, k \geq 0)$
$D\psi = [[a]]^k q' \leftarrow h_1, ..., h_m (k \geq 0)$
If there is a m.g.u. θ of $[[c]]^k q, [[a]]^k q'$, then the result is:
$G' = \leftarrow (g_1, ..., g_{i-1}, [[b]]^j h_1, ..., [[b]]^j h_m, g_{i+1}, ..., g_n)\theta$

For instance, let G and D be:

$G = \leftarrow [b]s$

$f(G) = [b]s$

$D = [b]s \leftarrow r$

Then, it is obtained $[[b]]^j = \epsilon$, $[[c]]^k = [b]$ and $[[a]]^k = [b]$. Since $[[a]]^k r = [[c]]^k r$, the modal f-reduction rule can be applied, giving the following result:

$G' = \leftarrow r$

Modal f-extension Let G be an objective clause, D a definite clause modal or non-modal and ψ a relabelling for D under the presence of G.

$G = \leftarrow g_1, ..., g_n$

$f(G) = g_i = [[a]]^k q (k \geq 0)$

$D\psi = [[b]]^j [[c]]^k q' \leftarrow h_1, ..., h_m (j > 0, k \geq 0)$

If there is a m.g.u. θ of $[[a]]^k q, [[c]]^k q'$, then the result is:

$G' = \leftarrow ([[b]]^j g_1, ..., [[b]]^j g_{i-1}, h_1, ..., h_m, [[b]]^j g_{i+1}, ..., [[b]]^j g_n)\theta$

Where $[[b]]^j$ is called **extension sequence of G'**.

For instance, let G and D be:

$G = \leftarrow p, s$

$f(G) = p$

$D = [b]p \leftarrow t$

Then, it is obtained $[[a]]^k = \epsilon$, $[[b]]^j = [b]$ and $[[c]]^k = \epsilon$. Since $[[a]]^k p = [[c]]^k p$, the modal f-extension rule can be applied, giving the following result:

$G' = \leftarrow t, [b]s$

Where $[[b]]^j = [b]$ is the *extension sequence of G'*.

Generic SLD(f)-modal deduction Let f be a selection function, S an pseudo-definite modal set and G an objective clause. A generic SLD(f)-modal deduction of G from S is a sequence of clauses $C = (C_1, ..., C_n)$ ending on G, a sequence $A = ([[s]]_0, ..., [[s]]_{n-r})$ of action sequences and $r < n$ such that:

i. $[[s]]_0 = \epsilon$
ii. For all $i < r$, C_i is a modal or non-modal definite clause belonging to S.
iii. C_r is the objective clause of S.
iv. For all $i > r$, C_i is obtained by applying modal f-reduction or modal f-extension rule to C_{i-1} and C_j, $j < r$, and:
    a. C_i is obtained by f-reduction $[[s]]_{i-r} = [[s]]_{i-r-1}$
    b. C_i is obtained by f-extension $[[s]]_{i-r} = [[b]]^j [[s]]_{i-r-1}$, where $[[b]]^j$ is the extension sequence of C_i.
    c. If there is $e (r < e < n)$ such that C_e is obtained by f-reduction to C_{e-1} and $C_j (j < r)$, where C_j is a fact, then no $C_i (i > e)$ is obtained by modal f-extension.

Generic SLD(f)-modal refutation A generic SLD(f)-modal refutation is the deduction of the empty clause from an pseudo-definite set S, with an objective clause G and a selection function f. On the deduction suffix, for each new objective clause must be placed:

1. From which rule the clause was obtained.
2. Which auxiliary clause of refutation was used.
3. The ψ relabelling of the auxiliary clause.
4. The m.g.u. used in the unification of the auxiliary and objective clauses.
5. The action sequence of the plan obtained on this derivation step.

In the table 3, it is presented an example of deduction.

Table 3. An example of deduction

Clause	Rule	Action sequence
1. $u \leftarrow$		
2. $r \leftarrow$		
3. $t \leftarrow$		
4. $[a]q \leftarrow p, s$		
5. $[b]s \leftarrow r$		
6. $[b]p \leftarrow t$		
7. $\leftarrow q$		$[[s]]_0 = \epsilon$
8. $\leftarrow p, s$	EXT(4)	$[[s]]_1 = [a]$
9. $\leftarrow t, [b]s$	EXT(6)	$[[s]]_2 = [b][a]$
10. $\leftarrow [b]s$	RED(3)	$[[s]]_3 = [b][a]$
11. $\leftarrow r$	RED(5)	$[[s]]_4 = [b][a]$
12. $\leftarrow$	RED(2)	$[[s]]_5 = [b][a]$

5 Extensions

In this section, we present some extensions of Game Prolog

5.1 Facts valid in every scenery

The original definition of Modal Prolog [4], [22] defines the facts as valid only on the initial scenery. However, it is desirable the definition of facts which are valid on every possible scenery. This feature could be achieved by adding rules on this style:

$$[action]\ fact(parameters) \leftarrow fact(parameters).$$

But this approach has a problem: it is usually necessary the value of parameters in an unification. However, in order to use the fact, it is necessary to obtain an action sequence, still unknown. When using any action sequence, the amount of required backtracking grows too much.

A solution used to implement Modal Prolog, in [16], is the use of a special mark for facts which are always valid. In this case, a modality containing suspension points, meaning that the fact will be valid after any action sequence, including the empty one. Thus, a fact can be defined on this way:

$$[...]\ fact(parameters).$$

Then, its parameters can be used inside an unification with the objective clause, using modal f-reduction. This approach reduces the unnecessary backtracking.

5.2 Player definition

When dealing with games, the specification of the action performed for some rule is not enough, being required the indication of the player which has performed it. In this way, it is proposed the operator :, that links the player to the executed action. For instance, the modality $[j : a]$ says that player j performed the action a.

In a First-Order refutation, the player may be variable, becoming unifiable with other elements of a clause, also allowing more complex turn functions.

