Unreal GOLOG Bots

Stefan Jacobs and **Alexander Ferrein** and **Gerhard Lakemeyer**
RWTH Aachen University
Computer Science Department
Ahornstr. 55, 52056 Aachen, Germany
{sjacobs,ferrein,gerhard}@cs.rwth-aachen.de

Abstract

Even though reasoning and, in particular, planning techniques have had a long tradition in Artificial Intelligence, these have only recently been applied to interactive computer games. In this paper we propose the use of READYLOG, a variant of the logic-based action language GOLOG, to build game bots. The language combines features from classical programming languages with decision-theoretic planning. The feasibility of the approach is demonstrated by integrating READYLOG with the game UNREAL TOURNAMENT.

1 Introduction

Interactive computer games have come a long way since the introduction of *Pac-Man* many years ago. In particular, the availability of sophisticated graphics engines for regular PCs has made it possibly to create complex realistic 3D scenarios with multiple players controlled by either the computer or by humans. Recent examples of such games are HALF LIFE 2 [Valve Corporation, 2005], MEDAL OF HONOR [Electronic Arts Inc., 2005], or UNREAL TOURNAMENT [Epic Games Inc., 2005], which is the focus of this paper. What makes these games challenging for both humans and computers is their fast pace and the fact that a player usually has only incomplete or uncertain knowledge about the world.

Artificial Intelligence so far has had rather little impact on the development of computer-controlled players, also called game bots. In the commercial world, for example, simple and easy-to-use scripting languages are often preferred to specify games and agents [Berger, 2002], and finite state machines are a popular means to specify reactive behavior [Fu and Houlette, 2004]. To turn such agents into challenging opponents for humans, they are often given more knowledge about the game situation and more powerful capabilities than are available to the human player.

Nevertheless, interactive games and, in particular, UNREAL TOURNAMENT have caught the attention of AI recently. Kaminka et al. [Kaminka *et al.*, 2002] have proposed UNREAL TOURNAMENT 2004 as a framework for research in multi-agent systems.[1] Their own work has focused on areas like navigation, mapping and exploration. In [Munoz-Avila and Fisher, 2004], hierarchical planning techniques are used to devise long-term strategies for UNREAL TOURNAMENT bots. Recently, Magerko at al. [Magerko *et al.*, 2004] have connected the powerful rule-based system SOAR [Lewis, 1999] to Haunt II, an extension of UNREAL TOURNAMENT.

Our own work fits into this line of work on symbolic reasoning applied to UNREAL TOURNAMENT, with a focus on real-time decision-theoretic planning techniques.[2] In particular, we propose to use READYLOG [Ferrein *et al.*, 2004], a variant of the action language GOLOG [Levesque *et al.*, 1997], for the specification of game bots. GOLOG has been applied to control animated characters before by Funge [1998]. However, there the scenario was much simpler with complete knowledge about the world and no uncertainty.

Roughly, READYLOG is a high-level programming language with the usual constructs found in imperative programming languages and additional ones which allow agents to choose among alternative actions. Here, actions are understood as in AI planning, with effects and preconditions specified in a logical language. Built into the language is also a decision-theoretic planning component, which can deal with uncertainty in the form of stochastic actions. In general, decision-theoretic planning can be very time consuming. As we will see, combining it with explicit programming of behavior, the user can control the amount of planning so that it becomes feasible for real-time applications. We have demonstrated this already by using READYLOG to control soccer playing robots [Ferrein *et al.*, 2004], and we are using the same highly-optimized Prolog implementation for the work described in this paper.

The rest of the paper is organized as follows. In the next section we give a brief introduction to UNREAL TOURNAMENT 2004, followed by a very brief discussion of READYLOG and its foundations. In Section 4 we discuss how UNREAL TOURNAMENT can be modelled in READYLOG. We then present some experimental results and conclude.

[1]In the area of strategy games, Buro [Buro, 2003] is also developing an open-source game to serve as a testbed for AI techniques.

