Adaptive Multi-Agent Programming in GTGolog

Alberto Finzi[1, 2] and Thomas Lukasiewicz[2, 1]

[1] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria

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

Abstract. We present a novel approach to adaptive multi-agent programming, which is based on an integration of the agent programming language GTGolog with adaptive dynamic programming techniques. GTGolog combines explicit agent programming in Golog with multi-agent planning in stochastic games. A drawback of this framework, however, is that the transition probabilities and reward values of the domain must be known in advance and then cannot change anymore. But such data is often not available in advance and may also change over the time. The adaptive generalization of GTGolog in this paper is directed towards letting the agents themselves explore and adapt these data, which is more useful for realistic applications. We use high-level programs for generating both abstract states and optimal policies, which benefits from the deep integration between action theory and high-level programs in the Golog framework.

1 Introduction

In the recent years, 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. An integration of both approaches for multi-agent systems has recently been proposed through the language GTGolog [7] (generalizing DTGolog [3]), which integrates explicit agent programming in Golog [20] with game-theoretic multi-agent planning in stochastic games [16]. It allows for partially specifying a high-level control program (for a system of two competing agents or two competing teams of agents) in a high-level language as well as for optimally filling in missing details through game-theoretic multi-agent planning.

However, a drawback of GTGolog (and also of DTGolog) is that the transition probabilities and reward values of the domain must be known in advance and then cannot change anymore. But such data often cannot be provided in advance in the model. It would thus be more useful for realistic applications to make the agents themselves capable of estimating, exploring, and adapting these data.

This is the main motivating idea behind this paper. We present a novel approach to adaptive multi-agent programming, which is an integration of GTGolog with reinforcement learning as in [13]. We use high-level programs for generating both abstract

states and policies over these abstract states. The generation of abstract states exploits the structured encoding of the domain in a basic action theory, along with the high-level control knowledge in a Golog program. A learning process then incrementally adapts the model to the executive context and instantiates the partially specified behavior.

To our knowledge, this is the first adaptive approach to Golog interpreting. Differently from classical Golog, here the interpreter generates not only complex sequences of actions, but also an abstract state space for each machine state. Similarly to [2,11], we rely on the situation calculus machinery for state abstraction, but in our system the state generation is driven by the program structure. Here, we can take advantage from the deep integration between the action theory and programs provided by Golog: deploying the Golog semantics and the domain theory, we can produce a tailored state abstraction for each program state. In this way, we can extend the scope of programmable learning techniques [4,17,5,1,14] to a logic-based agent [12,20,21] and multi-agent [6] programming framework: the choice points of partially specified programs are associated with a set of state formulas and are instantiated through reinforcement learning and dynamic programming constrained by the program structure.

2 Preliminaries

In this section, we recall the basics of the situation calculus and Golog, matrix games, stochastic games, and reinforcement learning.

2.1 The Situation Calculus and Golog

The situation calculus [15,20] is a first-order language for representing dynamically changing worlds. Its main ingredients are *actions*, *situations*, and *fluents*. An *action* is a first-order term of the form $a(u_1, \ldots, u_n)$, where the function symbol a is its *name* and the u_i's are its *arguments*. All changes to the world are the result of actions. For example, the action $moveTo(r, x, y)$ may stand for moving the agent r to the position (x, y). 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 do is a distinguished binary function symbol, 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 of s. For example, the situation $do(moveTo(r, 1, 2), do(moveTo(r, 3, 4), S_0))$ stands for executing $moveTo(r, 1, 2)$ after executing $moveTo(r, 3, 4)$ in the initial situation S_0. We write $Poss(a, s)$, where $Poss$ is a distinguished binary predicate symbol, to denote that the action a is possible to execute in the situation s. A *(relational) fluent* represents a world or agent property that may change when executing an action. It is a predicate symbol whose most right argument is a situation. For example, $at(r, x, y, s)$ may express that the agent r is at the position (x, y) in the situation s. In the situation calculus, a dynamic domain is represented by a *basic action theory* $AT = (\Sigma, \mathcal{D}_{una}, \mathcal{D}_{S_0}, \mathcal{D}_{ssa}, \mathcal{D}_{ap})$, where:

- Σ is the set of (domain-independent) foundational axioms for situations [20].
- $\mathcal{D}_{una}$ is the set of unique names axioms for actions, which express that different actions are interpreted in a different way.

- $\mathcal{D}_{S_0}$ is a set of first-order formulas describing the initial state of the domain (represented by S_0). For example, $at(r, 1, 2, S_0) \wedge at(r', 3, 4, S_0)$ may express that the agents r and r' are initially at the positions $(1, 2)$ and $(3, 4)$, respectively.
- $\mathcal{D}_{ssa}$ is the set of *successor state axioms* [20]. For each fluent $F(\boldsymbol{x}, s)$, it contains an axiom of the form $F(\boldsymbol{x}, do(a, s)) \equiv \Phi_F(\boldsymbol{x}, a, s)$, where $\Phi_F(\boldsymbol{x}, a, s)$ is a formula with free variables among $\boldsymbol{x}, a, s$. These axioms specify the truth of the fluent F in the next situation $do(a, s)$ in terms of the current situation s, and are a solution to the frame problem (for deterministic actions). For example, the axiom $at(r, x, y, do(a, s)) \equiv a = moveTo(r, x, y) \vee (at(r, x, y, s) \wedge \neg\exists x', y' \, (a = moveTo(r, x', y')))$ may express that the agent r is at the position (x, y) in the situation $do(a, s)$ iff either r moves to (x, y) in the situation s, or r is already at the position (x, y) and does not move away in s.
- $\mathcal{D}_{ap}$ is the set of *action precondition axioms*. For each action a, it contains an axiom of the form $Poss(a(\boldsymbol{x}), s) \equiv \Pi(\boldsymbol{x}, s)$, which characterizes the preconditions of the action a. For example, $Poss(moveTo(r, x, y), s) \equiv \neg\exists r' \, at(r', x, y, s)$ may express that it is possible to move the agent r to the position (x, y) in the situation s iff no other agent r' is at (x, y) in s.

We use the concurrent version of the situation calculus [20,18], which is an extension of the standard situation calculus by concurrent actions. A *concurrent action* c is a set of standard actions, which are concurrently executed when c is executed.