5.3 Operators for Combination of Actions

In order to increase the expressivity of Modal Prolog, operators for combination of actions are introduced, with their mappings to common clauses. These operators, which are similar to PDL [5] [11], are defined as follows: [1]

Choice Denoted by the symbol +, the modality $[a + b]$ means that is possible to do the action a or the action b non-deterministically.

For instance, the clause: $[a + b]q \leftarrow p_1, ..., p_n$

Will be converted to:

$[a]q \leftarrow p_1, ..., p_n$

$[b]q \leftarrow p_1, ..., p_n$

Composition Denoted by the symbol ;, the modality $[a; b]$ means that the actions a and b are done in sequence.

The clause: $[a; b]q \leftarrow p_1, ..., p_n$

Will be converted to:

$[a]q' \leftarrow p_1, ..., p_n$

$[b]q \leftarrow q'$

Where q' is a **new** predicate in the pseudo-definite set of clauses.

[1] For the sake of clarity the player identification is omitted in this section.

5.4 Converse Modality

The converse modality ([8], [9]) is a feature that allows the consultation to atoms inside a previous scenery, from which the deduction arrived by executing an action, The formula $[i : a-]p$ is true in the current scenery s if exists a previous scenery s' where p is true and if player i would have executed action a, then it would have reached state s.

Whenever performing a modal deduction, at the moment when an atom containing an action with converse modality is selected from the objective clause, there must be an entrance clause that has the correspondent action on its head and the literal inside its body. Then, this clause will be inverted, as shown in figure 3, for having the converse modality in the head and resolve it with the objective clause.

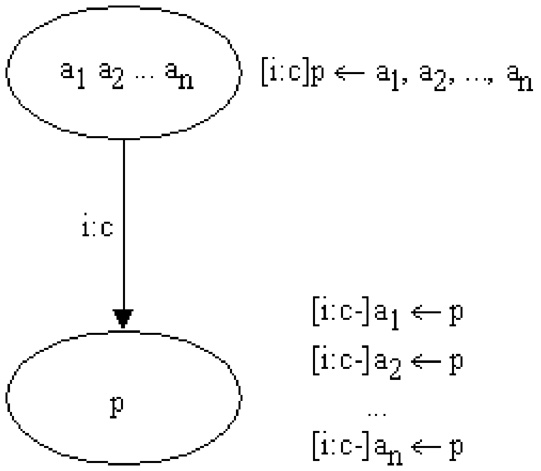

Fig. 3. The same clause from different viewpoints

In a deduction, at a point where an atom containing converse modality is selected from the objective clause, there must be an input clause that has the correspondent action on its head and the literal inside its body. Then, the following computation must be done.

1. $[i : c]p \leftarrow a_1, ..., a_n$

...

k. $\leftarrow [i : c-]a_1$

Here will be done the inversion of clause 1 in order to have the converse modality in the head:

1.1. $[i : c-]a_1 \leftarrow p$

...

1.n. $[i : c-]a_n \leftarrow p$

On this way, the (k) clause can be solved with the clause (1.1.) by modal f-reduction. Thus:

k+1. $\leftarrow p\ [[s]]_j\ (j > 0)$

On the next step, the modal f-extension of the objective clause obtained with the entrance clause inverted on the previous step.

k+2. $\leftarrow a_1, ..., a_n\ [[s]]_{j+1} = [i : c][[s]]_j$

Note After a modal f-reduction with a clause obtained from the inversion of an entrance clause, it is **mandatory** the modal f-extension of the objective clause with the entrance clause that has been inverted on the previous step. This is necessary to the deduction remaining in the path investigated previously. If other clause would be used, the result will be incorrect, once the possibility of this resolution hadn't existed before the inversion of the entrance clause.

In the table 4, it can be seen an example of deduction using Converse Modality.

Table 4. Deduction using Converse Modality

	Clause	Rule	Action sequence
1.	$p \leftarrow$		
2.	$q \leftarrow$		
3.	$[1 : a]r \leftarrow p$		
4.	$[1 : b]s \leftarrow p$		
5.	$[1 : b]t \leftarrow p$		
6.	$[2 : c]x \leftarrow [1 : a-]p, t$		
7.	$[2 : c]x \leftarrow s$		
8.	$[1 : a]t \leftarrow q$		
9.	$\leftarrow x$		$[[s]]_0 = \epsilon$
10.	$\leftarrow [1 : a-]p, t$	EXT(6)	$[[s]]_1 = [2 : c]$
3.1.	$[1 : a-]p \leftarrow r$	INV(3)	
11.	$\leftarrow r, t$	RED(3.1)	$[[s]]_2 = [2 : c]$
12.	$\leftarrow p, [1 : a]t$	EXT(3)	$[[s]]_3 = [1 : a][2 : c]$
13.	$\leftarrow [1 : a]t$	RED(1)	$[[s]]_4 = [1 : a][2 : c]$
14.	$\leftarrow q$	RED(8)	$[[s]]_5 = [1 : a][2 : c]$
15.	$\leftarrow$	RED(2)	$[[s]]_6 = [1 : a][2 : c]$

6 Winning Strategy

A winning strategy [17] is an action sequence which leads a player to the victory independent of the actions of the other players.

In this section, it will be presented methods for propositional and first-order search of a winning strategy. The first is similar to ATL [1] and ATEL [12], [13]. The last one is similar to the Negation by Failure of SLD Resolution [15].

It is initially assumed that the games handled by these algorithms are strictly competitive and, afterwards, it will be shown extensions that allows dealing with coalitions.

6.1 Propositional Search of a Winning Strategy

Method When an action sequence that leads player i to make a proposition β true, we must check if this sequence is a winning strategy.

To check this, when reaching a clause where an opponent j performs an action, a new deduction is opened, with several changes:

- It's used only a subset of the rules, which doesn't contain β neither in their head, nor in their tail. Since the opponent would try to avoid β, it doesn't make sense to test these rules.
- The propositions that will be tested are the ones that aren't in the tail of clauses that contain β in their head.
- The facts are the propositions in the body of the objective clause, after the action of player i was performed.