[2]Real-time planning is also considered in [Orkin, 2004], but without the decision-theoretic aspect.

2 UNREAL TOURNAMENT 2004

UNREAL II and UNREAL TOURNAMENT 2004 [Epic Games Inc., 2005] are two state-of-the-art interactive computer games. While the former is mainly a single-player game, the latter is a multi-player game. Here we focus on using and modifying the bot framework of UNREAL TOURNAMENT 2004 because the bots available therein are programmed to behave like human adversaries for training purposes.

The engine itself is mainly written in C++ and it cannot be modified. In contrast the complete Unreal Script (in the following USCRIPT) code controlling the engine is publicly available and modifiable for each game. For instance, introducing new kinds of game play like playing soccer in teams or the game of Tetris have been implemented on the basis of the Unreal Engine. All this can be defined easily in USCRIPT, a simple, object-oriented, Java-like language which is publicly available.

In UNREAL TOURNAMENT 2004 ten types of gameplay or game modes have been implemented and published. For our work the following game types are of interest:

- *Deathmatch* (DM) is a game type where each player is on its own and struggles with all other competitors for winning the game. The goal of the game is to score points. Scoring points is done by disabling competitors and secondary goal is not getting disabled oneself. If the player gets disabled he can choose to re-spawn[3] in a matter of seconds and start playing again. To be successful in this type of game one has to know the world, react quickly, and recognize the necessity to make a strategic withdrawal to recharge. An interesting subproblem here is the games where only two players or bots compete against each other in much smaller arenas. In this setting one can compare the fitness of different agents easily.
- *Team Deathmatch* (TDM) is a special kind of Deathmatch where two teams compete against each other in winning the game with the same winning conditions as in Deathmatch. This is the most basic game type where team work is necessary to be successful. Protecting teammates or cooperating with them to disable competitors of the other team are examples of fundamental strategies.
- *Capture the Flag* (CTF) is a strategical type of game play. The game is played by two teams. Both teams try to hijack the flag of the other team to score points. Each flag is located in the team base. In this base the team members start playing. Scoring points is done by taking the opposing team's flag and touching the own base with it while the own flag is located there. If the own flag is not at the home base no scoring is possible and the flag has to be recaptured first. If a player is disabled while carrying the flag he drops it and if it is touched by a player of an opponent team, the flag is carried further to the opponents home base. If the flag is touched by a teammate who owns the flag it is teleported back to its base.

  To win such a game the players of a team have to cooperate, to delegate offensive or defensive tasks, and to communicate with each other. This game type is the first one which rewards strategic defense and coordinated offense maneuvers.

Note that the above game types include similar tasks. A bot being able to play Team Deathmatch has to be able to play Deathmatch just in case a one-on-one situation arises. Furthermore Capture the Flag depends on team play just like the Team Deathmatch.

3 READYLOG

READYLOG is an extension of the action language GOLOG, which in turn is based on the situation calculus. We will briefly look at all three in turn.

The situation calculus, originally proposed in [McCarthy, 1963], is a dialect of first-order logic intended to represent dynamically changing worlds. Here we use the second-order variant of the situation calculus proposed by Reiter [Reiter, 2001].[4] In this language the world is seen as a sequence of *situations* connected by actions. Starting in some initial situation called S_0, each situation is characterized by the sequence of actions that lead to it from S_0. Given a situation s, the situation which is reached after performing an action a is denoted as $do(a, s)$. Situations are described by so-called *relational* and *functional fluents*, which are logical predicates and functions, respectively, that have as their last argument a situation term.

