Decision-Theoretic, High-level Agent Programming in the Situation Calculus

Craig Boutilier
Dept. of Computer Science
University of Toronto
Toronto, ON M5S 3H5
cebly@cs.toronto.edu

Ray Reiter
Dept. of Computer Science
University of Toronto
Toronto, ON M5S 3H5
reiter@cs.toronto.edu

Mikhail Soutchanski
Dept. of Computer Science
University of Toronto
Toronto, ON M5S 3H5
mes@cs.toronto.edu

Sebastian Thrun
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
thrun@cs.cmu.edu

Abstract

We propose a framework for robot programming which allows the seamless integration of explicit agent programming with decision-theoretic planning. Specifically, the *DTGolog* model allows one to partially specify a control program in a high-level, logical language, and provides an interpreter that, given a logical axiomatization of a domain, will determine the optimal completion of that program (viewed as a Markov decision process). We demonstrate the utility of this model with results obtained in an office delivery robotics domain.

1 Introduction

The construction of autonomous agents, such as mobile robots or software agents, is paramount in artificial intelligence, with considerable research devoted to methods that will ease the burden of designing controllers for such agents. There are two main ways in which the conceptual complexity of devising controllers can be managed. The first is to provide languages with which a programmer can specify a control program with relative ease, using high-level actions as primitives, and expressing the necessary operations in a natural way. The second is to simply specify goals (or an objective function) and provide the agent with the ability to plan appropriate courses of action that achieve those goals (or maximize the objective function). In this way the need for explicit programming is obviated.

In this paper, we propose a framework that combines both perspectives, allowing one to partially specify a controller by writing a program in a suitably high-level language, yet allowing an agent some latitude in choosing its actions, thus requiring a modicum of planning or decision-making ability. Viewed differently, we allow for the seamless integration of programming and planning. Specifically, we suppose that the agent programmer has enough knowledge of a given domain to be able to specify some (but not necessarily all) of the structure and the details of a good (or possibly optimal) controller. Those aspects left unspecified will be filled in by the agent itself, but must satisfy any constraints imposed by the program (or partially-specified controller). When controllers can easily be designed by hand, planning has no role to play. On the other hand, certain problems are more easily tackled by specifying goals and a declarative domain model, and allowing the agent to plan its behavior.

Our framework is based on the synthesis of Markov decisions processes (MDPs) [4, 13] with the Golog programming language [10]. Key to our proposal is the extension of the Golog language and interpreter, called *DTGolog*, to deal with uncertainty and general reward functions. The planning ability we provide is that of a decision-theoretic planner in which choices left to the agent are made by maximizing expected utility. Our framework can thus be motivated in two ways. First, it can be viewed as a decision-theoretic extension of the Golog language. Golog is a high-level agent programming language based on the situation calculus, with a clear semantics, and in which standard programming constructs (e.g., sequencing, nondeterministic choice) are used to write high-level control programs.

From a different standpoint, our contribution can be viewed as a language and methodology with which to provide "advice" to a decision-theoretic planner. MDPs are a conceptually and computationally useful model for decision-theoretic planning, but their solution is often intractable. We provide the means to *naturally* constrain the search for (ideally, optimal) policies with a Golog program. The agent can only adopt policies that are consistent with the execution of the program. The decision-theoretic Golog interpreter then solves the underlying MDP by making choices regarding the execution of the program through expected utility maximization. This viewpoint is fruitful when one considers that an agent's designer or "taskmaster" often has a good idea about the general structure of a good (or optimal) policy, but may be unable to commit to certain details. While we run the risk that the program may not allow for optimal behavior, this model has the clear advantage that the decision problem faced will generally be more tractable: it need only make those choices left open to it by the programmer. In contrast to existing models for constraining policies in MDPs, which use concepts such as local policies [11, 18] or finite-state machines [11], DTGolog provides a natural and well-understood formalism for *programming* behaviors.

Our approach is specifically targeted towards developing complex robotics software. Within robotics, the two major paradigms—planning and programming—have largely been pursued independently. Both approaches have their advantages (flexibility and generality in the planning paradigm, performance of programmed controllers) and scaling limitations (e.g., the computational complexity of planning approaches, task-specific design and conceptual complexity for programmers in the programming paradigm). MDP-style planning has been at the core of a range of fielded robot applications, such as two recent tour-guide robots [5, 19]. Its ability to cope with uncertain worlds is an essential feature for real-world robotic applications. However, MDP planning scales poorly to complex tasks and environments. By programming easy-to-code routines and leaving only those choices to the MDP planner that are difficult to program (e.g., because the programmer cannot easily determine appropriate or optimal behavior), the complexity of planning can be reduced tremendously. Note that such difficult-to-program behaviors may actually be quite easy to *implicitly* specify using goals or objectives.

To demonstrate the advantage of this new framework, we have developed a prototype mobile office robot that delivers mail, using a combination of pre-programmed behavior and decision-theoretic deliberation. An analysis of the relative trade-offs shows that the combination of programming and planning is essential for developing robust, scalable control software for robotic applications like the one described here.

We give brief overviews of MDPs and Golog in Sections 2 and 3. We describe the DTGolog representation of MDPs and programs and the DTGolog interpreter in Section 4, and illustrate the functioning of the interpreter by describing its implementation in a office robot in Section 5.

