I N F S Y S

R E S E A R C H

R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG WISSENSBASIERTE SYSTEME

GAME-THEORETIC AGENT PROGRAMMING IN GOLOG

ALBERTO FINZI THOMAS LUKASIEWICZ

INFSYS RESEARCH REPORT 1843-04-02

MAY 2004

Institut für Informationssysteme
Abtg. Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TECHNISCHE UNIVERSITÄT WIEN

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 1843-04-02, MAY 2004

GAME-THEORETIC AGENT PROGRAMMING IN GOLOG

(PRELIMINARY VERSION, JULY 26, 2004)

Alberto Finzi[1] Thomas Lukasiewicz[1 2]

Abstract. We present the agent programming language GTGolog, which integrates explicit agent programming in Golog with game-theoretic multi-agent planning in Markov games. It is a generalization of DTGolog to a multi-agent setting, where we have two competing single agents or two competing teams of agents. The language allows for specifying a control program for a single agent or a team of agents in a high-level logical language. The control program is then completed by an interpreter in an optimal way against another single agent or another team of agents, by viewing it as a generalization of a Markov game, and computing a Nash strategy. We illustrate the usefulness of this approach along a robotic soccer example. We also report on a first prototype implementation of a simple GTGolog interpreter.

[1]Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I-00198 Rome, Italy; e-mail: {finzi, lukasiewicz}@dis.uniroma1.it.

[2]Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; e-mail: lukasiewicz@kr.tuwien.ac.at.

Acknowledgements: This work was partially supported by the Austrian Science Fund under project Z29-N04 and by a Marie Curie Individual Fellowship of the European Union programme "Human Potential" under contract number HPMF-CT-2001-001286 (disclaimer: The authors are solely responsible for information communicated and the European Commission is not responsible for any views or results expressed).

Contents

1 Introduction **1**

2 Preliminaries **2**
2.1 Situation Calculus and Golog 2
2.2 Matrix Games 3
2.3 Markov Games 3

3 Game-Theoretic Golog **4**
3.1 Domain Theory 4
3.2 Syntax 5
3.3 Semantics 6
3.4 Representation and Optimality Results 8

4 Soccer Example **9**

5 Teams **10**

6 Implementation **11**
6.1 GTGolog Interpreter 11
6.2 Soccer Example 15

7 Summary and Outlook **18**

1 Introduction

During the recent decades, the development of controllers for autonomous agents has become increasingly important in AI. One way of designing such controllers is the programming approach, where a control program is specified through a language based on high-level actions as primitives. Another way is the planning approach, where goals or reward functions are specified and the agent is given a planning ability to achieve a goal or to maximize a reward function.

Recently, an integration of the programming and the planning approach has been suggested through the language DTGolog [2], which integrates explicit agent programming in Golog [11] with decision-theoretic planning in Markov decision processes (MDPs) [9]. DTGolog allows for partially specifying a control program in a high-level language as well as for optimally filling in missing details through decision-theoretic planning. It can thus be seen as a decision-theoretic extension to Golog, where choices left to the agent are made by maximizing expected utility. From a different perspective, DTGolog can also be seen as a formalism that gives advice to a decision-theoretic planner, since it naturally constrains the search.

The agent programming language DTGolog, however, is designed only for the single-agent framework. That is, the model of the world essentially consists of a single agent that we control by a DTGolog program and the environment that is summarized in "nature". But in realistic applications, we often encounter multiple agents, which may compete against each other, or which may also cooperate with each other. For example, in robotic soccer, we have two competing teams of agents, where each team consists of cooperating agents. Here, the optimal actions of one agent generally depend on the actions of all the other ("enemy" and "friend") agents. In particular, there is a bidirected dependence between the actions of two different agents, which generally makes it inappropriate to model enemies and friends of the agent that we control simply as a part of "nature".

Game theory [13] deals with optimal decision making in the multi-agent framework with competing and cooperating agents. In particular, Markov games [12, 5], also called stochastic games [7], generalize matrix games from game theory, and are multi-agent generalizations of MDPs with competing agents.

In this paper, we present a combination of explicit agent programming in Golog with game-theoretic multi-agent planning in Markov games. The main contributions of this paper are as follows:

- We define the language GTGolog, which integrates explicit agent programming in Golog with game-theoretic multi-agent planning in Markov games. GTGolog is a generalization of DTGolog [2] to a multi-agent setting, which allows for modeling two competing agents as well as two competing teams of cooperative agents.

- The language GTGolog allows for specifying a control program for a single agent or a team of agents, which is then completed by an interpreter in an optimal way against another single agent or another team of agents, by viewing it as a generalization of a Markov game, and computing a Nash equilibrium.

- We show that GTGolog generalizes Markov games. That is, GTGolog programs can represent Markov games, and a GTGolog interpreter can be used to compute their Nash equilibria. We also show that the GTGolog interpreter is optimal in the sense that it computes a Nash equilibrium of GTGolog programs.

- A robotic soccer example gives evidence of the usefulness of our approach in realistic applications. We also report on a first prototype implementation of a simple GTGolog interpreter.

The work closest in spirit to this paper is perhaps Poole's one [8], which shows that the independent choice logic can be used as a formalism for logically encoding games in extensive and normal form. Our view in this paper, however, is much different, as we aim at using game theory for optimal agent control in multi-agent systems.

2 Preliminaries

In this section, we recall the basic concepts of the situation calculus and Golog, of matrix games, and of Markov games.

2.1 Situation Calculus and Golog

The situation calculus [6, 11] is a first-order language for representing dynamic domains. Its main ingredients are *actions*, *situations*, and *fluents*. A situation is a first-order term encoding a sequence of actions. It is either a constant symbol or of the form $do(a, s)$, where a is an action and s is a situation. The constant symbol S_0 is the *initial situation*, and represents the empty sequence, while $do(a, s)$ encodes the sequence obtained from executing a after the sequence encoded in s. A *fluent* represents a world or agent property that may change when executing an action. It is a predicate symbol whose last argument is a situation. E.g., $at(pos, s)$ may express that an agent is at position pos in situation s.

In the situation calculus, a dynamic domain is expressed by a *basic action theory* $BAT = (\Sigma, \mathcal{D}_{S_0}, \mathcal{D}_{ssa}, \mathcal{D}_{una}, \mathcal{D}_{ap})$, where

- Σ is the set of foundational axioms for situations;
- $\mathcal{D}_{una}$ is the set of unique name axioms for actions, which express that different action terms stand for different actions;
- $\mathcal{D}_{S_0}$ is a set of first-order formulas describing the initial state of the domain (represented by S_0). For example, $at(\boldsymbol{a}, 1, 2, S_0) \wedge at(\boldsymbol{o}, 3, 4, S_0)$ may express that the agents $\boldsymbol{a}$ and $\boldsymbol{o}$ are initially at the positions $(1, 2)$ and $(3, 4)$, respectively;
- $\mathcal{D}_{ssa}$ specifies the *successor state axioms* [10, 11]: for each fluent $F(\vec{x}, s)$, one has an axiom of the form $F(\vec{x}, do(c, s)) \equiv \Phi_F(\vec{x}, c, s)$ providing both action effects and a solution to the frame problem (assuming deterministic actions). For example,

$$\begin{aligned} at(o, x, y, do(a, s)) \equiv a = moveTo(o, x, y) \vee \\ at(o, x, y, s) \wedge \neg(\exists x', y') a = moveTo(o, x', y')\,; \end{aligned} \tag{1}$$

- $\mathcal{D}_{ap}$ represents the action precondition axioms: for each simple action a, one has an axiom $Poss(a(\vec{x}), s) \equiv \Pi(\vec{x}, s)$. For example, $Poss(moveTo(o, x, y), s) \equiv \neg(\exists x', y', o') at(o', x', y', s)$.

Golog is an agent programming language that is based on the situation calculus. It allows for constructing complex actions from the primitive actions defined in a basic action theory BAT, where standard (and not so-standard) Algol-like control constructs can be used, in particular, (i) action sequences: $p_1; p_2$, (ii) tests: $\phi?$, (iii) nondeterministic action choices: $p_1|p_2$, (iv) nondeterministic choices of action argument: $(\pi x).p(x)$, and (v) conditionals, while loops, and procedure calls. An example of a Golog program is

$$\textbf{while } \neg at(\boldsymbol{a}, 1, 2) \textbf{ do } (\pi x, y) moveTo(\boldsymbol{a}, x, y).$$

Intuitively, the nondeterministic choice $(\pi x, y)moveTo(\boldsymbol{a}, x, y)$ is iterated until the agent $\boldsymbol{a}$ is at the position $(1, 2)$.

