On-line Decision-Theoretic Golog for Unpredictable Domains

Alexander Ferrein and **Christian Fritz** and **Gerhard Lakemeyer** [1]

Abstract. DTGolog was proposed by Boutilier et al. as an integration of decision-theoretic (DT) planning and the programming language Golog. Advantages include the ability to handle large state spaces and to limit the search space during planning with explicit programming. Soutchanski developed a version of DTGolog, where a program is executed on-line and DT planning can be applied to parts of a program only. One of the limitations is that DT planning generally cannot be applied to programs containing sensing actions. In order to deal with robotic scenarios in unpredictable domains, where certain kinds of sensing like measuring one's own position are ubiquitous, we propose a strategy where sensing during deliberation is replaced by suitable models like computed trajectories so that DT planning remains applicable. In the paper we discuss the necessary changes to DTGolog entailed by this strategy and an application of our approach in the ROBOCUP domain.

1 Introduction

Boutilier et al (2001) proposed DTGolog, an integration of Markov Decision Processes (MDPs) [12] and the programming language Golog [8], which is based on Reiter's variant of the situation calculus [13]. Golog is equipped with familiar control structures like sequence and while-loops, but also nondeterminism, which allow for complex combinations of actions operating on fluents (predicates and functions changing over time). DTGolog extends Golog by adding familiar MDP notions like stochastic actions and rewards. Moreover, decision-theoretic planning is incorporated in the form of an MDP-style optimization method, which takes a program ρ and computes a policy (another program), which follows the controls of ρ except that it chooses among nondeterministic actions in order to maximize expected utility up to a given horizon of actions. The advantage over traditional MDP's is that the state space need not be represented explicitly and that the search space can be narrowed effectively by Golog's control structures.

One serious limitation of DTGolog is that it does not account for sensing actions.[2] The reason for this limitation is that DTGolog operates in an off-line modus, that is, it computes a policy for the whole program, which is then handed to an execution module. When the program contains actions sensing fluents that can take on a large, perhaps infinite number of values, finding a policy quickly becomes infeasible, if not impossible. For this reason Soutchanski [15] introduced an on-line version of DTGolog, which interleaves policy optimization and execution. The main idea is that a user can specify for which parts of the program an MDP-style policy is to be computed. As an example, consider the program $optimize(\rho_1); sense(\phi)$; if ϕ then ρ_2 else ρ_3. The idea is, roughly, that first a policy is computed for the subprogram ρ_1, which is then executed, followed by an action sensing the truth value of ϕ. Finally, depending on the outcome either ρ_2 or ρ_3 is executed, both of which may themselves contain further occurrences of $optimize$.

In order to see that Soutchanski's approach is problematic for decision making in highly dynamic domains, it is useful to distinguish two very different forms of sensing, which we refer to as *active* and *passive* sensing. An example of active sensing is an automatic taxi driver asking a customer for her destination. Typically, this form of sensing happens only occasionally and should be part of the robot's control program. An example of passive sensing is keeping track of one's own position, which happens frequently, often in the order of tens of milliseconds. It would make little sense to explicitly represent such passive sensing actions in the robot's control program, for these would make up the bulk of the program and render reasoning about the program all but impossible. While Soutchanski does not say so explicitly, he clearly is concerned only with active sensing actions, as all his sensing actions are part of the control program.

In highly dynamic domains, passive sensing is ubiquitous as a robot has to constantly monitor its own position and its environment. The aim of this paper is to show how decision-theoretic planning can be adapted to account for this form of sensing. The starting point for our investigations is the work by Grosskreutz and Lakemeyer [5], who integrated passive sensing into Golog. The idea is, roughly, that when reasoning about a program (e.g. projecting its outcome) one uses *models* of how fluents like the robot's position change. (To model the movement of a robot they use simple linear functions of time to approximate the robot's trajectories.) During actual execution these models are replaced by passive sensing actions which are represented as so-called exogenous actions, which periodically update fluents like the position of the robot and which are inserted by the interpreter of the program.

Assuming we have appropriate models of how the relevant fluents change during deliberation, could we then simply adopt Soutchanski's approach or even the original DTGolog if we ignore active sensing? The answer, in short, is No. What is missing in both cases is that, after a policy has been computed, its execution must be carefully *monitored.* This is because the model of the world used during deliberation is only a rough approximation of the real world and things may very well turn out differently and may even result in aborting the current policy. For example, when a driver initiates passing a car and another vehicle suddenly appears speeding from behind, it may be advisable to let the other car pass first. Monitoring then means to compare assumptions made by the model of the world (such as com-

[1] Computer Science Department, Knowledge-Based Systems Group RWTH Aachen, D-52056 Aachen {gerhard,fritz, ferrein}@cs.rwth-aachen.de

[2] The only exception are sensing actions which are introduced by the optimizer to determine the state after a stochastic action.

puted agent trajectories) with the actual values obtained by sensing during execution. As we will see, this can be achieved by annotating the policy with appropriate information.