2 Markov Decision Processes

We begin with some basic background on MDPs (see [4, 13] for further details). We assume that we have a stochastic dynamical system to be controlled by some agent. A fully-observable MDP $M = \langle \mathcal{S}, \mathcal{A}, \Pr, R \rangle$ comprises the following components. $\mathcal{S}$ is a finite set of *states* of the system being controlled. The agent has a finite set of *actions* $\mathcal{A}$ with which to influence the system state. Dynamics are given by $\Pr : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$; here $\Pr(s_i, a, s_j)$ denotes the probability that action a, when executed at state s_i, induces a transition to s_j. $R : \mathcal{S} \rightarrow \Re$ is a real-valued, bounded *reward function*. The process is fully observable: though the agent cannot predict the outcome of an action with certainty, it can observe that state precisely once it is reached.

The decision problem faced by the agent in an MDP is that of forming an *optimal policy* (a mapping from states to actions) that maximizes expected total accumulated reward over some horizon of interest. An agent finding itself in state s^t at time t must choose an action a^t. The *expected value* of a course of action π depends on the specific objectives. A *finite-horizon* decision problem with horizon T measures the value of π as $E(\sum_{t=0}^{T} R(s^t)|\pi)$ (where expectation is taken w.r.t. Pr).[1] For an MDP with horizon T, a *(nonstationary) policy* $\pi : S \times \{1, \cdots, T\} \rightarrow A$ associates with each state s and stage-to-go $t \leq T$ an action $\pi(s, t)$ to be executed at s with t stages remaining. An *optimal* policy is one with maximum expected value at each state-stage pair.

The planning problem faced by an agent is that of forming an *optimal policy* (a mapping from states to actions) that maximizes expected total accumulated reward over some horizon. Dynamic programming methods are often used to solve MDPs [13], though one difficulty facing (the classical versions of) such algorithms is their reliance on an explicit state-space formulation; as such, their complexity is exponential in the number of state variables. However, "logical" representations such as STRIPS and dynamic Bayesian networks have recently been used to make the specification and solution of MDPs much easier [4]. The DTGolog representation goes further in this direction by specifying state transitions in first order logic. Restricting attention to reachable states using decision tree search can, in some circumstances, alleviate the computational difficulties of dynamic programming. Search-based approaches to solving MDPs can use heuristics, learning, sampling, and pruning to improve their efficiency [3, 6, 7, 8, 9]. Declarative search control knowledge, used successfully in classical planning [2], might also be used to prune the search space. In an MDP, this could be viewed as restricting the set of policies considered. This type of approach has been explored in the more general context of value iteration for MDPs in, e.g., [11, 18]: local policies or finite-state machines are used to model partial policies, and techniques are devised to find the optimal policy consistent with the constraints so imposed. In Section 4 we develop the DTGolog interpreter to capture similar intuitions, but adopt the Golog programming language as a means of specifying these constraints using natural programming constructs.

3 The Situation Calculus and Golog

The situation calculus is a first-order language for axiomatizing dynamic worlds. In recent years, it has been considerably extended beyond the "classical" language to include concurrency, continuous time, etc., but in all cases, its basic ingredients consist of *actions*, *situations* and *fluents*.

Actions are first-order terms consisting of an action function symbol and its arguments. In the approach to representing time in the situation calculus of [14], one of the arguments to such an action function symbol—typically, its last argument—is the time of the action's occurrence. For example, $\mathit{startGo}(l, l', 3.1)$ might denote the action of a robot starting to move from location l to l' at time 3.1. Following Reiter [14], all actions are instantaneous (i.e, with zero duration).[2]

A *situation* is a first-order term denoting a sequence of actions. These sequences are represented using a binary function symbol *do*: $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. The special constant S_0 denotes the *initial situation*, namely the empty action sequence. Therefore, the situation term

$$do(\mathit{endGo}(l, l', 7.3), do(\mathit{startGrasp}(o, 2), do(\mathit{startGo}(l, l', 2), S_0)))$$

denotes the following sequence of actions: $\mathit{startGo}(l, l', 2)$, $\mathit{startGrasp}(o, 2)$, $\mathit{endGo}(l, l', 7.3)$. Axioms for situations with time are given in [15].

Relations whose truth values vary from state to state are called *relational fluents*, and are denoted by predicate or function symbols whose last argument is a situation term. For

[1] We focus on finite-horizon problems to keep the presentation short, though everything we describe can be applied with little modification to discounted, infinite-horizon MDPs.

[2] Durations can be captured using processes, as shown below. A full exposition of time is not possible here.

example, $closeTo(x, y, s)$ might be a relational fluent, meaning that when the robot performs the action sequence denoted by the situation term s, x will be close to y.

A domain theory is axiomatized in the situation calculus with four classes of axioms:

Action precondition axioms: There is one for each action function $A(\vec{x})$, with syntactic form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$. Here, $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$. These are the preconditions of action A.

Successor state axioms: There is one for each relational fluent $F(\vec{x}, s)$, with syntactic form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $a, s, \vec{x}$. These characterize the truth values of the fluent F in the next situation $do(a, s)$ in terms of the current situation s, and they embody a solution to the frame problem for deterministic actions ([16]).

Unique names axioms for actions: These state that the actions of the domain are pairwise unequal.

Initial database: This is a set of sentences whose only situation term is S_0; it specifies the initial problem state.

Examples of these axioms will be seen in Section 4.1.

Golog [10] is a situation calculus-based programming language for defining complex actions in terms of a set of primitive actions axiomatized in the situation calculus as described above. It has the standard—and some not-so-standard—control structures found in most Algol-like languages.