The semantics of a Golog program σ is specified by a situation-calculus formula $Do(\sigma, s, s')$, which encodes that s' is a situation which can be reached from s by executing σ. Thus, Do represents a macro expansion to a situation calculus formula. For example, the action sequence is defined through $Do(p_1; p_2, s, s') = \exists s''(Do(p_1, s, s'') \wedge Do(p_2, s'', s'))$. For more details on the core situation calculus and Golog, we refer the reader to [11].

2.2 Matrix Games

We briefly recall two-player matrix games from classical game theory [13]. Intuitively, they describe the possible actions of two agents and the rewards that they receive when they simultaneously execute one action each. Formally, a *two-player matrix game* $G = (A, O, R_{\boldsymbol{a}}, R_{\boldsymbol{o}})$ consists of two nonempty finite sets of *actions* A and O for two agents $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively, and two *reward functions* $R_{\boldsymbol{a}}, R_{\boldsymbol{o}} \colon A \times O \to \mathbf{R}$ for $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively. The game G is *zero-sum* iff $R_{\boldsymbol{a}} = -R_{\boldsymbol{o}}$; we then often omit $R_{\boldsymbol{o}}$.

A pure (resp., mixed) strategy specifies which action an agent should execute (resp., which actions an agent should execute with which probability). Formally, a *pure strategy* for agent $\boldsymbol{a}$ (resp., $\boldsymbol{o}$) is any action from A (resp., O). If agents $\boldsymbol{a}$ and $\boldsymbol{o}$ play the pure strategies $a \in A$ and $o \in O$, respectively, then they receive the *rewards* $R_{\boldsymbol{a}}(a, o)$ and $R_{\boldsymbol{o}}(a, o)$, respectively. A *mixed strategy* for agent $\boldsymbol{a}$ (resp., $\boldsymbol{o}$) is any probability distribution over A (resp., O). If agents $\boldsymbol{a}$ and $\boldsymbol{o}$ play the mixed strategies $\pi_{\boldsymbol{a}}$ and $\pi_{\boldsymbol{o}}$, respectively, then the *expected reward* to agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ is $R_k(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) = \mathbf{E}[R_k(a, o)|\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}] = \sum_{a \in A,\, o \in O} \pi_{\boldsymbol{a}}(a) \cdot \pi_{\boldsymbol{o}}(o) \cdot R_k(a, o)$.

We are especially interested in pairs of mixed strategies $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$, which are called Nash equilibria, where no agent has the incentive to deviate from its half of the pair, once the other agent plays the other half. Formally, $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ is a *Nash equilibrium* (or *Nash pair*) for G iff (i) for any mixed strategy $\pi'_{\boldsymbol{a}}$, it holds $R_{\boldsymbol{a}}(\pi'_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) \leqslant R_{\boldsymbol{a}}(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$, and (ii) for any mixed strategy $\pi'_{\boldsymbol{o}}$, it holds $R_{\boldsymbol{o}}(\pi_{\boldsymbol{a}}, \pi'_{\boldsymbol{o}}) \leqslant R_{\boldsymbol{o}}(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$. Every two-player matrix game G has at least one Nash pair among its mixed (but not necessarily pure) strategy pairs, and many two-player matrix games have multiple Nash pairs, which can be computed by linear complementary programming and linear programming in the general and the zero-sum case, respectively. A *Nash selection function* f associates with every two-player matrix game G a unique Nash pair $f(G) = (f_{\boldsymbol{a}}(G), f_{\boldsymbol{o}}(G))$. The expected reward to agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ under $f(G)$ is denoted by $v_f^k(G)$.

In the zero-sum case, if $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ and $(\pi'_{\boldsymbol{a}}, \pi'_{\boldsymbol{o}})$ are Nash pairs, then $R_{\boldsymbol{a}}(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) = R_{\boldsymbol{a}}(\pi'_{\boldsymbol{a}}, \pi'_{\boldsymbol{o}})$, and also $(\pi_{\boldsymbol{a}}, \pi'_{\boldsymbol{o}})$ and $(\pi'_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ are Nash pairs. That is, the expected reward to the agents is the same under any Nash pair, and Nash pairs can be freely "mixed" to form new Nash pairs. Here, agent $\boldsymbol{a}$'s strategies in Nash pairs are given by the optimal solutions of following linear program: $\max v$ subject to (i) $v \leqslant \sum_{a \in A} \pi(a) \cdot R_{\boldsymbol{a}}(a, o)$ for all $o \in O$, (ii) $\sum_{a \in A} \pi(a) = 1$, and (iii) $\pi(a) \geqslant 0$ for all $a \in A$. Moreover, agent $\boldsymbol{a}$'s expected reward under a Nash pair is the optimal value of the above linear program.

2.3 Markov Games

Markov games [12, 5], or also called stochastic games [7], generalize both matrix games and MDPs.

Roughly, a Markov game consists of a set of states S, a matrix game for every state $s \in S$, and a transition function that associates with every state $s \in S$ and combination of actions of the agents a probability distribution on future states $s' \in S$. We only consider the two-player case here. Formally, a *two-player Markov game* $G = (S, A, O, P, R_{\boldsymbol{a}}, R_{\boldsymbol{o}})$ consists of a finite nonempty set of states S, two finite nonempty sets of actions A

and O for two agents $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively, a transition function $P\colon S \times A \times O \to PD(S)$, where $PD(S)$ denotes the set of all probability functions over S, and two *reward functions* $R_{\boldsymbol{a}}, R_{\boldsymbol{o}}\colon S \times A \times O \to \mathbf{R}$ for $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively. The game G is *zero-sum* iff $R_{\boldsymbol{a}} = -R_{\boldsymbol{o}}$; we then often omit $R_{\boldsymbol{o}}$.

Assuming a finite horizon $H \geqslant 0$, a pure (resp., mixed) time-dependent policy associates with every state $s \in S$ and number of steps to go $h \in \{0, \ldots, H\}$ a pure (resp., mixed) matrix-game strategy. Formally, a *pure policy* α (resp., ω) for agent $\boldsymbol{a}$ (resp., $\boldsymbol{o}$) assigns to each state $s \in S$ and number of steps to go $h \in \{0, \ldots, H\}$ an action from A (resp., O). The *H-step reward* to agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ under a start state $s \in S$ and the pure policies α and ω, denoted $G_k(H, s, \alpha, \omega)$, is defined as $R_k(s, \alpha(s,0), \omega(s,0))$, if $H = 0$, and $R_k(s, \alpha(s,H), \omega(s,H)) + \sum_{s' \in S} P(s' | s, \alpha(s,H), \omega(s,H)) \cdot G_k(H-1, s', \alpha, \omega)$, otherwise. A *mixed policy* $\pi_{\boldsymbol{a}}$ (resp., $\pi_{\boldsymbol{o}}$) for $\boldsymbol{a}$ (resp., $\boldsymbol{o}$) assigns to every state $s \in S$ and number of steps to go $h \in \{0, \ldots, H\}$ a probability distribution over A (resp., O). The *expected H-step reward* to agent k under a start state s and the mixed policies $\pi_{\boldsymbol{a}}$ and $\pi_{\boldsymbol{o}}$, denoted $G_k(H, s, \pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$, is $\mathbf{E}[R_k(s, a, o) \mid \pi_{\boldsymbol{a}}(s,0), \pi_{\boldsymbol{o}}(s,0)]$, if $H = 0$, and $\mathbf{E}[R_k(s, a, o) + \sum_{s' \in S} P(s' \mid s, a, o) \cdot G_k(H-1, s', \pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) \mid \pi_{\boldsymbol{a}}(s,H), \pi_{\boldsymbol{o}}(s,H)]$, otherwise.