If this deduction fail, the sequence initially found is a winning strategy. However, if this deduction succeed, it's necessary to verify if the player i can force β by changing his move. It can be performed by opening another deduction, with the full set of rules, and facts extracted from the propositions in the body of objective clause obtained after the action of previous opponent.

If a path leading to β is found, it must be checked whether the opponents can avoid β. This is a recursive procedure, with several nested deductions, whose effect is the verification of the sub-tree where the player i performs an action.

Modal interpretation of Winning Strategy operator The operators of Game Prolog language has a relationship with ATL [1] and ATEL ([12], [13]), as can be seen at table 5.

Table 5. Comparison between Game Prolog and ATL/ATEL

Game Prolog	ATL/ATEL	
$\{j\}\beta$	$\ll j \gg \Diamond\beta$	There is a subtree where player j can force β.
$[j : a]\beta$	$\ll j \gg \bigcirc\beta$	The player j can perform an action that makes β occur in the next state.

When a search for winning strategy for j is performed, the modality $[k : b]$, for all $k \neq j$, is related to expression $[[k]] \bigcirc \beta$, that asserts the impossibility of k avoid β, given the relation $[[k]] \bigcirc \beta \leftrightarrow \neg \ll k \gg \bigcirc\neg\beta$.

The operators $\ll j \gg \Diamond$ and $[[k]]\Diamond$ can be defined inductively in function of $\ll j \gg \bigcirc$ and $[[k]]\bigcirc$, as shown below. A similar definition relates the winning strategy operator $\{j\}$ to the action operator $[j : a]$.

– $\ll j \gg \Diamond\beta \leftrightarrow (\ll j \gg \bigcirc(\beta \vee ([[k_1]]\Diamond\beta \wedge ... \wedge [[k_n]]\Diamond\beta)))$
– $[[k_i]]\Diamond\beta \leftrightarrow ([[k_i]] \bigcirc (\ll j \gg \Diamond\beta \vee ([[k_1]]\Diamond\beta \wedge ... \wedge [[k_n]]\Diamond\beta)))$

Algorithms Below, the propositional version of algorithms are presented with a brief description.

Facts replacement The procedure of extracting the propositions from the body of objective clause obtained from an action of the previous player and replacing the original facts to these propositions means that the new deduction will be done from the scenery where the current player is doing the action instead of the initial state.

Function `searchWinningStrategy(`*rules, facts, player, proposition, subrules, subprops*`)`

```
Data   : rules is the full set of game rules;
         facts is the set of facts on the initial scenery of the game;
         player is that for whom we want to find a winning strategy;
         proposition is that we want to check whether the player can force;
         subrules is a subset of game rules that doesn't contain proposition;
         subprops is a subset of game propositions that aren't in tail of clauses
         that contain the tested proposition in their head;
begin
    while true do
        look for a path that leads proposition to the player;
        if there's no path then
            return false;
        failed := false;
        foreach (action in path) and not failed do
            if action was done by an opponent then
                extract the propositions in the tail of the clause used to
                perform the action;
                replace the facts in the game to the propositions extracted
                previously;
                if searchOpponentVictory(rules, facts, player, opponent,
                proposition, subrules, subprops) then
                    failed := true;
            end
        end
        if not failed then
            return true;
    end
end
```

Function ***searchWinningStrategy*** For a given path that leads player i to proposition β to be a winning strategy, it's necessary that the opponents can't avoid β.

On this way, for each action done by an opponent j, is performed a verification of non-existence of possibility of avoidance of β by player j, which is done by creating a new deduction where is done the replacement of facts by the propositions on the objective clause after the action of previous player.

Function ***searchOpponentVictory*** This function performs a search of a path where opponent can avoid β, by disabling clauses that contains β and propositions that are in the tail of clauses that contains β in their heads. This functions aims to verify whether even the opponent performing a different action, the player i can still force β.

Function `searchOpponentVictory`(*rules, facts, player, opponent, proposition, subrules, subprops*)

```
Data   : rules is the full set of game rules;
         facts is the set of facts on the initial scenery of the game;
         player is that for whom we originally wanted to find a winning
         strategy;
         opponent is a player for whom we'll check the existence of a winning
         strategy;
         proposition is that we originally wanted to check whether the player
         can force;
         subrules is a subset of game rules that doesn't contain proposition;
         subprops is a subset of game propositions that aren't in tail of clauses
         that contain the tested proposition in their head;
begin
    foreach prop in subprops do
        look for a path that leads prop to the opponent, using the rules in
        subrules;
        if there's no path then
            continue;
        failed := false;
        foreach (action in path) and not failed do
            if action is done by player then
                extract the propositions in the tail of the clause used to
                perform the action;
                replace the facts in the game to the propositions extracted
                previously;
                if searchWinningStrategy(rules, facts, player, proposition,
                subrules, subprops) then
                    failed := true;
            end
        end
        if not failed then
            return true;
    end
    return false;
end
```

Informal description of the proof In order to show the soundness of the algorithms of winning strategy search, it is assumed that Modal Prolog [4], [22] is correct.

The execution of the functions above performs a backward induction [17], since function `searchWinningStrategy` calls function `searchOpponentVictory` to verify whether an opponent can prevent current player from obtaining the desired result and, conversely, to check if current player can enforce a result, in spite of the opponent's trials to prevent this.

This method can be demonstrated by induction on the number of actions in the verified path:

Base In a path with one action, the player i has a winning strategy if it is his/her action and (s)he can reach the desired proposition, or if it is an opponent action and for every possible action available for him/her, the proposition will be obtained. Otherwise, there is not a winning strategy.

Induction hypotesis In a path with k actions, the algorithm replies correctly whether it is a winning strategy or not.