1. *Sequence:* $\alpha\ ;\ \beta$. Do action α, followed by action β.
2. *Test actions:* $p?$ Test the truth value of expression p in the current situation.
3. *Nondeterministic action choice:* $\alpha \mid \beta$. Do α or β.
4. *Nondeterministic choice of arguments:* $(\pi\ x)\alpha(x)$. Nondeterministically pick a value for x, and for that value of x, do action $\alpha(x)$.
5. *Conditionals* (*if-then-else*) and *while* loops.
6. *Procedures, including recursion.*

The semantics of Golog programs is defined by macro-expansion, using a ternary relation *Do*. $Do(\delta, s, s')$ is an *abbreviation* for a situation calculus formula whose intuitive meaning is that s' is one of the situations reached by evaluating the program δ beginning in situation s. Given a program δ, one *proves*, using the situation calculus axiomatization of the background domain, the formula $(\exists s)Do(\delta, S_0, s)$ to compute a plan. Any binding for s obtained by a constructive proof of this sentence is a legal execution trace, involving only primitive actions, of δ. A Golog interpreter for the situation calculus with time, implemented in Prolog, is described in [15].

Thus the interpreter will makes choices (if possible) that lead to successful computation of an execution trace of the program. With nondeterministic choice and the specification of postconditions corresponding to goals, Golog can be viewed as integrating planning and programming in deterministic domains. We will see examples of Golog programs in Section 5.

4 DTGolog: Decision-Theoretic Golog

As a planning model, MDPs are quite flexible and robust, dealing with uncertainty, multiple objectives, and so on, but suffer from several key limitations. While recent work in DTP has focused on the development of compact, natural representations for MDPs [4], little work has gone into the development of first-order languages for specifying MDPs (see [1, 12] for two exceptions). More importantly, the computational complexity of policy construction is prohibitive. As mentioned, one way to circumvent planning complexity is to allow explicit agent programming; yet little work has been directed toward integrating the ability to write programs or otherwise constrain the space of policies that are searched during planning. What work has been done (e.g., [11, 18]) fails to provide a language for imposing such constraints, and certainly offers no tools for *programming* agent behavior. We believe that natural, declarative *programming languages and methodologies* for (partially) specifying agent behavior are necessary for this approach to find successful application in real domains.

Golog, on the other hand, provides a very natural means for agent programming. With nondeterministic choice a programmer can even leave a certain amount of "planning" up to the interpreter (or agent being controlled). However, for applications such as robotics programming, the usefulness of Golog is severely limited by its inability to model stochastic domains, or reason decision-theoretically about appropriate choices. Despite these limitations, (deterministic) Golog has been successfully used to provide the high-level control of a museum tour-guide robot, controlling user interaction and scheduling more than 2,400 exhibits [5].

We have developed DTGolog, a decision-theoretic extension of Golog that allows one to specify MDPs in a first-order language, and provide "advice" in the form of high-level programs that constrain the search for policies. A program can be viewed as a partially-specified policy: its semantics can be viewed, informally, as the execution of the program (or the completion of the policy) that has highest expected value. DTGolog offers a synthesis of both planning and programming, and is in fact general enough to accommodate both extremes. One can write purely nondeterministic programs that allow an agent to solve an MDP optimally, or purely deterministic programs that leave no decisions in the agent's hands whatsoever. We will see, in fact, that a point between these ends of the spectrum is often the most useful way to write robot programs. DTGolog allows the appropriate point for any specific problem to be chosen with relative ease. Space precludes the presentation of many technical details, but we try to provide the basic flavor of DTGolog.

4.1 DTGolog: Problem Representation

The specification of an MDP requires the provision of a background *action theory*—as in Section 3—and a background *optimization theory*—consisting of the specification of a reward function and some optimality criterion (here we require only a horizon T). The unique names axioms and initial database have the same form as in standard Golog.

A background action theory in the decision-theoretic setting distinguishes between *deterministic* agent actions and *stochastic* agent actions. Both types are used to form programs and policies. However, the situation resulting from execution of a stochastic action is not determined by the action itself: instead each stochastic agent action is associated with a finite set of deterministic actions, from which "nature" chooses stochastically. Successor state axioms are provided for nature's actions directly (which are deterministic), not for stochastic agent actions (i.e., successor state axioms never mention stochastic agent actions). When a stochastic action is executed, nature chooses one of the associated actions with a specified probability, and the successor state is given by nature's action so chosen. The predicate $\mathit{stochastic}(a, s, n)$ relates a stochastic agent action a to one of nature's action n in a situation s, and $\mathit{prob}(n, p, s)$ denotes the probability with which n is chosen in s. Deterministic agent's actions are axiomatized using exactly the same precondition and successor state axioms. This methodology allows us to extend the axiomatization of a domain theory described in the previous section in a minimal way.

As an example, imagine a robot moving between different locations: the process of going is initiated by a deterministic action $\mathit{startGo}(l_1, l_2, t)$; but the terminating action $\mathit{endGo}(l_1, l_2, t)$ is stochastic (e.g., the robot may end up in some location other than l_2, say, the hallway). We give nature two choices, $\mathit{endGoS}(l_1, l_2, t)$ (successful arrival) and $\mathit{endGoF}(l_1, \mathit{Hall}, t)$ (end with failure), and include axioms such as $\mathit{stochastic}(\mathit{endGo}(l_1, l_2, t), s, \mathit{endGoS}(l_1, l_2, t))$ and $\mathit{prob}(\mathit{endGoS}(l_1, l_2, t), 0.9, s)$ (i.e., successful movement occurs with probability 0.9 in any situation). Let $\mathit{going}(l_1, l_2, s)$ be the relational fluent meaning that in the situation s the robot is in the process of moving between locations l_1 and l_2; and let $\mathit{robotLoc}(l, s)$ be a relational fluent denoting the robot's location. The following precondition and successor state axioms characterize these fluents, and the actions *startGo*, *endGoS*, *endGoF*:

$$
\begin{aligned}
&\mathit{Poss}(\mathit{startGo}(l_1, l_2, t), s) \equiv \neg(\exists l, l')\mathit{going}(l, l', s) \\
&\qquad \wedge \mathit{robotLoc}(l_1, s) \\
&\mathit{Poss}(\mathit{endGoS}(l_1, l_2, t), s) \equiv \mathit{going}(l_1, l_2, s), \\
&\mathit{Poss}(\mathit{endGoF}(l_1, l_2, t), s) \equiv \exists l'.\mathit{going}(l_1, l', s) \wedge l' \neq l_2 \\
&\mathit{going}(l, l', do(a, s)) \equiv (\exists t)a = \mathit{startGo}(l, l', t) \vee \\
&\qquad\qquad \mathit{going}(l, l', s) \wedge \neg(\exists t)a = \mathit{endGoS}(l, l', t) \vee \\
&\qquad\qquad \mathit{going}(l, l', s) \wedge \neg(\exists t, l'')a = \mathit{endGoF}(l, l'', t),
\end{aligned}
$$

The background action theory also includes a new class of axioms, *sense conditions axioms*, which assert atomic formulae using predicate $\mathit{senseCond}(n, \phi)$: this holds if ϕ is a logical condition that an agent uses to determine if the specific nature's action n occurred when some stochastic action was executed. We require such axioms in order to "implement" full observability. While in the standard MDP model one simply assumes that the successor state is known, in practice, one must force agents to disambiguate the state using sensor information. The sensing actions needed can be determined from sense condition axioms. The following distinguish successful from unsuccessful movement:

$$
\begin{aligned}
&\mathit{senseCond}(\mathit{endGoS}(l_1, l_2, t), \mathit{robotLoc}(l_2)) \\
&\mathit{senseCond}(\mathit{endGoF}(l_1, l_2, t), \mathit{robotLoc}(\mathit{Hall}))
\end{aligned}
$$

A DTGolog optimization theory contains axioms specifying the reward function.[3] In their simplest form, reward axioms use the function $\mathit{reward}(s)$ and assert costs and rewards as a function of the action taken, properties of the current situation, or both (note that the action taken can be recovered from the situation term). For instance, we might assert

$$\mathit{reward}(do(\mathit{giveCoffeeSuccessful}(\mathit{Jill}, t), s)) = 6.3$$

Because primitive actions have an explicit temporal argument, we can also describe time-dependent reward functions easily (associated with behaviors that extend over time). These can be dealt with in the interpreter because of our use of situation terms rather than states, from which time can be derived without having it explicitly encoded in the state. This often proves useful in practice. In a given temporal Golog program, the temporal occurrence of certain actions can be uniquely determined either by temporal constraints or by the programmer. Other actions may occur at any time in a certain interval determined by temporal inequalities; for any such action $A(\vec{x}, t)$, we can instantiate the time argument by maximizing the reward for reaching the situation $do(A(\vec{x}, t), s)$. For example, suppose the robot receives a reward $r = max(\frac{100-t}{distance(l_1, l_2)})$ for doing the action $\mathit{endGoS}(l_1, l_2, t)$ in s. With this reward function, the robot is encouraged to arrive at the destination as soon as possible and is also encouraged to go to nearby locations (because the reward is inversely proportional to distance).

Our representation for stochastic actions is related somewhat to the representations proposed in [1, 7, 12].

4.2 DTGolog: Semantics

In what follows, we assume that we have been provided with a background action theory and optimization theory. We interpret DTGolog programs relative to this theory. DTGolog programs are written using the same program operators as Golog programs. The semantics is specified in a similar fashion, with the predicate *BestDo* (described below) playing the role of *Do*. However, the structure of *BestDo* (and its Prolog implementation) is rather different than that of *Do*. One difference reflects the fact that primitive actions can be stochastic. Execution traces for a sequence of primitive actions need not be simple "linear" situation terms, but rather branching "trees." Another reflects the fact that DTGolog distinguishes otherwise legal traces according to expected utility. Given a choice between two actions (or subprograms) at some point in a program, the interpreter chooses the action with highest expected value, mirroring the structure of an MDP search tree. The interpreter returns a *policy*—an expanded Golog program—in which every nondeterministic choice point is *grounded* with the selection of an optimal choice. Intuitively, the semantics of a DTGolog program will be given by the *optimal execution of that program*.

The semantics of a DTGolog program is defined by a predicate $\mathit{BestDo}(prog, s, horiz, pol, val, prob)$, where $prog$ is a Golog program, s is a starting situation, pol is the optimal conditional policy determined by program $prog$ beginning in

[3] We require an optimality criterion to be specified as well. We assume a finite-horizon H in this work.

situation s, val is the expected value of that policy, $prob$ is the probability that pol will execute successfully, and $horiz$ is a prespecified horizon. Generally, an intepreter implementing this definition will be called with a given program $prog$, situation S_0, and horizon $horiz$, and the arguments pol, val and $prob$ will be instantiated by the interpreter. The policy pol returned by the interpreter is a Golog program consisting of the sequential composition (under ;) of agent actions, *senseEffect*(A) sensing actions (which serve to identify nature's choices whenever A is a stochastic agent action), and conditionals (**if** ϕ **then** pol_1 **else** pol_2).

Below we assume an MDP with finite horizon H: if a program fails to terminate before the horizon is reached, the interpreter produces the best (partial) H-step execution of the program. The interpreter can easily be modified to deal with programs that are guaranteed to terminate in a finite amount of time (so a bound H need not be imposed) or infinite-horizon, discounted problems (returning ε-optimal policies).