The notion of a finite-horizon Nash equilibrium for a Markov game is then defined as follows. A pair of mixed policies $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ is a *Nash equilibrium* (or *Nash pair*) for G iff (i) for any start state s and any $\pi'_{\boldsymbol{a}}$, it holds $G_{\boldsymbol{a}}(H, s, \pi'_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) \leqslant G_{\boldsymbol{a}}(H, s, \pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$, and (ii) for any start state s and any $\pi'_{\boldsymbol{o}}$, it holds $G_{\boldsymbol{o}}(H, s, \pi_{\boldsymbol{a}}, \pi'_{\boldsymbol{o}}) \leqslant G_{\boldsymbol{o}}(H, s, \pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$. Every two-player Markov game G has at least one Nash pair among its mixed (but not necessarily pure) policy pairs, and it may have exponentially many Nash pairs.

Nash pairs for G can be computed by finite value iteration from local Nash pairs of two-player matrix games as follows [4]. We assume an arbitrary Nash selection function f for two-player matrix games with the action sets A and O. For every state $s \in S$ and number of steps to go $h \in \{0, \ldots, H\}$, the two-player matrix game $G[s,h] = (A, O, Q_{\boldsymbol{a}}[s,h], Q_{\boldsymbol{o}}[s,h])$ is defined by $Q_k[s,0](a,o) = R_k(s,a,o)$ and $Q_k[s,h](a,o) = R_k(s,a,o) + \sum_{s' \in S} P(s'|s,a,o) \cdot v_f^k(G[s', h-1])$ for all $a \in A$, $o \in O$, and $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$. Let the mixed policy π_k be defined by $\pi_k(s,h) = f_k(G[s,h])$ for all $s \in S$, $h \in \{0, \ldots, H\}$, and $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$. Then, $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ is a Nash pair of G, and $G_k(H, s, \pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) = v_f^k(G(s,H))$ for all $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ and $s \in S$.

In the case of zero-sum G, by induction on $h \in \{0, \ldots, H\}$, it is easy to see that every $G[s,h]$, $s \in S$ and $h \in \{0, \ldots, H\}$, is also zero-sum. Moreover, all Nash pairs that are computed by the above finite value iteration produce the same expected H-step reward, and they can be freely "mixed" to form new Nash pairs.

3 Game-Theoretic Golog

In this section, we present the language GTGolog for the case of two competing agents. We first describe the domain theory and the syntax of GTGolog programs. We then define the semantics of GTGolog programs and provide representation and optimality results.

3.1 Domain Theory

GTGolog programs are interpreted w.r.t. a background action theory AT and a background optimization theory OT.

The background action theory AT is an extension of the basic action theory BAT of Section 2.1, where we also allow for stochastic actions. We assume two (zero-sum) competing agents $\boldsymbol{a}$ and $\boldsymbol{o}$ (called *agent* and *opponent*, respectively, where the former is under our control, while the latter is not). A *two-player action* is either an action a for agent $\boldsymbol{a}$, or an action b for agent $\boldsymbol{o}$, or two parallel actions $a \| b$, one for each agent. For example, $move(\boldsymbol{a}, M)$, $move(\boldsymbol{o}, O)$, and $move(\boldsymbol{a}, M) \| move(\boldsymbol{o}, O)$ are two-player actions. Note that

we introduce parallel actions as primitives to simplify the presentation, more complex parallel actions can be easily treated as well, deploying the concurrent version of the situation calculus [11].

Analogously to [2], we represent stochastic actions by means of a finite set of deterministic actions. When a stochastic action is executed, then "nature" chooses and executes with a certain probability exactly one of its deterministic actions. We use the predicate $stochastic(a, s, n)$ to associate the stochastic action a with the deterministic action n in situation s, and we use the function $prob(a, n, s) = p$ to encode that "nature" chooses n in situation s with probability p. A stochastic action s is indirectly represented by providing a *successor state axiom* for each associated nature choice n. Thus, BAT is extended to a probabilistic setting in a minimal way. For example, consider the stochastic action $moveS(a, x, y)$ such that $stochastic(moveS(a, x, y), s, moveTo(a, x, y')) \equiv 0 \leqslant y' - y \leqslant 1$, that is, $moveS(a, x, y)$ can slide to $y+1$. We specify $moveS$ by defining the precondition of $moveTo(a, x, y')$ and assuming the successor state axiom (1). Furthermore, in order to specify the probability distribution over the deterministic components, we define $prob(moveS(a, x, y), moveTo(a, x, y), s) = 0.9$ and $prob(moveS(a, x, y), moveTo(a, x, y + 1), s) = 0.1$.

Like [2], we assume that the domain is *fully observable*. To this end, we introduce *observability axioms*, which disambiguate the state of the world after executing a stochastic action. For example, after executing $moveS(a, x, y)$, we test the predicates $at(a, x, y, s)$ and $at(a, x, y + 1, s)$ to check which of the deterministic components was executed (that is, $n{=}moveTo(a, x, y)$ or $n{=}moveTo(a, x, y{+}1)$). This condition is denoted by the predicate $condSta(a, n, s)$, e.g., $condSta(moveS(a, x, y), moveTo(a, x, y{+}1), s) \equiv at(a, x, y + 1, s)$. Analogous observability axioms are needed to observe which actions the opponent and the agent have chosen. For this purpose, we introduce the predicate $cond(a, s)$, e.g., $cond(moveTo(o, x, y), s) \equiv at(o, x, y, s)$.

The optimization theory OT specifies a reward and a utility function. The former associates with every situation s and two-player action α, a reward to agent $\boldsymbol{a}$, denoted $reward(\alpha, s)$, e.g., $reward(moveTo(a, x, y), s) = y$. As we assume that the rewards to $\boldsymbol{a}$ and $\boldsymbol{o}$ are zero-sum, we need not explicitly specify the reward to $\boldsymbol{o}$. The utility function maps every reward and success probability to a real-valued utility $utility(v, pr)$. We assume that $utility(v, 1) = v$ for all v. An example is $utility(v, pr) = v \cdot pr$. The utility function suitably mediates between the agent reward and the failure of actions due to unsatisfied preconditions.

3.2 Syntax

Given the two-player actions represented by the domain theory, *programs* p in GTGolog are inductively built using the following constructs (ϕ is a condition, and p_1 and p_2 are programs):

1. *Deterministic or stochastic action:* α (a two-player action).
2. *Nondeterministic action choice:* $\alpha|...|\beta$. Execute α or ... or β, where α,..., β are two-player actions.
3. *Test action*: $\phi?$. Test the truth value of ϕ in the current situation.
4. *Nondeterministic choice of an argument.*
5. *Action sequence*: $p_1; p_2$. Do program p_1 followed by program p_2.
6. *Conditionals*: $\mathbf{if}\ \phi\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2$.
7. *While loops*: $\mathbf{while}\ \phi\ \mathbf{do}\ p_1$.

8. *Nondeterministic iteration*: ${p_1}^*$. Execute p_1 zero or more times.

9. *Procedures, including recursion.*

To clearly distinguish between the choices of the agent $\boldsymbol{a}$ and the opponent $\boldsymbol{o}$, we use $\mathbf{choice}(\boldsymbol{a}\colon a_1|\cdots|a_n) \,\|\, \mathbf{choice}(\boldsymbol{o}\colon o_1|\ldots|o_m)$ to denote $(\boldsymbol{a}{:}a_1\|\boldsymbol{o}{:}o_1)|(\boldsymbol{a}{:}a_2\|\boldsymbol{o}{:}o_1)|\cdots|(\boldsymbol{a}{:}a_n\|\boldsymbol{o}{:}o_m)$.

3.3 Semantics

The semantics of a GTGolog program p w.r.t. AT and OT is defined through the predicate $DoG(p, s, h, \pi, v, pr)$. Here, we have given as input the program p, a situation s, and a finite horizon $h \geqslant 0$. The predicate DoG then determines a strategy π for both agents $\boldsymbol{a}$ and $\boldsymbol{o}$, the reward to agent $\boldsymbol{a}$ under this strategy π, and the success probability $pr \in [0, 1]$ of π. Note that due to the finite horizon, if the program p fails to terminate before the horizon h is reached, then it is stopped, and the best partial strategy is returned. Intuitively, our aim is to control agent $\boldsymbol{a}$, which is given the strategy π that DoG computes for program p, and which then executes its part of π. We define $DoG(p, s, h, \pi, v, pr)$ by induction as follows:

1. Zero horizon and null program:

   $$DoG(p, s, 0, \pi, v, pr) =_{def} \pi = Nil \wedge v = 0 \wedge pr = 1$$
   $$DoG(Nil, s, h, \pi, v, pr) =_{def} \pi = Nil \wedge v = 0 \wedge pr = 1$$

   Intuitively, p ends when it is null or the horizon end is reached.