Given that we are motivated by robots operating in highly dynamic and unpredictable domains, deliberation and decision making should happen quickly, preferably in less than a second. For arbitrary Golog programs this clearly cannot be guaranteed.[3] Here we are concerned with control programs for robots that operate continuously over longer periods of time. In such scenarios it makes little sense to find optimal policies for the robot's actions from start to finish, since it is impossible to predict what the world will be like after even a few seconds. Instead one is content to peek into the future to plan perhaps only a handful of actions with highest utility, like passing another car. As we will demonstrate at the end of the paper, under these assumptions, efficient decision-theoretic planning is achievable and can lead to overall good performance.

Since an application like an automatic taxi driver is currently still out of reach, we have chosen robotic soccer, in particular, the ROBOCUP MIDDLE SIZE LEAGUE as a benchmark. While the environment is still fairly controlled (a fixed playing field with four mobile robots on each team), game situations are nevertheless challenging due to their dynamics and unpredictability. To keep things simple, we only consider the case of passive sensors, that is, Golog programs as supplied by a user do not contain explicit sensing actions.

In related work, Poole [11] incorporates a form of decision-theoretic planning into his independent choice logic. While he also distinguishes passive from active sensing, he does not consider the issue of on-line DT planning. Other action logics addressing uncertainty include [14], where abduction is the focus, and [6], which addresses symbolic dynamic programming and which itself is based on [1]. Finally, [7] also discuss ways of replacing sensing by models of the environment during deliberation.

The rest of the paper is organized as follows. First, we give a brief overview of DTGolog and its underlying semantics. Then we sketch out our approach to on-line decision-theoretic planning, followed by a discussion of applying decision-theoretic to ROBOCUP's MIDDLE SIZE LEAGUE and some concluding remarks.

2 The Situation Calculus and DTGolog

2.1 The Situation Calculus

Golog is based on Reiter's variant of the Situation Calculus [13, 10], a second-order language for reasoning about actions and their effects. Changes in the world are only due to actions so that a situation is completely described by the history of actions since the initial situation S_0. Properties of the world are described by *fluents*, which are predicates and functions with a situation term as their last argument. For each fluent the user defines a successor state axiom describing precisely when a fluent value changes or does not change after performing an action. These, together with precondition axioms for each action, axioms for the initial situation, and foundational axioms as well as unique names and domain closure assumption, form a so-called *basic action theory* [13].

2.2 Off-line DTGolog

DTGolog uses basic action theories to give meaning to primitive actions and it inherits Golog's programming constructs such as sequence, if-then-else, while-loops, and procedures, as well as nondeterministic actions. From MDPs DTGolog borrows the notion of *reward*, which is a real number assigned to situations indicating the desirability of reaching that situation, and *stochastic actions*. To see what is behind the latter, consider the action of intercepting a ball in robotic soccer. Such an action routinely fails and we assign a low probability (0.2) to its success. To model this in DTGolog, we define a stochastic action $intercept$. It is associated with two non-stochastic or deterministic actions $interceptS$ and $interceptF$ for a successful and failed intercept, respectively. Instead of executing $intercept$ directly, nature chooses to execute $interceptS$ with probability 0.2 and $interceptF$ with probability 0.8. The effect of $interceptS$ can be as simple as setting the robot's position to the position of the ball. The effect of $interceptF$ can be to teleport the ball to some arbitrary other position and setting the robot's position to the old ball position.[4]

While the original Golog merely looks for any sequence of primitive actions that corresponds to a successful execution of a program, DTGolog takes a program and converts it into another simplified program, called a policy, which is a tree of conditional actions. This policy, roughly, follows the advice of the original program in case of deterministic actions and settles on those choices among nondeterministic actions which maximize expected utility. The search for the right choices is very similar to the search for an optimal policy in an MDP. One advantage of using Golog compared to a regular MDP is that the search can be arbitrarily constrained by restricting the number of nondeterministic actions.

DTGolog is defined in terms of a macro $BestDo(p, s, h, \pi, v, pr)$, which ultimately translates into a situation calculus expression. Given a program p and a starting situation s, $BestDo$ computes a policy π with expected utility v and probability pr for a successful execution. h denotes a finite horizon, which provides a bound on the maximal depth of any branch in the policy. For space reasons we only consider the definition of $BestDo$ for nondeterministic choice and stochastic actions. (See [2] for more details.)