BestDo is defined inductively on the structure of its first argument, which is a Golog program:

1. **Zero horizon.**

$$\mathit{BestDo}(p, s, h, \pi, v, pr) \stackrel{def}{=} h = 0 \wedge \pi = Nil \wedge v = reward(s) \wedge pr = 1.$$

Give up on the program p if the horizon reaches 0.

2. **The null program**

$$\mathit{BestDo}(Nil, s, h, \pi, v, pr) \stackrel{def}{=} \pi = Nil \wedge v = reward(s) \wedge pr = 1.$$

3. **First program action is deterministic.**

$$\begin{aligned}&\mathit{BestDo}(a; p, s, h, \pi, v, pr) \stackrel{def}{=}\\ &\neg Poss(a, s) \wedge \pi = \mathit{Stop} \wedge pr = 0 \wedge v = \mathit{reward}(s) \vee\\ &Poss(a, s) \wedge\\ &\exists(\pi', v', pr') \mathit{BestDo}(p, do(a, s), h{-}1, \pi', v', pr') \wedge\\ &\quad \pi = a; \pi' \wedge v = reward(s) + v' \wedge pr = pr'.\end{aligned}$$

A program that begins with a deterministic agent action a (if a is possible in situation s) has its optimal execution defined as the optimal execution of the remainder of the program p in situation $do(a, s)$. Its value is given by the expected value of this continuation plus the reward in s (action cost for a can be included without difficulty), while its success probability is given by the success probability of its continuation. The optimal policy is a followed by the optimal policy for the remainder. If a is *not* possible at s, the policy is simply the *Stop* action, the success probability is zero, and the value is simply the reward associated with situation s. $Stop$ is a zero-cost action that takes the agent to a zero-cost absorbing state.[4]

4. **First program action is stochastic.**
When a is a stochastic agent action for which nature selects one of the actions in the set $\{n_1, \ldots, n_k\}$,

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

The resulting policy is $a; \mathit{senseEffect}(a); \pi'$ where π' is the policy delivered by *BestDoAux*. Intuitively, this policy says that the agent should first perform action a, at which point nature selects one of $n_1, \ldots, n_k$ to perform (with probabilities $prob(n_i, s)$), then the agent should sense the outcome of action a (which tells it which of nature's actions n_i actually occurred), then it should execute the policy delivered by *BestDoAux*.[5]

$$\begin{aligned}&\mathit{BestDoAux}(\{\,\}, p, s, h, \pi, v, pr) \stackrel{def}{=}\\ &\qquad \pi = Stop \wedge v = 0 \wedge pr = 0.\end{aligned}$$

Suppose $k \geq 1$. Suppose further that ϕ_1 is the sense condition for nature's action n_1, meaning that observing that ϕ_1 is true is necessary and sufficient for the agent to conclude that nature actually performed action n_1, among the choices $\{n_1, \ldots, n_k\}$ available to her by virtue of the agent having done stochastic action a. Then

$$\begin{aligned}&\mathit{BestDoAux}(\{n_1, \ldots, n_k\}, p, s, h, \pi, v, pr) \stackrel{def}{=}\\ &\neg Poss(n_1, s) \wedge \mathit{BestDoAux}(\{n_2, \ldots, n_k\}, p, s, h, \pi, v, pr)\\ &\vee Poss(n_1, s) \wedge\\ &\exists(\pi', v', pr').\mathit{BestDoAux}(\{n_2, \ldots, n_k\}, p, s, h, \pi', v', pr') \wedge\\ &\exists(\pi_1, v_1, pr_1).\mathit{BestDo}(p, do(n_1, s), h{-}1, \pi_1, v_1, pr_1) \wedge\\ &\quad \pi = \textbf{if } \phi_1 \textbf{ then } \pi_1 \textbf{ else } \pi' \wedge\\ &\quad v = v' + v_1 \cdot prob(n_1, s) \wedge\\ &\quad pr = pr' + pr_1 \cdot prob(n_1, s).\end{aligned}$$

BestDoAux determines a policy in the form of a conditional plan:

if ϕ_{i_1} **then** pol_1 **else if** ϕ_{i_2} **then** $pol_2 \cdots$
else if ϕ_{i_m} **then** pol_m **else** $Stop$.

Here, $n_{i_1}, \ldots, n_{i_m}$ are all of nature's actions among $\{n_1, \ldots, n_k\}$ that are possible in s, and pol_j is the policy returned by the program p, in situation $do(n_{i_j}, s)$.

5. **First program action is a test.**

$$\begin{aligned}&\mathit{BestDo}(\phi?; p, s, h, \pi, v, pr) \stackrel{def}{=}\\ &\quad \phi[s] \wedge \mathit{BestDo}(p, s, h, \pi, v, pr) \vee\\ &\quad \neg\phi[s] \wedge \pi = \mathit{Stop} \wedge pr = 0 \wedge v = reward(s)\end{aligned}$$

6. **First program action is the nondeterministic choice of two programs.**

$$\begin{aligned}&\mathit{BestDo}((p_1 \mid p_2); p, s, h, \pi, v, pr) \stackrel{def}{=}\\ &\exists(\pi_1, v_1, pr_1).\mathit{BestDo}(p_1; p, s, h, \pi_1, v_1, pr_1) \wedge\\ &\exists(\pi_2, v_2, pr_2).\mathit{BestDo}(p_2; p, s, h, \pi_2, v_2, pr_2) \wedge\\ &((v_1, pr_1) \leq (v_2, pr_2) \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2 \vee\\ &\quad (v_1, pr_1) > (v_2, pr_2) \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1).\end{aligned}$$

Given the choice between two subprograms p_1 and p_2, the optimal policy is determined by that subprogram with optimal execution. Note that there is some subtlety in the interpretation of a DTGolog program: on the one hand, we wish the interpreter to choose a course of action with maximal expected value; on the other, it should follow the advice provided by the program. Because certain choices may lead to abnormal termination—the $Stop$ action cor-

[4] This can be viewed as having an agent simply give up its attempt to execute the policy and await further instruction.