Dynamic worlds are described by so-called *basic action theories*. From a user's point of view their main ingredients are a description of the initial situation, action precondition axioms and successor state axioms, which describe when an action is executable and what the value of a fluent is after performing an action. For example,

$$Poss(moveto(r, x), s) \equiv HasEnergy(r, s)^5$$

may be read as agent r can move to position x if it has enough energy. A simple successor-state axiom for the location of r could be

$$\begin{aligned} loc(r, do(a, s)) = x \equiv a = moveto(r, x) \vee \\ loc(r, s) = x \wedge \neg(moveto(r, y) \wedge x \neq y) \end{aligned}$$

Roughly, this says that the location of r is x after some action just in case the action was a move-r-to-x action or r was already there and did not go anywhere else. We remark that successor state axioms encode both effect and frame problems and were introduced by Reiter as a solution to the frame problem [Reiter, 2001]. Furthermore, a large class of precondition and successor state axioms can easily be implemented in Prolog, just like GOLOG which we now turn to.

GOLOG [Levesque *et al.*, 1997] is a logic programming language based on the situation calculus. GOLOG offers control structures familiar from imperative programming languages like conditionals, loops, procedures, and others:

[3]'Re-spawning' means the reappearance of a player or an item such that it becomes active again.

[4]Second-order logic is needed to define while-loops of programs, among other things.

[5]All free variables are assumed to be universally quantified.

primitive actions: α denotes a primitive action, which is equivalent to an action of the situation calculus.

sequence: $[e_1, e_2, \ldots, e_n]$ is a sequence of legal GOLOG programs e_i.

test action: $?(\phi)$ tests if the logical condition ϕ holds.

nondeterministic choice of actions: $e_1|e_2$ executes either program e_1 or program e_2.

nondeterministic choice of arguments: $pi(v, e)$ chooses a term t, substitutes it for all occurrences of v in e, and then executes e;

conditionals: $if(\phi, e_1, e_2)$ executes program e_1 if ϕ is true, otherwise e_2;

nondeterministic repetition: $star(e)$ repeats program e an arbitrary number of times;

while loops: $while(\phi, e)$ repeats program e as long as condition ϕ holds:

procedures: $proc(p, e)$ defines a procedure with name p and body e. The procedure may have parameters and recursive calls are possible.

We remark that the semantics of these constructs is fully defined within the situation calculus (see [Levesque *et al.*, 1997] for details). Given a GOLOG program, the idea is, roughly, to find a sequence of primitive actions which corresponds to a successful run of the program. These actions are then forwarded to the execution module of the agent like moving to a particular location or picking up an object.

READYLOG [Ferrein *et al.*, 2004] extends the original GOLOG in many ways. It integrates features like probabilistic actions [Grosskreutz, 2000], continuous change [Grosskreutz and Lakemeyer, 2000], on-line execution [De Giacomo and Levesque, 1998], decision-theoretic planning [Boutilier *et al.*, 2000], and concurrency [De Giacomo *et al.*, 2000; Grosskreutz, 2002]. Its primary use so far has been as the control language of soccer playing robots in the ROBOCUP middle-size league.

For the purpose of controlling UNREAL game bots, perhaps the most interesting feature of READYLOG is its decision-theoretic component. It makes use of nondeterministic as well as stochastic actions, which are used to model choices by the agent and uncertainty about the outcome of an action. To make a reasoned decision about which actions to choose, these are associated with utilities and the decision-theoretic planner computes the optimal choices (also called policies) by maximizing the accumulated expected utility wrt a (finite) horizon of future actions. This is very similar to computing optimal policies in Markov decision processes (MDPs) [Puterman, 1994].[6]

4 Modelling UNREAL in READYLOG

The UNREAL bots are described by a variety of fluents which have to be considered while playing the game. All of the fluents have a time stamp associated such that the bot is able to know how old and how precise his state information are.

Identifier fluents: In the set of identifier fluents the bots name, the currently executed skill, together with unique ids describing the bot and the skill can be found, among others.

Location fluents: The location fluents represent the bots location in a level, its current orientation, and its velocity.

Bot Parameter fluents: Health, armor, adrenaline, the currently available inventory in which the items are stored, and the explicit amount of each inventory slot is saved in this set of fluents. In the inventory additional game objective items can be found such as a flag in CTF.