Suppose a program starts with a nondeterministic choice between two programs p_1 and p_2, written as $(p_1|p_2)$. Then

$$\begin{aligned}&BestDo((p_1|p_2); p, s, h, \pi, v, pr) \overset{def}{=}\\ &\quad \exists \pi_1, v_1, pr_1.BestDo(p_1; p, s, h, \pi_1, v_1, pr_1) \wedge\\ &\quad \exists \pi_2, v_2, pr_2.BestDo(p_2; p, s, h, \pi_2, v_2, pr_2) \wedge\\ &\quad ((v_1, p_1) \geq (v_2, p_2) \wedge \pi = \pi_1 \wedge pr = pr_1 \wedge v = v_1) \vee\\ &\quad (v_1, p_1) < (v_2, p_2) \wedge \pi = \pi_2 \wedge pr = pr_2 \wedge v = v_2)\end{aligned}$$

Here $BestDo$ commits the policy to the best choice among the two alternatives, where "best" is defined in terms of a multi-objective optimization of expected value and success probability. See [2] for an example of how $(v_i, p_i) \geq (v_j, p_j)$ can be defined.

Now suppose that a is a stochastic action with nature's choices $n_1, n_2, \ldots, n_k$.

$$\begin{aligned}&BestDo(a; p, s, h, \pi, v, pr) \overset{def}{=}\\ &\quad \exists \pi'.BestDoAux(\{n_1, \ldots, n_k\}, p, s, h, \pi', v, pr)\\ &\quad \pi = a; senseEffect(a); \pi'.\end{aligned}$$

Here the policy is $a; senseEffect(a); \pi'$ where π' is computed by $BestDoAux$ below. The action $senseEffect(a)$ is inserted in order to maintain the MDP assumption of full observability. Its job

[3] In the coffee-delivery example in [2], the robot needed several seconds or even minutes to find a policy.

[4] While this model is certainly simplistic, it suffices in most real game situations, since all that matters is that the ball is not in the robot's possession after a failed intercept.

is to make sure that after performing a the robot gathers enough information to distinguish between the outcomes n_i. In the case of $intercept$, the sensing would involve finding out whether the robot has the ball denoted by the fluent $haveBall(s)$.

$$
\begin{aligned}
&BestDoAux(\{n_1,\ldots,n_k\},p,s,h,\pi,v,pr) \stackrel{def}{=}\\
&\neg Poss(n_1,s) \;\wedge\; BestDoAux(\{n_2,\ldots,n_k\},p,s,h,\pi,v,pr) \;\vee\\
&Poss(n_1,s) \wedge\\
&\exists \pi',v',pr'.BestDoAux(\{n_2,\ldots,n_k\},p,s,h,\pi',v',pr') \wedge\\
&\exists \pi_1,v_1,pr_1.BestDo(n_1,do(n_1,s),h-1,\pi_1,v_1,pr_1) \wedge\\
&\pi = if(\varphi_1,\pi_1,\pi') \wedge\\
&v = v' + v_1 \cdot prob(n_1,s) \wedge\\
&pr = pr' + pr_1 \cdot prob(n_1,s)
\end{aligned}
$$

$$BestDoAux(\{\},p,s,h,\pi,v,pr) \stackrel{def}{=} \pi = Stop \wedge v = 0 \wedge pr = 0$$

Note that $BestDoAux$ produces a policy of the form $if(\varphi_1,\pi_1,if(\varphi_2,\pi_1,\ldots))$ accounting for all outcomes of nature's choices. The φ_i are user-defined tests which allow the robot to distinguish between them. In our intercept example, these could be $haveBall(s)$ for $interceptS$ and $\neg haveBall(s)$ for $interceptF$. A policy contains $Stop$ if in the respective branch no further actions can be executed, i.e. an action was not possible.

2.3 On-line DTGolog

The original version of DTGolog, which we just described, operates in an off-line modus, that is, it first computes a policy for the whole program and only then initiates execution. As was observed already in [4], this is not practical for large programs and certainly not for applications with tight real-time constraints such as ROBOCUP. In the extreme one would only want to reason about the next action of a program, execute it and then continue with the rest of the program. This is the basic idea of an on-line interpretation of a Golog program [4]. To make this work, a so-called transition semantics is needed, which takes a configuration consisting of a program and a situation and turns it into another configuration. Formally, one introduces a predicate $Trans(\delta,s,\delta',s')$, which first appeared in [3], expressing a possible transition of program δ in situation s to the program δ' leading to situation s' by performing an action. For space reasons we only consider the case of while-loops.

$$
\begin{aligned}
&Trans(while(\varphi,p),s,\delta',s') \equiv\\
&\qquad \exists \delta''.Trans(p,s,\delta'',s') \wedge \varphi[s] \wedge \delta' = \delta''; while(\varphi,p)^{5}
\end{aligned}
$$

Given such definitions for all constructs, the execution of a complete program can be defined in terms of the reflexive and transitive closure of $Trans$.[6]

A nice feature of on-line interpretation is that the step-wise execution of a program can easily be interleaved with other exogenous actions or events, which are supplied from outside. This is how we handle periodic sensor updates for position estimation, for example. (See [5] for details of how this can be done in Golog.)