Induction step We want to verify if in a path with $k + 1$ actions the algorithm continues to reply correctly. Then, there are two cases:

- If in the path with the last k actions, (s)he does not have winning strategy, (s)he would not have winning strategy in the $(k + 1)^{th}$ action, because the algorithm replied correctly in the path with k actions by induction hypotesis.
- If the player i has a winning strategy in the path with k actions, for the $(k + 1)^{th}$ action there are two possibilities:
  - If this action is done by player i, trivially this path is also a winning strategy, by performing the specified action.
  - If this action is done by an opponent, a call to `searchOpponentVictory` will be done, in order to try to discover a path which hinders player i to get the desired proposition. Again, there are two cases:
    - If such path does not exist, player i has a winning strategy.
    - If there is a path, it will be tested the existence of winning strategy in the states where player i makes an action, by calling the function `searchWinningStrategy`. These functions will be called recursively until a call to `searchWinningStrategy` reaches paths with length less or equal k actions, for which replies correctly whether exists winning strategy or not, by induction hypotesis. If for every tested path, player i can force the desired proposition, then player i has a winning strategy.

Examples

Success As indicated in figure 4, a search for a path that leads player 1 to the victory is performed, returning the action sequence $[1 : A_1][2 : B_1][1 : A_1]$. However, we must check whether $[1 : A_1]$ is a winning strategy.

In order to tackle this, the states where is the turn of an opponent (in this case, only the state where it would have been played B_1) are verified to check if (s)he can win or draw.

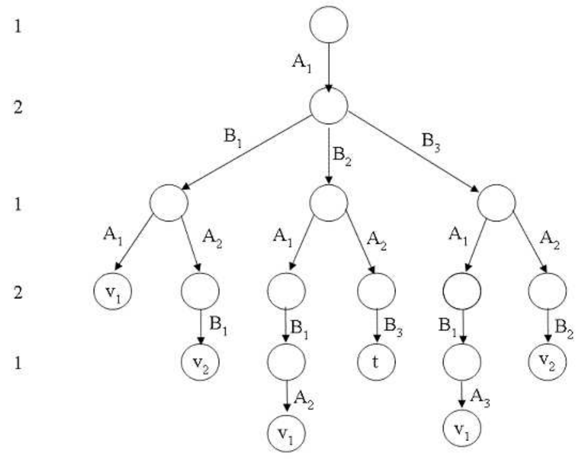

Fig. 4. Example of successful search

Search for opponent's victory The first candidate would be the path $[2 : B_1][1 : A_2][2 : B_1]$. But in the turn of player 1, it can do $[1 : A_1]$ and win. Thus, this possibility is discarded.

Another possible path is $[2 : B_3][1 : A_2][2 : B_2]$. However, player 1 can play A_1, and then perform the path $[1 : A_1][2 : B_1][1 : A_3]$ and win, without the possibility of player 2 do another action instead of B_1. Since there are no more possibilities for player 2, this search failed.

Search for tie There's a way for a tie if the action sequence $[2 : B_2][1 : A_3][2 : B_3]$ would be done. However, player 1 can perform A_1 and force the path $[1 : A_1][2 : B_1][1 : A_2]$, winning the game.

Since neither the victory of player 2, nor a tie is possible, this sub-tree is a winning strategy.

Failure With small changes in the presented sub-tree, as shown in figure 5, a winning strategy cease to exist. Again, there is the initial action sequence $[1 : A_1][2 : B_1][1 : A_1]$, with victory of player 1.

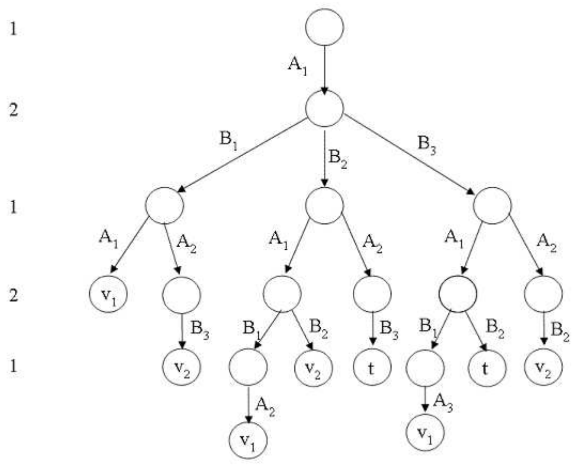

Fig. 5. Example of failed search

Search for opponent's victory To the victory of the opponent, there's the path $[2 : B_2][1 : A_1][2 : B_2]$. The only way for player 1 to avoid this path is execute A_2, but it will result in a tie. Thus, there is no more winning strategy.

Search for tie Two paths that leads to a tie can be verified: $[2 : B_2][1 : A_2][2 : B_3]$ and $[2 : B_3][1 : A_1][2 : B_2]$. In both cases, player 1 can't make another action without giving a winning strategy to the opponent.

In practice, it's enough that one of this cases holds in order to eliminate the possibility of existence of a winning strategy for a specific player.

6.2 First-Order Search of a Winning Strategy

Method The first-order versions of the algorithms for searching winning strategies have some differences from propositional versions initially presented. When an opponent j performs an action, the new deduction is done on this way:

- It's used the full set of rules, as it occurs when doing the evaluation for player i.
- It's tested only the predicate passed originally, and it's verified whether the opponent can force a different instantiation of the predicate parameters.
- The facts are the predicates in the body of the objective clause, after the action of player i was performed. The instantiation of the predicates variables, between the initial scenery and this scenery is kept, meaning that the path previously covered remained fixed.

Since a variable can't be unified with more than one value at the same time, we can assume that ground instatiations of predicates are mutually exclusive, As an example, the predicate $win(X)$ for players 1 and 2, could be instantiated on these manners: $win(1)$, $win(2)$ and $win(tie)$.

Although this method is more general than propositional one, the former is used for efficiency, since predicate may have different instantiations of its variables. Furthermore, this method requires mutual exclusion between the tested terms (ground predicates or propositions).

