I N F S Y S

R E S E A R C H

R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ARBEITSBEREICH WISSENSBASIERTE SYSTEME

ADAPTIVE GAME-THEORETIC AGENT PROGRAMMING IN GOLOG

ALBERTO FINZI THOMAS LUKASIEWICZ

INFSYS RESEARCH REPORT 1843-08-07

AUGUST 2008

Institut für Informationssysteme
AB 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

TU WIEN

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 1843-08-07, AUGUST 2008

ADAPTIVE GAME-THEORETIC AGENT PROGRAMMING IN GOLOG

AUGUST 29, 2008

Alberto Finzi [1] Thomas Lukasiewicz [2]

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 immediate rewards 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 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 present an algorithm for learning policies and show that it converges and produces optimal policies. This multi-agent learning algorithm includes as a special case a single-agent learning algorithm for DTGolog. 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]Institut für Informationssysteme, TU Wien, Favoritenstraße 9-11, 1040 Vienna, Austria. Dipartimento di Scienze Fisiche, Università di Napoli Federico II, Via Cinthia, 80126 Naples, Italy; e-mail: finzi@na.infn.it.

[2]Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, UK; e-mail: thomas.lukasiewicz@comlab.ox.ac.uk. Institut für Informationssysteme, TU Wien, Favoritenstraße 9-11, 1040 Vienna, Austria; e-mail: lukasiewicz@kr.tuwien.ac.at.

Acknowledgements: This work has been partially supported by the Austrian Science Fund (FWF) under the Project P18146-N04 and by the German Research Foundation (DFG) under the Heisenberg Programme.

Contents

1 Introduction

During the recent decade, the development of controllers for autonomous agents in real-world environments has become increasingly important in AI. In particular, there has been a significant research progress in the field of mobile robotics on the aspects of flexibility, autonomy, and human interaction. Several very successful real-world projects have shown in particular the feasibility of autonomous vehicles [7], office and museum tour-guide robots [4, 30], as well as robotic assistants for elderly people [26]. Furthermore, robotic vacuum cleaners (see especially http://www.irobot.com) and entertainment robots [40] in the form of toys and pets are already available on the market as successful consumer products. Rodney A. Brooks [3] summarizes "The weight of progress in so many forms of robots for unstructured environments leads to the conclusion that robots will be common in people's lives by the middle of the century if not significantly earlier".

One of the most crucial problems that we have to face in the development of controllers for autonomous agents in real-world environments is uncertainty, both about the initial situation of the agent's world and about the results of the actions taken by the agent. 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.

Towards combining the advantages of both ways of designing controllers, seminal works by Boutilier, Reiter, Soutchanski, and Thrun [36] and Soutchanski [37] present a generalization of Golog [22, 34], called *DTGolog*, where agent programming in Golog relative to stochastic action theories in the situation calculus [34] is combined with decision-theoretic planning in Markov decision process (MDPs) [33]. 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 (that is, the program may contain points with multiple possible actions, which are then replaced by a single optimal one). 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, it can also be seen as a formalism that gives advice to a decision-theoretic planner, since it naturally constrains the search space.

A limitation of DTGolog, however, is that it 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 summarized in "nature". But there are many applications where we 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 (adversary and friend) agents. That is, the agents can reason about and adapt to each other, but "nature" cannot do so. In particular, there is a bidirected dependence between the actions of two different agents, which generally makes it inappropriate to model adversaries and friends of the agent that we control simply as a part of "nature".

In [10, 11], we overcome this limitation of DTGolog by presenting the multi-agent programming language GTGolog, which integrates explicit agent programming in Golog with game-theoretic multi-agent planning in stochastic games [29]. GTGolog 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.

The main idea behind GTGolog can be roughly described as follows for the case of two competing agents. Suppose we want to control an agent and that, to this end, we write or we are already given a DTGolog program that specifies the agent's behavior in a partial way. If the agent acts alone in an environment,

then the DTGolog interpreter from [36] replaces all action choices of our agent in the DTGolog program by some actions that are guaranteed to be optimal. However, if our agent acts in an environment with an adversary, then the actions produced by the DTGolog interpreter are in general no longer optimal, since the optimal actions of our agent generally depend on the actions of its adversary, and conversely the actions of the adversary also generally depend on the actions of our agent. Hence, we have to enrich the DTGolog program for our agent by all the possible action moves of its adversary. Every such enriched DTGolog program is a GTGolog program. How do we then define the notion of optimality for the possible actions of our agent? We do this by defining the notion of a Nash equilibrium for GTGolog programs (and thus also for the above DTGolog programs enriched by the actions of the adversary). Every Nash equilibrium consists of a Nash policy for our agent and a Nash policy for its adversary. Since we assume that the rewards of our agent and of its adversary are zero-sum, we then obtain the important result that our agent always behaves optimally when following such a Nash policy, and this even when the adversary follows a Nash policy of another Nash equilibrium or no Nash policy at all (in the latter case, our agent can do no worse than its Nash policy guarantees).

Example 1 (Logistics Domain) Consider an agent $\boldsymbol{a}$ operating in a logistics domain with the goal of bringing to its base as many objects as it can while competing with an adversary $\boldsymbol{o}$ with the same objective. At each step, the two agents may either remain stationary, or move towards one location (for example, p_1, p_2, or p_3), or pick up or drop one object. Assume the two agents $\boldsymbol{a}$ and $\boldsymbol{o}$ can reach the position p_1 and compete for picking up the object obj in that position, or try to go somewhere else, for example, other two reachable positions p_2 or p_3. The possible choices of the agent $\boldsymbol{a}$ can then, for example, be specified by the following DTGolog procedure:

proc $getObject$
$move(p_1) \,|\, move(p_2) \,|\, move(p_3)$;
$pickUp \,|\, move(p_1) \,|\, move(p_2) \,|\, move(p_3)$
end.

Without adversary, the DTGolog interpreter determines one optimal action for each of the two action choices. In the presence of an adversary, however, the actions filled in by the DTGolog interpreter are in general no longer optimal. The GTGolog interpreter can be used for filling in optimal actions in DTGolog programs for agents with adversaries: We first enrich the DTGolog program by all the possible actions of the adversary. As a result, we obtain a GTGolog program, which looks as follows for the above procedure:

proc $getObject$
choice($\boldsymbol{a}$: $move(p_1) \,|\, move(p_2) \,|\, move(p_3)$ ||
  choice($\boldsymbol{o}$: $pickUp \,|\, move(p_1) \,|\, move(p_2) \,|\, move(p_3) \,|\, drop$) ;
choice($\boldsymbol{a}$: $pickUp \,|\, move(p_1) \,|\, move(p_2) \,|\, move(p_3)$ ||
  choice($\boldsymbol{o}$: $pickUp \,|\, move(p_1) \,|\, move(p_2) \,|\, move(p_3) \,|\, drop$)
end.

The GTGolog interpreter then specifies a Nash equilibrium for such programs. Each Nash equilibrium consists of a Nash policy for the agent $\boldsymbol{a}$ and a Nash policy for its adversary $\boldsymbol{o}$. The former specifies an optimal way of filling in missing actions in the original DTGolog program.

In addition to being a language for programming agents in multi-agent systems, GTGolog can also be considered as a new language for relational specifications of games: The background theory defines the

basic structure of a game, and any action choice contained in a GTGolog program defines the points where the agents can make one move each. In this case, rather than looking from the perspective of one agent that we program, we adopt an objective view on all the agents (as usual in game theory).

Example 2 (Logistics Domain cont'd) The following GTGolog program encodes the complete moves of the logistics domain.

proc $game$
while $\neg gameOver$ **do**
choice($\boldsymbol{a}$: $pickUp \mid move(p_1) \mid move(p_2) \mid move(p_3) \mid drop$) ||
  choice($\boldsymbol{o}$: $pickUp \mid move(p_1) \mid move(p_2) \mid move(p_3) \mid drop$)
end.

Informally, while the game is not over (that is, there is at least one object to be brought to the base), the two agents concurrently choose and execute one of the possible actions.

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

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 [23]. 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 execution 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, 19], 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 [5, 31, 6, 1, 24] to a logic-based agent [22, 34, 38] and multi-agent [8] 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.

The main contributions of this paper can be summarized as follows.

- We present the adaptive multi-agent programming language AGTGolog, which integrates the agent programming language GTGolog with adaptive dynamic programming techniques. We define the syntax of AGTGolog programs and their underlying first-order domain theories in the situation calculus. To our knowledge, this is the first work where high-level agent programming relative to logic-based action theories is combined with adaptive dynamic programming techniques.

- We then define a state partition for each machine state consisting of an AGTGolog program and a finite horizon. Furthermore, we present an algorithm for learning optimal policies in AGTGolog programs,

which uses these state partitions. This learning algorithm for optimal policies in AGTGolog programs includes as a special case a learning algorithm for optimal policies in DTGolog programs.

- We show that the policy and the expected utility computed by the learning algorithm converge with probability 1 against the policy and the utility, respectively, computed by the GTGolog interpreter (for fixed and explicitly given immediate rewards and transition probabilities). This also implies the optimality of the learned policies.

The rest of this paper is organized as follows. In Section 2, we recall the basic concepts of the situation calculus, concurrent actions, regression, Golog, DTGolog, matrix games, stochastic games, and Q-learning. Section 3 defines the domain theory and the syntax of AGTGolog programs. In Sections 4 and 5, we describe the generation of state partitions and the algorithm for learning optimal policies in AGTGolog programs relative to finite horizons, respectively. Section 6 illustrates the learning algorithm along an example, and Section 7 provides convergence and optimality results for the learning algorithm. Section 8 summarizes the main results and gives an outlook on future research. Note that detailed proofs of all results in the body of this paper are given in Appendix A.

2 Preliminaries

In this section, we first recall the basic concepts of the situation calculus, concurrent actions in the situation calculus, regression in the situation calculus, and the agent programming languages Golog and DTGolog. We then recall the basics of matrix games, stochastic games, and reinforcement learning.

2.1 The Situation Calculus

The situation calculus [25, 34] is a second-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. 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. 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*[1] represents a world or agent property that may change when executing an action. It is a predicate symbol whose rightmost argument is a situation. A situation calculus formula is *uniform* in a situation s iff (i) it does not mention the predicates $Poss$ and $\sqsubset$ (which denotes the proper subsequence relationship on situations), (ii) it does not quantify over situation variables, (iii) it does not mention equality on situations, and (iv) every situation in the situation argument of a fluent coincides with s (cf. [34]).

Example 3 The action $moveTo(r, x, y)$ may stand for moving the agent r to the position (x, y), while 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. The (relational) fluent $at(r, x, y, s)$ may express that the agent r is at the position (x, y) in the situation s.

[1] Note that we here do not consider fluents in the form of functions, called *functional fluents* [34].

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 [34].
- $\mathcal{D}_{una}$ is the set of unique names axioms for actions, which encode that different actions are interpreted differently. That is, actions with different names have a different meaning, and actions with the same name but different arguments have a different meaning.
- $\mathcal{D}_{S_0}$ is a set of first-order formulas that are uniform in S_0, which describe the initial state of the domain (represented by S_0).
- $\mathcal{D}_{ssa}$ is the set of *successor state axioms* [34]. For each fluent $F(\vec{x}, s)$, it contains an axiom of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula that is uniform in s with free variables among $\vec{x}, a, s$. These axioms specify the truth of the fluent F in the successor situation $do(a, s)$ in terms of the current situation s, and are a solution to the frame problem (for deterministic actions). More concretely, every successor state axiom is generally of the form

  $$F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)),$$

  where $\gamma_F^+(\vec{x}, a, s)$ (resp., $\gamma_F^-(\vec{x}, a, s)$) is a formula describing all the conditions under which performing the action a in the situation s results in the fluent F becoming true (resp., false) in the successor situation $do(a, s)$.
- $\mathcal{D}_{ap}$ is the set of *action precondition axioms*. For each action a, it contains an axiom of the form $Poss(a(\vec{x}), s) \equiv \Pi(\vec{x}, s)$, where Π is a formula that is uniform in s with free variables among $\vec{x}, s$. This axiom characterizes the preconditions of the action a.

Example 4 The formula $at(r, 1, 2, S_0) \wedge at(r', 3, 4, S_0)$ in $\mathcal{D}_{S_0}$ may express that the agents r and r' are initially at the positions $(1, 2)$ and $(3, 4)$, respectively.

The successor state axiom

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

in $\mathcal{D}_{ssa}$ 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. This successor state axiom can be constructed from the formulas $\gamma_{at}^+(\vec{x}, a, s)$ and $\gamma_{at}^-(\vec{x}, a, s)$, which describe all the conditions under which performing a in s lead at to become true resp. false in $do(a, s)$, and which are given by $a = moveTo(r, x, y)$ resp. $\exists x', y' ((x' \neq x \vee y' \neq y) \wedge a = moveTo(r, x', y'))$.

The action precondition axiom $Poss(moveTo(r, x, y), s) \equiv \neg\exists r'\, at(r', x, y, s)$ in $\mathcal{D}_{ap}$ may express that it is possible to move the agent r to the position (x, y) in the situation s iff no agent r' is at (x, y) in s (note that this also includes that the agent r is not at (x, y) in s).

2.2 Concurrent Actions in the Situation Calculus

We use a concurrent version of the situation calculus, which is an extension of the above standard situation calculus by concurrent actions [34, 32]. A *concurrent action* c is a set of standard actions, which are concurrently executed when c is executed. A situation is then a sequence of concurrent actions $do(c_m, \ldots, do(c_0,$

$S_0)$), which encodes the execution of the sequence of concurrent actions $c_0, \ldots, c_m$ in the situation S_0, where every execution of c_i means that all the simple actions a in c_i are executed at the same time. To encode concurrent actions, some slight modifications to standard basic action theories are necessary.

In particular, the successor state axioms in $\mathcal{D}_{ssa}$ are now defined relative to concurrent actions. Every successor state axiom is now generally of the form

$$F(\vec{x}, do(c, s)) \equiv \gamma_F^+(\vec{x}, c, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, c, s)) \,,$$

where $\gamma_F^+(\vec{x}, c, s)$ (resp., $\gamma_F^-(\vec{x}, c, s)$) is a formula describing all the conditions under which performing the concurrent action c in the situation s results in the fluent F becoming true (resp., false) in the successor situation $do(c, s)$. Such formulas $\gamma_F^+(\vec{x}, c, s)$ and $\gamma_F^-(\vec{x}, c, s)$ can be constructed by unifying the positive and negative effects of all the standard actions $a \in c$ and removing eventual collisions of positive and negative effects.

Example 5 The successor state axiom

$$\begin{aligned} &at(r, x, y, do(a, s)) \;\equiv\; a = moveTo(r, x, y) \vee \\ &\quad (at(r, x, y, s) \wedge \neg\exists x', y' \, ((x' \neq x \vee y' \neq y) \wedge a = moveTo(r, x', y'))) \end{aligned}$$

in $\mathcal{D}_{ssa}$ in the standard situation calculus is replaced by the successor state axiom

$$\begin{aligned} &at(r, x, y, do(c, s)) \;\equiv\; moveTo(r, x, y) \in c \vee \\ &\quad (at(r, x, y, s) \wedge \neg\exists x', y' \, ((x' \neq x \vee y' \neq y) \wedge moveTo(r, x', y') \in c)) \end{aligned}$$

in $\mathcal{D}_{ssa}$ in the situation calculus with concurrent actions.