With the basic transition mechanism in hand, it is, in principle, not hard to reintroduce off-line reasoning for parts of the program. In the case of DTGolog, Soutchanski proposed for that purpose an interleaving of off-line planning and on-line execution. We show an excerpt of his interpreter implemented in Prolog. We only consider the case of executing deterministic and sensing actions, leaving out stochastic actions:

```
online(E,S,H,Pol,U) :-
  incrBestDo(E, S, ER, H, Pol1, U1, Prob1),
  ( final(ER, S, H, Pol1, U1), Pol=Pol1, U=U1  ;
    reward(R, S), Pol1 = (A : Rest),
      %% deterministic action
    ( agentAction(A), doReally(A), !,
      online(ER, do(A,S), H, PolFut, UFut),
      Pol = (A : PolFut), U is R + UFut ;
      %% sensing action
      senseAction(A), doReally(A), !,
      online(ER, do(A,S), H, PolFut, UFut),
      Pol=(A: PolFut), U is R + UFut ;
      ...
    )
  ).
```

Roughly, the interpreter $online$ calculates a policy π for a given program e up to a given horizon h, executes its first action ($doReally(a)$) and recursively calls the interpreter with the remaining program again.

To control the search while optimizing Soutchanski proposes an operator $optimize$ defined by the following macro:

$$
\begin{aligned}
&IncrBestDo(optimize(p_1); p_2, s, p_r, h, \pi, u, pr) \stackrel{def}{=}\\
&\qquad \exists p'.IncrBestDo(p_1; Nil, s, p', h, \pi, u, pr) \wedge\\
&\qquad (p' \neq Nil \wedge p_r = (optimize(p'); p_2) \vee\\
&\qquad p' = Nil \wedge p_r = p_2).
\end{aligned}
$$

This has the effect that p_1 is optimized and the resulting policy is executed before p_2 is even considered. As mentioned already in the introduction, one advantage is that a user can deal with explicit (active) sensing actions by restricting $optimize$ to never go beyond the next sensing action.

Nevertheless the approach has a number of shortcomings. First note that, in the definition of the interpreter $online$, after executing only one action of a computed policy, the optimizer is called again. This means that large parts of the program are re-optimized over and over again, which is computationally too expensive for real-time decision making. Also note that it is only checked during the optimization phase whether an action is executable. Hence the interpreter ignores the possibility that an action, which was possible at planning time, becomes impossible to execute due to changes in the environment.

While there are easy fixes to these drawbacks, the fundamental problem of this approach is that it is not possible to do optimization ahead of sensing actions. Consider the program $p_1; sense(\varphi); if\ \varphi\ then\ p_2\ else\ p_3$. The condition must be evaluated before one is able to decide whether to execute p_2 or p_3, i.e. the sensing action must be executed online to evaluate the value of φ. The forementioned operator $optimize$ allows for limiting the search for an optimal policy for that case. Using $optimze$ the program can be rewritten as $optimize(p_1)$; $(sense(\varphi))$; $if\ \varphi\ then\ p_2\ else\ p_3$. This program instructs the interpreter to first optimize p_1 without regarding the rest of the program. Then, the sensing action is executed to get the needed value from the environment. Afterwards, the conditional can be optimized and executed, resp.

In real-time domains sensor updates arrive with high frequency. The proposed active sensing in Soutchanski's interpreter is not feasible as is renders the applicability of the decision-theoretic planning approach impossible.

We propose a different kind of online interpreting decision-theoretic plans in Golog which is feasible for real-time domains. Our approach differs mainly in that we use passive sensing instead of the active sensing proposed by Soutchanski. We therefore do not have

[5] $\varphi[s]$ denotes the situation calculus formula obtained from φ by restoring situation variable s as the suppressed situation argument for all fluent names mentioned in φ. Also note that free variables are universally quantified in the following formulas.

[6] One also needs the notion of a final configuration, an issue we ignore here for simplicity.

any restrictions with sensing actions. Instead we use models of the world in the planning phase. To be able to validate if the model assumptions hold while executing a plan we annotate the policies in a special way desribed in Section 3.1. In Section 3.2 we show how annotated policies are executed and how invalid policies are detected. We deployed our approach in the RoboCup domain and show some of the results in Section 4.