Bot Visibility fluents: Here information about the objects in the view range of the agent are found. These information are distinguished in a teammate and an opponent set. They contain the bots identifier and its current location. In games without team play the set of friends stays always empty during gameplay.

Item Visibility fluents: Here the information about the currently visible and non visible items can be found. If an item is not visible at its expected position a competitor took it away and it reappears after a specific time. The definite re-spawn time of the item is unknown in general. The explicit re-spawn time is only available, if the bot itself took the item.

Bots in UNREAL TOURNAMENT 2004 are able to use the skills *stop, celebrate, moveto, roam, attack, charge, moveattack, retreat*, and *hunt*. All actions from UNREAL are modelled in the READYLOG framework as stochastic actions and successor state axioms are defined for all the fluents. Details are left out for space reasons.

Our framework is very flexible and allows for modelling different tasks in various ways, combining decision-theoretic planning with explicit programming. We begin by showing two extreme ways to specify one task of the bot, collecting health items. One relies entirely on planning, where the agent has to figure out everything by itself, and the other on programming without any freedom for the agent to choose.

The example we use for describing the different approaches, their benefits, and their disadvantages is the collection of health items in an UNREAL level. Because collecting any one of them does not have a great effect the agent should try to collect as many as possible in an optimal fashion. Optimal means that the bot takes the optimal sequence which results in minimal time and maximal effect. Several constraints like the availability have to be taken into account.

The first and perhaps most intuitive example in specifying the collection of health packs is the small excerpt from a READYLOG program shown in Program 4.1. Using decision-theoretic planning alone, the agent is able to choose in which order to move to the items based upon the reward function. The search space is reduced by only taking those navigation nodes into account which contain a health item.

Note that in this first basic example all calculations are up to the agent. Information about availability of items, the distance or the time the agent has to invest to get to the item

[6]What makes READYLOG different from ordinary MDPs is that the state space can be represented in a compact (logical) form and the control structure of a program allow a user to drastically reduce the search space.

Program 4.1 READYLOG program to collect health powerups by setting up an MDP to solve. We scale down the search space by regarding the health nodes only. The reward function rewards low time usage and higher bot health.

```
  ...
  ?(getNavNodeHealthList(      HealthNodeList    ) ),
  solve(  while(  true,
                  pickBest(  healthnode,   HealthNodeList,
                             moveto(  epf_BotID,   healthnode   ) ) ),
          Horizon,   f_HealthReward    ),
  ...

function(   f_HealthReward,    Reward,
            and(  [   CurrentTime    = start,
                      TmpReward   = epf_BotHealth    - CurrentTime,
                      Reward   = max(   [TmpReward,    0]  ) ] )
        ). % of  simple_reward
```

become available to the agent as effects of the *moveto* action. While easy to formulate, the problem of Program 4.1 is its execution time. With increasing horizon the computation time increases exponentially in the size of the horizon. All combinations of visiting the nodes are generated and all stochastic outcomes are evaluated. For example, in a setting with $Horizon = 3$ and $\#HealthNodes = 7$ the calculation of the optimal path from a specific position takes about 50 seconds,[7] which makes this program infeasible at present.

The next idea in modelling the health collection is to further restrict the search by using only a subset of all available health nodes. The example shown previously took all health navigation nodes of the whole map into account, whereas a restriction of those nodes is reasonable. Items which are far away are not of interest to the agent. Because of this restriction the real-time demands are fulfilled in a better way but they are still not acceptable for the UNREAL domain. In the same setting as above ($Horizon = 3$ and $\#HealthNodes = 7$ from which only $Horizon + 1 = 4$ health navigation nodes are chosen) the calculation of the optimal path lasts about 8 seconds.

A much more efficient way to implement this action sequencing for arbitrary types is to program the search explicitly and not to use the underlying optimization framework. For example, filtering the available nodes and ordering them afterwards in an optimal way by hand is a much better way to perform on-line playing. The example described above is depicted in Program 4.2.