Moreover, the action preconditions in $\mathcal{D}_{ap}$ are extended by further axioms expressing (i) that a singleton concurrent action $c = \{a\}$ is executable if its standard action a is executable, (ii) that if a concurrent action is executable, then it is nonempty and all its standard actions are executable, and (iii) preconditions for concurrent actions. Note that precondition axioms for standard actions are in general not sufficient, since two standard actions may each be executable, but their concurrent execution may not be permitted. This *precondition interaction problem* [34] (see also [32]) requires some domain-dependent extra precondition axioms.

2.3 Regression in the Situation Calculus

We next recall the important concept of regression of formulas through actions [34]. Intuitively, the regression of a formula ϕ through an action a, denoted $Regr(\phi)$, is a formula ϕ' that holds before executing the action a, given that ϕ holds after executing a. More formally, the *regression* of ϕ whose situations are all of the form $do(a, s)$ is defined inductively using the successor state axioms $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$ as follows:

$$\begin{aligned} &Regr(F(\vec{x}, do(a, s))) = \Phi_F(\vec{x}, a, s) \,, \\ &Regr(\neg\phi) = \neg Regr(\phi) \,, \\ &Regr(\phi_1 \wedge \phi_2) = Regr(\phi_1) \wedge Regr(\phi_2) \,, \\ &Regr(\exists x \, \phi) = \exists x \, (Regr(\phi)) \,. \end{aligned}$$

Example 6 The regression of the formula $at(r, x, y, do(a, s))$ through the action $a = move\text{-}\ To(r, x, y)$ is simply the true formula $\top$. Intuitively, $at(r, x, y, do(a, s))$ always holds after executing $a = moveTo(r, x, y)$ in s, independently of what holds in s.

2.4 Golog

Golog [22, 34] is an agent programming language that is based on the situation calculus. It allows for constructing complex actions (also called *programs*) from (standard or concurrent) primitive actions that are defined in a basic action theory AT, where standard (and not-so-standard) Algol-like control constructs can be used. More precisely, *programs* p in Golog have one of the following forms (where c is a (standard or concurrent) primitive action, ϕ is a *condition* (which is obtained from a situation calculus formula that is uniform in s by suppressing the situation argument), $p, p_1, p_2, \ldots, p_n$ are programs, $P_1, \ldots, P_n$ are procedure names, and $x, \vec{x}_1, \ldots, \vec{x}_n$ are arguments):

1. *Primitive action*: c. Do c.
2. *Test action*: $\phi?$. Test the truth of ϕ in the current situation.
3. *Sequence*: $[p_1; p_2]$. Do p_1 followed by p_2.
4. *Nondeterministic choice of two programs*: $(p_1 \mid p_2)$. Do either p_1 or p_2.
5. *Nondeterministic choice of program argument*: $\pi x\,(p(x))$. Do any $p(x)$.
6. *Nondeterministic iteration*: $p^\star$. Do p zero or more times.
7. *Conditional*: **if** ϕ **then** p_1 **else** p_2. If ϕ is true in the current situation, then do p_1 else p_2.
8. *While-loop*: **while** ϕ **do** p. While ϕ is true in the current situation, do p.
9. *Procedures*: **proc** $P_1(\vec{x}_1)\ p_1$ **end** ; . . . ; **proc** $P_n(\vec{x}_n)\ p_n$ **end** ; p.

Example 7 The small Golog program

while $\neg at(r, 1, 2)$ **do** $\pi x, y\,(moveTo(r, x, y))$

may stand for "while the agent r is not at the position $(1, 2)$, move r to a nondeterministically chosen position (x, y)".

Golog has a declarative formal semantics, which is defined in the situation calculus. Given a Golog program p, its execution is represented by a situation calculus formula $Do(p, s, s')$, which encodes that the situation s' can be reached by executing the program p in the situation s. That is, Do represents a macro expansion to a situation calculus formula. For example, the semantics of the sequence is defined through

$$Do([p_1; p_2], s, s') \stackrel{def}{=} \exists s''(Do(p_1, s, s'') \wedge Do(p_2, s'', s'))\,.$$

For more details on (the core situation calculus and) Golog, we refer especially to [34].

2.5 Decision-Theoretic Golog (DTGolog)

Decision-theoretic Golog (DTGolog) [36, 37] is an extension of Golog that combines high-level agent programming in Golog with decision-theoretic planning in Markov decision process (MDPs) [33]. DTGolog programs have (nearly) the same syntax as standard Golog programs. They are specified relative to a background action theory and a background optimization theory in the situation calculus. The background action

theory defines deterministic and stochastic actions. The deterministic actions are defined through the axioms of a basic action theory in the situation calculus, while the stochastic actions are defined via their deterministic action components along with probabilities. The background optimization theory contains in particular axioms specifying a reward function.

Example 8 The small DTGolog program for a mail delivery robot from [36]

$$\begin{array}{l}\textbf{while } \exists p\,(\neg attempted(p) \wedge \exists n\, mailPresent(p,n)) \\ \quad \textbf{do } \pi[p\colon people]((\neg attempted(p) \wedge \exists n\, mailPresent(p,n))?\,;\, deliverTo(p))\end{array}$$

intuitively chooses people from a finite collection *people* for mail delivery.

The DTGolog interpreter translates a DTGolog program p into an optimal policy, which is obtained from p by replacing (i) every nondeterministic choice of two programs, (ii) every nondeterministic choice of a program argument, and (iii) every nondeterministic iteration by (i) an optimal choice among the two programs, (ii) an optimal choice of a program argument among the possible ones, and (iii) an optimal number of iterations, respectively, through maximizing expected utility. This semantics of a DTGolog program p is specified via a situation calculus formula $BestDo(p, s, h, \pi, v, pr)$, which encodes that, given a program p, starting from the situation s, and assuming a horizon h, the optimal policy obtained from p is given by π and has the expected value v and the success probability pr. For example, for the case of nondeterministic choice of two programs, $BestDo$ is defined as follows:

$$\begin{array}{l}BestDo([(p_1|p_2);p], s, h, \pi, v, pr) \stackrel{def}{=} \\ \quad \exists \pi_1, v_1, pr_1\,(BestDo([p_1;p], s, h, \pi_1, v_1, pr_1) \wedge \\ \quad \exists \pi_2, v_2, pr_2\,(BestDo([p_2;p], s, h, \pi_2, v_2, pr_2) \wedge \\ \quad ((\langle v_1, pr_1\rangle \succ \langle v_2, pr_2\rangle \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1) \vee \\ \quad (\langle v_1, pr_1\rangle \preccurlyeq \langle v_2, pr_2\rangle \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2)))\,.\end{array}$$

Informally, the optimal policy for $[(p_1|p_2);p]$ along with its expected value and success probability is given by the optimal policy among $[p_1;p]$ and $[p_2;p]$ along with its expected value and success probability. Here, the binary predicate $\succ$ and its negation $\preccurlyeq$ compare expected utilities, which are pairs $\langle v, pr\rangle$ consisting of an expected value v and a success probability $pr \in [0,1]$. More concretely, the preference order $\succ$ is defined by $\langle v_1, pr_1\rangle \succ \langle v_2, pr_2\rangle$ iff either (a) $pr_1 > 0$ and $pr_2 = 0$, or (b) $v_1 > v_2$ and $pr_1 = pr_2 = 0$, or (c) $v_1 > v_2$ and $pr_1, pr_2 > 0$. Observe that $\succ$ actually only distinguishes between zero and positive success probabilities; it thus treats success probabilities only as *success flags*. For more details on DTGolog and its semantics, we refer the reader especially to [36, 37].

Example 9 Consider again the DTGolog program for a mail delivery robot in Example 8. The order in which the robot delivers mail to people is such that it maximizes expected utility. So, for a person to be served first, mail delivery to that person must be associated with a high utility in the optimization theory behind the DTGolog program.

2.6 Matrix Games

Matrix games from classical game theory [41, 27, 28] 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,

		Player O	
		One Finger	Two Fingers
Player E	One Finger	$(2, -2)$	$(-3, 3)$
	Two Fingers	$(-3, 3)$	$(4, -4)$

Figure 1: Reward functions (R_E, R_O) for two-finger Morra.

and two *reward functions* $R_{\boldsymbol{a}}, R_{\boldsymbol{o}}\colon A \times O \to \mathbf{R}$ for $\boldsymbol{a}$ and $\boldsymbol{o}$. It is *zero-sum* iff $R_{\boldsymbol{a}} = -R_{\boldsymbol{o}}$; we then often omit $R_{\boldsymbol{o}}$ and abbreviate $R_{\boldsymbol{a}}$ by R.

The behavior of the agents in a matrix game is expressed through the notions of pure and mixed strategies. Pure strategies specify a single action that an agent should execute, while mixed strategies specify a set of possible actions to be executed, where every action in this set is associated with the probability with which it should be executed. Formally, 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. Let $PD(A)$ (resp., $PD(O)$) denote the set of all probability functions over A (resp., O), that is, the set of all mappings μ from A (resp., O) to $[0, 1]$ such that $\sum_{a \in A} \mu(a) = 1$ (resp., $\sum_{o \in O} \mu(o) = 1$). If the agents $\boldsymbol{a}$ and $\boldsymbol{o}$ play the mixed strategies $\mu_{\boldsymbol{a}} \in PD(A)$ and $\mu_{\boldsymbol{o}} \in PD(O)$, respectively, then the *expected reward* to agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ is $R_k(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}}) = \mathbf{E}[R_k(a, o) | \mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}}] = \sum_{a \in A,\, o \in O} \mu_{\boldsymbol{a}}(a) \cdot \mu_{\boldsymbol{o}}(o) \cdot R_k(a, o)$.

Towards optimal behavior of the agents in a matrix game, we are especially interested in pairs of mixed strategies $(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$, 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: $(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$ is a *Nash equilibrium* (or *Nash pair*) for G iff (i) $R_{\boldsymbol{a}}(\mu'_{\boldsymbol{a}}, \mu_{\boldsymbol{o}}) \leqslant R_{\boldsymbol{a}}(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$ for any mixed $\mu'_{\boldsymbol{a}}$, and (ii) $R_{\boldsymbol{o}}(\mu_{\boldsymbol{a}}, \mu'_{\boldsymbol{o}}) \leqslant R_{\boldsymbol{o}}(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$ for any mixed $\mu'_{\boldsymbol{o}}$. 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. The Nash pairs 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 each matrix game G a unique Nash equilibrium $f(G) = (f_{\boldsymbol{a}}(G), f_{\boldsymbol{o}}(G))$; the reward to $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ under $f(G)$ is then denoted by $v_f^k(G)$.

In the zero-sum case, if $(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$ and $(\mu'_{\boldsymbol{a}}, \mu'_{\boldsymbol{o}})$ are two Nash equilibria of G, then $R_{\boldsymbol{a}}(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}}) = R_{\boldsymbol{a}}(\mu'_{\boldsymbol{a}}, \mu'_{\boldsymbol{o}})$, and also $(\mu_{\boldsymbol{a}}, \mu'_{\boldsymbol{o}})$ and $(\mu'_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$ are Nash equilibria of G. That is, the expected reward to the agents is the same under any Nash equilibrium, and Nash equilibria can be freely "mixed" to form new Nash equilibria. The strategies of agent $\boldsymbol{a}$ in Nash equilibria are the optimal solutions of the following linear program:

$$
\begin{aligned}
&\max v \text{ subject to} \\
&\textstyle\sum_{a \in A} \mu(a) \cdot R_{\boldsymbol{a}}(a, o) \geqslant v \text{ for all } o \in O, \\
&\textstyle\sum_{a \in A} \mu(a) = 1, \text{ and} \\
&\mu(a) \geqslant 0 \text{ for all } a \in A.
\end{aligned}
$$

Furthermore, the expected reward to agent $\boldsymbol{a}$ under a Nash equilibrium is the optimal value of the above linear program.

Example 10 (Two-Finger Morra) In the zero-sum matrix game *two-finger Morra* [35], two players E and O simultaneously show one or two fingers. Let f be the total number of fingers shown. If f is odd, then O gets f dollars from E, and if f is even, then E gets f dollars from O (see Fig. 1). A pure strategy

for player E (or O) is to show two fingers, while a mixed strategy for player E (or O) is to show one finger with the probability $7/12$ and two fingers with the probability $5/12$. The mixed strategy profile where each player shows one finger with the probability $7/12$ and two fingers with the probability $5/12$ is a Nash pair.

2.7 Stochastic Games

Stochastic games [29], or also called Markov games [39, 23], generalize both matrix games [41] and (fully observable) Markov decision processes (MDPs) [33].

A (two-player) stochastic game consists of a set of states S, a matrix game for every state $s \in S$ (with common sets of actions for each agent), and a transition function that associates with every state $s \in S$ and joint action 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 $P\colon S \times A \times O \to PD(S)$ (which associates with every triple $(s, a, o) \in S \times A \times O$ a probability function $P(s, a, o)(\,\cdot\,)$ over S, abbreviated as $P(\,\cdot\,|s, a, o)$ or $P(\,\cdot\,|s, (a, o))$), and two *reward functions* $R_{\boldsymbol{a}}, R_{\boldsymbol{o}}\colon S \times A \times O \to \mathbf{R}$ for $\boldsymbol{a}$ and $\boldsymbol{o}$. It is *zero-sum* iff $R_{\boldsymbol{a}} = -R_{\boldsymbol{o}}$; we then often omit $R_{\boldsymbol{o}}$ and abbreviate $R_{\boldsymbol{a}}$ by R.

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. A *pure (resp., mixed) policy profile* $\pi = (\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ consists of a pure (resp., mixed) policy π_k for each agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$. The *$H$-step value* to agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ under a start state $s \in S$ and the pure policy profile $\pi = (\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$, denoted $G_k(H, s, \pi)$, is defined by:

$$G_k(H, s, \pi) = \begin{cases} R_k(s, \pi(s, H)) & \text{if } H = 0; \\ R_k(s, \pi(s, H)) + \sum_{s' \in S} P(s'|s, \pi(s, H)) \cdot G_k(H-1, s', \pi) & \text{if } H > 0. \end{cases}$$

The *expected H-step value* to agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ under a start state s and the mixed policy profile $\pi = (\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$, denoted $G_k(H, s, \pi)$, is defined by:

$$G_k(H, s, \pi) = \begin{cases} \mathbf{E}[R_k(s, a) \,|\, \pi(s, H)] & \text{if } H = 0; \\ \mathbf{E}[R_k(s, a) + \sum_{s' \in S} P(s'|s, a) \cdot G_k(H-1, s', \pi) \,|\, \pi(s, H)] & \text{if } H > 0. \end{cases}$$

Finite-horizon Nash equilibria for stochastic games are then defined as follows. A mixed policy profile $\pi = (\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ is a *(H-step) Nash equilibrium* (or *(H-step) Nash pair*) of G iff for every agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ and every start state $s \in S$, it holds that $G_k(H, s, \pi \lhd \pi'_k) \leqslant G_k(H, s, \pi)$ for every mixed policy π'_k, where $\pi \lhd \pi'_k$ is obtained from π by replacing π_k by π'_k. Every stochastic game G has at least one Nash pair among its mixed (but not necessarily pure) policy profiles, and it may have exponentially many Nash pairs.