The *regression* of a formula ϕ through an action a, denoted $Regr(\phi)$, is a formula ϕ' that holds before executing a, given that ϕ holds after executing a. The regression of ϕ whose situations are all of the form $do(a, s)$ is defined inductively using the successor state axioms $F(\boldsymbol{x}, do(a, s)) \equiv \Phi_F(\boldsymbol{x}, a, s)$ as follows:

$$Regr(F(\boldsymbol{x}, do(a, s))) = \Phi_F(\boldsymbol{x}, a, s), \quad Regr(\neg\phi) = \neg Regr(\phi),$$
$$Regr(\phi_1 \wedge \phi_2) = Regr(\phi_1) \wedge Regr(\phi_2), \text{ and } \quad Regr(\exists x \, \phi) = \exists x \, (Regr(\phi)) \, .$$

Golog [12,20] 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 AT, where standard (and not so standard) Algol-like control constructs can be used, in particular, (i) sequence: $p_1; p_2$; (ii) test action: $\phi?$; (iii) nondeterministic choice of two programs: $(p_1 \mid p_2)$; (iv) nondeterministic choice of an argument: $\pi x \, (p(x))$; and (v) conditional, while-loop, and procedure. For example, the Golog program **while** $\neg at(r, 1, 2)$ **do** $\pi x, y \, (moveTo(r, x, y))$ repeats moving the agent r to a nondeterministically chosen position (x, y) while r is not at $(1, 2)$. The semantics of a Golog program p is specified by a situation-calculus formula $Do(p, s, s')$, which encodes that s' is a situation which can be reached from s by executing p. That is, 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 [20].

2.2 Matrix Games

Matrix games from classical game theory [23] describe the possible actions of two agents and the rewards that they receive when they simultaneously execute one action

each. Formally, a *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* $R_{\boldsymbol{a}}, R_{\boldsymbol{o}}: A \times O \to \mathbf{R}$ for $\boldsymbol{a}$ and $\boldsymbol{o}$. The matrix game G is *zero-sum* iff $R_{\boldsymbol{a}} = -R_{\boldsymbol{o}}$; we then often omit $R_{\boldsymbol{o}}$ and abbreviate $R_{\boldsymbol{a}}$ by R.

A *pure* (resp., *mixed*) *strategy* specifies which action an agent should execute (resp., which actions an agent should execute with which probability). If the 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. If the agents $\boldsymbol{a}$ and $\boldsymbol{o}$ play the mixed strategies $\pi_{\boldsymbol{a}} \in PD(A)$ and $\pi_{\boldsymbol{o}} \in PD(O)$, 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 $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$, 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: $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ is a *Nash equilibrium* (or *Nash pair*) for G iff (i) $R_{\boldsymbol{a}}(\pi'_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) \leq R_{\boldsymbol{a}}(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ for any mixed $\pi'_{\boldsymbol{a}}$, and (ii) $R_{\boldsymbol{o}}(\pi_{\boldsymbol{a}}, \pi'_{\boldsymbol{o}}) \leq R_{\boldsymbol{o}}(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ for any mixed $\pi'_{\boldsymbol{o}}$. Every two-player matrix game G has at least one Nash pair among its mixed (but not necessarily pure) strategy pairs, and many have multiple Nash pairs.

2.3 Stochastic Games

Stochastic games [16], or also called Markov games [22,13], generalize both matrix games [23] and (fully observable) Markov decision processes (MDPs) [19].

They consist 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 $s' \in S$. Formally, a *(two-player) stochastic 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 $P: S \times A \times O \to PD(S)$, and two *reward functions* $R_{\boldsymbol{a}}, R_{\boldsymbol{o}}: S \times A \times O \to \mathbf{R}$ for $\boldsymbol{a}$ and $\boldsymbol{o}$. The stochastic game G is *zero-sum* iff $R_{\boldsymbol{a}} = -R_{\boldsymbol{o}}$; we then often omit $R_{\boldsymbol{o}}$.

Assuming a finite horizon $H \geq 0$, 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. 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 $G_k(0, s, \alpha, \omega) = R_k(s, \alpha(s, 0), \omega(s, 0))$ and $G_k(H, s, \alpha, \omega) = R_k(s, \alpha(s, H), \omega(s, H)) + \sum_{s' \in S} P(s' | s, \alpha(s, H), \omega(s, H)) \cdot G_k(H-1, s', \alpha, \omega)$ for $H > 0$. The notions of an *expected H-step reward* for mixed policies and of a *Nash pair* can then be defined in a standard way. Every two-player stochastic 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.

2.4 Learning Optimal Policies

Q-learning [24] is a reinforcement learning technique, which allows to solve an MDP without a model (that is, transition and reward functions) and can be used on-line. The value $Q(s, a)$ is the expected discounted sum of future payoffs obtained by executing a from the state s and following an optimal policy. After being initialized to arbitrary numbers, the Q-values are estimated through the agent experience. For each execution of an action a leading from the state s to the state s', the agent receives a reward r, and

the Q-value update is $Q(s,a) := (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a' \in A} Q(s',a'))$, where γ (resp., α) is the discount factor (resp., the learning rate). This algorithm converges to the correct Q-values with probability 1 assuming that every action is executed in every state infinitely many times and α is decayed appropriately.

Littman [13] extends Q-learning to learning an optimal mixed policy in a zero-sum two-player stochastic game. Here, the Q-value update is $Q(s,a,o) := (1-\alpha) \cdot Q(s,a,o) + \alpha \cdot (r + \gamma \cdot \max_{\pi \in PD(A)} \min_{o' \in O} \sum_{a' \in A} Q(s',a',o') \cdot \pi(a'))$, where the "maxmin"-term gives the expected reward of a Nash pair for a zero-sum matrix game.

3 Adaptive GTGolog (AGTGolog)

In this section, we first define the domain theory behind Adaptive GTGolog (AGTGolog) and then the syntax of AGTGolog.

3.1 Domain Theory of AGTGolog

A *domain theory* $DT = (AT, ST, OT)$ of AGTGolog consists of a basic action theory AT, a *stochastic theory* ST, and an *optimization theory* OT, as defined below.