[5] It is these sensing actions that "implement" the assumption that the MDP is fully observable.

responding to an incomplete execution of the program—with varying probabilities, the success probability associated with a policy can be loosely viewed as the degree to which the interpreter adhered to the program. Thus we have a multi-objective optimization problem, requiring some tradeoff between success probability and expected value of a policy. The predicate $\leq$ compares pairs of the form (p, v), where p is a success probability and v is an expected value.[6]

7. **Conditionals.**

$$\mathit{BestDo}((\textbf{if } \phi \textbf{ then } p_1 \textbf{ else } p_2); p, s, h, \pi, v, pr) \stackrel{def}{=} \mathit{BestDo}((\phi?; p_1 \mid \neg\phi?; p_2); p, s, h, \pi, v, pr)$$

This simply says that a conditional **if** ϕ **then** p_1 **else** p_2 is an abbreviation for $\phi?; p_1 \mid \neg\phi?; p_2$.

8. **Nondeterministic finite choice of action arguments.**

$$\mathit{BestDo}((\pi(x : \tau)p); p', s, h, pol, v, pr) \stackrel{def}{=} \mathit{BestDo}(p|^x_{c_1} \mid \cdots \mid p|^x_{c_n}); p', s, h, pol, v, pr)$$

The programming construct $\pi(x : \tau)p$ requires the nondeterministic choice of an element x from the finite set $\tau = \{c_1, \ldots, c_n\}$, and for that x, do the program p. It therefore is an abbreviation for the program $p|^x_{c_1} \mid \cdots \mid p|^x_{c_n}$, where $p|^x_c$ means substitute c for all free occurrences of x in p.

9. **Associate sequential composition to the right.**

$$\mathit{BestDo}((p_1; p_2); p_3, s, h, \pi, v, pr) \stackrel{def}{=} \mathit{BestDo}(p_1; (p_2; p_3), s, h, \pi, v, pr).$$

This is needed to massage the program to a form in which its first action is one of the forms suitable for application of rules 2-8.

There is also a suitable expansion rule when the first program action is a procedure call. This is almost identical to the rule for Golog procedures [10], and requires second-order logic to characterize the standard fixed point definition of recursive procedures. Because it is a bit on the complicated side, and because it is not central to the specification of policies for DTGolog, we omit this expansion rule here. While loops can be defined using procedures.

4.3 Computing Optimal Policies

$\mathit{BestDo}(prog, s, horiz, pol, val, prob)$ is, analogously to the case for Golog, an *abbreviation* for a situation calculus formula whose intuitive meaning is that pol is an optimal policy resulting from evaluating the program $prog$ beginning in situation s, that val is its value, and $prob$ the probability of a successful execution of this policy. Therefore, given a program δ, and horizon H, one *proves*, using the situation calculus axiomatization of the background domain described in Section 4.1, the formula

$$\exists(pol, val, prob)\mathit{BestDo}(\delta; Nil, S_0, H, pol, val, prob).$$

Any binding for pol, val and $prob$ obtained by a constructive proof of this sentence determines the result of the program computation.

4.4 Implementing a DTGolog Interpreter

Just as an interpreter for Golog is almost trivial to implement in Prolog, when given its situation calculus specification, so also is an interpreter for DTGolog. One simply translates each of the above rules into an almost identical Prolog clause. For example, here is the implementation for rules 3 and 6:

```
% First action is deterministic.
bestDo(A : E,S,H,Pol,V,Prob) :-
 agentAction(A), deterministic(A),
 (not poss(A,S), Pol=stop, Prob is 0, reward(V,S);
  poss(A,S), bestDo(E,do(A,S),H-1,RestPol,Vfuture,Prob),
     reward(R,S), V is R + Vfuture, Pol = (A : RestPol)).

% Nondeterministic choice between E1 and E2
bestDo((E1 # E2) : E,S,Pol,V,P,k) :-
      bestDo(E1 : E,S,Pol1,V1,P1,k),
      bestDo(E2 : E,S,Pol2,V2,P2,k),
      ( lesseq(V1,P1,V2,P2), Pol=Pol2, P=P2, V=V2;
        greater(V1,P1,V2,P2), Pol=Pol1, P=P1, V=V1).
```

The entire DTGolog interpreter is in this style, and is extremely compact and transparent.

5 Robot Programming

A key advantage of DTGolog as a framework for robot programming and planning is its ability to allow behavior to be specified at any convenient point along the programming/planning spectrum. By allowing the specification of stochastic domain models in a declarative language, DTGolog not only allows the programmer to specify programs naturally (using robot actions as the base level primitives), but also permits the programmer to leave gaps in the program that will be filled in optimally by the robot itself. This functionality can greatly facilitate the development of complex robotic software. Planning ability allows for the scheduling of complex behaviors that are difficult to preprogram. It also obviates the need to reprogram a robot to adapt its behavior to reflect environmental changes or changes in objective functions. Programming, in contrast, is crucial in alleviating the computational burden of uninformed planning.

To illustrate these points, we have developed a mobile delivery robot, tasked to carry mail and coffee in our office building. The physical robot is an RWI B21 robot, equipped with a laser range finder. The robot navigates using BeeSoft [5, 19], a software package that includes methods for map acquisition, localization, collision avoidance, and online path planning. Figure 1d shows a map, along with a delivery path (from the main office to a recipient's office).