Nash pairs for G can be computed by finite-horizon value iteration from local Nash pairs of matrix games as follows [20]. We assume an arbitrary Nash selection function f for matrix games $G = (A, O, R_{\boldsymbol{a}}, R_{\boldsymbol{o}})$. For every state $s \in S$ and every number of steps to go $h \in \{0, \ldots, H\}$, the matrix game $G[s, h] = (A, O, Q_{\boldsymbol{a}}[s, h], Q_{\boldsymbol{o}}[s, h])$ is defined by:

$$Q_k[s, h](a) = \begin{cases} R_k(s, a) & \text{if } h = 0; \\ R_k(s, a) + \sum_{s' \in S} P(s'|s, a) \cdot v^i_f(G[s', h-1]) & \text{if } h > 0, \end{cases}$$

for every joint action $a \in A = \bigtimes_{i \in I} A_i$ and every agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$. For every agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$, let the mixed policy π_k be defined by $\pi_k(s, h) = f_k(G[s, h])$ for every $s \in S$ and $h \in \{0, \ldots, H\}$. Then, $\pi = (\pi_{\boldsymbol{a}}, \pi_{\boldsymbol{o}})$ is

a H-step Nash pair of G, and it holds $G_k(H, s, \pi) = v_f^k(G[s, H])$ for every agent $k \in \{\boldsymbol{a}, \boldsymbol{o}\}$ and every state $s \in S$.

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

2.8 Learning Optimal Policies

Q-learning [42, 43] is a reinforcement learning technique, which allows to solve an MDP without a model (that is, transition and reward functions) and can be used online. 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's experience. For each execution of an action a leading from the state s to the (observed) state s', the agent receives a reward r (from the environment), 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 $\gamma \in (0, 1)$ (resp., $\alpha \in [0, 1)$) is the discount factor (resp., 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 that α is decayed appropriately. Convergence also holds in the non-discounting case ($\gamma = 1$) in the presence of reward-free absorbing states, which are eventually reached from any non-absorbing state with positive probability.

Littman [23] extends Q-learning to learning an optimal mixed policy in a zero-sum two-player stochastic game. Here, the Q-value update is

$$Q(s, a, o) := (1 - \alpha) \cdot Q(s, a, o) + \alpha \cdot (r + \gamma \cdot \max_{\mu \in PD(A)} \min_{o' \in O} \sum_{a' \in A} Q(s', a', o') \cdot \mu(a')),$$

where the "maxmin"-term is 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 the agent programming language 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-agent action* is any concurrent action c over $A \cup O$ such that $|c \cap A| \leqslant 1$ and $|c \cap O| \leqslant 1$. We often write a, o, and $a \| o$ to abbreviate $\{a\} \subseteq A$, $\{o\} \subseteq O$, and $\{a, o\} \subseteq A \cup O$, respectively.

Example 11 The concurrent actions $\{moveTo(\boldsymbol{a}, 1, 2)\} \subseteq A$ and $\{moveTo(\boldsymbol{o}, 2, 3)\} \subseteq O$ are single-agent actions of $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively, and thus also two-agent actions, while the concurrent action $\{moveTo(\boldsymbol{a}, 1, 2), moveTo(\boldsymbol{o}, 2, 3)\} \subseteq A \cup O$ is only a two-agent action, but not a single-agent action of $\boldsymbol{a}$ or $\boldsymbol{o}$.

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

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

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

We next define the stochastic theory. As usual [36, 18, 2], every stochastic action a is expressed by a finite number of deterministic actions $n_1, \ldots, n_k$. When the stochastic action a is executed in the situation s, then "nature" chooses and executes with a certain probability exactly one of its deterministic actions n_i with $i \in \{1, \ldots, k\}$.

Example 12 The stochastic action $moveS(ag, x, y)$ of moving the agent ag to the position (x, y) in the situation s may have the effect that ag moves to (x, y) with the probability p and that ag moves to $(x, y+1)$ with the probability $1 - p$, which is realized by making "nature" choose and execute exactly one of the two deterministic actions $moveTo(ag, x, y)$ and $moveTo(ag, x, y+1)$ with the probabilities p and $1 - p$, respectively.

We use the predicate $stochastic$ to connect the stochastic action a in the situation s to the deterministic actions $n_1, \ldots, n_k$, that is, $stochastic(a, s, n_i)$ is true for all $i \in \{1, \ldots, k\}$. We also specify a state partition $P_{pr}^{a,n}(\vec{x}, s) = \{\phi_j^{a,n}(\vec{x}, s) \mid j \in \{1, \ldots, m\}\}$ to group together situations s with common p such that "nature" chooses n in s with probability p, denoted $prob(a(\vec{x}), n(\vec{x}), s) = p$:[2]

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

A stochastic action s is indirectly represented by providing a *successor state axiom* for each associated nature choice n; see also Example 14. 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. This condition is represented by the predicate $condStAct(c, s, n)$, where c is a stochastic action, s is a situation, n is a deterministic component of c, and $condStAct(c, s, n)$ is true iff executing c in s has resulted in actually executing n. Similar axioms are introduced to observe which actions the two agents have chosen.

Example 13 After executing $c = moveS(ag, x, y)$ in the situation s, we test the predicates $at(ag, x, y, do(c, s))$ and $at(ag, x, y+1, do(c, s))$ to check which of the two deterministic actions (that is, either $moveTo(ag, x, y)$ or $moveTo(ag, x, y+1)$) was actually executed. The predicate $condStAct(c, s, n)$ for the stochastic action $c = moveS(ag, x, y)$ is defined by:

$$condStAct(c, s, \{moveTo(ag, x, y)\}) \stackrel{def}{=} at(ag, x, y, do(c, s))\,,$$
$$condStAct(c, s, \{moveTo(ag, x, y{+}1)\}) \stackrel{def}{=} at(ag, x, y{+}1, do(c, s))\,.$$

[2]Note that we specify only partitions of state formulas that group together situations with common transition probabilities, but not the transition probabilities themselves.

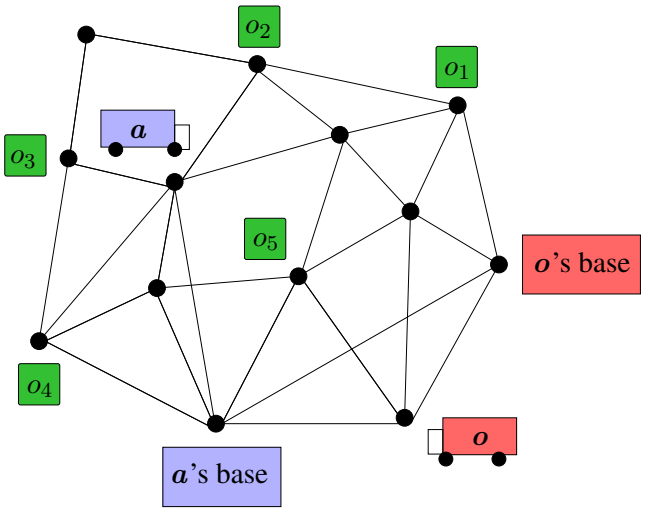

Figure 2: Logistics Domain.

As for the optimization theory, for every two-agent action α, we specify a state partition $P^{\alpha}_{rw}(\vec{x}, s) = \{\phi^{\alpha}_{k}(\vec{x}, s) \mid k \in \{1, \ldots, q\}\}$ to group together situations s with common r such that $\alpha(\vec{x})$ and s assign the reward r to $\boldsymbol{a}$, denoted $reward(\alpha(\vec{x}), s) = r$:[3]

$$\exists r_1, \ldots, r_q \, (\textstyle\bigwedge_{k=1}^{q} (\phi^{\alpha}_{k}(\vec{x}, s) \Leftrightarrow reward(\alpha(\vec{x}), s) = r_k)) \, .$$

Example 14 (Logistics Domain cont'd) Consider the following scenario in a multi-agent logistics domain. We have an undirected graph of n locations connected by m edges (see Fig. 2). There are two agents, denoted $\boldsymbol{a}$ and $\boldsymbol{o}$, which occupy one location each and move along the edges. The two agents can move one step to one of the directly connected locations, or remain stationary. Each agent can also pick up one object, and drop it 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, $\boldsymbol{a}$ and $\boldsymbol{o}$ receive (from the environment) the (zero-sum) rewards $r_{\boldsymbol{a}} - r_{\boldsymbol{o}}$ and $r_{\boldsymbol{o}} - r_{\boldsymbol{a}}$, respectively, where r_{ag}, $ag \in \{\boldsymbol{a}, \boldsymbol{o}\}$, is -1, 2, and 10 when ag moves to a new location or remains stationary, ag picks up a new object, and ag brings an object to its base, respectively.

The domain theory $DT = (AT, ST, OT)$ for the above logistics domain is defined as follows. As for the basic action theory AT, we assume the deterministic actions $move(ag, x)$ (the agent ag moves to the location x), $pickUp(ag, obj)$ (the agent ag picks up the object obj), $drop(ag, obj)$ (the agent ag drops the object obj), and nop (no action), as well as the relational fluents $at(q, x, s)$ (the agent or object q is at the location x in the situation s), $onFloor(obj, x, s)$ (the object obj is on the floor at the location x in the situation s), and $holds(ag, obj, s)$ (the agent ag holds the object obj in the situation s), which are defined

[3]Note that we specify only partitions of state formulas that group together situations with common rewards, but not the rewards themselves.

through the following successor state axioms:

$$
\begin{aligned}
&at(ag, x, do(c, s)) \equiv at(ag, x, s) \wedge \neg\exists y\,(move(ag, y) \in c) \vee move(ag, x) \in c\,;\\
&onFloor(obj, x, do(c, s)) \equiv (onFloor(obj, x, s) \wedge \neg\exists ag\,(pickUp(ag, obj) \in c) \vee\\
&\quad \exists ag\,((drop(ag, obj) \in c \vee collision(c, s)) \wedge at(ag, x, s) \wedge holds(ag, obj, s)))\,;\\
&holds(ag, obj, do(c, s)) \equiv holds(ag, obj, s) \wedge drop(ag, obj) \notin c \wedge \neg collision(c, s) \vee\\
&\quad pickUp(ag, obj) \in c\,.
\end{aligned}
$$

Here, $collision(c, s)$ encodes that the concurrent action c causes a collision between the two agents $\boldsymbol{a}$ and $\boldsymbol{o}$ in the situation s:

$$collision(c, s) \stackrel{def}{=} \exists x\,(move(\boldsymbol{a}, x) \in c \wedge move(\boldsymbol{o}, x) \in c)\,.$$

The deterministic actions $move(ag, x)$, $pickUp(ag, obj)$, $drop(ag, obj)$, and $nop(ag)$ are associated with precondition axioms as follows:

$$
\begin{aligned}
&Poss(move(ag, x), s) \equiv \exists y\,(at(ag, y, s) \wedge connects(y, x))\,;\\
&Poss(pickUp(ag, obj), s) \equiv \exists y\,(at(ag, y, s) \wedge onFloor(obj, y, s)) \wedge \neg\exists x\; holds(ag, x, s)\,;\\
&Poss(drop(ag, obj), s) \equiv holds(ag, obj, s)\,;\\
&Poss(nop(ag), s) \equiv \top\,.
\end{aligned}
$$

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

$$
\begin{aligned}
&Poss(\{pickUp(ag, obj), pickUp(ag', obj')\}, s) \equiv\\
&\quad \exists x, x'\,(at(ag, x, s) \wedge at(ag', x', s) \wedge x \neq x' \wedge obj \neq obj')\,.
\end{aligned}
$$

As for the stochastic theory ST, we assume the stochastic actions $moveS(ag, m)$ (the agent ag moves to one of the possible locations m), $pickUpS(ag, obj)$ (the agent ag picks up the object obj), $dropS(ag, obj)$ (the agent ag drops the object obj), which may succeed or fail. We assume the state partition $P_{pr}^{a,n} = \{\top\}$ for each pair consisting of a stochastic action a and one of its deterministic components n:

$$
\begin{aligned}
&\exists p\,(p \geqslant 0 \wedge prob(moveS(ag, d), move(ag, d), s) = p \wedge\\
&\quad prob(moveS(ag, d), nop(ag), s) = 1 - p)\,;\\
&\exists p\,(p \geqslant 0 \wedge prob(pickUpS(ag, obj), pickUp(ag, obj), s) = p \wedge\\
&\quad prob(pickUpS(ag, obj), nop(ag), s) = 1 - p)\,;\\
&\exists p\,(p \geqslant 0 \wedge prob(dropS(ag, obj), drop(ag, obj), s) = p \wedge\\
&\quad prob(dropS(ag, obj), nop(ag), s) = 1 - p)\,;\\
&\exists p\,(prob(a \| o, a' \| o', s) = p \equiv\\
&\quad \exists p_1, p_2\,(prob(a, a', s) = p_1 \wedge prob(o, o', s) = p_2 \wedge p = p_1 \cdot p_2))\,.
\end{aligned}
$$

As for the optimization theory OT, we define the reward function $reward$ as follows:

$$
\begin{aligned}
&reward(a, s) = r \equiv\\
&\quad \exists r_{\boldsymbol{a}}, r_{\boldsymbol{o}}\,(rewAg(\boldsymbol{a}, a, s) = r_{\boldsymbol{a}} \wedge rewAg(\boldsymbol{o}, a, s) = r_{\boldsymbol{o}} \wedge r = r_{\boldsymbol{a}} - r_{\boldsymbol{o}})\,;\\
&\exists r_1, \ldots, r_m\,(\textstyle\bigwedge_{j=1}^{m}(\phi_j^{ag,a}(s) \Leftrightarrow rewAg(ag, a, s) = r_j))\,.
\end{aligned}
$$

Here, the state partitions $P_{rw}^{ag,a}(s) = \{\phi_j^{ag,a}(s) \mid j \in \{1, \ldots, m\}\}$ for the agent ag and the different actions $a \in \{moveS(ag, x), pickUpS(ag, obj), dropS(ag, obj)\}$ are as follows:

- If $a = moveS(ag, x)$, then $P_{rw}^{ag,a}(s) = \{\top\}$. Informally, for each context, a moving action does not affect the immediate reward. Hence, we do not distinguish between different states, and we get the trivial partition defined by the true formula.
- If $a = pickUpS(ag, obj)$, then $P_{rw}^{ag,a}(s) = \{\neg h \wedge ato, \neg h \wedge \neg ato, h\}$, where $h = \exists x(holds(ag, x, s))$ (the agent ag holds an object) and $ato = \exists y, obj\ (at(ag, y, s) \wedge on\text{-}\ Floor(obj, y, s))$ (the agent ag is close to an object). Informally, the immediate reward depends on whether
  - ag holds no object but is close to an object ($\neg h \wedge ato$),
  - ag holds no object and is also far from an object ($\neg h \wedge \neg ato$), or
  - ag holds an object (h).
- If $a = dropS(ag, obj)$, then $P_{rw}^{ag,a}(s) = \{h \wedge atb, h \wedge \neg atb, \neg h\}$, where h is as above and $atb = atBase(ag, s) = \exists y\,(at(ag, y, s) \wedge base(ag, y))$ (the agent ag is at its base). Informally, the immediate reward depends on whether
  - ag holds an object and is at its base ($h \wedge atb$),
  - ag holds an object and is not at its base ($h \wedge \neg atb$), or
  - ag holds no object ($\neg h$).