Algorithms Below, the first-order version of algorithms are presented with a brief description.

Generation of combinations of constants The combinations which will be tested in the function `searchOpponentVictory` are generated from the constants existent in the entrance program. Thus, the amount of combinations will be finite, although it can grow exponentially according to the number of parameters of the tested predicate.

Fact replacement The procedure of extracting the predicates from the body of objective clause obtained from an action of the previous player and replacing the original facts to these predicates means that the new deduction will be done from the scenery where the current player is doing the action instead of the initial state. The freezing of unification performed on the previous deduction asserts that the previous path can't be undone.

The procedure to perform the fact replacement in the First-Order mode is now presented:

1. Extract the predicates of objective clause where the player has done an action.

2. Undo the unifications performed between the current objective clause and the empty clause (Note: It is possible to exist variables inside the generated facts).

3. Store the actions performed between the current objective clause and the empty clause, and the facts of previous initial scenery.

4. After the new deduction, verify the instantiations of the obtained facts.

4.1. Assign the new instantiations to the objective clause that had originated the facts of the new deduction.

4.2. Perform the deduction of this clause to the empty clause, using the previously stored set of facts and verify whether the deduction succeeds and the action sequence is the same to the sequence obtained initially.

Function `searchWinningStrategy` For a given path that leads player i to predicate $\beta(\alpha_1, ..., \alpha_n)$ to be a winning strategy, it's necessary to be sure that the opponents can't avoid $\beta(\alpha_1, ..., \alpha_n)$.

On this way, for each action done by an opponent j, it is verified whether (s)he cannot avoid $\beta(\alpha_1, ..., \alpha_n)$, which is done by creating a new deduction where is done the replacement of facts by the propositions on the objective clause after the action of previous player.

Function ***searchOpponentVictory*** This function performs a search of a path where opponent can avoid $\beta(\alpha_1, ..., \alpha_n)$, by testing the $\beta(\gamma_1, ..., \gamma_n)$ predicate with $(\gamma_1, ..., \gamma_n) \neq (\alpha_1, ..., \alpha_n)$. This function aims to verify whether even the opponent performing a different action, the player i can still force $\beta(\alpha_1, ..., \alpha_n)$.

Function searchWinningStrategy(*rules, facts, player, predicate, parameters, constants, previousActions, previousFacts*)

```
Data   : rules is the full set of game rules;
         facts is the set of facts on the initial scenery of the game;
         player is that for whom we want to find a winning strategy;
         predicate is that we want to check whether the player can force the desired
         values;
         parameters is that we want to check whether the player can force in the
         predicate;
         constants is a set of sets of constants which can occur in the predicate
         parameters;
         previousActions is the action sequence between the objective clause that
         generated the facts and empty clause;
         previousFacts is the set of facts used on the previous deduction;
begin
    while true do
        look for a path that leads the player to the predicate with parameters;
        if there's no path then
            return false;

        else if previousActions and previousFacts are not null then
            make an objective clause using the facts with the new instantiation;
            make another deduction using previousFacts as initial scenery;
            if deduction failed or (obtained action sequence $\neq$ previousActions) then
                return false;

        failed := false;
        foreach (action in path) and not failed do
            if action was done by an opponent then
                extract the predicates in the tail of the clause used to perform the
                action;
                newFacts := The predicates extracted previously;
                initActions := The actions performed between the current objective
                clause and the empty clause;
                if searchOpponentVictory(rules, newFacts, player, opponent,
                predicate, parameters, constants, initActions, facts) then
                    failed := true;
            end
        end
        if not failed then
            return true;
    end
end
```

Function `searchOpponentVictory(`*rules, facts, player, opponent, predicate, parameters, constants, previousActions, previousFacts*`)`

```
Data : rules is the full set of game rules;
       facts is the set of facts on the initial scenery of the game;
       player is that for whom we originally wanted to find a winning
       strategy;
       opponent is a player for whom we'll check the existence of a winning
       strategy;
       predicate is that we want to check whether the player can force the
       desired values;
       parameters is that we originally wanted to check whether the player
       can force in the predicate;
       constants is a set of sets of constants which can occur in the predicate
       parameters;
       previousActions is the action sequence between the objective clause
       that generated the facts and empty clause;
       previousFacts is the set of facts used on the previous deduction;
begin
    foreach combination of constants different from parameters do
        look for a path that leads the the opponent to the predicate with the
        combination;
        if there's no path then
            continue;

        else if previousActions and previousFacts are not null then
            make an objective clause using the facts with the new
            instantiation;
            make another deduction using previousFacts as initial scenery;
            if deduction failed or (obtained action sequence ≠
            previousActions) then
                continue;

        failed := false;
        foreach (action in path) and not failed do
            if action is done by a member of the coalition then
                extract the predicates in the tail of the clause used to perform
                the action;
                newFacts := The predicates extracted previously;
                initActions := The actions performed between the current
                objective clause and the empty clause;
                if searchWinningStrategy(rules, newFacts, player, predicate,
                parameters, constants, initActions, facts) then
                    failed := true;
            end
        end
        if not failed then
            return true;
    end
    return false;
end
```

Informal description of the proof In the same way of propositional mode, the execution of the functions above performs a backward induction [17], by calling recursively the functions `searchWinningStrategy` and `searchOpponentVictory`.

6.3 Extending the Winning Strategy search for coalitions

In games with more than two players is possible the formation of coalitions in order to a subset of players reach a result which wouldn't reach acting separately. Recently, logics for dealing with coalitions have been proposed [20].

This section presents an extension of the algorithms introduced in the previous section to deal with coalitions.

Required changes In order to allow the formation of coalitions, some changes are required:

– The winning strategy operator starts to accept more than one player, separated by commas, as in $\{p, q\}$
– When testing if an opponent can avoid the victory of the current player, and thereby, of coalition, check also if, given the opponent's action, another member of coalition, instead of only the current player, can reach the victory.