This example of how modelling health collection can be done is far from optimal from a decision-theoretic point of view. There are no backup actions available if something goes wrong and no projection of outcomes is applied during execution. On the other hand, the execution of Program 4.2 is computationally inexpensive. Arbitrary horizons for collecting an item can be given to the program without an exponential blow-up.

Given the pros and cons of the two examples above, it seems worthwhile to look for a middle ground. The idea is to allow for some planning besides programmed actions and to further abstract the domain so that the search space becomes more manageable. Instead of modelling each detail for every

[7]The experiments were carried out on a Pentium 4 PC with 1.7GHz and 1GB main memory.

Program 4.2 READYLOG program to collect health items which is able to be applied on-line. The method getNextVisNavNodes returns a list of navigation nodes with length Horizon which are of type Type ordered with increasing distance from location Loc and a minimal confidence of availability of either 0.9 or 0.5. The ordering is done by the underlying Prolog predicate sort. If one item matches the mentioned requirements, the agent travels there, and recursively calls the collect method again until Horizon is reached.

```
proc(  collect(   Type,  Horizon   ),
       if(  neg(  Horizon   = 0 ),
            [
               ?(  and(  [   Loc  = epf_BotLocation,
                             getNextVisNavNodes(      Loc,  Horizon,   Type,
                                                      0.9,  TmpVisList    ),
                             lif(  TmpVisList    = [],
                                   getNextVisNavNodes(      Loc,  Horizon,    Type,
                                                            0.5,  VisList    ),
                                   VisList    = TmpVisList    ),
                             lif(  neg(  VisList    = [] ),
                                   VisList    = [HeadElement|_TailElements],
                                   HeadElement     = nothing    ),
                             NewHorizon     = Horizon    - 1
                         ] ) ),
               if(  neg(  VisList    = [] ),
                    [  moveto(   epf_BotID,    HeadElement     ),
                       collect(   Type,  NewHorizon     )
                    ] )
            ] )
       ). % of  collect(   Type,   Horizon    )
```

action simpler models are introduced which do not need that much computational effort when planning.

To illustrate this we use an excerpt from our actual implementation of the *deathmatch agent* (Program 4.3). Here an agent was programmed which chooses at each action choice point between the outcomes of a finite set of actions. It has the choice between collecting a weapon, retreating to a health item, and so on based on a given reward function. The main part of the agent is the non-deterministic choice which represents the action the agent performs next. It has the choice between roaming and collecting items, attacking an opponent, or collecting several specific items. The decision which action to take next is performed based on the reward of the resulting state. Note also that the non-deterministic choices are restricted by suitable conditions attached to each choice. This way many choices can be ruled out right away, which helps prune the search space considerably.

5 Experimental Results

In our implementation we connected READYLOG and UNREAL via a TCP connection for each game bot. With this connection the programs transmit all information about the world asynchronously to provide the game-bot with the latest world information and receive the action which the bot shall perform next until a new action is received. With this setup and after implementing an agent to play different styles of play, we conducted several experiments.

The most important thing to be mentioned before attending to the explicit results is that the game is highly influenced by luck. Letting two original UNREAL bots compete in the game can result in a balanced game which is interesting to observe or in an unbalanced game where one bot is much more lucky than the others and wins unchallenged with healthy margin. Because of that we did run every test several times to substan-