3.2 Syntax of AGTGolog

In the sequel, let DT be a domain theory. AGTGolog has the same syntax as standard GTGolog, which is defined by induction as follows. A *program* p in AGTGolog has one of the following forms (where α is a two-agent action or the empty action nop (which is always executable and does not change the state of the world), ϕ is a condition, p, p_1, p_2 are programs without procedure declarations, $P_1, \ldots, P_n$ are procedure names, $x, \vec{x}_1, \ldots, \vec{x}_n$ are arguments, and $a_1, \ldots, a_n$ and $o_1, \ldots, o_m$ are actions of agents $\boldsymbol{a}$ and $\boldsymbol{o}$, respectively, and $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is a finite nonempty set of ground terms):

1. *Deterministic or stochastic action:* α.
   Do α.
2. *Nondeterministic action choice of agent* $\boldsymbol{a}$*:* **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 agent* $\boldsymbol{o}$*:* **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) \,\|\, $ **choice**$(\boldsymbol{o} : o_1 | \cdots | o_m)$.
   Do every action $a_i \| o_j$ with an optimal probability $\mu_{\boldsymbol{a}}(a_i) \cdot \mu_{\boldsymbol{o}}(o_j)$.
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 an optimal program among p_1 and p_2.

8. *Nondeterministic choice of program argument:* $\pi[x:\tau](p(x))$.
   Do an optimal program $p(x)$ with $x \in \tau$.

9. *Nondeterministic iteration*: $p^\star$.
   Do p optimally zero or more times.

10. *Conditional*: **if** ϕ **then** p_1 **else** p_2.
    If ϕ is true in the current situation, then do p_1 else do p_2.

11. *While-loop*: **while** ϕ **do** p.
    While ϕ is true in the current situation, do p.

12. *Procedures*: **proc** $P_1(\vec{x}_1)\ p_1$ **end**; ...; **proc** $P_n(\vec{x}_n)\ p_n$ **end**; p.
    The procedures named $P_1, \ldots, P_n$, respectively, are declared and can be used in p.

Hence, compared to Golog, we now also have two-agent actions (instead of only primitive or concurrent actions) and stochastic actions (instead of only deterministic actions). Moreover, we now additionally have three different kinds of nondeterministic action choices for the two agents in (2)–(4), where one or both of the two agents can choose among a finite set of single-agent actions. The two-agent actions and the choice operators in (3) and (4) are also new compared to DTGolog. The formal semantics of (2)–(4) is defined in such a way that an optimal action is chosen for each of the two agents. Similarly, an optimal program is chosen in (7)–(9). If there are several optimal action or program choices, then we additionally assume a total order on these choices and take the preferred choice according to this order (for example, choose the leftmost optimal action in (2)–(4) and the leftmost optimal program in (7)). As usual, the sequence operator ";" is associative (for example, $[[p_1; p_2]; p_3]$ and $[p_1; [p_2; p_3]]$ mean the same), and "$p_1; p_2$", "**if** ϕ **then** p_1", and "$\pi x\,(p(x))$" often abbreviate "$[p_1; p_2]$", "**if** ϕ **then** p_1 **else** nop", and $\pi[x:\tau](p(x))$, respectively, when there is no danger of confusion.

Example 15 (Logistics Domain cont'd) We define some AGTGolog programs relative to the domain theory $DT = (AT, ST, OT)$ of Example 14. The main task of the agent $\boldsymbol{a}$ (and the opponent $\boldsymbol{o}$) is to take one object and drop it at the base. This can be described by the following AGTGolog procedure:

proc $task(\boldsymbol{a}, \boldsymbol{o})$
$getObject(\boldsymbol{a}, \boldsymbol{o})\,;\ carryToBase(\boldsymbol{a}, \boldsymbol{o})$
end.

Here, $getObject(\boldsymbol{a}, \boldsymbol{o})$ is an AGTGolog procedure for searching and picking up an object, while $carryToBase(\boldsymbol{a}, \boldsymbol{o})$ describes a partially specified behavior where the agent $\boldsymbol{a}$ (competing with $\boldsymbol{o}$) is trying to move to its base in order to drop an object:

proc $getObject(\boldsymbol{a}, \boldsymbol{o})$
$\pi[y{:}\ loc](\pi[x{:}\ obj](moveS(\boldsymbol{a}, y)\,;\ ?(onFloor(x, y))\,;\ pickProc(\boldsymbol{a}, \boldsymbol{o}, x)))$
end.

proc $carryToBase(\boldsymbol{a}, \boldsymbol{o})$
$\pi[y{:}\ loc](moveS(\boldsymbol{a}, y)\,;$ **if** $atBase(\boldsymbol{a})$ **then** $\pi[x{:}\ obj](dropS(\boldsymbol{a}, x))$
else $carryToBase(\boldsymbol{a}, \boldsymbol{o}))$
end.

Here, loc (resp., obj) is the set of possible locations (resp., objects). The subsequent procedure $pickProc(\boldsymbol{a}, \boldsymbol{o}, x)$ 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 the agent $\boldsymbol{a}$ can directly use the primitive action $pickUpS(\boldsymbol{a}, x)$:

proc $pickProc(\boldsymbol{a}, \boldsymbol{o}, x)$
if $atSameLocation(\boldsymbol{a}, \boldsymbol{o})$ **then** $tryToPickUp(\boldsymbol{a}, \boldsymbol{o}, x)$
  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 $tryToPickUp(\boldsymbol{a}, \boldsymbol{o}, x)$ (which will be instantiated by a mixed policy):

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

Note that in the case of a concurrent attempt of picking up the same object, by the concurrent precondition axiom, at least one of the two agents is forced to fail.

4 State Partition Generation

In this section, we define joint state partitions for AGTGolog programs relative to finite horizons. Informally, during the learning of action rewards and transition probabilities for an AGTGolog program p, we randomly choose an initial state of the environment and start the execution of the program p, during which we receive action rewards and make nature choose the outcome of probabilistic transitions. Hence, the initially chosen state of the environment should fix in advance appropriate elements of the state partitions for the action rewards and the transition probabilities. Furthermore, whenever the program p arrives at a point with several action or program alternatives, the initially chosen state of the environment should allow for choosing any of these alternatives. That is, we have to construct a joint state partition from all state partitions of action rewards and transition probabilities, considering also all action and program alternatives, in the program p within a finite horizon.

More formally, a *machine state* (p, h) consists of an AGTGolog program p and a horizon $h \geqslant 0$. A machine state (p, h) *precedes* another machine state (p', h'), denoted $(p, h) \rhd (p', h')$, iff p' is a possible remaining of p after running p for $h - h'$ steps. A *joint state* (ϕ, p, h) consists of a state formula ϕ and a machine state (p, h). Informally, a machine state represents the executive state of the agent, while a joint state represents both the state of the environment and the executive state of the agent. Every machine state (p, h) is now associated with a state partition, denoted $SF(p, h) = \{\phi_1(\vec{x}, s), \ldots, \phi_m(\vec{x}, s)\}$, which is defined by induction on the structure of AGTGolog programs as follows:

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 $\{\top\}$. Indeed, when the program or the horizon ends, then no more choices are needed, and therefore we have a unique context representing the end of the program.

2. *Deterministic first program action:*

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

Here, the state partition for $a; p'$ with horizon h is obtained as the product of the reward partition $P^a_{rw}(\vec{x}, s)$, 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):*

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

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^{a,n_i}_{pr}.

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

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

where $a_1, \ldots, a_n$ are two-agent actions. That is, the state partition for a single choice of actions is the product of the state partitions for the possible choices.

5. *Nondeterministic joint action choice:*

$$SF(\mathbf{choice}(\boldsymbol{a}\colon a_1 | \cdots | a_n) \,\|\, \mathbf{choice}(\boldsymbol{o}\colon o_1 | \cdots | o_m); p', h) \;=\; \bigotimes_{i=1,j=1}^{n,m} SF(a_i \| o_j; p', h)\,,$$

where $a_1, \ldots, a_n$ and $o_1, \ldots, o_m$ are the possible choices for $\boldsymbol{a}$ and $\boldsymbol{o}$ respectively. The associated state partition is obtained from the partitions generated by the possible concurrent executions $a_i \| o_j$.

6. *Test action:*

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

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).

7. *Nondeterministic choice of two programs:*

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

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

8. *Nondeterministic choice of program argument:*

$$SF(\pi[x : \{\tau_1, \ldots, \tau_n\}](p(x)); p', h) \;=\; \{\phi_1(\tau_1) \wedge \cdots \wedge \phi_n(\tau_n) \mid \phi_1(x), \ldots, \phi_n(x) \;\in SF(p(x : \{\tau_1, \ldots, \tau_n\}); p', h)\} \setminus \{\bot\}\,.$$

The state partition for a nondeterministic choice of program argument is obtained as the product of the state partitions associated with the possible program instantiations. Here, $SF(p(x : \{\tau_1, \ldots, \tau_n\}); p', h)$ denotes a state partition that is parameterized via the variable x, which may take on any value from $\{\tau_1, \ldots, \tau_n\}$.

9. *Nondeterministic iteration:*

$$SF(p^\star; p', h) \;=\; SF(p; p^\star; p', h) \otimes SF(p', h)\,.$$

This case is reduced to procedures and nondeterministic choice of two programs.

10. *Conditional:*

$$\begin{aligned} &SF(\textbf{if } \psi \textbf{ then } p_1 \textbf{ else } p_2; p', h) \\ &\qquad = \; (\{\psi\} \otimes SF(p_1; p', h)) \cup (\{\neg\psi\} \otimes SF(p_2; p', h))\,. \end{aligned}$$

The state partition for "**if** ψ **then** p_1 **else** $p_2; p', h$" is obtained by composing the partition $\{\psi, \neg\psi\}$ with the state partitions for "$p_1; p', h$" and "$p_2; p', h$".

11. *While-loop:*

$$\begin{aligned} &SF(\textbf{while } \psi \textbf{ do } p; p', h) \\ &\qquad = \; (\{\psi\} \otimes SF(p; \textbf{while } \psi \textbf{ do } p; p', h)) \cup (\{\neg\psi\} \otimes SF(p', h))\,. \end{aligned}$$

The state partition for "**while** ψ **do** $p; p', h$" is obtained by composing the partition $\{\psi, \neg\psi\}$ with the state partitions for "p; **while** ψ **do** $p; p', h$" and "p', h".

12. *Procedures:*

$$\begin{aligned} &SF(\textbf{proc } P_1(\vec{x}_1)\ p_1 \textbf{ end}\,; \ldots; \textbf{proc } P_n(\vec{x}_n)\ p_n \textbf{ end}\,; p, h) \\ &\qquad = \; SF(p\langle \textbf{proc } P_1(\vec{x}_1)\ p_1 \textbf{ end}\,; \ldots; \textbf{proc } P_n(\vec{x}_n)\ p_n \textbf{ end}\rangle; p', h)\,; \\ &SF(P_i(\vec{x}_i); p'\langle d\rangle, h) \;=\; SF(p_d(P_i(\vec{x}_i)); p', h)\,. \end{aligned}$$

We consider the cases of (1) handling procedure declarations and (2) handling procedure calls. To this end, we slightly extend the first argument of SF by a store for procedure declarations, which can be safely ignored in all the other above cases. Here, $p_d(P_i(\vec{x}_i))$ denotes the code from d for the procedure call $P_i(\vec{x}_i)$.

Example 16 (Logistics Domain cont'd) Some joint states are (ϕ_1, p, h) and (ϕ_2, p, h), where (p, h) is the machine state consisting of the AGTGolog program $move(\boldsymbol{a}, x) \parallel move(\boldsymbol{o}, y)\,;\, nil$ and the horizon $h = 1$, and ϕ_1 and ϕ_2 are given as follows:

$$\begin{aligned} \phi_1 \;=\;& \exists k\,(onFloor(k, x, s)) \wedge \exists l\,(onFloor(l, y, s)) \wedge \\ & \exists z\,(at(\boldsymbol{a}, z, s) \wedge connects(z, x)) \wedge \exists w\,(at(\boldsymbol{o}, w, s) \wedge connects(w, y))\,; \\ \phi_2 \;=\;& \neg\exists k\,(onFloor(k, x, s)) \wedge \exists l\,(onFloor(l, y, s)) \wedge \\ & \exists z\,(at(\boldsymbol{a}, z, s) \wedge connects(z, x)) \wedge \exists w\,(at(\boldsymbol{o}, w, s) \wedge connects(w, y))\,. \end{aligned}$$

As for the worst-case number of state formulas in the generated state partitions, it is not difficult to verify that the worst-case cardinality of $SF(p, h)$ is in $O((m\,n)^{a^h})$, where (i) n is the maximal number of state formulas in the input probability and the input reward partitions; (ii) a is the maximum among (ii.a) the maximal number of actions of an agent in nondeterministic (single or joint) action choices in p, (ii.b) the maximal number of choices of nature after stochastic actions in p, and (ii.c) the *nondeterministic branching*

factor of p, which is the maximal number of alternative program choices (via nondeterministic choices of two programs, nondeterministic choices of program arguments, and nondeterministic iterations) at any step within h; and (iii) m is the *conditional branching factor* of p, which is the maximal number of branchings via conditionals and while-loops of p at any step within h. Although this worst-case number seems to be quite large, observe that many of the generated state formulas will be the false formula, especially when the input state partitions are logically very similar. Furthermore, an upper bound for the number of state formulas in state partitions is also the size of the state space. Finally, in many applications in practice, one can assume that the horizon is very small and bounded by a constant, and that it is not necessary to explore the whole state space in the learning algorithm.

5 Learning Optimal Policies

We now show how to learn optimal policies for AGTGolog programs relative to finite horizons. Intuitively, given an AGTGolog program p and a horizon $h \geqslant 0$, an h-step policy π for p relative to a domain theory DT 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. Note that the learning algorithm also implicitly learns the transition probabilities and rewards. Note also that the convergence and optimality of the learning algorithm is proved in Section 7.

We first describe the overall learning algorithm. We then define selection functions and describe the updating step of the learning algorithm. We finally sketch how success probabilities / flags can be added and describe a (very) preliminary implementation.

5.1 Learning Algorithm

The main learning algorithm is $Learn$ in Algorithm 1. The algorithm takes as input an AGTGolog program p and a finite horizon $h \geqslant 0$ (note that h is the maximal number of steps to go, that is, the maximal number of actions to be executed). It generates as output an optimal h'-step policy $\pi(\sigma)$ along with its expected utility $v(\sigma)$, for each joint state $\sigma = (\phi, p', h')$ such that $\phi \in SF(p', h')$ and $(p, h) \rhd (p', h')$.

We use a hierarchical version of Q-learning. In line 1, we initialize the *learning rate* α to 1, which decays at each learning cycle according to $decay$. In lines 2 and 3, we also initialize to 0 the variables $v(\sigma)$ representing the current expected utilities. At each cycle, the current state $\phi \in SF(p, h)$ is evaluated (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 5.3) executes the program p with horizon h, and updates and refines the expected utilities $v(\sigma)$ and the policies $\pi(\sigma)$. 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 evaluated, 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 convergence. That is, the agent executes the program (p, h) several times refining its pairs of expected utilities $v(\sigma)$ and its policies $\pi(\sigma)$ until they do not change anymore.

5.2 Selection Functions

In the updating step, we use selection functions for minimal arguments in minimizations, maximal arguments in maximizations, and Nash equilibria of zero-sum matrix games of the form $G = (I, (A_i)_{i \in I}, R)$,

Algorithm 1 $Learn(p, h)$

Require: AGTGolog program p and a finite horizon $h \geqslant 0$.