We first give some preliminaries. 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). The set of primitive actions is partitioned into the sets of primitive actions A and O of agents $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively. A *two-player action* is any concurrent action c over $A \cup O$ such that $|c \cap A| \leq 1$ and $|c \cap O| \leq 1$. For example, the concurrent actions $\{moveTo(\boldsymbol{a}, 1, 2)\} \subseteq A$, $\{moveTo(\boldsymbol{o}, 2, 3)\} \subseteq O$, and $\{moveTo(\boldsymbol{a}, 1, 2),\ moveTo(\boldsymbol{o}, 2, 3)\} \subseteq A \cup O$ are all two-player actions. We often write a, o, and $a \| o$ to abbreviate $\{a\} \subseteq A$, $\{o\} \subseteq O$, and $\{a, o\} \subseteq A \cup O$, respectively.

A *state formula* over $\boldsymbol{x}, s$ is a formula $\phi(\boldsymbol{x}, s)$ in which all predicate symbols are fluents, and the only free variables are the non-situation variables $\boldsymbol{x}$ and the situation variable s. A *state partition* over $\boldsymbol{x}, s$ is a nonempty set of state formulas $P(\boldsymbol{x}, s) = \{\phi_i(\boldsymbol{x}, s) \mid i \in \{1, \ldots, m\}\}$ such that (i) $\forall \boldsymbol{x}, s\, (\phi_i(\boldsymbol{x}, s) \Rightarrow \neg \phi_j(\boldsymbol{x}, s))$ is valid for all $i, j \in \{1, \ldots, m\}$ with $j > i$, (ii) $\forall \boldsymbol{x}, s\, (\bigvee_{i=1}^{m} \phi_i(\boldsymbol{x}, s))$ is valid, and (iii) every $\exists \boldsymbol{x}, s\, (\phi_i(\boldsymbol{x}, s))$ is satisfiable. For state partitions P_1 and P_2, we define their *product* as follows:

$$P_1 \otimes P_2 = \{\psi_1 \wedge \psi_2 \mid \psi_1 \in P_1,\ \psi_2 \in P_2,\ \psi_1 \wedge \psi_2 \neq \bot\}\,.$$

We often omit the arguments of a state formula when they are clear from the context.

We next define the stochastic theory. As usual [3,10,2], stochastic actions are expressed by 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. We also specify a state partition $P_{pr}^{a,n}(\boldsymbol{x}, s) = \{\phi_j^{a,n}(\boldsymbol{x}, s) \mid j \in \{1, \ldots, m\}\}$ to group together situations s with common p such that "nature" chooses n in s with probability p, denoted $prob(a(\boldsymbol{x}), n(\boldsymbol{x}), s) = p$:

$$\exists p_1, \ldots, p_m\, (\textstyle\bigwedge_{j=1}^{m} (\phi_j^{a,n}(\boldsymbol{x}, s) \Leftrightarrow prob(a(\boldsymbol{x}), n(\boldsymbol{x}), s) = p_j))\,.$$

A stochastic action s is indirectly represented by providing a *successor state axiom* for each associated nature choice n. Thus, AT is extended to a probabilistic setting in a minimal way. We assume that the domain is *fully observable*. For this reason, 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 $at(a,x,y,s)$ and $at(a,x,y+1,s)$ to see which of the deterministic actions was executed (that is, $moveTo(a,x,y)$ or $moveTo(a,x,y+1)$). This condition is denoted $condSta(a,n,s)$. For example, $condSta(moveS(a,x,y), moveTo(a,x,y+1), s) \equiv at(a,x,y+1,s)$. Similar axioms are introduced to observe which actions the two agents have chosen.

As for the optimization theory, for every two-player action a, we specify a state partition $P^a_{rw}(\boldsymbol{x},s) = \{\phi^a_k(\boldsymbol{x},s) \mid k \in \{1,\ldots,q\}\}$ to group together situations s with common r such that $a(\boldsymbol{x})$ and s assign the reward r to $\boldsymbol{a}$, denoted $reward(a(\boldsymbol{x}),s) = r$:

$$\exists r_1,\ldots,r_q \left(\textstyle\bigwedge_{k=1}^{q}(\phi^a_k(\boldsymbol{x},s) \Leftrightarrow reward(a(\boldsymbol{x}),s) = r_k)\right).$$

Moreover, a utility function associates with every reward v and success probability pr a real-valued utility $utility(v,pr)$. We assume that $utility(v,1) = v$ and $utility(v,0) = 0$ for all v. An example of such a function is $utility(v,pr) = v \cdot pr$.

Example 3.1 (Stratagus Domain). Consider the following scenario inspired by [14]. The stratagus field consists of 9×9 positions (see Fig. 1). There are two agents, denoted $\boldsymbol{a}$ and $\boldsymbol{o}$, which occupy one position each. The stratagus field has designated areas representing two *gold-mines*, one *forest*, and one *base* for each agent (see Fig. 1). The two agents can move one step in one of the directions *N*, *S*, *E*, and *W*, or remain stationary. Each of the two agents can also pick up one unit of wood (resp., gold) at the forest (resp., gold-mines), and drop these resources at its base. Each action of the two agents can fail, resulting in a stationary move. Any carried object drops when the two agents collide. After each step, the agents $\boldsymbol{a}$ and $\boldsymbol{o}$ receive the (zero-sum) rewards $r_{\boldsymbol{a}} - r_{\boldsymbol{o}}$ and $r_{\boldsymbol{o}} - r_{\boldsymbol{a}}$, respectively, where r_k for $k \in \{\boldsymbol{a},\boldsymbol{o}\}$ is 0, 1, and 2 when k brings nothing, one unit of wood, and one unit of gold to its base, respectively.

The domain theory $DT = (AT, ST, OT)$ for the above stratagus domain is defined as follows. As for the basic action theory AT, we assume the deterministic actions $move(\alpha,m)$ (agent α performs m among N, S, E, W, and $stand$), $pickUp(\alpha,o)$ (agent α picks up the object o), and $drop(\alpha,o)$ (agent α drops the object o), as well as the relational fluents $at(q,x,y,s)$ (agent or object q is at the position (x,y) in the situation s), and $holds(\alpha,o,s)$ (agent α holds the object o in the situation s), which are defined through the following successor state axioms:

$$\begin{aligned}
&at(q,x,y,do(c,s)) \equiv agent(q) \wedge (at(q,x,y,s) \wedge move(q,stand) \in c \vee\\
&\quad \exists x',y'\,(at(q,x',y',s) \wedge \exists m\,(move(\alpha,m) \in c \wedge \phi(x,y,x',y',m)))) \vee\\
&\quad object(q) \wedge (at(q,x,y,s) \wedge \neg\exists\alpha\,(pickUp(\alpha,q) \in c) \vee\\
&\quad \exists\alpha\,((drop(\alpha,q) \in c \vee collision(c,s)) \wedge at(\alpha,x,y,s) \wedge holds(\alpha,q,s)));\\
&holds(\alpha,o,do(c,s)) \equiv holds(\alpha,o,s) \wedge\\
&\quad drop(\alpha,o) \notin c \wedge \neg collision(c,s) \vee pickUp(\alpha,o) \in c.
\end{aligned}$$