3 On-line DTGolog for Passive Sensing

As the re-optimization of a remaining program is generally not feasible in real-time environments, our first modification of on-line DTGolog is to make sure that the whole policy and not just the first action is executed. For this purpose we introduce the following operator $solve(p, h)$ for a program p and a fixed horizon h.

$$\begin{aligned} &\mathit{Trans}(solve(p,h), s, \delta', s') \equiv \\ &\quad \exists \pi, v, pr.\mathit{BestDo}(p, s, h, \pi, v, pr) \\ &\quad \wedge \delta' = \mathit{applyPol}(\pi) \wedge s' = s. \end{aligned}$$

The predicate BestDo first calculates the policy for the whole program p. For now the reader may assume the definition of the previous section, but we will see below that it needs to be modified. This policy is then scheduled for execution as the remaining program. However, as discussed in the introduction, the policy is generated using an abstract model of the world to avoid sensing, and we need to monitor whether π remains valid during execution. To allow for this special treatment, we use the special construct $\mathit{applyPol}$, whose definition is deferred until later.

3.1 Annotated Policies

In order to see why we need to modify the original definition of BestDo and, for that matter, the one used by Soutchanski, we need to consider, in a little more detail, the idea of using a model of the world when planning vs. using sensor data during execution. The following fragment of the control program of our soccer robots might help to illustrate the problem:

```
while $game\_on$ do ...;
    $solve(...;$
        if $\exists x, y(ball\_pos(x, y) \wedge reachable(x, y))$
        then $intercept$
        else ...; ... , $h)$
endwhile
```

While the game is still on, the robots execute a loop where they determine an optimal policy for the next few (typically less than six) actions, execute the policy and then continue the loop. One of the choices is intercepting the ball which requires that the ball is reachable, which can be defined as a clear trajectory between the robot and the ball. Now suppose BestDo determines that the if-condition is true and that $intercept$ has the highest utility. In that case, since $intercept$ is a stochastic action, the resulting policy π contains $\ldots intercept; senseEffect(intercept); \ldots$. Note, in particular, that the if-condition of the original program is not part of the policy. And this is where the problem lies. For during execution of the policy it may well be the case that the ball is no longer reachable because an opponent is blocking the way. In that case $intercept$ will fail and it makes sense to abort the policy and start planning for the next moves. For that, the if-condition should be re-evaluated using the most up-to-date information about the world provided by the sensors and compared to the old value. Hence we need to make sure that the if-condition and the old truth value are remembered in the policy.

In general, this means we need to modify the definition of BestDo for those cases involving the evaluation of logical formulas. Here we consider if-then-else and test actions. While-loops are treated in a similar way.

$$\begin{aligned} &\mathit{BestDo}(if(\varphi, p_1, p_2); p, s, h, \pi, v, pr) \overset{def}{=} \\ &\quad \varphi[s] \wedge \exists \pi_1.\mathit{BestDo}(p_1; p, s, h, \pi_1, pr) \wedge \\ &\quad \pi = \mathfrak{M}(\varphi, true); \pi_1 \vee \\ &\quad \neg\varphi[s] \wedge \exists \pi_2.\mathit{BestDo}(p_2; p, s, h, \pi_2, v, pr) \wedge \\ &\quad \pi = \mathfrak{M}(\varphi, false); \pi_2 \end{aligned}$$

The only difference compared to the original BestDo is that we prefix the generated policy with a marker $\mathfrak{M}(\varphi, true)$ in case the φ turned out to be true in s and $\mathfrak{M}(\varphi, false)$ if it is false. The treatment of a test action $?(\varphi)$ is even simpler, since only the case where φ is true matters. If φ is false, the current branch of the policy is terminated, which is indicated by the Stop action.

$$\begin{aligned} &\mathit{BestDo}(?(\varphi); p, s, h, \pi, v, pr) \overset{def}{=} \\ &\quad \varphi[s] \wedge \exists \pi'.\mathit{BestDo}(p, s, h, \pi', v, pr) \wedge \\ &\quad \pi = \mathfrak{M}(\varphi, true); \pi' \vee \\ &\quad \neg\varphi[s] \wedge \pi = \mathit{Stop} \wedge pr = 0 \wedge v = \mathit{reward}(s) \end{aligned}$$

In the next subsection, we will see how our annotations will allow us to check at execution time whether the truth value of conditions in the program at planning time are still the same and what to do about it when they are not. Before that, however, it should be mentioned that explicit tests are not the only reason for a possible mismatch between planning and execution. To see that note that when a primitive action is entered into a policy, its executability has been determined by BestDo. Of course, it could happen that the same action is no longer possible at execution time. It turns out that this case can be handled without any special annotation.