Ensure: optimal h'-step policy $\pi(\sigma)$ and its expected utility $v(\sigma)$, for each $\sigma = (\phi, p', h')$ such that $\phi \in SF(p', h')$ and $(p, h) \rhd (p', h')$.

```
1: α := 1;
2: for each σ = (φ, p', h') such that φ ∈ SF(p', h') and (p, h) ▷ (p', h')
3:   do v(σ) := 0;
4: repeat
5:   estimate φ ∈ SF(p, h);
6:   Update(φ, p, h);
7:   α := α · decay
8: until α < ε;
9: return (v(φ, p', h'), π(φ, p', h'))_{φ∈SF(p',h'), (p,h)▷(p',h')}.
```

where $I = \{\boldsymbol{a}, \boldsymbol{o}\}$ and $A_{\boldsymbol{a}} = \{a_1, \ldots, a_m\}$ (resp., $A_{\boldsymbol{o}} = \{o_1, \ldots, o_n\}$) is a nonempty set of single-agent actions of agent $\boldsymbol{a}$ (resp., $\boldsymbol{o}$).

As for selecting minimal (resp., maximal) arguments in minimizations (resp., maximizations), we define $prefArgmin(r_1, \ldots, r_n)$ (resp., $prefArgmax(r_1, \ldots, r_n)$), where $n \geqslant 1$, as the unique index $l \in \{1, \ldots, n\}$ such that (i) r_l is a minimal (resp., maximal) element among $r_1, \ldots, r_n$ and (ii) the index l is minimal with (i). Intuitively, arguments with lower index are preferred to arguments with higher index. But we use slightly modified versions of these functions, which additionally take into account numerical imprecision due to convergence with probability 1: We define ε-$prefArgmin(r_1, \ldots, r_n)$ (resp., ε-$prefArgmax(r_1, \ldots, r_n)$), where $n \geqslant 1$ and $\varepsilon > 0$, as the unique index $l \in \{1, \ldots, n\}$ such that (i) r_l is within the ε-range of a minimal (resp., maximal) element among $r_1, \ldots, r_n$ and (ii) the index l is minimal with (i). These ε-variants of $prefArgmin$ and $prefArgmax$ then allow for pushing through the convergence with probability 1 of policies (see also Proposition 19).

As for selecting Nash pairs of zero-sum matrix games $G = (I, (A_i)_{i \in I}, R)$, where $I = \{\boldsymbol{a}, \boldsymbol{o}\}$ and $A_{\boldsymbol{a}} = \{a_1, \ldots, a_m\}$ (resp., $A_{\boldsymbol{o}} = \{o_1, \ldots, o_n\}$) as above, the Nash selection function $prefNash$ selects the Nash pair $\mu^{pref} = (\mu_{\boldsymbol{a}}^{pref}, \mu_{\boldsymbol{o}}^{pref})$ of G such that for every other Nash pair $\mu = (\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$ of G there exist some $k \in \{1, \ldots, m\}$ and $l \in \{1, \ldots, n\}$ such that:

- $\mu_{\boldsymbol{a}}^{pref}(a_i) = \mu_{\boldsymbol{a}}(a_i)$ for all $i \in \{1, \ldots, k-1\}$,
- $\mu_{\boldsymbol{a}}^{pref}(a_k) > \mu_{\boldsymbol{a}}(a_k)$,
- $\mu_{\boldsymbol{o}}^{pref}(o_j) = \mu_{\boldsymbol{o}}(o_j)$ for all $j \in \{1, \ldots, l-1\}$, and
- $\mu_{\boldsymbol{o}}^{pref}(o_l) > \mu_{\boldsymbol{o}}(o_l)$.

Intuitively, actions with lower index are given a higher probability than arguments with higher index. We compute $prefNash(G)$ by linear programming. More concretely, let v be the expected reward to agent $\boldsymbol{a}$ under a Nash pair of G. Then, $prefNash(G)$ is the pair of mixed strategies $\mu^{pref} = (\mu_{\boldsymbol{a}}^{pref}, \mu_{\boldsymbol{o}}^{pref}) \in PD(A_{\boldsymbol{a}}) \times PD(A_{\boldsymbol{o}})$ such that:

- $\mu_{\boldsymbol{a}}^{pref}(a_i)$, for every $i \in \{1, \ldots, m\}$, is the maximum of $\mu_{\boldsymbol{a}}(a_i)$ subject to

$\sum_{i=1}^{m} \mu_{\boldsymbol{a}}(a_i) \cdot R(a_i, o_j) \geqslant v$ for all $j \in \{1, \ldots, n\}$,
$\sum_{i=1}^{m} \mu_{\boldsymbol{a}}(a_i) = 1$,
$\mu_{\boldsymbol{a}}(a_i) \geqslant 0$ for all $i \in \{1, \ldots, m\}$, and
$\mu_{\boldsymbol{a}}(a_l) = \mu_{\boldsymbol{a}}^{pref}(a_l)$ for every $l \in \{1, \ldots, i-1\}$;

- $\mu_{\boldsymbol{o}}^{pref}(o_j)$, for every $j \in \{1, \ldots, n\}$, is the maximum of $\mu_{\boldsymbol{o}}(o_j)$ subject to

  $\sum_{j=1}^{n} \mu_{\boldsymbol{o}}(o_j) \cdot R(a_i, o_j) \leqslant v$ for all $i \in \{1, \ldots, m\}$,
  $\sum_{j=1}^{n} \mu_{\boldsymbol{o}}(o_j) = 1$,
  $\mu_{\boldsymbol{o}}(o_j) \geqslant 0$ for all $j \in \{1, \ldots, n\}$, and
  $\mu_{\boldsymbol{o}}(o_l) = \mu_{\boldsymbol{o}}^{pref}(o_l)$ for every $l \in \{1, \ldots, j-1\}$.

To additionally take into account numerical imprecision due to convergence with probability 1, as in the case of $prefArgmin$ and $prefArgmax$, we also use an ε-variant of $prefNash$, denoted ε-$prefNash$, where $\varepsilon > 0$, which is defined by ε-$prefNash(G) = prefNash(G')$, where the matrix game G' is obtained from G be replacing any collection of rewards that lie within an ε-range of each other by their additive average. This ε-variant of $prefNash$ allows for pushing through the convergence with probability 1 of policies (see also Proposition 20).

5.3 Updating Step

The procedure $Update(\phi, p, h)$ in Algorithms 2 and 3 implements the execution and update step of a Q-learning algorithm. Each joint state σ of the program p within the horizon h is associated with a variable $v(\sigma)$, which store the current expected utility at σ, and a variable $\pi(\sigma)$, which stores the current optimal policy at σ. The procedure $Update(\phi, p, h)$ updates these variables during an execution of the program p within the horizon h from a state $\phi \in SF(p, h)$. It is recursive, following the structure of the program.

Algorithm 2 shows the first part of the procedure $Update(\phi, p, h)$. Lines 1–4 encode the basis of the induction: if the program is empty or the horizon is zero, then we set the expected utility and the policy to 0 and nil, respectively. In lines 5–8, we consider the non-executable cases: if a primitive action a is not executable in the current state formula (here, $\neg Poss(a, \phi)$ abbreviates $DT \cup \phi \models \neg Poss(a, s)$) or a test fails in the current state formula (here, $\neg\psi[\phi]$ stands for $DT \cup \phi \models \neg\psi(s)$), then we set the expected utility and the policy to 0 and $stop$ (which stops the execution), respectively. In lines 9–14, we consider deterministic actions a (here, $Poss(a, \phi)$ is a shortcut for $DT \cup \phi \models Poss(a, s)$): after executing a, the agent receives the reward $reward$ from the environment. Then, after executing the rest of the program from the next state formula (that is, $do(a, \phi)$, which is the state formula $\phi' \in SF(p', h-1)$ such that $Regr(\phi'(do(a, s))) = \phi(s)$ relative to DT) via $Update(do(a, \phi), p', h-1)$, the variables $v(\sigma)$ and $\pi(\sigma)$ for $\sigma = (\phi,\ a; p',\ h)$ are updated by adding the received reward to the current expected utility of the next joint state and adding a to the current policy of the next joint state, respectively. In lines 15–21, we consider stochastic actions a: after executing the stochastic action a, we observe the deterministic component n_q chosen by "nature", and we update the expected utilities similarly as in Q-learning, where the current utility is obtained as the sum of the old utility with factor $1 - \alpha$ and of the updated utility (for the executed component) with factor α. The generated policy is a conditional plan where each possible execution is considered. Here, ϕ_i are the conditions to discriminate

Algorithm 2 $Update(\phi, p, h)$

Require: state formula ϕ, AGTGolog program p, and finite horizon $h \geqslant 0$.
Ensure: updates $v(\sigma)$ and $\pi(\sigma)$, where $\sigma = (\phi, p, h)$.

if $p = nil \vee h = 0$ **then**
  $v(\sigma) := 0$;
  $\pi(\sigma) := nil$
end if;
if $p = a; p' \wedge \neg Poss(a, \phi) \vee p = \psi?; p' \wedge \neg\psi[\phi]$ **then**
  $v(\sigma) := 0$;
  $\pi(\sigma) := stop$
end if;
if $p = a; p' \wedge Poss(a, \phi)$ and a is deterministic **then**
  execute a and observe $reward$;
  $Update(do(a, \phi), p', h-1)$;
  $v(\sigma) := v(do(a, \phi), p', h-1) + reward$;
  $\pi(\sigma) := a; \pi(do(a, \phi), p', h-1)$
end if;
if $p = a; p' \wedge Poss(a, \phi)$ and a is stochastic **then**
  "nature" selects any deterministic action n_q of the action a;
  $Update(\phi, n_q; p', h)$;
  $v(\sigma) := (1 - \alpha) \cdot v(\sigma) + \alpha \cdot v(\phi, n_q; p', h)$;
  $\pi(\sigma) := a;$ **if** ϕ_1 **then** $\pi(\phi, n_1; p', h)$ ...
    else if ϕ_k **then** $\pi(\phi, n_k; p', h)$
end if;
if $p =$ **choice**$(\boldsymbol{a}: a_1 | \cdots | a_n); p'$ **then**
  select any $q \in \{1, \ldots, n\}$ with strategy *explore* (see below);
  $Update(\phi, \boldsymbol{a}{:}a_q; p', h)$;
  $v(\sigma) := \max_{i \in \{1,\ldots,n\}} v(\phi, \boldsymbol{a}{:}a_i; p', h)$;
  $k := \varepsilon\text{-}prefArgmax_{i \in \{1,\ldots,n\}} v(\phi, \boldsymbol{a}{:}a_i; p', h)$;
  $\pi(\sigma) := \boldsymbol{a}{:}a_k;$ **if** ϕ_1 **then** $\pi(do(\boldsymbol{a}{:}a_1, \phi), p', h-1)$ ...
    else if ϕ_n **then** $\pi(do(\boldsymbol{a}{:}a_n, \phi), p', h-1)$
end if;
if $p =$ **choice**$(\boldsymbol{o}: o_1 | \cdots | o_m); p'$ **then**
  select any $q \in \{1, \ldots, m\}$ with strategy *explore* (see below);
  $Update(\phi, \boldsymbol{o}{:}o_q; p', h)$;
  $v(\sigma) := \min_{i \in \{1,\ldots,m\}} v(\phi, \boldsymbol{o}{:}o_i; p', h)$;
  $k := \varepsilon\text{-}prefArgmin_{i \in \{1,\ldots,m\}} v(\phi, \boldsymbol{o}{:}o_i; p', h)$;
  $\pi(\sigma) := \boldsymbol{o}{:}o_k;$ **if** ϕ_1 **then** $\pi(do(\boldsymbol{o}{:}o_1, \phi), p', h-1)$ ...
    else if ϕ_m **then** $\pi(do(\boldsymbol{o}{:}o_m, \phi), p', h-1)$
end if;
if $p =$ **choice**$(\boldsymbol{a}: a_1 | \cdots | a_n) \,\|\, $ **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* (see below);
  $Update(\phi, \boldsymbol{a}{:}a_r \| \boldsymbol{o}{:}o_s; p', h)$;
  $v(\sigma) :=$ expected reward under a Nash pair of $\{r_{i,j} = v(\phi, \boldsymbol{a}{:}a_i \| \boldsymbol{o}{:}o_j; p', h) \mid i, j\}$;
  $(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}}) := \varepsilon\text{-}prefNash(\{r_{i,j} = v(\phi, \boldsymbol{a}{:}a_i \| \boldsymbol{o}{:}o_j; p', h) \mid i, j\})$;
  $\pi(\sigma) := \mu_{\boldsymbol{a}} \| \mu_{\boldsymbol{o}};$ **if** $\phi_1 \wedge \psi_1$ **then** $\pi(do(\boldsymbol{a}{:}a_1 \| \boldsymbol{o}{:}o_1, \phi), p', h-1) \ldots$
    else if $\phi_n \wedge \psi_m$ **then** $\pi(do(\boldsymbol{a}{:}a_n \| \boldsymbol{o}{:}o_m, \phi), p', h-1)$
end if;
▷ $Update(\phi, p, h)$ is continued in Algorithm 3.

Algorithm 3 $Update(\phi, p, h)$ **(cont'd)**

if $p = \psi?; p' \wedge (\phi = \psi \wedge \phi')$ **then**
  $Update(\phi', p', h)$;
  $v(\sigma) := v(\phi', p', h)$;
  $\pi(\sigma) := \pi(\phi', p', h)$
end if;
if $p = (p_1 \mid p_2); p'$ **then**
  select any $i \in \{1, 2\}$ with strategy *explore* (see below);
  $Update(\phi, p_i; p', h)$;
  $v(\sigma) := \max_{i \in \{1,2\}} v(\phi, p_i; p', h)$;
  $k := \varepsilon\text{-}prefArgmax_{i \in \{1,2\}} v(\phi, p_i; p', h)$;
  $\pi(\sigma) := \pi(\phi, p_k; p', h)$
end if;
if $p = \pi[x : \{\tau_1, \ldots, \tau_n\}](p(x)); p'$ **then**
  select any $i \in \{1, \ldots, n\}$ with strategy *explore* (see below);
  $v(\sigma) := \max_{i \in \{1,\ldots,n\}} v(\phi, p(\tau_i); p', h)$;
  $k := \varepsilon\text{-}prefArgmax_{i \in \{1,\ldots,n\}} v(\phi, p(\tau_i); p', h)$;
  $\pi(\sigma) := \pi(\phi, p(\tau_k); p', h)$
end if;
if $p = p^\star; p'$ **then**
  select any $p_1 = p; p^\star; p'$ or $p_2 = p'$ with strategy *explore* (see below);
  $Update(\phi, p_i, h)$;
  $v(\sigma) := \max_{i \in \{1,2\}} v(\phi, p_i, h)$;
  $k := \varepsilon\text{-}prefArgmax_{i \in \{1,2\}} v(\phi, p_i, h)$;
  $\pi(\sigma) := \pi(\phi, p_k, h)$
end if;
if $p =$ "**if** ψ **then** p_1 **else** $p_2; p'$" $\wedge\, ((\phi = \psi \wedge \phi') \vee (\phi = \neg\psi \wedge \phi'))$ **then**
  if $\phi = \psi \wedge \phi'$ **then** $Update(\phi', p_1; p', h)$ **else** $Update(\phi', p_2; p', h)$
end if;
if $p =$ "**while** ψ **do** $p; p'$" $\wedge\, ((\phi = \psi \wedge \phi') \vee (\phi = \neg\psi \wedge \phi'))$ **then**
  if $\phi = \psi \wedge \phi'$ **then** $Update(\phi', p;$ **while** ψ **do** $p; p', h)$ **else** $Update(\phi', p', h)$
end if;
if $p =$ "**proc** $P_1(\vec{x}_1)\ p_1$ **end**; ...; **proc** $P_n(\vec{x}_n)\ p_n$ **end**; p" **then**
  $decl :=$ "**proc** $P_1(\vec{x}_1)\ p_1$ **end**; ...; **proc** $P_n(\vec{x}_n)\ p_n$ **end**";
  $Update(\phi, p, h)$
end if;
if $p = P_i(\vec{x}_i); p'$ **then**
  extract the code $p_{decl}(P_i(\vec{x}_i))$ for the procedure call $P_i(\vec{x}_i)$ from $decl$;
  $Update(\phi, p_{decl}(P_i(\vec{x}_i)); p', h)$
end if.

the executed component (represented by the observability axioms). In lines 22–37, the code deals with the agent (resp., opponent) choice construct, and encodes how the agent learns an optimal action from the possible ones. Here, we first select one possible action according to an exploration strategy $explore$: with some probability $p \in (0,1)$, the agent (resp., opponent) selects randomly, and with the probability $1-p$, the agent (resp., opponent) selects an action according to the current optimal policy $\pi(\sigma)$. That is, $explore$ controls how often the agent (resp., opponent) deviates from the current optimal policy, ensuring a suitable exploration of the state space. After executing the selected action via $Update$, the expected utility $v(\sigma)$ is updated with the utilities of a maximal (resp., minimal) choice, and $\pi(\sigma)$ is updated with a conditional plan starting with this maximal (resp., minimal) choice. In lines 38–45, we consider the joint action choice of both agents. We select one action from the possible ones for each agent, using the strategy $explore$, similarly as described above. After executing the procedure $Update$ along the selected joint action, $v(\sigma)$ is updated with the expected reward under the pair of mixed strategies $\mu_{\boldsymbol{a}} \| \mu_{\boldsymbol{o}}$ specified by ε-$prefNash$ from the matrix game of the possible joint actions (that is, the current expected utilities of the possible joint actions). Furthermore, $\pi(\sigma)$ is updated with a conditional plan where each possible execution is considered. Here, ϕ_i and ψ_j are the conditions to discriminate the executed component for the agent and the opponent, respectively.