Here, $\phi(x,y,x',y',m)$ represents the coordinate change due to m, and $collision(c,s)$ encodes that the concurrent action c causes a collision between the agents $\boldsymbol{a}$ and $\boldsymbol{o}$ in

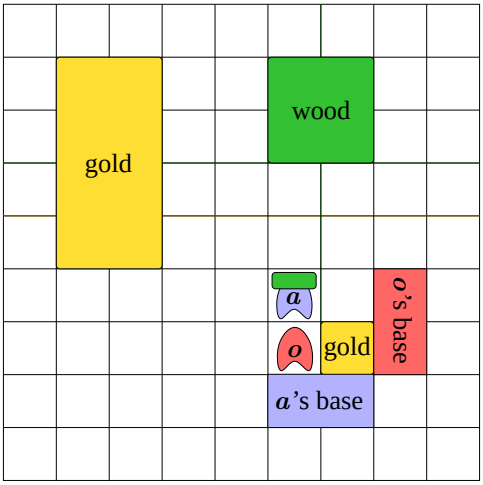

Fig. 1. Stratagus Domain

the situation s. The deterministic actions $move(\alpha, m)$, $drop(\alpha, o)$, and $pickUp(\alpha, o)$ are associated with precondition axioms as follows:

$$Poss(move(\alpha, m), s) \equiv \top\,;$$
$$Poss(drop(\alpha, o), s) \equiv holds(\alpha, o, s)\,;$$
$$Poss(pickUp(\alpha, o), s) \equiv \neg\exists x\; holds(\alpha, x, s)\,.$$

Furthermore, we assume the following additional precondition axiom, which encodes that two agents cannot pick up the same object at the same time (where $\alpha \neq \beta$):

$$Poss(\{pickUp(\alpha, o), pickUp(\beta, o)\}, s) \equiv$$
$$\exists x, y, x', y'(at(\alpha, x, y, s) \wedge at(\beta, x', y', s) \wedge (x \neq x' \vee y \neq y'))\,.$$

As for the stochastic theory ST, we assume the stochastic actions $moveS(\alpha, m)$ (agent α executes m among N, S, E, W, and $stand$), $pickUpS(\alpha, o)$ (agent α picks up the object o), $dropS(\alpha, o)$ (agent α drops the object o), which may succeed or fail. We assume the state partition $P^{a,n}_{pr} = \{\top\}$ for each pair consisting of a stochastic action and one of its deterministic components:

$$\exists p\,(prob(pickUpS(\alpha, o), pickUp(\alpha, o), s) = p)\,;$$
$$\exists p\,(prob(pickUpS(\alpha, o), move(\alpha, stand), s) = p)\,;$$
$$\exists p\,(prob(dropS(\alpha, o), drop(\alpha, o), s) = p)\,;$$
$$\exists p\,(prob(dropS(\alpha, o), move(\alpha, stand), s) = p)\,;$$
$$\exists p\,(prob(moveS(\alpha, d), move(\alpha, d), s) = p)\,;$$
$$\exists p\,(prob(moveS(\alpha, d), move(\alpha, stand), s) = p)\,;$$
$$\exists p\,(prob(a \| o, a' \| o', s) = p \equiv \exists p_1, p_2\,(prob(a, a', s) = p_1 \wedge$$
$$prob(o, o', s) = p_2 \wedge p = p_1 \cdot p_2))\,.$$

As for the optimization theory OT, we use the product as the utility function $utility$. Furthermore, we define the reward function $reward$ as follows:

$$reward(\alpha, a, s) = r \equiv \exists r_\alpha, r_\beta \, (rewAg(\alpha, a, s) = r_\alpha \wedge \\ \exists \beta \, (rewAg(\beta, a, s) = r_\beta) \wedge r = r_\alpha - r_\beta) \, ; \\ \exists r_1, \ldots, r_m \, (\textstyle\bigwedge_{j=1}^{m} (\phi_j^{\alpha,a}(s) \Leftrightarrow rewAg(\alpha, a, s) = r_j)) \, .$$

Here, $\phi_j^{\alpha,a}(\boldsymbol{x}, s)$ belongs to $P_{rw}^{\alpha,a}$, which is defined as follows. If $a = moveS(\alpha, x, y)$, then $P_{rw}^{\alpha,a} = \{\top\}$; if $a = pickUpS(\alpha, o)$, then $P_{rw}^{\alpha,a} = \{\neg h \wedge atg, \neg h \wedge atw, \neg h \wedge ato, h\}$; if $a = dropS(\alpha, o)$, then $P_{rw}^{\alpha,a} = \{hw \wedge atb, hg \wedge atb, \neg atb \wedge h, \neg h\}$, where h, atg, atw, atb, hg, hw, ato are formulas that stand for α holding something, being at the gold-mine, being at the wood, being at the base, holding gold, holding wood, and being close to an object, respectively.

3.2 Syntax of AGTGolog

AGTGolog has the same syntax as standard GTGolog: Given the actions specified by a domain theory DT, a *program* p in AGTGolog has one of the following forms (where α is a two-player action, ϕ is a condition, p, p_1, p_2 are programs, and $a_1, \ldots, a_n$ and $o_1, \ldots, o_m$ are actions of agents $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively):