3.2 Execution and Monitoring

Now that we have modified BestDo so that we can discover problems at execution time, all that is left to do is to define the actual execution of a policy. Given our initial definition of $\mathit{Trans}(solve(p, h), s, \delta', s')$, this means that we need to define Trans for the different cases of $\mathit{applyPol}(\pi)$. To keep the definitions simple, let us assume that every branch of a policy ends with Stop or nil, where nil represents the empty program.

$$\begin{aligned} \mathit{Trans}(\mathit{applyPol}(Nil), s, \delta', s') &\equiv s = s' \wedge \delta' = nil \\ \mathit{Trans}(\mathit{applyPol}(\mathit{Stop}), s, \delta', s') &\equiv s = s' \wedge \delta' = nil \end{aligned}$$

Given the fact that configurations with nil as the program are always final, that is, execution may legally terminate, this simply means that nothing needs to be done after Stop or nil.

In case a marker was inserted into the policy we have to check the test performed at planning time still yields the same result. If this is the case we are happy and continue executing the policy, that is, $\mathit{applyPol}$ remains in effect in the successor configuration. But what should we do if the test turns out different? We have chosen to simply abort the policy, that is, the successor configuration has nil as its program. While this may seem simplistic, it seems the right approach for applications like ROBOCUP. For consider the case of an intercept. If we find out that the path is blocked, the intercept will likely fail and all subsequent actions in the policy become meaningless. Moreover, a quick abort will enable immediate replanning according to the control program, which is not a bad idea under the circumstances.

$$\begin{aligned}
&Trans(applyPol(\mathfrak{M}(\varphi, v); \pi), s, \delta', s') \equiv s = s' \wedge \\
&\quad (v = true \wedge \varphi[s] \wedge \delta' = applyPol(\pi) \vee \\
&\quad v = false \wedge \neg\varphi[s] \wedge \delta' = applyPol(\pi) \vee \\
&\quad v = true \wedge \neg\varphi[s] \wedge \delta' = nil \vee \\
&\quad v = false \wedge \varphi[s] \wedge \delta' = nil)
\end{aligned}$$

If the next construct in the policy is a primitive action other than a stochastic action or a $senseEffect$, then we execute the action and continue executing the rest of the policy. As discussed above, due to changes in the world it may be the case that a has become impossible to execute. In this case we again abort the rest of the policy with the successor configuration $\langle nil, s\rangle$.

$$\begin{aligned}
&Trans(applyPol(a; \pi), s, \delta', s') \equiv \\
&\quad \exists \delta''. Trans(a; \pi, s, \delta'', s') \wedge \delta' = applyPol(\delta'') \vee \\
&\quad \neg Poss(a[s], s) \wedge \delta' = nil \wedge s' = s
\end{aligned}$$

If a is a stochastic action, we obtain

$$\begin{aligned}
&Trans(applyPol(a; senseEffect(a); \pi), s, \delta', s') \equiv \\
&\quad \exists \delta''. Trans(senseEffect(a); \pi, s, \delta'', s') \wedge \\
&\quad \delta' = applyPol(\delta''))
\end{aligned}$$

Note the subtlety that a is ignored by $Trans$. This has to do with the fact that stochastic actions have no direct effects according to the way they are modeled in DTGolog. Instead one needs to perform $senseEffect$ to find out about the actual effects. Of course, even though $Trans$ ignores a, care must be taken by the implementation that it is executed in the real world.[7] As in the original DTGolog we also assume that $senseEffect$ actions are always executable.

Finally, if we encounter an if-construct, which was inserted into the policy due to a stochastic action, we determine which branch of the policy to choose and go on with the execution of that branch.

$$\begin{aligned}
&Trans(applyPol(if(\varphi, \pi_1, \pi_2)), s, \delta', s') \equiv \\
&\quad \varphi[s] \wedge Trans(applyPol(\pi_1), s, \delta', s') \vee \\
&\quad \neg\varphi[s] \wedge Trans(applyPol(\pi_2), s, \delta', s')
\end{aligned}$$

We end this section with a few notes about the implementation of our on-line decision-theoretic interpreter called Readylog.[8]

We begin with a (very) rough sketch of the main loop of the interpreter.

```
/******** Interpreter mainloop ******/
/* (1)- exogenous action occured */
icpgo(E,H) :- exog_occurs(Act,H), exog_action(Act),!,
              icpgo(E,[Act|H]).

/* (2) - performing a step in program execution */
icpgo(E,H) :- trans(E,H,E1,H1),icpxeq(H,H1,H2),!,
              icpgo(E1,H2).

/* (3) - program is final -> execution finished */
icpgo(E,H) :- final(E,H).
```

[7] This can be done similar to Soutchanski's interpreter by inserting an appropriate $doReally(a)$ literal (see Section 2.3 or $icpxeq$ in our case (see above)).