Algorithm 3 shows the second part of the procedure $Update(\phi, p, h)$. Lines 47–51 define the successful test execution, that is, if $DT \cup \phi \models \psi(s)$, then after executing the rest of the program (p', h), we update the expected utility and the policy with the ones for the program (p', h) from ϕ'. Lines 52–58 encode the choice among two programs: one of the two programs is selected with exploration strategy *explore*, similarly as described above. After executing the selected program through $Update$, the variables $v(\sigma)$ and $\pi(\sigma)$ are updated considering the maximal among the utilities of the two programs. Finally, in lines 59–82, we essentially reduce all the remaining constructs to the previous ones.

5.4 Adding Success Probabilities / Flags

To extend the semantics and the learning algorithm of AGTGolog to also account for success probabilities / flags as in DTGolog, expected utilities $u = \langle v, pr \rangle$ consisting of an expected value v and a success probability $pr \in [0,1]$ are represented by the value $f(u) = v$ (resp., $f(u) = v + M$) iff $pr = 0$ (resp., $pr > 0$), where M is a sufficiently large number (which is greater than any possible expected value v). It is then not difficult to verify that $u_1 \prec u_2$ iff $f(u_1) < f(u_2)$ for all possible utilities u_1 and u_2. To correctly compute such expected utilities in AGTGolog, one then has to slightly adapt some computations of expected utilities in the learning algorithm (in a similar way as in the GTGolog interpreter; see [11]).

5.5 Implementation

We have a (very) preliminary implementation, where the state formula generator is implemented in Eclipse Prolog, and the learning algorithm is realized in Eclipse Prolog embedded in C++, using the *gltk* library for linear programming (for the Nash solver). The learning algorithm uses the Eclipse engine for state formula evaluation and utility/policy update.

6 Example

We now illustrate the overall system working in the Logistics Domain.

Machine States		State Partitions	
$(p, 3)$		$SF(p,3) = (\{asl\} \otimes SF(p_1,3)) \cup (\{\neg asl\} \otimes SF(p'_1,3))$	
$(p_1, 3)$	$(p'_1, 3)$	$SF(p_1,3) = \bigotimes_{i,j \in \{1,2\}} SF(p_{i,j},3)$	$SF(p'_1,3) = (P^a_{rw}(\vec{x}, s) \otimes \ldots$
$(p_{i,j}, 3)$		$SF(p_{i,j},3) = \ldots$	
$(p_2, 2)$		$SF(p_2,2) = \bigotimes_{q \in loc} SF(p_q,2)$	
$(p(y), 2)$		$SF(p(y),2) = (\{atb^y\} \otimes SF(p_3,1)) \cup (\{\neg atb^y\} \otimes SF(p'_3,1))$	
$(p_3, 1)$		$SF(p_3,1) = \ldots$	

Figure 3: Machine states and state partitions for the program $p = pickProc(\boldsymbol{a}, \boldsymbol{o}, x); carryToBase(\boldsymbol{a}, \boldsymbol{o})$ and the horizon $h = 3$.

Example 17 (Logistics Domain cont'd) Consider again the domain theory $DT = (AT, ST, OT)$ for the Logistics Domain of Example 14 and the AGTGolog procedures of Example 15. Let the AGTGolog program p be given by

$$p \;=\; pickProc(\boldsymbol{a}, \boldsymbol{o}, x); carryToBase(\boldsymbol{a}, \boldsymbol{o})\,,$$

and assume the horizon $h = 3$. Recall that a policy for p within h is obtained from p by replacing every single-agent choice in p within h by a single action, and every multi-agent choice in p within h by a collection of probability distributions, one over the actions of each agent. Learning an optimal policy now works as follows. First, we generate the state partition $SF(p', h')$ for every machine state (p', h') of p within h, that is, for every machine state (p', h') such that $(p, h) \rhd (p', h')$. Then, we use Algorithm *Learn* for p and h to learn an optimal policy $\pi(\sigma)$ for each joint state $\sigma = (\phi, p', h')$ such that $\phi \in SF(p', h')$ and $(p, h) \rhd (p', h')$: we run several times p within h for $\phi \in SF(p, h)$ until the expected utility v and the policy π stabilize for each joint state $\sigma = (\phi, p', h',)$ such that $\phi \in SF(p', h')$ and $(p, h) \rhd (p', h')$.

Machine States. Given the program p, we consider the following machine states:

$$\begin{array}{llll}
(p, & 3) & \text{with} & p = pickProc(\boldsymbol{a}, \boldsymbol{o}, x); carryToBase(\boldsymbol{a}, \boldsymbol{o})\,,\\
(p_1, & 3) & \text{with} & p_1 = tryToPickUp(\boldsymbol{a}, \boldsymbol{o}, x); carryToBase(\boldsymbol{a}, \boldsymbol{o})\,,\\
(p_{i,j}, & 3) & \text{with} & p_{i,j} = a_i \| o_j; carryToBase(\boldsymbol{a}, \boldsymbol{o})\,,\\
(p'_1, & 3) & \text{with} & p'_1 = pickUpS(\boldsymbol{a}, x);\ carryToBase(\boldsymbol{a}, \boldsymbol{o})\,,\\
(p_2, & 2) & \text{with} & p_2 = carryToBase(\boldsymbol{a}, \boldsymbol{o})\,,\\
(p(y), & 2) & \text{with} & p(y) = moveS(\boldsymbol{a}, y); \textbf{if}\ atBase(\boldsymbol{a})\ \textbf{then}\ p_3\ \textbf{else}\ p_2\,,\\
(p_3, & 1) & \text{with} & p_3 = \pi[x\colon obj]\, dropS(\boldsymbol{a}, x)\,.
\end{array}$$

These are in the following relations:

$$(p, 3) \rhd (p_1, 3) \rhd (p_{i,j}, 3) \rhd (p_2, 2) \rhd (p_q, 2) \rhd (p_3, 1)\,,$$
$$(p, 3) \rhd (p'_1, 3) \rhd (p_2, 2) \rhd (p_q, 2) \rhd (p_3, 1)\,.$$

State Partitions and Policies. We now consider the state partitions and policies associated with the above machine states.

- $(p, 3)$: The state partition for the machine state $(p, 3)$ is given as follows:

$$SF(p,3) \;=\; (\{asl\} \otimes SF(p_1,3)) \cup (\{\neg asl\} \otimes SF(p_1',3))\,,$$

with $asl = atSameLocation(\boldsymbol{a}, \boldsymbol{o})$. In the learning algorithm, we then associate with every joint state $(\phi, p, 3)$, where $\phi = asl \wedge \phi' \in SF(p,3)$ (resp., $\phi = \neg asl \wedge \phi' \in SF(p,3)$), the policy $\pi(\phi', p_1, 3)$ (resp., $\pi(\phi', p_1', 3)$), distinguishing the two different cases of the agents being (resp., not being) at the same location.

- $(p_1, 3)$: In the machine state $(p_1, 3)$, we reduce $p_1 = tryToPickUp(\boldsymbol{a}, \boldsymbol{o}, x)$; $carryTo\text{-}\ Base(\boldsymbol{a}, \boldsymbol{o})$ to the program:

$$\textbf{choice}(\boldsymbol{a}: a_1 \mid a_2) \,\|\, \textbf{choice}(\boldsymbol{o}: o_1 \mid o_2); p_2\,,$$

where $a_1 = pickUpS(\boldsymbol{a}, x)$, $a_2 = nop(\boldsymbol{a})$, $o_1 = pickUpS(\boldsymbol{o}, x)$, $o_2 = nop(\boldsymbol{o})$, and $p_2 \;=\; carryTo\text{-}Base(\boldsymbol{a}, \boldsymbol{o})$. Informally, we have a joint action choice of both agents, where the agents have to learn a mixed policy over their two possible actions (to pick up an object or to wait) against each other. We thus obtain the four program choices $p_{i,j} \;=\; a_i \| o_j\,; p_2$, where $i, j \in \{1, 2\}$, and the state partition for p_1 and $h = 3$ is given by:

$$SF(p_1,3) = \bigotimes_{i,j \in \{1,2\}} SF(p_{i,j},3)\,.$$

In the learning algorithm, for each state formula $\phi \in SF(p_1, 3)$, we have the associated expected utility $v_{i,j}$ and one policy $\pi_{i,j}$ for each program choice $p_{i,j}$ with $i, j \in \{1, 2\}$ and the horizon 3. We then determine a Nash pair

$$(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}}) = (a_1, a_2 \mapsto p, 1{-}p\,;\; o_1, o_2 \mapsto q, 1{-}q)$$

via the Nash selection function $\varepsilon\text{-}prefNash$ for the following matrix game:

		Agent $\boldsymbol{o}$	
		$o_1; p_2$	$o_2; p_2$
Agent $\boldsymbol{a}$	$a_1; p_2$	$(v_{1,1}, -v_{1,1})$	$(v_{1,2}, -v_{1,2})$
	$a_2; p_2$	$(v_{2,1}, -v_{2,1})$	$(v_{2,2}, -v_{2,2})$

The expected utility for p_1 and $h = 3$ is finally given by the expected value of $\{v_{i,j} | i, j \in \{1, 2\}\}$ under the Nash pair $(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}})$, and the corresponding policy is given by:

$$\begin{aligned}\pi(\phi, p_1, 3) \;=\; \mu_{\boldsymbol{a}} \| \mu_{\boldsymbol{o}}\,; \quad & \textbf{if}\, \alpha_1 \wedge \omega_1 \,\textbf{then}\, \pi(do(a_1 \| o_1, \phi), p_2, 2) \;\ldots \\ & \textbf{else if}\, \alpha_2 \wedge \omega_2 \,\textbf{then}\, \pi(do(a_2 \| o_2, \phi), p_2, 2)\,,\end{aligned}$$

where the $\alpha_i \wedge \omega_j$'s are the conditions in Algorithm 3 used to discriminate the executed joint actions, that is, α_i (resp., ω_j) is true iff a_i (resp., o_j) is the executed action among $\{a_i \mid i \in \{1, 2\}\}$ (resp., $\{o_j \mid j \in \{1, 2\}\}$), e.g., for $a_1 = pickUpS(\boldsymbol{a}, x)$, the condition is $\alpha_1 = holding(\boldsymbol{a}, x)$. For example, for the state formula:

$$\phi_1 = \neg asl \wedge ato \wedge atb^{q_1} \wedge \cdots \wedge atb^{q_m} \wedge \neg h_{\boldsymbol{a}} \wedge h_{\boldsymbol{o}} \in SF(p_1, 3),$$

where asl, ato, h_{ag}, and atb^q stand for the following formulas:

$$asl = atSameLocation(\boldsymbol{a}, \boldsymbol{o}), \quad ato = \exists y, obj\ (at(\boldsymbol{a}, y, s) \wedge onFloor(obj, y, s)),$$
$$h_{ag} = \exists x\ (holds(ag, x, s)), \qquad atb^q = Regr(atBase(\boldsymbol{a}, do(moveS(\boldsymbol{a}, q), s))),$$

and $q_1, \ldots, q_m$ are possible locations (see below), the algorithm produces the policy

$$\begin{aligned}\pi(\phi_1, p_1, 3) \;=\;& [\mu_{\boldsymbol{a}} \colon a_1, a_2 \mapsto 1, 0] \parallel [\mu_{\boldsymbol{o}} \colon o_1, o_2 \mapsto 0, 1]\,; \\ & \textbf{if}\ \alpha_1 \wedge \omega_1\ \textbf{then}\ \pi(do(a_1 \| o_1, \phi_1), p_2, 2)\ \ldots \\ & \textbf{else if}\ \alpha_2 \wedge \omega_2\ \textbf{then}\ \pi(do(a_2 \| o_2, \phi_1), p_2, 2)\,.\end{aligned}$$

Informally, the joint action $pickUpS(\boldsymbol{a}, x) \parallel nop(\boldsymbol{o})$ is selected with probability 1. In contrast, for the state formula:

$$\phi_2 = asl \wedge ato \wedge atb^{q_1} \wedge \cdots \wedge atb^{q_m} \wedge \neg h_{\boldsymbol{a}} \wedge \neg h_{\boldsymbol{o}} \in SF(p_1, 3),$$

the learning algorithm produces the following policy:

$$\begin{aligned}\pi(\phi_2, p_1, 3) \;=\;& [\mu_{\boldsymbol{a}} \colon a_1, a_2 \mapsto 0, 1] \parallel [\mu_{\boldsymbol{o}} \colon o_1, o_2 \mapsto 0, 1]\,; \\ & \textbf{if}\ \alpha_1 \wedge \omega_1\ \textbf{then}\ \pi(do(a_1 \| o_1, \phi_2), p_2, 2)\ \ldots \\ & \textbf{else if}\ \alpha_2 \wedge \omega_2\ \textbf{then}\ \pi(do(a_2 \| o_2, \phi_2), p_2, 2)\,.\end{aligned}$$

Informally, the joint action $nop(\boldsymbol{a}) \parallel nop(\boldsymbol{o})$ is selected with probability 1.