1. *Deterministic or stochastic action:* α. Do α.
2. *Nondeterministic action choice of* $\boldsymbol{a}$*:* **choice**$(\boldsymbol{a} : a_1 | \cdots | a_n)$.
   Do an optimal action (for agent $\boldsymbol{a}$) among $a_1, \ldots, a_n$.
3. *Nondeterministic action choice of* $\boldsymbol{o}$*:* **choice**$(\boldsymbol{o} : o_1 | \cdots | o_m)$.
   Do an optimal action (for agent $\boldsymbol{o}$) among $o_1, \ldots, o_m$.
4. *Nondeterministic joint action choice:*
   choice$(\boldsymbol{a} : a_1 | \cdots | a_n) \parallel$ **choice**$(\boldsymbol{o} : o_1 | \cdots | o_m)$.
   Do any action $a_i \| o_j$ with an optimal probability $\pi_{i,j}$.
5. *Test action*: $\phi?$. Test the truth of ϕ in the current situation.
6. *Sequence*: $p_1; p_2$. Do p_1 followed by p_2.
7. *Nondeterministic choice of two programs:* $(p_1 \,|\, p_2)$. Do p_1 or p_2.
8. *Nondeterministic choice of an argument:* $\pi x \, (p(x))$. Do any $p(x)$.
9. *Nondeterministic iteration*: $p^\star$. Do p zero or more times.
10. *Conditional*: **if** ϕ **then** p_1 **else** p_2.
11. *While-loop*: **while** ϕ **do** p.
12. *Procedures, including recursion.*

Example 3.2 (Stratagus Domain cont'd). We define some AGTGolog programs relative to the domain theory $DT = (AT, ST, OT)$ of Example 3.1. The following AGTGolog procedure $carryToBase$ describes a partially specified behavior where agent $\boldsymbol{a}$ is trying to move to its base in order to drop down an object:

proc $carryToBase$
choice$(\boldsymbol{a} : moveS(\boldsymbol{a}, N) \,|\, moveS(\boldsymbol{a}, S) \,|\, moveS(\boldsymbol{a}, E) \,|\, moveS(\boldsymbol{a}, W))$;
if $atBase$ **then** $\pi x \, (dropS(\boldsymbol{a}, x))$
  else $carryToBase$
end.

The subsequent procedure $pickProc(x)$ encodes that if the two agents $\boldsymbol{a}$ and $\boldsymbol{o}$ are at the same location, then they have to compete in order to pick up an object, otherwise agent $\boldsymbol{a}$ can directly use the primitive action $pickUpS(\boldsymbol{a}, x)$:

proc $pickProc(x)$
if $atSameLocation(\boldsymbol{a}, \boldsymbol{o})$ **then** $tryToPickUp(x)$
  else $pickUpS(\boldsymbol{a}, x)$
end.

Here, the joint choices of the two agents $\boldsymbol{a}$ and $\boldsymbol{o}$ when they are at the same location are specified by the following procedure $tryToPickUp(x)$ (which will be instantiated by a *mixed policy*):

proc $tryToPickUp(x)$
choice$(\boldsymbol{a}: pickUpS(\boldsymbol{a}, x) \mid moveS(\boldsymbol{a}, stand)) \parallel$
  choice$(\boldsymbol{o}: pickUpS(\boldsymbol{o}, x) \mid moveS(\boldsymbol{o}, stand))$
end.

4 Learning Optimal Policies

We now define state partitions SF for finite-horizon AGTGolog programs p. We then show how to learn an optimal policy for p. Intuitively, given a horizon $h \geq 0$, an h-step policy π of p relative to a domain theory is obtained from the h-horizon part of p by replacing every single-agent choice by a single action, and every multi-agent choice by a collection of probability distributions, one over the actions of each agent.

4.1 State Partition Generation

Given a GTGolog program p, a *machine state* consists of a subprogram p' of p and a horizon h. A *joint state* (ϕ, p', h) consists of a state formula ϕ and a machine state (p', h). Note that the joint state represents both the state of the environment and the executive state of the agent. Every machine state (p, h) is associated with a state partition, denoted $SF(p, h) = \{\phi_1(\boldsymbol{x}, s), \ldots, \phi_m(\boldsymbol{x}, s)\}$, which is inductively defined relative to the main constructs of AGTGolog (and naturally extended to all the other constructs) by:

1. Null program or zero horizon:

   $$SF(nil, h) = SF(p, 0) = \{\top\}.$$

   At the program or horizon end, the state partition is given by $\{\top\}$.
2. Deterministic first program action:

   $$SF(a; p', h) = P^a_{rw}(\boldsymbol{x}, s) \otimes \{Regr(\phi(\boldsymbol{x}, do(a, s)) \wedge Poss(a, s) \mid \phi(\boldsymbol{x}, s) \in SF(p', h-1)\} \cup \{\neg Poss(a, s)\} \setminus \{\bot\}.$$

   Here, the state partition for $a; p'$ with horizon h is obtained as the product of the reward partition $P^a_{rw}(\boldsymbol{x}, s)$, the state partition $SF(p', h-1)$ of the next machine state $(p', h-1)$, and the executability partition $\{\neg Poss(a, s),\ Poss(a, s)\}$.

3. Stochastic first program action (nature choice):

$$SF(a; p', h) = \bigotimes_{i=1}^{k} (SF(n_i; p', h) \otimes P_{pr}^{a,n_i}(\boldsymbol{x}, s)),$$

where $n_1, \ldots, n_k$ are the deterministic components of a. That is, the partition for $a; p'$ in h, where a is stochastic, is the product of the state partitions $SF(n_i; p', h)$ relative to the deterministic components n_i of a combined with the partitions P_{pr}^{a,n_i}.

4. Nondeterministic first program action (choice of agent k):

$$SF(\textbf{choice}(k\colon a_1 | \cdots | a_n); p', h) = \bigotimes_{i=1}^{n} SF(a_i; p', h),$$

where $a_1, \ldots, a_n$ are two-player actions. That is, the state partition for a single choice of actions is the product of the state partitions for the possible choices. Note that the state partition for the joint choice of both agents is defined in a similar way.

5. Nondeterministic choice of two programs:

$$SF((p_1 \mid p_2); p', h) = SF(p_1; p', h) \otimes SF(p_2; p', h).$$

The state partition for a nondeterministic choice of two programs is obtained as the product of the state partitions associated with the possible programs.

6. Test action:

$$SF(\phi?; p', h) = \{\phi, \neg\phi\} \otimes SF(p', h).$$

The partition for $(\phi?; p', h)$ is obtained by composing the partition $\{\phi, \neg\phi\}$ induced by the test $\phi?$ with the state partition for (p', h).

4.2 Learning Algorithm

The main learning algorithm is $Learn$ in Algorithm 1. For each joint state $\sigma = (\phi, p, h)$, where (p, h) is a machine state and $\phi \in SF(p, h)$, it generates an optimal h-step policy of p in ϕ, denoted $\pi(\sigma)$. We use a hierarchical version of Q-learning.