Program 4.3 Part of READYLOG program implementing an agent which is able to play Deathmatch games in UNREAL. The agent has several choices available and projects to choose the best action to execute. The results of this agent are presented in table 1.

```
proc( agent_dm( Horizon ),
   [ while( true,
         [ solve( [ nondet([if( f_SawOpponent = false,
                                   roam( epf_BotID ) ),
                              if( f_SawOpponent = true,
                                   moveattack( epf_BotID,
                                        f_GetNextOppBot ) ),
                              .....
                              if( f_ItemTypeAvailable( health ),
                                 collect( health, 1 ) ),
                              if(and([f_BotHasGoodWeapon = false,
                                f_ItemTypeAvailable(weapon) = true
                                       ] ),
                                 collect( weapon, 1 ) )
                              ] )
                    ], Horizon, f_DMReward ),
              exogf_Update
            ] )
   ] ). % of agent_dm( Horizon )

function( f_DMReward, Reward,
        and([.....
             lif( epf_BotHealth < 150, RewardHealth1 = -1 ),
             lif( epf_BotHealth < 100, RewardHealth2 = -5 ),
             .....
             lif( epf_BotArmor > 135, RewardArmor4 = 20 ),
             .....
             RewardScore = -200*(CurrentMaxScore-MyScore),
             .....
             Reward = RewardHealth1 + RewardHealth2 + . + RewardScore
            ] ) ). % of f_DMReward
```

Table 1: UNREAL deathmatch results generated in our framework. The setting was as follows: GoalScore = 9, LevelTime = 6 min, SkillLevel = skilled. We present the median result of five experiments for each entry here.

Level Name	#Player	RB only	RB vs. UB
Training Day	2	9:6	8 : 9
Albatross	2	9:5	8 : 9
Albatross	4	9:8:5:3	8:1 : 9:5
Crash	2	8:7	7 : 8
Crash	4	9:7:5:3	8:5 : 9:6

tiate our results.

Table 1 shows the results of the deathmatch agent which we described in the last section. In this and the next table the first column contains the name of the level we used for testing. The second column shows the total number of players competing in this level. In the following columns the results of different settings of the game are represented. The token UB stands for the original UNREAL bot. RB represents the READYLOG bot. "RB only" means that only READYLOG bots competed. "RB vs. UB" means that the READYLOG bots compete against the UNREAL bots.

Next we consider the capture-the-flag agent, which was implemented based on the team deathmatch agent. Here we focused on the implementation of a multi-agent strategy to be able to play Capture the Flag on an acceptable level.

We introduced two roles to implement a strategy for this type of game which we called *attacker* and *defender*. The attacker's task is to try to catch the opponents flag and to hinder the opponents from building up their game. The defender's task is to stay in the near vicinity of the own flag and to guard it. If the own flag is stolen its job is to retrieve it as fast as possible.

Table 2: UNREAL Capture the Flag results generated in our framework. The setting was as follows: GoalScore = 5, LevelTime = 6 min, SkillLevel = skilled. We present here the median result of five experiments for each entry.

Level Name	#Players	RB only	RB vs. UB	Mixed
Joust	2	5:3	5:3	-
Maul	4	1:0	0:1	2:1
Face Classic	6	2:1	0:1	2:1

Each role was implemented based on a simple set of rules based on the state of each team's flag. The two flags can each be in three states, *at home, carried*, or *dropped*. For each of the resulting nine combinations of the two flags we implemented a small program for each role. E.g. if the own flag is in the state dropped the defender's task is to recapture it by touching the flag.

For several states we introduced nondeterministic choices for the agent. It is able to choose between collecting several items or trying to do its role-related tasks.

The results can be interpreted as follows: In the one-on-one level *Joust* the READYLOG bot is surprisingly strong in gameplay. We confirmed those results in other one-on-one levels. We think this is due to the goal directed behavior of our attacker. The agent does not care much about items and mainly fulfills its job to capture the flag and recapture the own flag.

There exist several problems which we describe here but could not attend to because of time constraints. First of all the bots always choose the same paths in the map. This is not a big problem in games against UNREAL bots but humans observe and learn this behavior fast and are able to use this to their advantage.

6 Conclusions

We implemented a framework which enables us to control UNREAL game bots using the logic-based action language READYLOG, which enables the user to mix programmed actions decision-theoretic planning for intelligent game play. Different game types were implemented and experiments were carried out, where READYLOG bot is able to compete with the original UNREAL bot.

While our current bots can certainly be improved in many ways, perhaps the most important message of this paper is that logic-based action languages, which for the most part have only been considered in the theoretical KR community, can actually be used in challenging environments like interactive computer games.