- $(p_1', 3)$: In the machine state $(p_1', 3)$, we have the case of a deterministic first program action a, where $a = pickUpS(\boldsymbol{a}, x)$. So, the state partition for $(p_1', 3)$ is of the form

$$\begin{aligned}SF(p_1', 3) \;=\;& (P^a_{rw}(\vec{x}, s) \otimes \{Regr(\phi(\vec{x}, do(a, s)) \wedge Poss(a, s) \mid \\ & \phi(\vec{x}, s) \in SF(p_2, 2)\}) \cup \{\neg Poss(a, s)\} \setminus \{\bot\}\,.\end{aligned}$$

Depending on the state formula $\phi \in SF(p_1', 3)$, either $Poss(a, \phi)$ or $\neg Poss(a, \phi)$ holds, that is, the action a is either executable or not. In the first case, the policy for the joint state $(\phi, p_1', 3)$ is given by $a; \pi(do(a, \phi), p_2, 2)$, while in the second case, a is not executable, and therefore the policy for the joint state $(\phi, p_1', 3)$ is given by $stop$.

- $(p_2, 2)$: In the machine state $(p_2, 2)$, we reduce $p_2 = carryToBase(\boldsymbol{a}, \boldsymbol{o})$ to the nondeterministic program choices

$$p(y) = moveS(\boldsymbol{a}, y); \textbf{if}\ atBase(\boldsymbol{a})\ \textbf{then}\ p_3\ \textbf{else}\ p_3',$$

where y ranges over the possible locations in $loc = \{q_1, \ldots, q_m\}$. So, the state partition for the machine state $(p_2, 2)$ is

$$SF(p_2, 2) = \bigotimes_{q \in loc} SF(p(y), 2)\,,$$

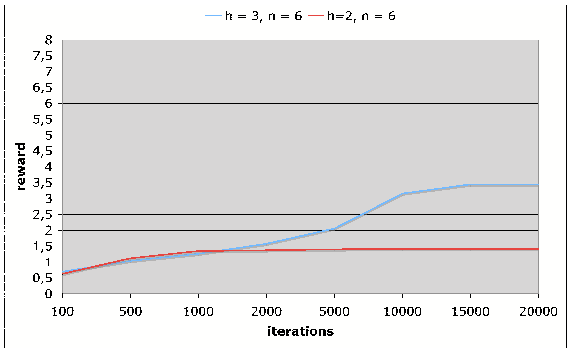

Figure 4: Average total reward gained by the program $p = pickProc(\boldsymbol{a}, \boldsymbol{o}, x); carryToBase(\boldsymbol{a}, \boldsymbol{o})$ with the horizons $h = 3$ (blue) and $h = 2$ (red) after x iterations in a Logistics Domain with six nodes. We considered the generated policies obtained after 100, 500, 1000, 2000, 5000, 10000, 15000, and 20000 iterations (horizontal axis), collecting the average total reward (vertical axis) after 100 runs for each case.

where the state partitions for the machine states $(p(y), 2)$ are

$$SF(p(y), 2) = (\{atb^y\} \otimes SF(p_3, 1)) \cup (\{\neg atb^y\} \otimes SF(p'_3, 1))$$

and $atb^y = Regr(atBase(\boldsymbol{a}, do(moveS(\boldsymbol{a}, y), s)))$. For each state formula $\phi \in SF(p_2, 2)$, the policy for the joint state $(\phi, p_2, 2)$ is then defined as the policy for $(\phi, p(y), 2)$ of an optimal program choice $p(y)$.

Experimental Results. We have run the learning algorithm on the above program p:

$$p \;=\; pickProc(\boldsymbol{a}, \boldsymbol{o}, x); carryToBase(\boldsymbol{a}, \boldsymbol{o})\,,$$

considering the horizons $h = 2$ and $h = 3$ in a logistics domain with six locations. In this setting, we have considered two randomly positioned objects and two base locations, one for each agent. At each iteration of the algorithm $Update$, a new intial state is randomly created by generating a new graph and the initial positions of the agent, the opponent, the two objects, and the two bases. The simulation environment was set as follows: the success of picking up an object and dropping it to a base was associated with the probability 0.9 and the rewards 4 and 20, respectively; the execution attempt of a non-executable action is penalized with -3; instead, the execution of the action $moveS$ had a cost of -1. Figure 4 illustrates the average total reward gained by the agent $\boldsymbol{a}$ that executes p against the opponent $\boldsymbol{o}$, considering n iterations of the algorithm $Update$. Here, the generated policy is evaluated at different stages of the learning algorithm: after n iterations of the algorithm $Update$, we executed the current policy and considered the average total reward collected in 100 runs.

7 Convergence Result

In this section, we show that the learning algorithm $Learn$ converges with probability 1 on every AGTGolog program p and horizon $h \geqslant 0$. For this result, we assume that (i) every subprogram of p within h is executed

infinitely many times in every state formula $\phi \in SF(p,h)$, and that (ii) the learning rate α is decayed appropriately. The main convergence result is obtained by induction on the structure of AGTGolog programs. It is based on several preparative results, which informally show that convergence with probability 1 can be pushed through the updating step, the selections of maximal and minimal arguments within an ε-range and the selections of Nash pairs within an ε-range in the learning algorithm.

The first preparative result informally shows that convergence with probability 1 can be pushed through the updating step for stochastic actions in $Learn$.

Proposition 18 *Let α_n, $n \in \{1,2,\ldots\}$, be a sequence of real numbers such that $0 \leqslant \alpha_n < 1$, $\sum_{n=1}^{\infty} \alpha_n = \infty$, and $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$. Let $k \geqslant 1$, and let ξ_n, $n \in \{1,2,\ldots\}$, be a sequence of random variables with the possible outcomes $a_{n,1},\ldots,a_{n,k}$ along with the outcome probabilities $p_1,\ldots,p_k$ ($p_1+\cdots+p_k=1$), respectively. Suppose that every sequence $a_{n,i}$, $i \in \{1,\ldots,k\}$, converges with probability 1 against a_i. Let the sequence x_n, $n \in \{1,2,\ldots\}$, be defined by $x_{n+1} = (1-\alpha_n)\cdot x_n + \alpha_n \cdot \xi_n$ for all $n \in \{1,2,\ldots\}$. Then, x_n converges with probability 1 against $\sum_{i=1}^{k} p_i \cdot a_i$.*

The second preparative result informally shows that convergence with probability 1 can be pushed through the maximizations and the selections of maximal arguments within an ε-range for single-agent choice constructs in $Learn$. A similar result holds for the minimizations and selections of minimal arguments within an ε-range in $Learn$.

Proposition 19 *Let the sequences $a_{n,i}$, $i \in \{1,\ldots,k\}$, converge with probability 1 against a_i. Then, (a) the sequence $b_n = \max(a_{n,1},\ldots,a_{n,k})$ converges with probability 1 against $b = \max(a_1,\ldots,a_k)$; and (b) if additionally $\varepsilon > 0$ is sufficiently small, then ε-$prefArgmax(a_{n,1},\ldots,a_{n,k})$ converges with probability 1 against $prefArgmax(a_1,\ldots,a_k)$.*

The third preparative result informally shows that convergence with probability 1 can be pushed through the computations of expected rewards of Nash pairs and the selections of Nash pairs within an ε-range for joint action choice constructs in $Learn$.

Proposition 20 *Let the sequences $a_{n,i,j}$, where $i \in \{1,\ldots,k\}$ and $j \in \{1,\ldots,l\}$, converge with probability 1 against $a_{i,j}$. Then, (a) the expected reward v_n of the matrix game $a_n = (a_{n,i,j})_{i,j}$ under a Nash pair converges with probability 1 against the expected reward v of the matrix game $a = (a_{i,j})_{i,j}$ under a Nash pair; and (b) if additionally $\varepsilon > 0$ is sufficiently small, then ε-$prefNash(a_n)$ converges with probability 1 against $prefNash(a)$.*

We are now ready to formulate the main convergence result, which says that the expected utility and policy computed by the learning algorithm $Learn$ on every AGTGolog program p and horizon $h \geqslant 0$ converges with probability 1 against the expected utility and policy, respectively, specified by the GTGolog interpreter for p and h [11] (where the immediate rewards and transition probabilities of the learning process are fixed and explicitly given in the domain theory). Observe that since the GTGolog interpreter for p relative to h generates an optimal policy, this also means that the policy learned by $Learn(p,h)$ is optimal. Furthermore, the result implies that in the single-agent case, the expected utility and policy computed by the learning algorithm converge against the expected utility and policy, respectively, specified by a variant of the DTGolog interpreter (for fixed and explicitly given immediate rewards and transition probabilities). That is, the multi-agent learning algorithm for AGTGolog includes as a special case a single-agent learning algorithm for DTGolog.

Theorem 21 *Let $DT = (AT, ST, OT)$ be a domain theory, p be an AGTGolog program w.r.t. DT, and $h \geqslant 0$ be a horizon. Let every subprogram of p within h be executed infinitely many times in every $\phi \in SF(p, h)$. Let α_n be the learning rate α at the n-th call of $Learn(p, h)$. Let $0 \leqslant \alpha_n < 1$, $\sum_{n=1}^{\infty} \alpha_n = \infty$, and $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$. Let $\varepsilon > 0$ be sufficiently small. Let $v_n(\sigma)$ (resp., $\pi_n(\sigma)$), for all $\sigma = (\phi, p, h)$ and $\phi \in SF(p, h)$, denote $v(\sigma)$ (resp., $\pi(\sigma)$) after the n-th call of $Learn(p, h)$. Let v (resp., π) be the expected utility (resp., optimal policy) specified by the GTGolog interpreter for p w.r.t. the domain theory DT' (obtained from DT by additionally specifying rewards and transition probabilities) with horizon h at a situation s satisfying ϕ. Then, v_n (resp., π_n) converge with probability 1 against v (resp., π).*

8 Conclusion

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 online instantiate a partially specified behavior playing against an adversary. Differently from the classical Golog approach, the interpreter generates not only complex sequences of actions (the policy), but also the state abstraction induced by the program at the different executive states (machine states). In this way, the Golog integration between action theory and programs allows to naturally combine the advantages of symbolic techniques [2, 19] with the strength of hierarchical reinforcement learning [31, 6, 1, 24]. 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.

The main focus of this paper was on AGTGolog as a language for learning against an adversary in DTGolog programs (where we are initially given a DTGolog program for the agent that we control, which is then completed to a GTGolog program by filling in all possible actions of the adversary) and for multi-agent learning in GTGolog programs (where we are initially given a GTGolog program that specifies the joint behavior of two competing agents). It is important to point out that AGTGolog can be easily extended to learning in a case between the above two cases, where each of the two competing agents has its own individual DTGolog program. Here, a joint policy is learned from the cross-product of the two DTGolog programs, which is possible due to the finite horizon assumption.

Although we have considered only the case of two competing agents, the framework can be easily extended to two competing teams of cooperative agents, where the agents of the same team all have the same rewards, and the agents of different teams have zero-sum rewards (which can be done similarly as for GTGolog; see [11] for further details).

An interesting topic for future research is to explore whether the presented approach can be extended to the partially observable case (which may be done along the lines of [17]).

Acknowledgments. This work was supported by the Austrian Science Fund under the project P18146-N04 and by the German Research Foundation (DFG) under the Heisenberg Programme. We thank the reviewers of this paper and its ECAI-2006 poster and KI-2006 abstract for their constructive comments, which helped to improve this work.

Appendix A: Proofs

Proof of Proposition 18. Observe first that since every sequence $a_{n,i}$, $i \in \{1, \ldots, k\}$, converges with probability 1 against a_i, the sequence $(a_{n,1}, \ldots, a_{n,k})$ converges with probability 1 against $(a_1, \ldots, a_k)$. That

is, for every $\varepsilon > 0$, some $n_\varepsilon \in \{1, 2, \ldots\}$ exists such that $|a_{n,i} - a_i| < \frac{\varepsilon}{2}$, for all $i \in \{1, \ldots, k\}$ and all $n \geqslant n_\varepsilon$, with probability 1.

Observe then that a standard result in stochastic convergence (e.g., Theorem 2.3.1 of [21]) says that the sequence $x_{n+1} = (1 - \alpha_n) \cdot x_n + \alpha_n \cdot y_n$, where (1) $0 \leqslant \alpha_n < 1$, (2) $\sum_{n=1}^{\infty} \alpha_n = \infty$, (3) $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$, and (4) y_n, $n \in \{1, 2, \ldots\}$, is a sequence of bounded random variables with mean μ, converges with probability 1 against μ. Hence, in particular, the sequence $x_{n+1} = (1 - \alpha_n) \cdot x_n + \alpha_n \cdot \xi'_n$, where ξ'_n, $n \in \{1, 2, \ldots\}$, is a sequence of random variables with the possible outcomes $a_1, \ldots, a_k$ along with the outcome probabilities $p_1, \ldots, p_k$ ($p_1 + \cdots + p_k = 1$), respectively, converges with probability 1 against $\mu = \sum_{i=1}^{k} p_i \cdot a_i$. That is, for every $\varepsilon > 0$, there exists some $n'_\varepsilon \in \{1, 2, \ldots\}$ such that $|x_n - \mu| < \frac{\varepsilon}{2}$, for all $n \geqslant n'_\varepsilon$, with probability 1.

In summary, we thus obtain the following for all $n \geqslant \max(n_\varepsilon, n'_\varepsilon)$:

$$\begin{array}{rclcccl} x_{n+1} & = & (1 - \alpha_n) \cdot x_n + \alpha_n \cdot \xi_n & < & (1 - \alpha_n) \cdot x_n + \alpha_n \cdot \xi'_n + \frac{\varepsilon}{2} & < & \mu + \varepsilon\,, \\ x_{n+1} & = & (1 - \alpha_n) \cdot x_n + \alpha_n \cdot \xi_n & > & (1 - \alpha_n) \cdot x_n + \alpha_n \cdot \xi'_n - \frac{\varepsilon}{2} & > & \mu - \varepsilon\,, \end{array}$$

both with probability 1. That is, for all $n \geqslant \max(n_\varepsilon, n'_\varepsilon) + 1$, it holds $|x_n - \mu| < \varepsilon$ with probability 1. That is, x_n converges with probability 1 against $\mu = \sum_{i=1}^{k} p_i \cdot a_i$. □

Proof of Proposition 19. (a) Immediate by the observation that the convergence with probability 1 of every sequence $a_{n,i}$, $i \in \{1, \ldots, k\}$, against a_i implies the convergence with probability 1 of the sequence $(a_{n,1}, \ldots, a_{n,k})$ against $(a_1, \ldots, a_k)$. Informally, then there exists some n_0 such that for all $n \geqslant n_0$, every maximal argument of $\max(a_{n,1}, \ldots, a_{n,k})$ corresponds to some maximal argument of $\max(a_1, \ldots, a_k)$. Since any of the former converges with probability 1 against the latter, it thus follows that $\max(a_{n,1}, \ldots, a_{n,k})$ converges with probability 1 against $\max(a_1, \ldots, a_k)$.

(b) If additionally $\varepsilon > 0$ is sufficiently small, then some n_0 exists such that for all $n \geqslant n_0$, it holds that $\{i \in \{1, \ldots, k\} \mid a_{n,i} \in \max(a_{n,1}, \ldots, a_{n,k}) + [-\varepsilon, +\varepsilon]\}$ coincides with $\{i \in \{1, \ldots, k\} \mid a_i = \max(a_1, \ldots, a_k)\}$ with probability 1. Hence, $\varepsilon\text{-}prefArgmax(a_{n,1}, \ldots, a_{n,k})$ converges with probability 1 against $pref\text{-}Argmax(a_1, \ldots, a_k)$. □

Proof of Proposition 20. (a) Immediate by the observation that the convergence with probability 1 of every sequence $a_{n,i,j}$, $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, l\}$, against $a_{i,j}$ implies the convergence with probability 1 of the sequence $(a_{n,i,j})_{i,j}$ against $(a_{i,j})_{i,j}$. Hence, $\max_{\pi \in PD(\{1,\ldots,k\})} \min_{j \in \{1,\ldots,l\}} \sum_{i \in \{1,\ldots,k\}} a_{n,i,j} \cdot \pi(i)$ converges with probability 1 against $\max_{\pi \in PD(\{1,\ldots,k\})} \min_{j \in \{1,\ldots,l\}} \sum_{i \in \{1,\ldots,k\}} a_{i,j} \cdot \pi(i)$.