More concretely, the algorithm takes as input a program state (p, h) and generates as output an optimal policy π for each associated joint state (ϕ, p, h). In line 1, we initialize the *learning rate* α to 1; it decays at each learning cycles according to $decay$. In line 2, we also initialize to $\langle 1, 1\rangle$ the variables $\langle v, pr\rangle$ representing the current *value function* (or *v-function*). At each cycle, the current state $\phi \in SF(p, h)$ is estimated (that is, the agent evaluates which of the state formulas describes the current state of the world). Then, from the joint state $\sigma = (\phi, p, h)$, the procedure $Update(\phi, p, h)$ (see Section 4.3) executes the program p with horizon h, and updates and refines the v-function $\langle v, pr\rangle$ and the policy π. At the end of the execution of $Update$, if the learning rate is greater than a suitable threshold ε, then the current state ϕ is estimated and a new learning cycle starts. At the end of the algorithm $Learn$, for suitable $decay$ and ε, each possible execution of (p, h), from each ϕ, is performed often enough to obtain the convergency. That is, the agent executes the program (p, h) several times refining its v-function $\langle v, pr\rangle$ and policy π until an optimal behavior is reached.

Algorithm 1 $Learn(p, h)$

Require: AGTGolog program p and finite horizon h.
Ensure: optimal policy $\pi(\phi, p, h)$ for all $\phi \in SF(p, h)$.

1: $\alpha := 1$;
2: **for each** joint state σ **do** $\langle v, pr\rangle(\sigma) := \langle 1, 1\rangle$;
3: **repeat**
4: estimate $\phi \in SF(p, h)$;
5: $Update(\phi, p, h)$;
6: $\alpha := \alpha \cdot decay$
7: **until** $\alpha < \varepsilon$;
8: **return** $(\pi(\phi, p, h))_{\phi \in SF(p,h)}$.

4.3 Updating Step

The procedure $Update(\phi, p, h)$ in Algorithms 2 and 3 (parts 1 and 2, respectively) implements the execution and update step of a Q-learning algorithm. Here, each joint state σ of the program is associated with a variable $\langle v, pr\rangle(\sigma)$, which stores the current value of the v-function, and the variable $\pi(\sigma)$, which contains the current optimal policy at σ. Notice here that $\langle v, pr\rangle(\sigma)$ collects the cumulated reward v and probability pr of successful execution of σ, and $utility(\langle v, pr\rangle)$ is the associated utility. The procedure $Update(\phi, p, h)$ updates these value during an execution of a program p with horizon h, from a state $\phi \in SF(p, h)$. It is recursive, following the structure of the program.

Algorithm 2 describes the first part of the procedure $Update(\phi, p, h)$. Lines 1–4 encode the base of the induction: if the program is empty or the horizon is 0, then we set the v-function to $\langle 0, 1\rangle$, that is, reward 0 and success probability 1. In lines 5–8, we consider the nonexecutable cases: if a primitive action a is not executable in the current state (here, $\neg Poss(a, \phi)$ abbreviates $DT \cup \phi \models \neg Poss(a, s)$) or a test failed in the current situation (here, $\neg\psi[\phi]$ stands for $DT \cup \phi \models \neg\psi(s)$), then we have the reward 0 and the success probability 0. In lines 9–15, we describe the execution of a primitive action a from $(\phi,\ a; p,\ h)$ (here, $Poss(a, \phi)$ is a shortcut for $DT \cup \phi \models Poss(a, s)$): after the execution, the agent receives a $reward$ from the environment. Here, the update of the v-function and of the policy π is postponed to the execution $Update(do(a, \phi), p', h-1)$ of the rest of the program, $(p', h-1)$, from the next state formula $do(a, \phi)$, that is, the state formula $\phi' \in SF(p', h-1)$ such that $Regr(\phi'(do(a, s)))$ equals to $\phi(s)$ relative to DT. Then, the v-function $\langle v, pr\rangle$ is updated as for Q-learning ($v(do(a, \phi), p', h)$ is for a Q-value for a in σ), while the success probability pr is inherited from the next joint state. In lines 16–22, we consider the stochastic action execution: after the execution, we observe a reward and the executed deterministic component n_q, then we update as in the deterministic case. The generated strategy is a conditional plan where each possible execution is considered. Here, ϕ_i are the conditions to discriminate the executed component (represented by the *observability axioms*).

The core of the learning algorithm (lines 24–55) is in Algorithm 3, where we show the second part of the procedure $Update(\phi, p, h)$. This code collects the agent choice constructs and describes how the agent learns an optimal probability distribution over the possible options in the choice points. Here, the algorithm selects one possible

Algorithm 2 $Update(\phi, p, h)$: Part 1

Require: state formula ϕ, AGTGolog program p, and finite horizon h.
Ensure: updates $\langle v, pr\rangle(\sigma)$ and $\pi(\sigma)$, where $\sigma = (\phi, p, h)$.

1: **if** $p = nil \vee h = 0$ **then**
2: $\quad\langle v, pr\rangle(\sigma) := \langle 0, 1\rangle$;
3: $\quad\pi(\sigma) := stop$
4: **end if**;
5: **if** $p = a; p' \wedge \neg Poss(a, \phi) \vee p = \psi?; p' \wedge \neg\psi[\phi]$ **then**
6: $\quad\langle v, pr\rangle(\sigma) := \langle 0, 0\rangle$;
7: $\quad\pi(\sigma) := stop$
8: **end if**;
9: **if** $p = a; p' \wedge Poss(a, \phi)$ and a is deterministic **then**
10: $\quad$execute a and observe $reward$;
11: $\quad Update(do(a, \phi), p', h{-}1)$;
12: $\quad\langle v, pr\rangle(\sigma) := \langle(1 - \alpha) \cdot v(\sigma) + \alpha \cdot (reward +$
13: $\qquad\qquad\gamma \cdot v(do(a, \phi), p', h{-}1)), pr(do(a, \phi), p', h{-}1)\rangle$;
14: $\quad\pi(\sigma) := a; \pi'(do(a, \phi), p', h{-}1)$
15: **end if**;
16: **if** $p = a; p' \wedge Poss(a, \phi)$ and a is stochastic **then**
17: $\quad$"nature" selects any deterministic action n_q of the action a;
18: $\quad Update(\phi, n_q; p', h)$;
19: $\quad\langle v, pr\rangle(\sigma) := \langle v, pr\rangle(\phi, n_q; p', h)$;
20: $\quad\pi(\sigma) := a;$ **if** ϕ_1 **then** $\pi(\phi, n_1; p', h)$ ...
21: $\qquad\qquad$ **else if** ϕ_k **then** $\pi(\phi, n_k; p', h)$
22: **end if**;
23: $\quad\triangleright$ The algorithm is continued in Alg. 3, where the agent choices are described.