2. Deterministic first program action:

   $$\begin{array}{l} DoG(a; p, s, h, \pi, v, pr) =_{def} \\ \quad \neg Poss(a, s) \wedge \pi = Stop \wedge v = 0 \wedge pr = 0 \vee \exists \pi', v' : \\ \quad Poss(a, s) \wedge DoG(p, do(a, s), h{-}1, \pi', v', pr) \wedge \\ \qquad \pi = a; \pi' \wedge v = v' {+} reward(s, a) \end{array}$$

   Informally, if a is not executable, then p stops with success probability 0. As in [2], $Stop$ is a fictitious action of zero-cost, which stops the program execution. If a is executable, then the optimal execution of $a; p$ in s depends on that one of p in $do(a, s)$.

3. Stochastic first program action (nature choice):

   $$\begin{array}{l} DoG(a; p, s, h, \pi, v, pr) =_{def} \\ \quad \neg Poss(a, s) \wedge \pi = Stop \wedge v = 0 \wedge pr = 0 \vee \exists \pi_q, v_q, pr_q : \\ \quad Poss(a, s) \wedge \bigwedge_{q=1}^{k} DoG(n_q; p, s, h, n_q; \pi_q, v_q, pr_q) \wedge \\ \quad \pi = a; \mathbf{if}\ \phi_1\ \mathbf{then}\ \pi_1\ \mathbf{else\ if}\ \phi_2\ \mathbf{then}\ \pi_2\ \ldots \\ \qquad\qquad \mathbf{else\ if}\ \phi_k\ \mathbf{then}\ \pi_k \wedge \\ \quad v = \sum_{q=1}^{k} v_q \cdot prob(a, n_q, s) \wedge pr = \sum_{q=1}^{k} pr_q \cdot prob(a, n_q, s) \end{array}$$

   where $\{n_q | 1 \leqslant q \leqslant k\}$ are the nature choices associated to a (which have the same reward function as a), and $\phi_1, \ldots, \phi_k$ are the relative conditions (represented by the *observability axioms*). The generated strategy is a conditional plan, that is, a cascade of if-then-else statements, where each possible stochastic action execution is considered.

4. Nondeterministic first program action (choice of agent $\boldsymbol{a}$):

$$
\begin{array}{l}
DoG(\mathbf{choice}(\boldsymbol{a}\colon a_1|\ldots|a_n); p, s, h, \pi, v, pr) =_{def} \\
\quad \exists \pi_i, v_i, pr_i\colon \bigwedge_{i=1}^{n} DoG(\boldsymbol{a}{:}a_i; p, s, h, \pi_i, v_i, pr_i) \wedge \\
\quad utility(v_k, pr_k) = \max\{utility(v_i, pr_i) \mid i \in \{1, \ldots, n\}\} \wedge \\
\quad \pi = \pi_k \wedge v = v_k \wedge pr = pr_k
\end{array}
$$

Given several possible actions for agent $\boldsymbol{a}$, the best action is the one where the action execution has the best utility.

5. Nondeterministic first program action (choice of opponent $\boldsymbol{o}$):

$$
\begin{array}{l}
DoG(\mathbf{choice}(\boldsymbol{o}\colon o_1|\ldots|o_m); p, s, h, \pi, v, pr) =_{def} \\
\quad \exists \pi_j, v_j, pr_j\colon \bigwedge_{j=1}^{m} DoG(\boldsymbol{o}{:}o_j; p, s, h, \pi_j, v_j, pr_j) \wedge \\
\quad \pi = \mathbf{if}\ \psi_1\ \mathbf{then}\ \pi_1\ \mathbf{else\ if}\ \psi_2\ \mathbf{then}\ \pi_2\ \ldots \\
\qquad\quad \mathbf{else\ if}\ \psi_m\ \mathbf{then}\ \pi_m \wedge \\
\quad utility(v_k, pr_k) = \min\{utility(v_j, pr_j) \mid j \in \{1, \ldots, m\}\} \wedge \\
\quad v = v_k \wedge pr = pr_k
\end{array}
$$

Informally, agent $\boldsymbol{a}$ assumes a rational behavior of $\boldsymbol{o}$, which is connected to its minimal reward (we consider a zero-sum setting). The ψ_i's are the conditions (defined by the *observability axioms*) that agent $\boldsymbol{a}$ has to test to observe the choice of opponent $\boldsymbol{o}$.

6. Nondeterministic first program action (choice of both $\boldsymbol{a}$ and $\boldsymbol{o}$):

$$
\begin{array}{l}
DoG(\mathbf{choice}(\boldsymbol{a}\colon a_1|\ldots|a_n)\|\mathbf{choice}(\boldsymbol{o}\colon o_1|\ldots|o_m); \\
p, s, h, \pi, v, pr) =_{def} \exists \pi_{i,j}, v_{i,j}, pr_{i,j}, \pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}\colon \\
\quad \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} DoG(\boldsymbol{a}{:}a_i\|\boldsymbol{o}{:}b_j; p, s, h, \boldsymbol{a}{:}a_i\|\boldsymbol{o}{:}b_j; \pi_{i,j}, v_{i,j}, pr_{i,j}) \wedge \\
\quad (\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) = selectNash(\{r_{i,j} = utility(v_{i,j}, pr_{i,j}) \mid i, j\}) \wedge \\
\quad \pi = \pi_{\boldsymbol{a}}\|\pi_{\boldsymbol{o}}; \mathbf{if}\ \phi_1{\wedge}\psi_1\ \mathbf{then}\ \pi_{1,1}\ \mathbf{else\ if}\ \phi_2{\wedge}\psi_1\ \mathbf{then}\ \pi_{2,1}\ \ldots \\
\qquad\quad \mathbf{else\ if}\ \phi_n{\wedge}\psi_m\ \mathbf{then}\ \pi_{n,m} \wedge \\
\quad v = \sum_{i=1}^{n} \sum_{j=1}^{m} v_{i,j} \cdot \pi_{\boldsymbol{a}}(a_i) \cdot \pi_{\boldsymbol{o}}(o_j) \wedge \\
\quad pr = \sum_{i=1}^{n} \sum_{j=1}^{m} pr_{i,j} \cdot \pi_{\boldsymbol{a}}(a_i) \cdot \pi_{\boldsymbol{o}}(o_j)
\end{array}
$$

Intuitively, we compute a Nash strategy by finite horizon value iteration for Markov games [4]. For each possible pair of action choices, the optimal strategy is calculated. Then, a Nash strategy is locally extracted from a matrix game by using the function $selectNash$. Here, $\pi_{\boldsymbol{a}}$ and $\pi_{\boldsymbol{o}}$ are probability distributions over $\{a_1, \ldots, a_n\}$ and $\{o_1, \ldots, o_m\}$, respectively. Moreover, ψ_i and ϕ_j are the conditions defined by the *observability axioms* to observe what $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively, have actually executed.

7. Test action:

$$
\begin{array}{l}
DoG(\phi?; p, s, h, \pi, v, pr) =_{def} \phi[s] \wedge DoG(p, s, h, \pi, v, pr) \vee \\
\quad \neg\phi[s] \wedge \pi = Stop \wedge v = 0 \wedge pr = 0
\end{array}
$$

8. The semantics of nondeterministic iterations, conditionals, while loops, procedures, argument selection, and associate sequential composition is defined in the standard way.

3.4 Representation and Optimality Results

The following result shows that every zero-sum two-player Markov game can be represented in GTGolog, and that DoG computes one of its finite-horizon Nash equilibria and its expected finite-step reward.