[8] Readylog stands for "real-time dynamic Golog."

```
solve(nondet(
 [kick(ownNumber, 40),
  dribble_or_move_kick(ownNumber),
  dribble_to_points(ownNumber),
  if(isKickable(ownNumber),
     pickBest(var_turnAngle, [-3.1, -2.3, 2.3, 3.1],
       [turn_relative(ownNumber, var_turnAngle, 2),
        nondet([[intercept_ball(ownNumber, 1),
           dribble_or_move_kick(ownNumber)],
           [intercept_ball(no_ByRole(supporter), 1),
            dribble_or_move_kick(no_ByRole(supp.))]])]),
     nondet([[intercept_ball(ownNumber, 1),
           dribble_or_move_kick(ownNumber)],
           intercept_ball(ownNumber, 0.0, 1)]) ) ]), 4)
```

Figure 1. The *bestInterceptor* program performed by an offensive player. Here, $nondet(\Sigma)$ denotes the nondeterministic choice of actions.

```
/* (4) - waiting for an exogenous action to happen */
icpgo(E,H) :- wait_for_exog_occurs, !, icpgo(E,H).

/******** Executing actions ******/
/* (1) - No action was performed so
                             we don't execute anything */
icpxeq(H,H,H).

/* (2) - The action is not a sensing one:
                          exec. and ignore its sensing */
icpxeq(H,[Act|H],H1) :- not senses(Act,_),
                           execute(Act,_,H), H1=[Act|H]).

/* (3) - The action is a sensing one for F:
                                 execute sensing action*/
icpxeq(H,[Act|H],H1):-senses(Act,F),
                    execute(Act,Sr,H),H1=[e(F,Sr),Act|H].
```

First, it checks whether an exogenous event occurred and if so inserts it into the history. Next, it is checked if a transition to a new configuration can be made executing the next possible action. If there is no successor configuration reachable a test for a final configuration is conducted. In case (4) where none of the previous cases apply the interpreter waits until some exogenous event occurs, e.g. the robot has reaches a certain position.

For the execution the predicate $icpxeq$ exists checking whether the action to be executed is a sensing action or not. This is very similar to Soutchanski's $doReally$. Note that we still allow for passive sensing actions we only avoid them during plan generation.

We put a lot of effort into tuning the perfomance of the interpreter. One major speed-up was achieved by integrating a progression mechanism for the internal database in the spirit of Lin and Reiter [9] (step 5 in the loop). For space reasons we leave out all details except to say that this is indispensable for maintaining tractability because the action history would otherwise grow beyond control very quickly.

Additional speed-ups were obtained by using a Readylog preprocessor. It takes a complete domain axiomatization as input and generates optimized Prolog code, i.e. run-time invariants like static conditions are evaluated at compile-time to save the time of evaluating them many times at run-time.

4 Empirical Results in RoboCup

We used the described version of Readylog for our ROBOCUP MIDDLE SIZE robot team at the world championships 2003 in Padua, Italy, and at the German Open 2004 in Paderborn. In this Section we show some details of the implementation of our soccer agent and present some results of the use of decision-theoretic planning in Golog in the soccer domain.

Among the basic actions we used the most important were $goto\_pos(x, y, \theta)$, $turn(\theta)$, $dribble\_to(x, y, \theta)$, $intercept$, $kick(power)$, and $move\_and\_kick(x, y, \theta)$.

While the goalie was controlled without Readylog in order to maintain the highest possible level of reactivity, all other players of our team had an individual Readylog procedure for playing. We assigned fixed roles to the three field players: defender, supporter, and

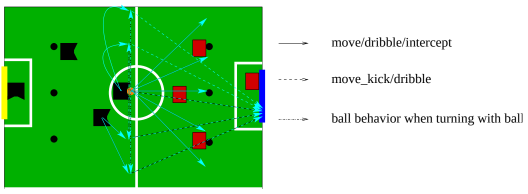

Figure 2. The set of alternatives for the attacker when it is in ball possession. The red boxes denote opponents, the black ones are teammates. Everything else are fi eld markings.

attacker. Only the best positioned player to the ball started to deliberate, i.e. solved the MDP given by the program in Fig. 1, otherwise it performed a program according to its role like defending the team's goal.

The set of alternatives made up from the *bestInterceptor* program is best described by Figure 2. In line 3 of Fig. 1 the choice is between a dribbling to the free goal corner or to dribble thereto but finishing the action with a shot as soon as the goal is straight ahead. In line 6 the agent decides among four angles to turn to in order to push the ball to either side where it can be intercepted by a teammate, using the $pickBest$ construct.

Figure 3 shows an example decision tree made up from this program. For readability we pruned some similar branches. The root node stands for the situation were the agent switched to off-line mode, i.e., the current situation. The boxes symbolize agent choices, i.e. the agent can decide which of the alternatives to take. The circles are nature's choices, denoting the possible outcomes of stochastic actions. Numbers of outgoing edges in these nodes are the probabilities for the possible outcomes. The numbers in the boxes are the rewards for the corresponding situation. The actually best policy in the situation of the example is marked by a thick line.