choice with the exploration strategy *explore*: with probability α, the agent selects randomly, and with probability $1{-}\alpha$, the agent selects according to the current policy $\pi(\sigma)$. Upon the execution of the selected action through the procedure $Update$, the v-function $\langle v, pr\rangle(\sigma)$ is updated. In the case of an agent (resp., opponent) choice (see lines 24–30 (resp., 31–38)), the current policy $\pi(\sigma)$ selects the current maximal (resp., minimal) choice; in the case of joint choices (see lines 39–48), following [13], an optimal current mixed policy is given by the Nash pair computed by a Nash selection function $selectNash$ from the matrix game defined by the possible joint choices. Then, depending on the case, the v-function is updated accordingly. Lines 49–55 encode the agent choice among programs. Finally, lines 55–60 define the successful test execution.

4.4 Example

We now illustrate the learning algorithm in the Stratagus Domain.

Example 4.1 (Stratagus Domain cont'd). Let the AGTGolog program p and the horizon h be given by $p = PickProc(x); carryToBase$ and $h = 3$, respectively. The learning

Algorithm 3 $Update(\phi, p, h)$: Part 2

if $p = \textbf{choice}(\boldsymbol{a} : a_1 | \cdots | a_n); p'$ **then**
  select any $q \in \{1, \ldots, n\}$ with strategy *explore*;
  $Update(\phi, \boldsymbol{a}{:}a_q; p', h)$;
  $k := \text{argmax}_{i \in \{1,\ldots,n\}} utility(\langle v, pr \rangle(\phi, \boldsymbol{a}{:}a_i; p', h))$;
  $\langle v, pr \rangle(\sigma) := \langle v, pr \rangle(\phi, \boldsymbol{a}{:}a_k; p', h)$;
  $\pi(\sigma) := \pi(\phi, \boldsymbol{a}{:}a_k; p', h)$
end if;
if $p = \textbf{choice}(\boldsymbol{o} : o_1 | \cdots | o_m); p'$ **then**
  select any $q \in \{1, \ldots, m\}$ with strategy *explore*;
  $Update(\phi, \boldsymbol{o}{:}o_q; p', h)$;
  $k := \text{argmin}_{i \in \{1,\ldots,m\}} utility(\langle v, pr \rangle(\phi, \boldsymbol{o}{:}o_i; p', h))$;
  $\langle v, pr \rangle(\sigma) := \langle v, pr \rangle(\phi, \boldsymbol{o}{:}o_k; p', h)$;
  $\pi(\sigma) :=$ **if** ϕ_1 **then** $\pi(\phi, \boldsymbol{o}{:}o_1; p', h)$ ...
    else if ϕ_m **then** $\pi(\phi, \boldsymbol{o}{:}o_m; p', h)$
end if;
if $p = \textbf{choice}(\boldsymbol{a} : a_1 | \cdots | a_n) \parallel \textbf{choice}(\boldsymbol{o} : o_1 | \cdots | o_m); p'$ **then**
  select any $r \in \{1, \ldots, n\}$ and $s \in \{1, \ldots, m\}$ with strategy *explore*;
  $Update(\phi, \boldsymbol{a}{:}a_r \| \boldsymbol{o}{:}o_s; p', h)$;
  $(\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}}) := selectNash(\{r_{i,j} = utility(\langle v, pr \rangle(\phi, \boldsymbol{a}{:}a_i \| \boldsymbol{o}{:}o_j; p', h)) \mid i, j\}$;
  $\langle v, pr \rangle(\sigma) := \sum_{i=1}^{n} \sum_{j=1}^{m} \pi_{\boldsymbol{a}}(a_i) \cdot \pi_{\boldsymbol{o}}(o_j) \cdot \langle v, pr \rangle(\phi, \boldsymbol{a}{:}a_i \| \boldsymbol{o}{:}o_j; p', h)$;
  $\pi(\sigma) := \pi_{\boldsymbol{a}} \| \pi_{\boldsymbol{o}}$; **if** $\phi_1 \wedge \psi_1$ **then** $\pi(\phi, \boldsymbol{a}{:}a_1 \| \boldsymbol{o}{:}o_1; p', h) \ldots$
    else if $\phi_n \wedge \psi_m$ **then** $\pi(\phi, \boldsymbol{a}{:}a_n \| \boldsymbol{o}{:}o_m; p', h)$
end if;
if $p = (p_1 \mid p_2); p'$ **then**
  select any $i \in \{1, 2\}$ with strategy *explore*;
  $Update(\phi, p_i; p', h)$;
  $k := \text{argmax}_{i \in \{1,2\}} utility(\langle v, pr \rangle(\phi, p_i; p', h))$;
  $\langle v, pr \rangle(\sigma) := \langle v, pr \rangle(\phi, p_k; p', h)$;
  $\pi(\sigma) := \pi(\phi, p_k; p', h)$
end if;
if $p = \psi?; p' \wedge \psi[\phi]$ **then**
  $Update(\phi, p', h)$;
  $\langle v, pr \rangle(\sigma) := \langle v, pr \rangle(\phi, p', h)$;
  $\pi(\sigma) := \pi(\phi, p', h)$
end if.

algorithm for this input (that is, $Learn(p,3)$) then works as follows. The agent runs several times p with horizon 3, playing against the opponent, until the learning ends and the variables $\langle v, pr\rangle$ are stabilized for each joint state (ϕ, p, h) associated with $(p,3)$ obtaining the relative policies $\pi(\phi, p, h)$.