Correspondence with ATL/ATEL On the same way of the strictly competitive case, given a coalition with members $\{c_1, ..., c_j\}$, we have the relation $\{c_1, ..., c_j\}\beta \equiv \ll c_1, ..., c_j \gg \Diamond\beta$.

The inductive definition of operators $\ll C \gg \Diamond\beta$ and $[[\Sigma \setminus C]]\Diamond\beta$ become more complex, due to the fact that, in the verification of the possibility of a non-member of coalition to avoid β, to be necessary to check the action of all coalition members in the direction of force β, instead of a single player.

– $\ll C \gg \Diamond\beta \leftrightarrow (\bigvee_{i \in C} \ll i \gg \bigcirc(\beta \vee \bigvee_{j \in C} \ll j \gg \Diamond\beta \vee \bigwedge_{k \notin C} [[k]]\Diamond\beta))$
– $[[q]]\Diamond\beta \leftrightarrow ([[q]] \bigcirc (\bigvee_{j \in C} \ll j \gg \Diamond\beta \vee \bigwedge_{k \notin C} [[k]]\Diamond\beta))\ (q \notin C)$

Changed Algorithms In all algorithms, an extra parameter is inserted, containing the members of coalition from which the player belongs. Both versions of algorithms ought to handle winning strategies for all members of coalition.

The first-order versions of functions `searchWinningStrategy` and `searchOpponentVictory` are now written on this way:

Function `searchWinningStrategy`(*rules, facts, player, coalition, predicate, parameters, constants, previousActions, previousFacts*)

```
Data   : rules is the full set of game rules;
         facts is the set of facts on the initial scenery of the game;
         player is that for whom we want to find a winning strategy;
         coalition is that from the player belongs;
         predicate is that we want to check whether the player can force the
         desired values;
         parameters is that we want to check whether the player can force in
         the predicate;
         constants is a set of sets of constants which can occur in the predicate
         parameters;
         previousActions is the action sequence between the objective clause
         that generated the facts and empty clause;
         previousFacts is the set of facts used on the previous deduction;
begin
    while true do
        look for a path that leads the player to the predicate with parameters;
        if there's no path then
            return false;

        else if previousActions and previousFacts are not null then
            make an objective clause using the facts with the new
            instantiation;
            make another deduction using previousFacts as initial scenery;
            if deduction failed or (obtained action sequence $\neq$
            previousActions) then
                return false;

        failed := false;
        foreach (action in path) and not failed do
            if action was done by a non-member of coalition then
                extract the predicates in the tail of the clause used to perform
                the action;
                newFacts := The predicates extracted previously;
                initActions := The actions performed between the current
                objective clause and the empty clause;
                if searchOpponentVictory(rules, newFacts, player, coalition,
                opponent, predicate, parameters, constants, initActions, facts)
                then
                    failed := true;
            end
        end
        if not failed then
            return true;
    end
end
```

Function `searchOpponentVictory`(*rules, facts, player, coalition, opponent, predicate, parameters, constants, previousActions, previousFacts*)

```
Data   : rules is the full set of game rules;
         facts is the set of facts on the initial scenery of the game;
         player is that for whom we originally wanted to find a winning strategy;
         coalition is that from the player belongs;
         opponent is a player for whom we'll check the existence of a winning strategy;
         predicate is that we want to check whether the player can force the desired
         values;
         parameters is that we originally wanted to check whether the player can force
         in the predicate;
         constants is a set of sets of constants which can occur in the predicate
         parameters;
         previousActions is the action sequence between the objective clause that
         generated the facts and empty clause;
         previousFacts is the set of facts used on the previous deduction;
begin
    foreach combination of constants different from parameters do
        look for a path that leads the the opponent to the predicate with the
        combination;
        if there's no path then
            continue;

        else if previousActions and previousFacts are not null then
            make an objective clause using the facts with the new instantiation;
            make another deduction using previousFacts as initial scenery;
            if deduction failed or (obtained action sequence ≠ previousActions) then
                continue;

        failed := false;
        foreach (action in path) and not failed do
            if action is done by a member of the coalition then
                extract the predicates in the tail of the clause used to perform the
                action;
                newFacts := The predicates extracted previously;
                initActions := The actions performed between the current objective
                clause and the empty clause;
                if searchWinningStrategy(rules, newFacts, player, coalition, predicate,
                parameters, constants, initActions, facts) then
                    failed := true;
            end
        end
        if not failed then
            return true;
    end
    return false;
end
```

7 Example of first-order modelling

In this section, the game *Cluedo* [10], [2] is represented in Game Prolog clauses.

7.1 Description

In this game, the players have to solve a murder happened in a mansion, by discovering the murderer, the weapon and the room where the crime took place.

Are considered suspects of the murder the guests who had been in the house during the weekend. In the beginning, three cards (suspect, weapon, room) are removed from the set of cards.

Each player will receive some cards, which cannot be seen by the other players. During the game, the players will have several chances to make an assumption on the murderer's cards, by questioning the other players whether they hold one of the asked cards.

The possible actions are enumerated on the table 6.

Table 6. Possible actions on *Cluedo*

[P:ask(S, W, R)]	The player P ask the others if they posses the suspect S or the weapon W or the room R.
[Q:nonshow]	The player Q replies it doesn't hold any of the cards previously asked.
[Q:show(P, C)]	The player Q shows to player P the card C.
[P:accuse(S, W, R)]	Player P affirms that suspect S, weapon W and room R are the murderer cards.
[P:pass]	Player P simply pass its turn.
[0:success]	This action asserts that an accusation is correct. Since the murderer's cards are assumed to belong to player 0, the player is fixed here.

A definition of auxiliary predicates, as defined in table 7, will be needed in the modelling of *Cluedo*.

7.2 Building the clauses

After having specified what predicates would be needed in the *Cluedo* modelling, the assembly of the clauses can be started. There must be a special attention on the predicates which should be passed to the next sceneries.