Theorem 3.1 *Let $G=(S,A,O,P,R_{\boldsymbol{a}})$ be a zero-sum two-player Markov game, and let $H\geqslant 0$ be a horizon. Then, there exists an action theory AT, an optimization theory OT, and a GTGolog program p relative to them such that $\vec{\pi}=(\vec{\pi}_{\boldsymbol{a}},\vec{\pi}_{\boldsymbol{o}})$ is a Nash equilibrium for G, where $\vec{\pi}_k(s,h)=\pi_k$, $k\in\{\boldsymbol{a},\boldsymbol{o}\}$, is given by $DoG(p,s,h{+}1,\pi_{\boldsymbol{a}}\|\pi_{\boldsymbol{o}};\pi',v,pr)$ for every $s\in S$ and $h\in\{0,\ldots,H\}$. Furthermore, for every $H\geqslant 0$ and $s\in S$, it holds that $G(H,s,\vec{\pi}_{\boldsymbol{a}},\vec{\pi}_{\boldsymbol{o}})=v$ is given by $DoG(p,s,H{+}1,\pi_{\boldsymbol{a}}\|\pi_{\boldsymbol{o}};\pi',v,pr)$.*

Proof. The background action theory AT comprises a fluent $state(s,sit)$, which associates with every situation sit a state $s\in S$ (such that the set of all states S partitions the set of all situations into equivalence classes), and one deterministic action n_s for every state $s\in S$, which performs a transition into a situation associated with s. Every pair of actions $(a,o)\in A\times O$ then yields a probabilistic two-player action, and P is encoded using n_s, $stochastic$, and $prob$, while $R_{\boldsymbol{a}}$ is encoded using $reward$ as follows. The relationship $P(s,a,o)=p$ is encoded as $stochastic(a\|o,sit_s,n_s)$ and $prob(a\|o,n_s,sit_s)=p$, while $R_{\boldsymbol{a}}(s,a,o)=r$ is encoded as $reward(a\|o,sit_s)=r$, for all situations sit_s associated with s. The program p is a sequence of $H{+}1$ constructs $\mathbf{choice}(\boldsymbol{a}\colon a_1|\cdots|a_n)\,\|\,\mathbf{choice}(\boldsymbol{o}\colon o_1|\cdots|o_m)$, where $A=\{a_1,\ldots,a_n\}$ and $O=\{o_1,\ldots,o_m\}$. It is then easy to verify that $pr=1$ for every success probability pr computed in DoG for such p. By induction on $H\geqslant 0$, it follows that DoG encodes the finite value iteration in [4]. □

We next show that DoG produces optimal results. Given a finite horizon $H\geqslant 0$, a *strategy* π for a GTGolog program p is obtained from the H-horizon part of p by replacing agent and opponent choices by single actions, and choices of both by probability distributions over their actions. The notions of an *expected H-step reward* $G(p,s,H,\pi)$, with a situation s, and of a finite-horizon *Nash equilibrium* can then be defined in a straightforward way as for Markov games. The next theorem shows that DoG is optimal in the sense that it computes a Nash equilibrium and its expected finite-step reward.

Theorem 3.2 *Let AT be an action theory, OT be an optimization theory, and p be a GTGolog program relative to them. Let $DoG(p,s,h{+}1,\pi,v,pr)$ for a situation s and $h\geqslant 0$. Then, π is a Nash equilibrium, and $utility(v,pr)$ is its expected h-step reward.*

Proof. Recall that syntactically the strategies π in the output of DoG have the form of programs, except that choices of single agents are replaced by one-player actions, and choices of both agents are replaced by two probability distributions over the possible one-player actions of each agent. The notion of an expected H-step reward G for such strategies is then defined in a similar way as DoG, except that (i) we now have strategies rather than programs in the input, (ii) no strategy is in the output, (iii) items (4) and (5) are removed (since single-agent choices in a program have been transformed into one-player actions in a strategy), and (iv) item (6) is adapted such that the two probability distributions are already given in a strategy and not computed as Nash equilibria from the choices of both agents in a program. Since the expected h-step reward G is similar to DoG as far as the computation of the outputs v and pr are concerned, it follows that $utility(v,pr)$ is the expected h-step reward of the strategy π computed by DoG. Unilaterally changing the strategy π for one of the two agents yields a possibly lower expected h-step reward for that agent, since the computations in items (4), (5), and (6) are optimal (in the sense of maximum, minimum, and Nash equilibrium, respectively). This shows that π is also a Nash equilibrium. □

In general, there may be exponentially many Nash equilibria. We assume that the opponent is rational, and thus follows a Nash equilibrium. But we do not know which one it actually uses. The following theorem shows that this is not necessary, as far as the opponent computes its Nash half in the same way as we do. That is, different Nash equilibria computed by DoG can be freely "mixed". The result follows from a similar result for matrix games [13] and Theorem 3.2.

Theorem 3.3 *Let AT be an action theory, OT be an optimization theory, and p be a GTGolog program relative to them. Let π and π' be strategies computed by DoG using different Nash selection functions. Then, π and π' have the same expected finite-step reward, and the strategy obtained by mixing π and π' is also a Nash equilibrium.*

Proof. Immediate by Theorem 3.2 and the result that in zero-sum matrix games, the expected reward is the same under any Nash pair, and Nash pairs can be freely "mixed" to form new Nash pairs [13]. □

4 Soccer Example

We consider a slightly modified version of the soccer example by Littman [5] (see Fig. 1): The soccer field is a 4×5 grid. There are two players, A and B, each occupying a square, and each able to do one of the following actions on each turn: N, S, E, W, and stand (move up, move down, move left, move right, and no move, respectively). The ball is represented by a circle and also occupies a square. A player is a *ball owner* iff it occupies the same square as the ball. The ball follows the moves of the ball owner, and we have a goal when the ball owner steps into the adversary goal. When the ball owner goes into the square occupied by the other player, if the other player stands, possession of ball changes. Therefore, a good defensive maneuver is to stand where the other agent wants to go. To axiomatize this domain, we introduce the action $move(\alpha, m)$

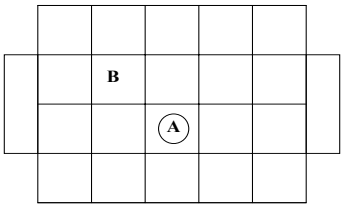

Figure 1: Soccer Example

(agent α executes $m \in \{N, S, E, W, stand\}$) and the fluents $at(\alpha, x, y, s)$ and $haveBall(\alpha, s)$ defined by the following successor state axioms:

$$
\begin{aligned}
&at(\alpha, x, y, do(a, s)) \equiv at(\alpha, x, y, s) \wedge a = move(\alpha, stand) \vee \\
&\quad at(\alpha, x', y', s) \wedge (\exists m).a = move(\alpha, m) \wedge \phi(x, y, x', y', m); \\
&haveBall(\alpha, do(a, s)) \equiv (\exists \alpha').haveBall(\alpha', s) \wedge \\
&\quad (\alpha = \alpha' \wedge \neg cngBall(\alpha', a, s) \vee \alpha \neq \alpha' \wedge cngBall(\alpha, a, s)).
\end{aligned}
$$

Here, $\phi(x, y, x', y', m)$ represents the coordinate change due to m, and $cngBall(\alpha, a, s)$ is true iff the ball possession changes after an action a of agent α in situation s. We can now define $goal(\alpha, s) \equiv (\exists x, y)$ $haveBall(\alpha, s) \wedge at(\alpha, x, y, s) \wedge goalpos(\alpha, x, y)$. Here, $goalpos(\alpha, x, y)$ represents the coordinates for

the α adversary goal. We next define a zero-sum reward function as follows:

$$
\begin{array}{l}
reward(\alpha, s){=}r \equiv \\
\ (\exists\alpha')goal(\alpha', s) \wedge (\alpha'{=}\alpha \wedge r{=}M \vee \alpha'{\neq}\alpha \wedge r{=}-M) \vee \\
\ (\exists\alpha')\neg goal(\alpha', s) \wedge haveBall(\alpha', s) \wedge at(\alpha', x, y, s) \wedge \\
\ \ (\exists r')evalPos(x, y, r') \wedge (\alpha{=}\alpha' \wedge r{=}r' \vee \alpha'{\neq}\alpha \wedge r{=}-r').
\end{array}
$$