Initially, the robot moves to the main office, where someone loads mail on the robot, as shown in Figure 1a. DTGolog then chooses a recipient by utility optimization. Figure 1b shows the robot traveling autonomously through a hallway. If the person is in his office, he acknowledges the receipt of

[6] How one defines this predicate depends on how one interprets the advice embodied in a program. In our implementation, we use a mild lexicographic preference where $(p_1, v_1) < (p_2, v_2)$ whenever $p_1 = 0$ and $p_2 > 0$ (so an agent cannot choose an execution that guarantees failure). If both p_1 and p_2 are zero, or both are greater than zero, than the v-terms are used for comparison. It is important to note that certain multiattribute preferences could violate the dynamic programming principle, in which case our search procedure would have to be revised (as would any form of dynamic programming). This is not the case with our lexicographic preference.

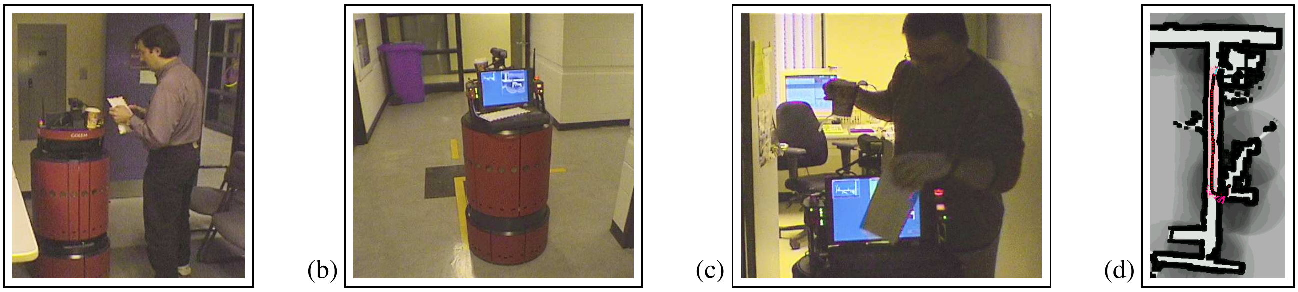

Figure 1: Mail delivery: (a) A person loads mail and coffee onto the robot. (b) DTGolog sends the robot to an office. (c) The recipient accepts the mail and coffee, acknowledging the successful delivery by pressing a button. (d) The map learned by the robot, along with the robot's path (from the main office to recipient).

the items by pressing a button on the robot as shown in Figure 1c; otherwise, after waiting for a certain period of time, the robot marks the delivery attempt as unsuccessful and continues with the next delivery. The task of DTGolog, thus, is to schedule the individual deliveries in the face of stochastic action effects arising from the fact that people may or may not be in their office at the time of delivery. It must also contend with different priorities for different people and balance these against the domain uncertainty.

The underlying MDP for this relatively simple domain grows rapidly as the number of people requiring deliveries increases. The state space is characterized by fluents such as *hasMail*$(person, s)$, *mailPresent*$(person, n, s)$, *robotLoc*(loc, s), and so on. In a domain with P people, L locations, and N as the maximum number of pieces of mail (and *ignoring* the temporal aspect of the problem), our MDP has a state space of size $2^N \cdot (6N+6)^P \cdot L^3$ when formulated in the most appropriate way. Even restricting the MDP to one piece (or bundle) of mail per person, the state space complexity, $24^P \cdot L^3$, grows exponentially in P. Actions include picking up mail, moving from location to location, giving mail and so on. Uncertainty is associated with the *endGo* action as described above, as well as with the outcome of giving mail (see below).

The robot's objective function is given by a reward function that associates an independent, additive reward with each person's successful delivery. Each person has a different deadline, and the reward decreases linearly with time until the deadline (when it becomes zero). The relative priority associated with different recipients is given by this function; e.g., we might use *reward*$(Ray, t, s) = 30 - t/10$, where the initial reward (30) and rate of decrease (1/10) indicates relative priority. Given a situation term corresponding to any branch of the tree, it is straightforward to maximize value with respect to choice of temporal arguments assigned to actions in the sequence. We do not delve into details here.

Our robot is provided with the following simple DTGolog program:

while $(\exists p.\neg attempted(p) \wedge \exists n$ *mailPresent(p,n)*$)$
  $\pi(p,\ people,$
    $(\neg attempted(p) \wedge \exists n$ *mailPresent(p,n)*$)?\ ;\ deliverTo(p)\ )$
endWhile

Intuitively, this program chooses people from the finite range `people` for mail delivery and delivers mail in the order that maximizes expected utility (coffee delivery can be incorporated readily). `deliverTo` is itself a complex procedure involving picking up items for a person, moving to the person's office, giving the items, and returning to the mailroom. But this sequence is a very obvious one to handcode in our domain, whereas the optimal ordering of delivery is not (and can change, as we'll see). We have included a guard condition $\neg attempted(p) \wedge \exists n$ *mailPresent(p,n)* in the program to prevent the robot from repeatedly trying to deliver mail to a person who is out of her office. This program constrains the robot to just one attempted mail delivery per person, and is a nice example of how the programmer can easily impose domain specific restrictions on the policies returned by a DTGolog program.

Several things emerged from the development of this code. First, the same program determines different policies—and very different qualitative behavior—when the model is changed or the reward function is changed. As a simple example, when the probability that Ray (high priority) is in his office is 0.8, his delivery is scheduled before Craig's (low priority); but when that probability is lowered to 0.6, Craig's delivery is scheduled beforehand. Such changes in the domain would require a change in the control program if not for the planning ability provided by DTGolog. The computational requirements of this decision making capability are much less than those should we allow completely arbitrary policies to be searched in the decision tree.