References

[Berger, 2002] L. Berger. Scripting: Overview and Code Generation. In *AI Game Programming Wisdom*, volume 1, pages 505–510. Charles River Media, 2002.

[Boutilier *et al.*, 2000] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-Theoretic, High-Level Agent

Programming in the Situation Calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 355–362, Menlo Park, CA, USA, July 2000. AAAI Press.

[Buro, 2003] M. Buro. Real Time Strategy Games: A new AI Research Challenge. In *Proceedings of the International Joint Confercence on AI*, Acapulco, Mexico, 2003. AAAI Press.

[De Giacomo and Levesque, 1998] G. De Giacomo and H. J. Levesque. An Incremental Interpreter for High-Level Programs with Sensing. Technical report, Department of Computer Science, University of Toronto, Toronto, Canada, 1998.

[De Giacomo *et al.*, 2000] G. De Giacomo, Y. Lesperance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, 2000.

[Electronic Arts Inc., 2005] Electronic Arts Inc. http://www.ea.com/, last visited in January 2005.

[Epic Games Inc., 2005] Epic Games Inc. http://www.unrealtournament.com/, last visited in February 2005.

[Ferrein *et al.*, 2004] A. Ferrein, C. Fritz, and G. Lakemeyer. On-line decision-theoretic golog for unpredictable domains. In *Proc. of 27th German Conference on AI*, 2004.

[Fu and Houlette, 2004] D. Fu and R. Houlette. The Ultimate Guide to FSMs in Games. In *AI Game Programming Wisdom*, volume 2, pages 3–14. Charles River Media, 2004.

[Funge, 1998] J. Funge. *Making Them Behave: Cognitive Models for Computer Animation*. PhD thesis, University of Toronto, Toronto, Canada, 1998.

[Grosskreutz and Lakemeyer, 2000] H. Grosskreutz and G. Lakemeyer. cc-Golog: Towards More Realistic Logic-Based Robot Controllers. In *AAAI-00*, 2000.

[Grosskreutz, 2000] H. Grosskreutz. Probabilistic Projection and Belief Update in the pGOLOG Framework. In *CogRob-00 at ECAI-00*, 2000.

[Grosskreutz, 2002] H. Grosskreutz. *Towards More Realistic Logic-Based Robot Controllers in the GOLOG Framework*. PhD thesis, RWTH Aachen University, Knowledge-based Systems Group, Aachen, Germany, 2002.

[Kaminka *et al.*, 2002] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholder, and S. Tejada. Game Bots: A Flexible Test Bed for Multiagent Research. *Communications of the ACM*, 45(2):43–45, 2002.

[Levesque *et al.*, 1997] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31:59–84, April-June 1997.

[Lewis, 1999] R. L. Lewis. Cognitive modeling, symbolic. In *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press, Cambridge, Massachusetts, USA, 1999.

[Magerko *et al.*, 2004] B. Magerko, J. E. Laird, M. Assanie, A. Kerfoot, and D. Stokes. AI Characters and Directors for Interactive Computer Games. In *Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference, San Jose, CA*. AAAI Press, 2004.

[McCarthy, 1963] J. McCarthy. Situations, Actions and Causal Laws. Technical report, Stanford University, 1963.

[Munoz-Avila and Fisher, 2004] H. Munoz-Avila and T. Fisher. Strategic Planning for Unreal Tournament Bots. In *AAAI Workshop on Challenges in Game AI*, San Jose, CA, USA, July 2004.

[Orkin, 2004] J. Orkin. Symbolic Representation of Game World State: Toward Real-Time Planning in Games. In *AAAI Workshop on Challenges in Game AI*, San Jose, CA, USA, July 2004.

[Puterman, 1994] M. Puterman. *Markov Decision Processes: Discrete Dynamic Programming*. Wiley, New York, USA, 1994.

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

[Valve Corporation, 2005] Valve Corporation. http://www.valvesoftware.com/, last visited in January 2005.