Here, the reward is high (M stands for a "big" integer), if a player scores a goal, and the reward depends on $evalPos(x, y, r)$, that is, the ball-owner position (roughly, r is high if the ball-owner is close to the adversary goal), otherwise. A game session can then be described by the following Golog procedure:

$$
\begin{array}{l}
\mathbf{proc}(game, \mathbf{while}\neg(\exists x)goal(x)\ \mathbf{do}\ (\\
\quad \mathbf{choice}(\boldsymbol{a}\colon move(\boldsymbol{a}, E)|move(\boldsymbol{a}, S)| \\
\quad\ move(\boldsymbol{a}, N)|move(\boldsymbol{a}, O)|move(\boldsymbol{a}, stand))\| \\
\quad \mathbf{choice}(\boldsymbol{o}\colon move(\boldsymbol{o}, E)|move(\boldsymbol{o}, S)| \\
\quad\ move(\boldsymbol{o}, N)|move(\boldsymbol{o}, O)|move(\boldsymbol{o}, stand)))\colon nil).
\end{array}
$$

Intuitively, while a goal is not reached, the two players (agent $\boldsymbol{a}$ and opponent $\boldsymbol{o}$) can choose a possible move. Consider the situation in Fig. 1 where A is the agent $\boldsymbol{a}$ and B the opponent $\boldsymbol{o}$. The initial situation can be described by $at(a, 3, 2, S_0) \wedge at(o, 2, 3, S_0) \wedge haveBall(a, S_0)$. Assuming the horizon $h = 3$, a strategy π can be calculated by searching for a constructive proof of $AT \models (\exists v, \pi, pr)DoG(game, S_0, 3, \pi, v, pr)$. Here, the maximal reward is achieved by following the pure strategy π that leads the agent $\boldsymbol{a}$ to score a goal after executing three times $move(\boldsymbol{a}, O)$. Note that if we consider $at(o, 1, 3, S_0)$ as the initial position of $\boldsymbol{o}$, then any pure strategy of $\boldsymbol{a}$ can be blocked by $\boldsymbol{o}$ and the only solution is a randomized strategy. The $game$ procedure introduced above represents a generic soccer game. However, more specialized game playing behavior can also be written. For instance, the agent $\boldsymbol{a}$ could discriminate game situations Φ_i where the game can be simplified (that is, possible agent and/or opponent behaviors are restricted):

$$
\begin{array}{l}
\mathbf{proc}(game', \mathbf{while}\neg(\exists x)goal(x)\ \mathbf{do} \\
\ (\mathbf{if}\ ?\Phi_1\colon schema_1\ \mathbf{else}\ (\mathbf{if}\ ?\Phi_2\colon schema_2\ \mathbf{else}\ game))\colon nil).
\end{array}
$$

For example, consider an attacking ball owner, which is closer to the opponent's goal than the opponent (that is, $\Phi(s) = (\exists x, y, x', y')\ at(a, x, y, s) \wedge at(o, x', y', s) \wedge x' > x$). In this situation, since the opponent is behind, the best agent strategy is to move quickly towards the goal. This strategy can be encoded as a Golog program $schema'$. Note that $game'$ lies between a specification of the game rules and a sketchy denotation of the agent strategy. This is in the same spirit of a standard Golog program, which can balance the tradeoff between a planner and a deterministic program: GTGolog allows for the specification of a partial agent strategy, which can be optimally completed once interpreted.

5 Teams

We now generalize to the case where we have two competing teams, rather than only two competing agents. Every team consists of a set of cooperating agents. Hence, all the members of every team have the same reward, while the rewards of two members of different teams are zero-sum. Formally, we assume two teams $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{o}}$, where $\vec{\boldsymbol{a}} = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n)$ consists of $n \geqslant 1$ agents $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, while $\vec{\boldsymbol{o}} = (\boldsymbol{o}_1, \ldots, \boldsymbol{o}_m)$ consists of $m \geqslant 1$ opponent agents $\boldsymbol{o}_1, \ldots, \boldsymbol{o}_m$. A *one-team action* is the parallel combination of at most one action for each member of one of the two teams. A *two-team action* is either a one-team action $\vec{a}$ for team $\vec{\boldsymbol{a}}$, or a one-team action $\vec{b}$ for team $\vec{\boldsymbol{o}}$, or two parallel one-team actions $\vec{a} \parallel \vec{b}$, one for each team.

We then easily extend the presented GTGolog for two competing agents to the case of two competing teams of agents, by using two-team actions, rather than two-player actions. Then, a one-agent choice of the form $\mathbf{choice}(\boldsymbol{a}\colon a_1|\ldots|a_k)$ turns into a team choice $\mathbf{choice}(\boldsymbol{a}_i\colon a_{i,1}|\ldots|a_{i,k_i})$ for $i\in\{1,\ldots,n\}$, which is written as $\mathbf{choice}(\vec{\boldsymbol{a}}\colon \vec{a}_1|\ldots|\vec{a}_k)$. Thus, in the one-agent choice, rather than a basic action a_i, we choose a combined action $\vec{a}_i$, which consists of at most one basic action for each member of the team $\vec{\boldsymbol{a}}$, while in the two-agent choice, rather than one probability distribution over the possible actions of each agent, we choose at most one distribution over the possible actions of each member of the two teams.

In general, every team has several options for how to act optimally, and two such options cannot be "freely" mixed for different team members. It is thus necessary that there is some form of coordination to agree on one common optimal strategy inside a team (see e.g. [1] for coordination in multi-agent systems). We assume that the coordination is done by centrally controlling a team. Alternatives are either allowing for local communication between team members, or having a total order on optimal strategies, which allows the team members to independently select a common preferred optimal strategy.

6 Implementation

We have implemented a GTGolog interpreter for two competing agents, where we make use of linear programming to calculate the Nash equilibrium at each choice step. The interpreter is implemented as a constraint logic program in Eclipse 5.7 and uses the eplex library to define and solve the linear programs for the Nash equilibria. Similarly as for standard Golog, the interpreter has been obtained by translating the rules of Section 3 into Prolog clauses.