(b) If additionally $\varepsilon > 0$ is sufficiently small, then some n_0 exists such that any "slipping" of probabilities along the expected reward under a Nash pair that is possible for the matrix game a is also possible with probability 1 for every matrix game a_n with $n \geqslant n_0$. Hence, $\varepsilon\text{-}prefNash(a_n)$ converges with probability 1 against $prefNash(a)$. □

Proof of Theorem 21. We prove the theorem by induction on the structure of AGTGolog programs. We use the results in Propositions 18, 19, and 20 that convergence with probability 1 can be pushed through the updating step, the selections of maximal and minimal arguments within an ε-range, and the selections of Nash pairs within an ε-range, respectively. For a detailed specification of the GTGolog interpreter, we refer the reader to [11].

Basis: If p is the empty program, or h is the zero horizon, then $v_n(\sigma) = 0$ and $\pi_n(\sigma) = nil$ trivially converge with probability 1 against the results $v = 0$ and $\pi = nil$, respectively, of the GTGolog interpreter for these two cases. Similarly, if $p = a; p'$ and $\neg Poss(a, \phi)$, or $p = \psi?; p'$ and $\neg\psi[\phi]$, then $v_n(\sigma) = 0$ and $\pi_n(\sigma) = stop$

converge with probability 1 against the results $v = 0$ and $\pi = stop$, respectively, of the GTGolog interpreter for these two cases.

Induction: If p is of the form $a; p'$ such that (i) $Poss(a, \phi)$ and (ii) a is deterministic, then by the induction hypothesis, $v_n(do(a, \phi), p', h-1)$ and $\pi_n(do(a, \phi), p', h-1)$ converge with probability 1 against the results v and π, respectively, of the GTGolog interpreter for p' relative to $do(a, s_\phi)$ and $h-1$. Let $reward = reward(a, s_\phi)$ be the reward to agent $\boldsymbol{a}$ when executing a in s_ϕ. Hence,

$$\begin{array}{lcl} v_{n+1}(\sigma) & = & v_n(do(a, \phi), p', h-1) + reward \text{ and} \\ \pi_{n+1}(\sigma) & = & a; \pi_n(do(a, \phi), p', h-1) \end{array}$$

converge with probability 1 against the results $v + reward(a, s_\phi)$ and $a; \pi$, respectively, of the GTGolog interpreter for $a\,; p'$ relative to s_ϕ and h.

If $p = a; p'$ such that (i) $Poss(a, \phi)$ and (ii) a is stochastic, then by the induction hypothesis, $v_n(\phi, n_q\,; p', h)$ and $\pi_n(\phi, n_q\,; p', h)$, $q \in \{1, \ldots, k\}$, converge with probability 1 against the results v_q and π_q, respectively, of the GTGolog interpreter for $n_q\,; p'$ relative to s_ϕ and h. Let nature choose each n_q, $q \in \{1, \ldots, k\}$, with probability $prob(a, s_\phi, n_q)$. By Proposition 18,

$$\begin{array}{lcl} v_{n+1}(\sigma) & = & (1-\alpha) \cdot v_n(\sigma) + \alpha \cdot v_n(\phi, n_q; p', h) \text{ and} \\ \pi_{n+1}(\sigma) & = & a; \textbf{if } \phi_1 \textbf{ then } \pi_n(\phi, n_1; p', h) \ldots \textbf{ else if } \phi_k \textbf{ then } \pi_n(\phi, n_k; p', h) \end{array}$$

converge with probability 1 against the results

$$\begin{array}{l} \sum_{q=1}^{k} v_q \cdot prob(a, s_\phi, n_q) \text{ and} \\ a\,; \textbf{if } \phi_1 \textbf{ then } \pi_1 \ldots \textbf{ else if } \phi_k \textbf{ then } \pi_k\,, \end{array}$$

respectively, of the GTGolog interpreter for $a\,; p'$ relative to s_ϕ and h.

If $p = \textbf{choice}(\boldsymbol{a}\colon a_1 | \cdots | a_m); p'$, then by the induction hypothesis, $v_n(\phi, \boldsymbol{a}\colon a_i\,; p', h)$ and $\pi_n(\phi, \boldsymbol{a}\!:\!a_i\,; p', h)$ converge with probability 1 against the results v_i and $\boldsymbol{a}\!:\!a_i\,; \pi_i$, respectively, of the GTGolog interpreter for $\boldsymbol{a}\!:\!a_i\,; p'$ relative to s_ϕ and h. By Proposition 19,

$$\begin{array}{lcl} v_{n+1}(\sigma) & = & v_n(\phi, \boldsymbol{a}\!:\!a_k; p', h) \text{ and} \\ \pi_{n+1}(\sigma) & = & \boldsymbol{a}\!:\!a_k\,; \textbf{if } \phi_1 \textbf{ then } \pi_n(do(\boldsymbol{a}\!:\!a_1, \phi), p', h-1) \ldots \\ & & \qquad \textbf{else if } \phi_m \textbf{ then } \pi_n(do(\boldsymbol{a}\!:\!a_m, \phi), p', h-1)\,, \end{array}$$

where $k = \varepsilon\text{-}prefArgmax_{i \in \{1,\ldots,m\}} v_n(\phi, \boldsymbol{a}\!:\!a_i; p', h)$, converge with probability 1 against the results

$$\begin{array}{l} v_l \text{ and} \\ \boldsymbol{a}\!:\!a_l\,; \textbf{if } \phi_1 \textbf{ then } \pi_1 \ldots \textbf{ else if } \phi_m \textbf{ then } \pi_m\,, \end{array}$$

respectively, where $l = prefArgmax_{i \in \{1,\ldots,m\}} v_i$, of the GTGolog interpreter for $\textbf{choice}(\boldsymbol{a}\colon\ a_1 | \cdots | a_m); p'$ relative to s_ϕ and h. The line of argumentation is similar for the cases $p = \textbf{choice}(\boldsymbol{o}\colon o_1 | \cdots | o_m); p'$, $p = (p_1 \,|\, p_2); p'$, and $p = p^\star; p'$.

If $p = \textbf{choice}(\boldsymbol{a}\colon a_1 | \cdots | a_l) \,\|\, \textbf{choice}(\boldsymbol{o}\colon o_1 | \cdots | o_m); p'$, then by the induction hypothesis, $v_n(\phi, \boldsymbol{a}\!:\!a_i \| \boldsymbol{o}\!:\!o_j; p', h)$ and $\pi_n(\phi, \boldsymbol{a}\!:\!a_i \| \boldsymbol{o}\!:\!o_j; p', h)$, where $i \in \{1, \ldots, l\}$ and $j \in \{1, \ldots, m\}$, converge with probability 1 against the results $v_{i,j}$ and $\boldsymbol{a}\!:\!a_i \| \boldsymbol{o}\!:\!o_j; \pi_{i,j}$, respectively, of the GTGolog interpreter for $\boldsymbol{a}\!:\!a_i \| \boldsymbol{o}\!:\!o_j; p'$ relative to s_ϕ and h. So, $\varepsilon\text{-}prefNash(v(\phi, \boldsymbol{a}\!:\!a_i \| \ \boldsymbol{o}\!:\!o_j; p', h) \mid i, j\})$ converges against $prefNash(\{v_{i,j} \mid i, j\})$

with probability 1. Thus,

$$\begin{array}{rcl} v_{n+1}(\sigma) & = & \sum_{i=1}^{l}\sum_{j=1}^{m}\mu_{\boldsymbol{a}}(a_i)\cdot\mu_{\boldsymbol{o}}(o_j)\cdot v_n(\phi,\, \boldsymbol{a}{:}a_i\|\boldsymbol{o}{:}o_j; p', h) \text{ and} \\ \pi_{n+1}(\sigma) & = & \mu_{\boldsymbol{a}}\|\mu_{\boldsymbol{o}};\ \textbf{if } \phi_1{\wedge}\psi_1 \textbf{ then } \pi_n(do(\boldsymbol{a}{:}a_1\|\boldsymbol{o}{:}o_1, \phi), p', h-1)\ldots \\ & & \qquad \textbf{else if } \phi_l{\wedge}\psi_m \textbf{ then } \pi_n(do(\boldsymbol{a}{:}a_l\|\boldsymbol{o}{:}o_m, \phi), p', h-1)\,, \end{array}$$

where $(\mu_{\boldsymbol{a}}, \mu_{\boldsymbol{o}}) = \varepsilon\text{-}prefNash(\{v(\phi, \boldsymbol{a}{:}a_i\|\boldsymbol{o}{:}o_j; p', h) \mid i, j\})$, converge with probability 1 against the results

$$\begin{array}{l} \sum_{i=1}^{l}\sum_{j=1}^{m}\mu'_{\boldsymbol{a}}(a_i)\cdot\mu'_{\boldsymbol{o}}(o_j)\cdot v_{i,j} \text{ and} \\ \mu'_{\boldsymbol{a}}\|\mu'_{\boldsymbol{o}};\ \textbf{if } \phi_1{\wedge}\psi_1 \textbf{ then } \pi_{1,1}\ldots \textbf{ else if } \phi_l{\wedge}\psi_m \textbf{ then } \pi_{l,m}\,, \end{array}$$

respectively, where $(\mu'_{\boldsymbol{a}}, \mu'_{\boldsymbol{o}}) = prefNash(\{v_{i,j} \mid i, j\})$, of the GTGolog interpreter for **choice**($\boldsymbol{a}{:}\ a_1|\cdots|a_l$) $\|$ **choice**($\boldsymbol{o}{:}\ o_1|\cdots|o_m$); p' relative to s_ϕ and h.

If $p = \psi?; p'$ and $\phi = \psi \wedge \phi'$, then by the induction hypothesis, $v_n(\phi', p', h)$ and $\pi_n(\phi', p', h)$ converge with probability 1 against the results v and π, respectively, of the GTGolog interpreter for p' relative to $s_{\phi'}$ and h. So, $v_{n+1}(\sigma) = v_n(\phi, p', h)$ and $\pi_{n+1}(\sigma) = \pi_n(\phi, p', h)$ trivially converge with probability 1 against the results v and π, respectively, of the GTGolog interpreter for $\psi?; p'$ relative to s_ϕ and h.

All the other cases are reduced to one of the former cases. □

References

[1] D. Andre and S. J. Russell. State abstraction for programmable reinforcement learning agents. In *Proceedings AAAI-2002*, pp. 119–125. AAAI Press, 2002.

[2] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings IJCAI-2001*, pp. 690–700. Morgan Kaufmann, 2001.

[3] R. A. Brooks. Humanoid robots. *Communications of the ACM*, 45(3):33–38, 2002.

[4] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artif. Intell.*, 114(1–2):3–55, 1999.

[5] P. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Proceedings NIPS-1993*, pp. 271–278. Morgan Kaufmann, 1994.

[6] T. G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings ICML-1998*, pp. 118–126. Morgan Kaufmann, 1998.

[7] H. Durrant-Whyte. Autonomous guided vehicle for cargo handling applications. *International Journal of Robotics Research*, 15(5):407–440, 1996.

[8] A. Ferrein, C. Fritz, and G. Lakemeyer. Using Golog for deliberation and team coordination in robotic soccer. *Künstliche Intelligenz*, 1:24–43, 2005.

[9] A. Finzi and T. Lukasiewicz. Structure-based causes and explanations in the independent choice logic. In *Proceedings UAI-2003*, pp. 225–232. Morgan Kaufmann, 2003.

[10] A. Finzi and T. Lukasiewicz. Game-theoretic agent programming in Golog. In *Proceedings ECAI-2004*, pp. 23–27. IOS Press, 2004.

[11] A. Finzi and T. Lukasiewicz. Game-theoretic agent programming in Golog. Technical Report INFSYS RR 1843-04-02, Institut für Informationssysteme, TU Wien, July 2008.

[12] A. Finzi and T. Lukasiewicz. Relational Markov games. In *Proceedings JELIA-2004*, Vol. 3229 of *LNCS/LNAI*, pp. 320–333. Springer, 2004.

[13] A. Finzi and T. Lukasiewicz. Game-theoretic Golog under partial observability. In *Proceedings AAMAS-2005*, pp. 1301–1302. ACM Press, 2005.

[14] A. Finzi and T. Lukasiewicz. Game-theoretic agent programming in Golog under partial observability. In *Proc. KI-2006*, Vol. 4314 of *LNCS/LNAI*, pp. 389–403. Springer, 2007.

[15] A. Finzi and T. Lukasiewicz. Adaptive multi-agent programming in GTGolog (poster). In *Proceedings ECAI-2006*, pp. 753–754. IOS Press, 2006.

[16] A. Finzi and T. Lukasiewicz. Adaptive multi-agent programming in GTGolog. In *Proceedings KI-2006*, Vol. 4314 of *LNCS/LNAI*, pp. 113–127. Springer, 2007.

[17] A. Finzi and T. Lukasiewicz. Team programming in Golog under partial observability. In *Proceedings IJCAI-2007*, pp. 2097–2102. AAAI Press / IJCAI, 2007.

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

[19] C. Gretton and S. Thiebaux. Exploiting first-order regression in inductive policy selection. In *Proceedings UAI-2004*, pp. 217–225. AUAI Press, 2004.

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

[21] H. J. Kushner and D. S. Clark. *Stochastic Approximation Methods for Constrained and Unconstrained Systems*. Springer, 1978.

[22] 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.

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

[24] B. Marthi, S. J. Russell, D. Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In *Proceedings IJCAI-2005*, pp. 779–785. Professional Book Center, 2005.

[25] 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.

[26] M. Montemerlo, J. Pineau, N. Roy, S. Thrun, and V. Verma. Experiences with a mobile robotic guide for the elderly. In *Proceedings AAAI/IAAI-2002*, pp. 587–592. AAAI Press/MIT Press, 2002.

[27] R. B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, 1997.

[28] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

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

[30] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Probabilistic algorithms and the interactive museum tour-guide robot Minerva. *International Journal of Robotics Research*, 19(11):972–999, 2000.

[31] R. Parr and S. J. Russell. Reinforcement learning with hierarchies of machines. In *Proceedings NIPS-1997*, Vol. 10, pp. 1043–1049. MIT Press, 1998.

[32] J. Pinto. Integrating discrete and continuous change in a logical framework. *Computational Intelligence*, 14(1):39–88, 1998.

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

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

[35] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, 2nd edition. Prentice Hall, 2002.

[36] 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. AAAI Press / MIT Press, 2000.

[37] M. Soutchanski. *High-Level Robot Programming in Dynamic and Incompletely Known Environments*. Ph.D. Thesis, University of Toronto, Canada, 2003.

[38] M. Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5:533–565, 2005.

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

[40] M. M. Veloso. Entertainment robotics. *Communications of the ACM*, 45(3):59–63, 2002.

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

[42] C. J. C. H. Watkins. *Learning from Delayed Rewards*. Ph.D. Thesis, King's College, Cambridge, UK, 1989.

[43] C. J. C. H. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8:279–292, 1992.