Full MDP planning can be implemented within DTGolog by running it with the program that allows *any* (feasible) action to be chosen at *any* time. This causes a full decision tree to be constructed. Given the domain complexity, this unconstrained search tree could only be completely evaluated for problems with a maximum horizon of seven (in about 1 minute)—this depth is barely enough to complete the construction of a policy to serve one person. With the program above, the interpreter finds optimal completions for a 3-person domain in about 1 second (producing a policy with success probability 0.94), a 4-person domain in about 9 seconds (success probability 0.93) and a 5-person domain in about 6 minutes (success probability 0.88). This latter corresponds to a horizon of about 30; clearly the decision tree search would be infeasible without the program constraints (with size well over 10^{30}). We note that the MDP formulation of this problem, with 5 people and 7 locations, would require more than 2.7 billion states. So dynamic programming could not be used to solve this MDP without program constraints (or exploiting some other form of structure).

We note that our example programs restrict the policy that the robot can implement, leaving only one choice (the choice of person to whom to deliver mail) available to the robot, with the rest of the robot's behavior fixed by the program. While these programs are quite natural, structuring a program this way may preclude optimal behavior. For instance, by restricting the robot to serving one person at a time, the simultaneous delivery of mail to two people in nearby offices won't be considered. In circumstances where interleaving is impossible (e.g., the robot can carry only one item at a time), this program admits optimal behavior—it describes *how* to deliver an item, leaving the robot to decide only on the order of deliveries. But even in settings where simultaneous or interleaved deliveries are feasible, the "nonoverlapping" program may have sufficiently high utility that restricting the robot's choices is acceptable (since it allows the MDP to be solved much more quickly).

These experiments illustrate the benefits of integrating programming and planning for mobile robot programming. We conjecture that the advantage of our framework becomes even more evident as we scale up to more complex tasks. For example, consider a robot that serves dozens of people, while making decisions as to when to recharge its batteries. Mail and coffee requests might arrive sporadically at random points in time, not just once a day (as is the case for our current implementation). Even with today's best planners, the complexity of such tasks is well beyond what can be tackled in reasonable time. DTGolog is powerful enough to accommodate such scenarios. If supplied with programs of the type described above, we expect DTGolog to make the (remaining) planning problem tractable—with minimal effort on the programmer's side.

6 Concluding Remarks

We have provided a general first-order language for specifying MDPs and imposing constraints on the space of allowable policies by writing a program. In this way we have provided a natural framework for combining decision-theoretic planning and agent programming with an intuitive semantics. We have found this framework to be very flexible as a robot programming tool, integrating programming and planning seamlessly and permitting the developer to choose the point on this spectrum best-suited to the task at hand. While Golog has proven to be an ideal vehicle for this combination, our ideas transcend the specific choice of language.

A number of interesting directions remain to be explored. The decision-tree algorithm used by the DTGolog interpreter is clearly subject to computational limitations.[7] However, the basic intuitions and foundations of DTGolog are not wedded to this particular computational model. We are currently integrating integrating efficient algorithms and other techniques for solving MDPs into this framework (dynamic programming, abstraction, sampling, etc.). We emphasize that even with these methods, the ability to naturally constrain the search for good policies with explicit programs is crucial. Other avenues include: incorporating realistic models of partial observability (a key to ensuring wider applicability of the model); extending the expressive power of the language to include other extensions already defined for the classical Golog model (e.g., concurrency); incorporating declaratively-specified heuristic and search control information; monitoring of on-line execution of DTGolog programs [17]; and automatically generating sense conditions for stochastic actions.

[7] Note, however, that program constraints often make otherwise intractable MDPs reasonably easy to solve using search methods.

References

[1] F. Bacchus, J. Halpern, and H. Levesque. Reasoning about noisy sensors in the situation calculus. *IJCAI-95*, pp.1933–1940, Montreal, 1995.

[2] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab, A. Milani, eds., *New Directions in Planning*, pp.141–153, 1996. IOS Press.

[3] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Art. Intel.*, 72:81–138, 1995.

[4] C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *J. Art. Intel. Res.*, 11:1–94, 1999.

[5] W. Burgard, A. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Art. Intel.*, 114, 1999.

[6] R. Dearden and C. Boutilier. Abstraction and approximate decision theoretic planning. *Art. Intel.*, 89:219–283, 1997.

[7] H. Geffner and B. Bonet. High-level planning and control with incomplete information using POMDPs. *AAAI Fall Symp. on Cognitive Robotics*, Orlando, 1998.

[8] M. Kearns, Y. Mansour, and A. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *IJCAI-99*, Stockholm, 1999.

[9] S. Koenig and R. Simmons. Real-time search in nondeterministic domains. *IJCAI-95*, pp.1660–1667, Montreal, 1995.

[10] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: a logic programming language for dynamic domains. *J. Logic Prog.*, 31(1-3):59–83, 1997.

[11] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. *NIPS-10*, pp.1043–1049. MIT Press. 1998.

[12] D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Art. Intel.*, 94:7–56, 1997.

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

[14] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. *KR'96*, pp.2–13, Cambridge, 1996.

[15] R. Reiter. Sequential, temporal GOLOG. *KR'98*, pp.547–556, Trento, 1998.

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

[17] M. Soutchanski. Execution monitoring of high–level temporal programs. *IJCAI-99 Workshop on Robot Action Planning*, Stockholm, 1999.

[18] R. Sutton. TD models: Modeling the world at a mixture of time scales. *ICML-95*, pp.531–539, Lake Tahoe, 1995.

[19] S. Thrun, M. Bennewitz, W. Burgard, A. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A second generation mobile tour-guide robot. *ICRA-99*, 1999.