6.1 GTGolog Interpreter

The interpreter code is as follows.

```
%%  A game-theoretic Golog interpreter.

:- lib(eplex).

:- eplex_instance(ep).

:- set_flag(print_depth,100).
:- nodbgcomp.
:- dynamic(proc/2).     % Compiler directives.
:- set_flag(all_dynamic, on).

:- op(800, xfy, [&]).   % Conjunction
:- op(850, xfy, [v]).   % Disjunction
:- op(870, xfy, [=>]).  % Implication
:- op(880,xfy, [<=>]).  % Equivalence
:- op(950, xfy, [:]).   % Action sequence.
:- op(960, xfy, [#]).   % Nondeterministic action choice.

%%%%%%%%%%%%%% Matrix game: computing Nash equilibria in eplex.

selectNash(StrA, StrO, UtMatrix,Cost) :-
        nash_ag(UtMatrix,StrA,V,Cost),
        traspMatrix(UtMatrix, TUtMatrix),
        nash_ag(TUtMatrix,StrO,V1,Cost1).
```

```
nash_ag(MatrixRew,ListVar,V,Cost) :-
        ep: (sum(ListVar) $= 1),
        gtzeroCnst(ListVar),
        systemRew(MatrixRew,ListVar,V),
        ep: eplex_solver_setup(max(V)),
        ep: eplex_solve(Cost), get_solution(ListVar), ep: eplex_cleanup.

nash_opp(ListRew,ListVarO,V,Cost) :-
        ep: (sum(ListVarO) $= 1),
        gtzeroCnst(ListVarO),
        ep: eplex_solver_setup(min(V)),
        ep: (ListRew * ListVarO $=< V),
        ep: eplex_solve(Cost), get_solution(ListVarO), ep: eplex_cleanup.

systemRew([ListRew|MatrixRew],ListVar,V) :-
        ep: (ListRew * ListVar $>= V),
        systemRew(MatrixRew,ListVar,V).
systemRew([], ListVar, V).

gtzeroCnst([]).
gtzeroCnst([X|L]) :- ep: (X $>=0), gtzeroCnst(L).

prodM([],DistA,[]).
prodM([L|MatrixRew],DistA,[E|VectorRew]) :-
      prodV(L,DistA,E), prodM(MatrixRew,DistA,VectorRew).
prodV([],[],0).
prodV([X|L],[Y|M],P) :- prodV(L,M,P1), P is P1 + X * Y.

get_solution(ListVar) :-
        ListVar = [],!;
        ListVar = [A|L], ep: eplex_var_get(A,typed_solution,A), get_solution(L).

%% Matrix transposition.

selectLine([],[],[]).
selectLine([X|L],[[X|L1]|M1],[L1| M2]) :-
        selectLine(L,M1,M2).

traspMatrix([[]|L],[]).
traspMatrix(M,[L|MT]) :- selectLine(L,M,M1), traspMatrix(M1,MT).

%%%%%%%%%%%%%% doG

doG(P, S, 0, Pi, V, Pr) :- Pi = nil, V = 0, Pr = 1.
doG(nil, S, H,nil,V,Pr) :- V = 0, Pr = 1.

%% Sequence

doG((A : B) : C,S,H,Pi,V,Pr) :- doG(A : (B : C),S,H,Pi,V,Pr).

doG(A : C, S, H, Pi, V, Pr) :- concurrentAction(A),
        ( poss(A,S), H1 is H - 1, doG(C, do(A,S), H1,Pi1,V1,Pr1), agent(Ag),
                reward(Ag,R,A,S),
                seq(A, Pi1,Pi), V is V1 + R, Pr = Pr1;
```

```
         not poss(A,S), Pi = stop, V = 0, Pr = 0).

doG(A : B, S, H, Pi, V,Pr) :- stochastic(A),
        (not poss(A,S), Pi = stop, V = 0, Pr = 0;
         poss(A,S), nChoice(A,C), doGAux(A,C,B,S, H,Pi1,V,Pr), seq(A, Pi1,Pi)).

doG(A : B,S, H,Pi,R,Pr) :- proc(A,C), doG(C : B, S, H, Pi,R,Pr).

%% Nondeterministic choice of argument.

doG(pi(V,E) : B, S, H, Pi,R,Pr) :- sub(V,_,E,E1), doG(E1 : B, S, H, Pi, R,Pr).

%% Test

doG(?(T) : A, S, H, Pi, R, Pr) :- holds(T,S), !, doG(A, S, H, Pi, R, Pr) ;
                          Pi = stop, V = 0, Pr = 0. % Program can't continue.
                                                    % Create a leaf.
%% Conditional

doG(if(T,A,B) : C,S, H, Pi, R, Pr) :- holds(T,S), !, doG(A : C, S, H, Pi,R,Pr) ;
                                doG(B : C, S, H, Pi, R, Pr).

%% Loop

doG(while(T,A) : B, S, H, Pi, R, Pr) :- holds(T,S), !,
                                 doG(A : while(T,A) : B, S, H, Pi,R,Pr) ;
                                 doG(B, S, H, Pi, R, Pr).

%% Agent choice

doG([choice(Ag,C1)] : E, S, H, Pi,R, Pr) :-
        agent(Ag), doMax(C1,E, S, H, Pi,R, Pr);
        opponent(Ag), doMin(C1,E, S, H, Pi,R, Pr).

doG([choice(Ag1,C1), choice(Ag2,C2)] : E, S, H, Pi,R, Pr) :-
         agent(Ag1), opponent(Ag2), doMinMax(C1,C2, E, S, H, Pi,R, Pr);
         agent(Ag2), opponent(Ag1), doMinMax(C2,C1, E, S, H, Pi,R, Pr).

doMinMax(C1,C2,E, S, H, Pi,R, Pr) :-
        doMatrix(C2,C1, E, S, H, PiMatrix,RMatrix,UtMatrix, PrMatrix),
        genListVar(C1,C2,StrA,StrO),
        selectNash(StrA, StrO, UtMatrix, R),
        probNash(StrA,StrO,PrMatrix,Pr), strNash(C1,C2,StrA, StrO, PiMatrix, Pi).

doMatrix([], B, E, S, H, [],[],[],[]).
doMatrix([A|L], B, E, S, H, [PiLine| PiSubMatrix],[RLine | RSubMatrix],
                            [UtLine| UtSubMatrix], [PrLine |PrSubMatrix]) :-
        doLine(A, B, E, S, H, PiLine,RLine,UtLine,PrLine),
        doMatrix(L,B, E, S, H, PiSubMatrix, RSubMatrix, UtSubMatrix, PrSubMatrix).

doLine(A,[], E, S, H, [],[],[],[]).
doLine(A,[B|L], E, S, H, [Pi|PiM],[R|RM],[Ut|UtM], [Pr|PrM]) :-
        doG([A,B] : E, S, H, Pi1,R,Pr), seq([A,B],Pi,Pi1),
        doLine(A,L,E, S, H, PiM,RM,UtM,PrM),
        utility(Ut, R, Pr).
```

```
doMax([A],E, S, H, Pi,R, Pr) :- doG([A] : E, S, H, Pi, R, Pr),!.
doMax([A|L],E, S, H, Pi,R, Pr) :-
        doG([A] : E, S, H, Pi1,R1, Pr1),
        doMax(L,E, S, H, Pi2,R2, Pr2), utility(Ut1, R1, Pr1),
        utility(Ut2, R2, Pr2),
        (Ut1 >= Ut2, Pi = Pi1, R = R1, Pr = Pr1;
         Ut1 < Ut2, Pi = Pi2, R = R2, Pr = Pr2).

doMin([A],E, S, H, Pi,R, Pr) :- doG(A : E, S, H, Pi,R, Pr).
doMin([A|L],E, S, H, Pi,R, Pr) :- not L = [],
        doG(A : E, S, H, Pi1,R1, Pr1),
        doMax([A|L],E, S, H, Pi2,R2, Pr2), utility(Ut1, R1, Pr1),
        utility(Ut2, R2, Pr2),
        (Ut2 >= Ut1, Pi = Pi1, R = R1, Pr = Pr1;
         Ut2 < Ut1, Pi = Pi2, R = R2, Pr = Pr2).

%%%%%%%%%%%%%

doGAux(A,[],B,S, H,[],0,0).
doGAux(A,[C1|LC],B, S, H, Pi,V,Pr) :-
        doG(C1 :B, S, H, Pi1,V1,Pr1),
        doGAux(LC,B, S, H, Pi2,V2,Pr2),
        prob(C1,A,S,Pr3),Pi = if(obsNature(A,C1),Pi1,Pi2),
        Pr is Pr1 * Pr3 + Pr2,
        V is V1 * Pr1 * Pr3 + V2 * Pr2.

prob(C,A,S,P) :- choice(A,C), poss(C,S), !, prob0(C,A,S,P) ; P = 0.0.
utility(Ut,R,Pr) :- Ut is R * Pr.

stochastic(A) :- nChoice(A,N), !.

%%%%%%%%%%%%%

probNash(StrA,StrO,PrMatrix,Pr) :-
        prodM(PrMatrix, StrA,PrLine), prodV(PrLine, StrO, Pr).

strNash(C1,C2,StrA, StrO, PiMatrix, Pi) :-
        genNashStrategy(C1,C2,PiMatrix,Pi1),
        Pi = alea([[C1,StrA] ,[C2, StrO]], Pi1).

genNashStrategy(LA,[O], [PiL],Pi) :- genNashStrategy1(O, LA, PiL, Pi),!.
genNashStrategy(LA,[O|CO], [PiL | PiMatrix],Pi) :-
        genNashStrategy1(O, LA, PiL, Pi2),
      genNashStrategy(LA, CO, PiMatrix,Pi3),
      Pi = if(obsChoice(O),Pi2,Pi3).

genNashStrategy1(O,[A],[Pi],Pi):-!.
genNashStrategy1(O,[A|LA],[Pi1|PiL], Pi) :-
        genNashStrategy1(O, LA, PiL, Pi2), Pi = if(obsChoice(A),Pi1,Pi2).

genListVar([],[],[],[]).
genListVar([],[X2|LC2],[],[O1|StrO]) :- genListVar([],LC2,[],StrO).
genListVar([X1|LC1],[],[A1|StrA],[]) :- genListVar(LC1,[],StrA,[]).
genListVar([X1|LC1],[X2|LC2], [A1|StrA], [O1|StrO]) :-
```

```
                                genListVar(LC1,LC2,StrA,StrO).

%% sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New.

sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

%% The holds predicate implements the revised Lloyd-Topor
%% transformations on test conditions.

holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :- holds(-((P => Q) & (Q => P)),S).
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- not holds(some(V,P),S).  % Negation
holds(-P,S) :- isAtom(P), not holds(P,S).       % by failure.
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).

%% The following clause treats the holds predicate for non fluents,
%% including Prolog system predicates.

holds(A,S) :- restoreSitArg(A,S,F), F ;
              not restoreSitArg(A,S,F), isAtom(A), A.

seq(A,Pi1, A : Pi1).

isAtom(A) :- not (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
       A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).

restoreSitArg(poss(A),S,poss(A,S)).

concurrentAction([A | C]) :- not A = choice(_,_).
```