The state partition of $(p,3)$ is given by $SF(p,3) = SF(p_1,3) \otimes \{asl\} \cup SF(p'_1,3) \otimes \{\neg asl\}$, where $p_1 = tryToPickUp(x)$; $carryToBase$ and $p'_1 = pickUpS(\boldsymbol{a}, x)$; $carry-ToBase$. In the machine state $(p_1,3)$, we have the joint choices $c^h_{\boldsymbol{a}} \| c^k_{\boldsymbol{o}}$, where $c^{pk}_{\alpha} = pickUpS(\alpha, x)$ and $c^{pk}_{\alpha} = moveS(\alpha, stand)$, and the agent is to learn the probability distributions of the relative mixed policies. The choices $c^h_{\boldsymbol{a}} \| c^k_{\boldsymbol{o}}$ are associated with the programs $p_{h,k} = c^h_{\boldsymbol{a}} \| c^k_{\boldsymbol{o}}; p_2$, where $p_2 = carryToBase$. In the machine state $(p_2,2)$, we have another choice point over the possible moves $m(q) = moveS(\boldsymbol{a}, q)$ towards the base. Each choice $m(q)$ is associated with the program $p_q = m(q); p_3$, where $p_3 =$ **if** atb **then** $dropS(\boldsymbol{a}, x)$ **else** p_2 (atb abbreviates $atBase$). Here, the partition is $SF(p_q,2) = \{atb^q \wedge hw,\ atb^q \wedge hg,\ \neg atb^q,\ \neg h\}$, where atb^q represents atb after $m(q)$ obtained form $Regr(ab, m(q))$. If $\neg atb$ is represented by atb^0, and $atb^{q_1} \wedge \ldots \wedge atb^{q_4}$ is represented by $atb^{q_1,\ldots,q_4}$, then $SF(p_2,2) = \otimes_q SF(p_q,2) = \{\neg h, atb^{0\ldots0}\} \cup \{atb^{k_1\ldots k_4} \wedge hg \mid \exists i : k_i \neq 0\} \cup \{atb^{h_1\ldots h_4} \wedge hw \mid \exists i : k_i \neq 0\}$. For each state formula $\phi \in SF(p_2,2)$, the algorithm $Learn(p,3)$ continuously refines, through the $Update$ step, the probability distribution over the policies $\pi(\phi, p_2, 2)$. For example, training the agent against a random opponent, in the state $\phi = atb^{S,0,0,0} \wedge hg$, the algorithm produces a policy $\pi(\phi, p_2, 2)$ assigning probability 1 for the component $q = S$, and probability 0 for $q \neq S$. Analogously, in the choice point $(p_1,3)$, $Learn(p,3)$ defines a mixed policy $\pi(\phi, p_1, 3)$ for each state $\phi \in SF(p_1,3) = \otimes_{h,k}\ SF(p_{h,k},3)$. For example, given $\phi_1 \in SF(p_1,3)$ equal to $\neg asl \wedge ag \wedge atb^{k_1,\ldots,k_4} \wedge \neg h_{\boldsymbol{a}} \wedge h_{\boldsymbol{o}}$, we get probability 1 for the choice $c_{pk,s}$ in $\pi(\phi_1, p_1, 3)$, instead, for $\phi_2 \in SF(p_1,3)$ equal to $asl \wedge ag \wedge atb^{k_1,\ldots,k_4} \wedge \neg h_{\boldsymbol{a}} \wedge \neg h_{\boldsymbol{o}}$, we get probability 0 for $c_{pk,s}$, $c_{s,pk}$, $c_{pk,pk}$, and probability 1 for $c_{s,s}$.

5 Summary and Outlook

We have presented a framework for adaptive multi-agent programming, which integrates high-level programming in GTGolog with adaptive dynamic programming. It allows the agent to on-line instantiate a partially specified behavior playing against an adversary. Differently from the classical Golog approach, here the interpreter generates not only complex sequences of actions (the policy), but also the state abstraction induced by the program at the different executive stages (machine states). In this way, we show how the Golog integration between action theory and programs allows to naturally combine the advantages of symbolic techniques [2,11] with the strength of hierarchical reinforcement learning [17,5,1,14]. This work aims at bridging the gap between programmable learning and logic-based programming approaches. To our knowledge, this is the first work exploring this very promising direction.

An interesting topic of future research is to explore whether the presented approach can be extended to the partially observable case.

Acknowledgments. This work was supported by the Austrian Science Fund Project P18146-N04 and by a Heisenberg Professorship of the German Research Foundation (DFG). We thank the reviewers for their comments, which helped to improve this work.

References

1. D. Andre and S. J. Russell. State abstraction for programmable reinforcement learning agents. In *Proceedings AAAI-2002*, pp. 119–125.
2. C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings IJCAI-2001*, pp. 690–700.
3. C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings AAAI-2000*, pp. 355–362.
4. P. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Proc. NIPS-1993*, pp. 271–278.
5. T. G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings ML-1998*, pp. 118–126.
6. A. Ferrein, C. Fritz, and G. Lakemeyer. Using Golog for deliberation and team coordination in robotic soccer. *Künstliche Intelligenz*, 1:24–43, 2005.
7. A. Finzi and T. Lukasiewicz. Game-theoretic agent programming in Golog. In *Proceedings ECAI-2004*, pp. 23–27.
8. A. Finzi and T. Lukasiewicz. Relational Markov games. In *Proceedings JELIA-2004*, Vol. 3229 of *LNCS/LNAI*, pp. 320–333.
9. A. Finzi and T. Lukasiewicz. Game-theoretic Golog under partial observability. In *Proceedings AAMAS-2005*, pp. 1301–1302.
10. A. Finzi and F. Pirri. Combining probabilities, failures and safety in robot control. In *Proceedings IJCAI-2001*, pp. 1331–1336.
11. C. Gretton and S. Thiebaux. Exploiting first-order regression in inductive policy selection. In *Proceedings UAI-2004*, pp. 217–225.
12. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Logic Program.*, 31(1–3):59–84, 1997.
13. M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings ICML-1994*, pp. 157–163.
14. B. Marthi, S. J. Russell, D. Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In *Proceedings IJCAI-2005*, pp. 779–785.
15. J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In *Machine Intelligence*, Vol. 4, pp. 463–502. Edinburgh University Press, 1969.
16. G. Owen. *Game Theory: Second Edition*. Academic Press, 1982.
17. R. Parr and S. J. Russell. Reinforcement learning with hierarchies of machines. In *Proceedings NIPS-1997*, Vol. 10, pp. 1043–1049.
18. J. Pinto. Integrating discrete and continuous change in a logical framework. *Computational Intelligence*, 14(1):39–88, 1998.
19. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.
20. R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.
21. Michael Thielscher. Programming of reasoning and planning agents with FLUX. In *Proceedings KR-2002*, pp. 435–446.
22. J. van der Wal. *Stochastic Dynamic Programming*, Vol. 139 of *Mathematical Centre Tracts*. Morgan Kaufmann, 1981.
23. J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, 1947.
24. C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.