Initial scenery In the initial scenery, the distribution of the cards (six suspects, six weapons and nine rooms) among the players will be done, setting aside the cards assigned to the murderer (player 0). A clause *maximum*, with the number of players is also inserted. These facts ought to be valid in any scenery.

The turn is initially assigned to player 1, and the clauses about the knowledge are made on this way:

Table 7. Auxiliaries predicates

maximum(P)	This predicate keeps the maximum ID player.
card(P, T, V)	Player P holds the card of type T and value V. It will be assumed that the murderer's cards belongs to agent 0.
mayBe(P, T, V)	Player P believes that card with type T and value V may belong to the murderer.
turn(P)	This predicate asserts it is the turn of player P
toReply(P, Q, S, W, R)	It is the time to player Q reply to player P whether holds the suspect S or weapon W or room R.
finishedReplies(P)	All the other players replied to player P the action *ask* by the actions *show* or *nonshow*
accusation(P, S, W, R)	This predicate holds an accusation of player P over suspect S, weapon W and room R.

– If a player doesn't hold a card $card(player, Type, Value)$, it can't be sure whether a specific card is the murderer's one or not. Thus, a fact $mayBe(player, Type, Value)$ is inserted.
– Otherwise, the player already knows this is not the murderer's card. Then, it is not necessary to create a fact to represent this.

Rules

Procedure increase This procedure is made just to pass the turn of players.
$increase(MaxPlayer, 1) \leftarrow maximum(MaxPlayer).$
$increase(Player, NextPlayer) \leftarrow NextPlayer$ is $Player + 1.$

ask(Player, Suspect, Weapon, Room) The action ask, performed inside a turn of a player, starts a round of replies of the other players about the asked cards, which the current player is not sure whether or not they are the murderer's cards. The knowledge of all players about the cards remains unchanged.
$[P : ask(S, W, R)]\ toReply(P, 1, S, W, R) \leftarrow turn(P),\ mayBe(P, suspect, S),$ $mayBe(P, weapon, W),\ mayBe(P, room, R).$
$[P : ask(S, W, R)]\ turn(P) \leftarrow turn(P).$
$[P : ask(S, W, R)]\ mayBe(Q, T, V) \leftarrow mayBe(Q, T, V).$

nonshow This action is performed as a reply of *ask(P, S, W, R)*, when the player doesn't have any card among the asked ones. The knowledge of the players also remains unchanged.
$[Q : nonshow]\ finishedReplies(P) \leftarrow maximum(Q), toReply(P, Q, S, W, R),$ NOT $card(Q, suspect, S)$, NOT $card(Q, weapon, W)$, NOT $card(Q, room, R)$.
$[Q : nonshow]\ finishedReplies(P) \leftarrow maximum(P),\ increase(Q, P),$ $toReply(P, Q, S, W, R)$, NOT $card(Q, suspect, S)$, NOT $card(Q, weapon, W)$, NOT $card(Q, room, R)$.

$[Q : nonshow]$ $toReply(P, NQ, S, W, R) \leftarrow increase(Q, NQ)$, $toReply(P, Q, S, W, R)$, NOT $card(Q, suspect, S)$, NOT $card(Q, weapon, W)$, NOT $card(Q, room, R)$.

$[Q : nonshow]$ $mayBe(P, T, V) \leftarrow mayBe(P, T, V)$.

$[Q : nonshow]$ $turn(P) \leftarrow turn(P)$.

show(Player, Card) If player Q has at least one of the asked cards, it will show one of them. The knowledge of player P will change after this action, since the shown card could not be of the murderer.

% These clauses are for the reply of the last player.

$[Q : show(P, S)]$ $finishedReplies(P) \leftarrow maximum(Q), toReply(P, Q, S, W, R)$, $card(Q, suspect, S)$.

$[Q : show(P, W)]$ $finishedReplies(P) \leftarrow maximum(Q), toReply(P, Q, S, W, R)$, $card(Q, weapon, W)$.

$[Q : show(P, R)]$ $finishedReplies(P) \leftarrow maximum(Q), toReply(P, Q, S, W, R)$, $card(Q, room, R)$.

% The asker player may be the last. In this case, when the player N - 1 replies, the cycle of replies should be finished.

$[Q : show(P, S)]$ $finishedReplies(P) \leftarrow maximum(P)$, $increase(Q, P)$, $toReply(P, Q, S, W, R)$, $card(Q, suspect, S)$.

$[Q : show(P, W)]$ $finishedReplies(P) \leftarrow maximum(P)$, $increase(Q, P)$, $toReply(P, Q, S, W, R)$, $card(Q, weapon, W)$.

$[Q : show(P, R)]$ $finishedReplies(P) \leftarrow maximum(P)$, $increase(Q, P)$, $toReply(P, Q, S, W, R)$, $card(Q, room, R)$.

% This is the general case.

$[Q : show(P, S)]$ $toReply(P, NQ, S, W, R) \leftarrow increase(Q, NQ)$, $toReply(P, Q, S, W, R)$, $card(Q, suspect, S)$.

$[Q : show(P, W)]$ $toReply(P, NQ, S, W, R) \leftarrow increase(Q, NQ)$, $toReply(P, Q, S, W, R)$, $card(Q, weapon, W)$.

$[Q : show(P, R)]$ $toReply(P, NQ, S, W, R) \leftarrow increase(Q, NQ)$, $toReply(P, Q, S, W, R)$, $card(Q, room, R)$.

% The beliefs of the current player are changed after the reply. All other knowledge remains unchanged.

$[Q : show(P, C)]$ $mayBe(P, T, V) \leftarrow C <> V$, $mayBe(P, T, V)$.

$[Q : show(P, C)]$ $turn(P) \leftarrow turn(P)$.

$[Q : show(P, C)]$ $mayBe(R, T, V) \leftarrow R <> P$, $mayBe(R, T, V)$.

pass Here the player just pass its turn, keeping the knowledge unchanged.

$[P : pass]$ $turn(NP) \leftarrow finishedReplies(P)$, $increase(P, NP)$.