6.2 Soccer Example

The Soccer Example of Section 4 can be implemented by the following Prolog program

```
%%      Soccer Example

agent(a). opponent(b).

%%      Actions

poss(move(Ag,X,Y),S) :- X = 0 ; Y = 0.
```

```
poss(C,S) :- allPoss(C,S).
allPoss([],S).
allPoss([A | R],S) :- poss(A,S), allPoss(R,S).

%%      Fluents

at(Ag,X,Y,do(C,S)) :-
        at(Ag, X, Y, S), not member(move(Ag,X1,Y1),C);
        at(Ag, X1, Y1, S), member(move(Ag,Z,T),C), X is Z + X1, Y is T + Y1.

haveBall(Ag,do(C,S)) :- (agent(Ag);opponent(Ag)),
         (haveBall(Ag,S), not looseBall(Ag,C,S);
          haveBall(Ag1,S), not Ag1 = Ag, looseBall(Ag1,C,S)).

looseBall(Ag1,C,S) :-
        (agent(Ag1), opponent(Ag2); agent(Ag2), opponent(Ag2) ),
                at(Ag1,X,Y,S),
                at(Ag2,X1,Y,S),
                at(Ag2,X1,Y,do(C,S)),
                at(Ag1,X1,Y,do(C,S)).

%%      Reward

goal(Ag,S) :- haveBall(Ag,S), at(Ag,X,Y,S), goalPos(Ag,X,Y).

reward(Ag,Rw,A,S) :- goal(Ag1,do(A,S)),
            (Ag1 = Ag, Rw is 1000; not Ag1 = Ag, Rw is -1000 ),!;
             haveBall(Ag1,do(A,S)), at(Ag1,X,Y,do(A,S)), evalPos(Ag1,X,Y,R),
             (Ag1 = Ag, Rw is R ; not Ag1 = Ag, Rw is -R ).

evalPos(Ag,X,Y,R) :-
        Ag = b, (X >0,X<7,!, R is X; R is 0);
        Ag = a, (X >0,X<7,!, R is 6-X; R is 0).

goalPos(a,0,Y) :- Y = 2; Y = 3.
goalPos(b,6,Y) :- Y = 2; Y = 3.
```

For example, consider an initial situation S_0, where agent a is in position $(2, 3)$ and has the ball, and agent b is in position $(1, 3)$:

```
at(a,2,3,s0). at(b,1,3,s0). haveBall(a,s0).
```

Consider then the following program schema, where first agent a can move by either $(-1, 0)$ or $(0, -1)$, while agent b can move by either $(0, -1)$ or $(0, 0)$, and then agent a can move by either $(-1, 0)$ or $(0, -1)$, while agent b can move by either $(0, 0)$ or $(0, -1)$, and finally agent a moves by $(-1, 0)$:

```
proc(schema,
        [choice(a,[move(a,-1,0),move(a,0,-1)]),choice(b,[move(b,0,-1),move(b,0,0)])]:
        [choice(a,[move(a,-1,0),move(a,0,-1)]),choice(b,[move(b,0,0),move(b,0,-1)])]:
        [move(a,-1,0)]).
```

Informally, the agent and the opponent are facing each other. The former has to perform a dribbling in order to score a goal, while the latter can try to guess the agent's move in order to change the ball possession. This action requires a mixed strategy, which can be generated by the following query:

```
doG(schema:nil,s0,H,Pi,R,Pr).
```

The computed results are as follows (where `alea` is a construct representing the probability distribution on possible choices, and `if` encodes a standard if-then-else statement):

```
Pi = alea([[[move(a,-1,0),move(a,0,-1)],[0.495785391289137,0.504214608710863]],
           [[move(b,0,-1),move(b,0,0)],[0.50371686477334,0.49628313522666]]],
   if(obsChoice(move(b,0,-1)),
      if(obsChoice(move(a,-1,0)),
         alea([[[move(a,-1,0),move(a,0,-1)],[0.0,1.0]],
               [[move(b,0,0),move(b,0,-1)],[0.0,1.0]]],
            if(obsChoice(move(b,0,0)),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil))),
         alea([[[move(a,-1,0),move(a,0,-1)],[0.00592300098716692,0.994076999012833]],
               [[move(b,0,0),move(b,0,-1)],[0.989141164856861,0.0108588351431392]]],
            if(obsChoice(move(b,0,0)),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil)))),
      if(obsChoice(move(a,-1,0)),
         alea([[[move(a,-1,0),move(a,0,-1)],[0.0,1.0]],
               [[move(b,0,0),move(b,0,-1)],[0.0,1.0]]],
            if(obsChoice(move(b,0,0)),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil))),
         alea([[[move(a,-1,0),move(a,0,-1)],[1.0,0.0]],
               [[move(b,0,0),move(b,0,-1)],[0.0,1.0]]],
            if(obsChoice(move(b,0,0)),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil),
               if(obsChoice(move(a,-1,0)),
                  [move(a,-1,0)]:nil,
                  [move(a,-1,0)]:nil))))))

R = 507.26518401539317
Pr = 1.0
Yes (0.27s cpu, solution 1, maybe more)
```

The above combined strategy yields the following two single-agent strategies for agents a and b:

```
%% Extracted strategy for agent a:

[move(a,-1,0),move(a,0,-1)]:[0.495785391289137,0.504214608710863];
if obsChoice(move(a,-1,0)) then move(a,0,-1)
   else if obsChoice(move(b,0,-1))
```

```
      then [move(a,-1,0),move(a,0,-1)]:[0.00592300098716692,0.9940769990128 33]
      else move(a,-1,0);
move(a,-1,0).


%% Extracted strategy for agent b:

[move(b,0,-1),move(b,0,0)]:[0.50371686477334,0.49628313522666];
if obsChoice(move(b,0,-1)) and obsChoice(move(a,0,-1))
   then [move(b,0,0),move(b,0,-1)]:[0.989141164856861,0.0108588351431392]
   else move(b,0,-1);
nil.
```

7 Summary and Outlook

We have presented the agent programming language GTGolog, which integrates explicit agent programming in Golog with game-theoretic multi-agent planning in Markov games. It is a generalization of DTGolog to a multi-agent setting, where we have two competing single agents or two competing teams of agents. The language allows for specifying a control program for a single agent or a team of agents in a high-level logical language. The control program is then completed by an interpreter in an optimal way against another single agent or another team of agents, by viewing it as a generalization of a Markov game, and computing a Nash strategy. We have illustrated the usefulness of this approach along a robotic soccer example. We have also described a prototype implementation of a GTGolog interpreter for the case of two competing agents.

An interesting topic of future research is to explore how the presented GTGolog framework can be generalized to also allow for partial observability.

References

[1] C. Boutilier. Sequential optimality and coordination in multiagent systems. In *Proceedings IJCAI-1999*, pp. 478–485. Morgan Kaufmann, 1999.

[2] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings AAAI/IAAI-2000*, pp. 355–362. AAAI Press / MIT Press, 2000.

[3] A. Finzi and F. Pirri. Combining probabilities, failures and safety in robot control. In *Proceedings IJCAI-2001*, pp. 1331–1336. Morgan Kaufmann, 2001.

[4] M. Kearns, Y. Mansour, and S. Singh. Fast planning in stochastic games. In *Proceedings UAI-2000*, pp. 309–316. Morgan Kaufmann, 2000.

[5] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings ICML-1994*, pp. 157–163. Morgan Kaufmann, 1994.

[6] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In *Machine Intelligence*, volume 4, pp. 463–502. Edinburgh University Press, 1969.

[7] G. Owen. *Game Theory: Second Edition*. Academic Press, 1982.

[8] D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell.*, 94(1–2):7–56, 1997.

[9] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[10] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 359–380. Academic Press, 1991.

[11] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[12] J. van der Wal. *Stochastic Dynamic Programming*, volume 139 of *Mathematical Centre Tracts*. Morgan Kaufmann, 1981.

[13] J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, 1947.