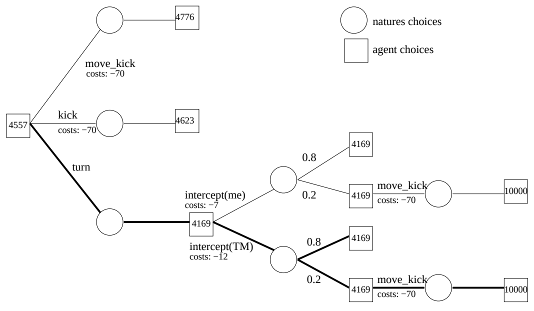

Figure 3. A (pruned) example decision tree for the *bestInterceptor* program.

Indispensable for a successful acting agent using decision-theoretic planning is a reasonable reward function. For this scenario, however, we used a rather primitive reward function based solely on the velocity, relative position and distance of the ball towards the opponents goal. In future work this could be refined for improving the overall play.

Naturally, the time a player spent deliberating depended highly on the number of alternatives that were possible. In this respect, whether or not the ball was kickable made the most difference (all times in seconds):

	examples	min	avg	max
without ball	698	< 0.01	0.094	0.450
with ball	117	0.170	0.536	2.110

The hardware used was an on-board Pentium III-933.

With the described decision making method we shot 13 goals at the world championships and scratched the final round by one goal. In the end we placed 10th out of 24 and ended 5th (or 7th depending on ranking algorithm) out of 13.

5 Conclusion

In this paper, we proposed a novel method of on-line decision-theoretic planning and execution in Golog, which is particularly suited for robotic applications with frequent sensor updates like measuring one's own position and that of other agents and objects. We overcame the problem of Soutchanski's approach, which cannot plan past the next sensing action, by eliminating explicit sensor updates of the above kind from a robot control program altogether and replacing them instead with models that allow the approximate calculation of otherwise sensed values during planning. However, this also required annotating policies with information about the models so that discrepancies with the real world could be detected while executing the policy. Our approach was applied in the ROBOCUP domain with encouraging results.

One weakness of the current implementation is that rewards are assigned manually in a rather ad-hoc manner. In the future we hope to employ learning methods to improve the overall performance.

REFERENCES

[1] C. Boutilier, R. Reiter, and B. Price, 'Symbolic dynamic programming for fi rst-order MDPs', in *IJCAI*, pp. 690–700, (2001).
[2] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun, 'Decision-theoretic, high-level agent programming in the situation calculus', in *Proc. of AAAI-00*, pp. 355–362. AAAI Press, (2000).
[3] G. De Giacomo, Y. Lésperance, and H. Levesque, 'ConGolog, A concurrent programming language based on situation calculus', *Artificial Intelligence*, **121**(1–2), 109–169, (2000).
[4] G. De Giacomo and H. Levesque, 'An incremental interpreter for high-level programs with sensing', in *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, eds., Hector J. Levesque and Fiora Pirri, 86–102, Springer, Berlin, (1999).
[5] Henrik G. and G. Lakemeyer, 'On-line execution of cc-Golog plans', in *Proc. of IJCAI-01*, (2001).
[6] A. Großmann, S. Hölldobler, and O. Skvortsova, 'Symbolic dynamic programming with the Fluent Calculus', in *Proceedings of the IASTED (ACI-2002)*, pp. 378–383, (2002).
[7] Y. Lespérance and H.-K. Ng, 'Integrating planning into reactive high-level robot programs', in *Proc. (CogRob-2000)*, pp. 49–54, (2000).
[8] H. Levesque, R. Reiter, Y. Lesperance, Fangzhen Lin, and Richard B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**(1-3), 59–83, (1997).
[9] F. Lin and R. Reiter, 'How to progress a database', *Artificial Intelligence*, **92**(1-2), 131–167, (1997).
[10] J. McCarthy, 'Situations, actions and causal laws', Technical report, Stanford University, (1963).
[11] D. Poole, 'The independent choice logic for modelling multiple agents under uncertainty', *Artificial Intelligence*, **94**(1-2), 7–56, (1997).
[12] M. Puterman, *Markov Decision Processes: Discrete Dynamic Programming*, Wiley, New York, 1994.
[13] R. Reiter, *Knowledge in Action*, MIT Press, 2001.
[14] M. Shanahan, 'The event calculus explained', *Lecture Notes in Computer Science*, **1600**, (1999).
[15] M. Soutchanski, 'An on-line decision-theoretic golog interpreter', in *Proc. IJCAI-2001*, Seattle, Washington, (August 2001).