$[P : pass]$ $mayBe(Q, T, V) \leftarrow mayBe(Q, T, V)$.

accuse(Player, Suspect, Weapon, Room) If the player is sure, (s)he can make the accusation. This is the case where there are only one suspect, one weapon and one room that may be the murderer's card.

$[P : accuse(S, W, R)]\ accusation(P, S, W, R) \leftarrow turn(P), finishedReplies(P), mustBe(P, suspect, S), mustBe(P, weapon, W), mustBe(P, room, R).$

The auxiliary procedure mustBe(Player, Type, Value) This procedure is performed to test whether there are only one card of a given type that could be the murderer one.

$mustBe(P, T, V) \leftarrow mayBe(P, T, V)$, NOT $thereIsOther(P, T, V)$.
$thereIsOther(P, T, V) \leftarrow mayBe(P, T, W),\ V <> W.$

success This is a simple action, which tests whether an accusation is correct or not.

$[0 : success]\ win(P) \leftarrow accusation(P, S, W, R), card(0, suspect, S), card(0, weapon, W), card(0, room, R).$

8 Conclusion

This work proposes the use of a language based on modal Horn clauses, in order to specify Extensive Games. This language is an extension Modal Prolog. Some new features, like explicit player specification and operators of choice and composition were inserted just to make notation more practical, but Converse Modality and Winning Strategy operator are not originally inside Modal Prolog.

Since there are games where is desirable the knowledge of past actions, and what has held before them, the Converse Modality is developed to provide this behavior.

An important feature of Game Prolog is the operator of Winning Strategy. In this paper, it was presented algorithms that verify if a player or coalition can force a result in spite of the other players of the game.

There are still more features that may be incorporated in Game Prolog, like payoff evaluation. In particular, a "weaker" search of strategy, returning a "reasonable" action sequence, instead of the optimal, which requires too much processing, may be present on the next extensions of Game Prolog.

References

1. R. Alur, T. A. Henzinger, O. Kupferman, *Alternating-time Temporal Logic*, In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS).* pp.100–109, 1997
2. M. Benevides, M. Carlini, C. Delgado, *Specification of Knowledge-Based Multi-Agent Systems*, to appear.
3. M. Benevides, R. Ribeiro, *Game Prolog* (Extended Abstract). *Sixth Conference on Logic and the Foundations of Game and Decision Theory (LOFT 6)*, Leipzig, Germany, July 2004.

4. M. Benevides, O. T. Rodrigues, *PROMAL: Programming in Modal Action Logic*. In: *Proceeding of the 12th Brazilian Symposium on Artificial Intelligence*. pp. 101–111, October, 1995.
5. J. van Benthem, *Games in Dynamic-Epistemic Logic*. In: G. Bonanno & W. van der Hoek, eds., *Bulletin of Economic Research 53:4*, pp 219–248 (Proceedings LOFT–4, Torino). June, 2000.
6. P. Blackburn, M. de Rijke, and Y. Venema, *Modal Logic*, Cambridge, Cambridge University Press, 2001.
7. G. Bonanno, *Branching Time Logic, Perfect Information Games and Backward Induction*. In: *Games and Economic Behavior*, 36 (1), pp. 57–73 July, 2001.
8. G. Bonanno, *Memory of past beliefs and actions*, *Studia Logica*, 75 (1), pp. 7–30, October 2003.
9. G. Bonanno, *Memory and perfect recall in extensive games*, *Games and Economic Behavior*, 47 (2), pp. 237–256, May 2004.
10. Cluedo Fan, *The Original Rules of Cluedo* [online]. Available on Internet via WWW. URL: http://www.cluedofan.com/origrule.htm. Retrieved in September 28th, 2004.
11. B. P. Harrenstein, W. van der Hoek, J. J. Ch. Meyer, Cees Witteveen *On Modal Logic Interpretations of Games*, in F. van Harmelen (ed.), *ECAI 2002, 15th European Conference on Artificial Intelligence*, IOS Press, Amsterdam, pages 28–32. July, 2002.
12. W. van der Hoek, M. J. Wooldridge, *Tractable Multi Agent Planning for Epistemic Goals*, in C. Castelfranchi and W.L. Johnson (eds), *Proceedings of the First Internation Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, ACM Press, New York, USA, pp. 1167–1174, 2002.
13. W. van der Hoek, M. Wooldridge, *Cooperation, Knowledge and Time: Alternating-time Temporal Epistemic Logic and its Applications*, *Studia Logica*, 75:1, pp. 125–157, 2003.
14. D. Leivant, *Logics of Programs* [online]. Available on Internet via WWW. URL: http://www.cs.indiana.edu/classes/b619/pdl.pdf. Retrieved in March 18th, 2004.
15. J. W. Lloyd, *Foundations of Logic Programming*, Berlin, Springer-Verlag, 1984.
16. A. S. Machado, *MProlog - Compilador Prolog com Modalidade de Ação*. Undergraduation Project, IM/UFRJ, Rio de Janeiro, Brazil, 1995. In Portuguese.
17. M. J. Osborne, A. Rubinstein, *A Course in Game Theory*, 4th Printing, Massachussets, The MIT Press, 1994.
18. R. Parikh, *The Logic of Games and its applications*, *A. Discrete Mathematics* 24, pp 111–140, 1985.
19. M. Pauly, *Game logic for game theorists*. In: *Report from Centrum voor Wiskunde en Informatica* September, 2000.
20. M. Pauly, *Logic for Social Software*. Ph.D. Thesis, *Universiteit van Amsterdam*, Amsterdam, Netherlands, 2001.
21. M. Pauly, *Some Game Theory for Logicians*. *University of Amsterdam*, 2003.
22. O. T. Rodrigues, *Prolog Modal de Ação e Revisão de Crenças em Conjuntos Definidos*. M.Sc. Thesis, COPPE/UFRJ, Rio de Janeiro, Brazil, 1993. In Portuguese.