# 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, but also 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 by describing 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) [5, 17] with the Golog programming language [11]. 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 [13], with a clear logical semantics, and in which standard programming constructs (e.g., sequencing, nondeterministic choice) are used to write a 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 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.

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 [6, 23]. 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 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 [5, 17] for further details on MDPs). 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 with certainty the state that will be reached when an action is taken, 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 $\pi$ depends on the specific objectives. A *finite-horizon* decision problem with horizon $T$ measures the value of $\pi$ 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.

A simple algorithm for constructing optimal policies is *value iteration* [4, 17]. Define the $t$-stage-to-go value function $V^t$ by setting $V^0(s_i) = R(s_i)$ and, for all $1 \leq t \leq T$:

$$V^t(s_i) = R(s_i) + \max_{a \in \mathcal{A}} \{ \sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) V^{t-1}(s_j) \} \quad (1)$$

By setting $\pi(s_i, t)$ to the action $a$ maximizing the right-hand term, the resulting policy $\pi$ will be optimal.

One difficulty faced by (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 [5].

When the system is known to start in a given state $s_0$, the *reachability structure* of the MDP can also be exploited for computational gain. Reachability analysis allows one to restrict value and policy computations to states reachable by some sequence of actions. This form of *directed value iteration* can be effected by building a search tree rooted at state $s_0$: its successors at level 1 of the tree are possible actions; the successors at level 2 of any action node are those states that can be reached with nonzero probability when that action is taken at state $s_0$; and deeper levels of the tree are defined recursively in the same way. For an MDP with finite horizon $k$, the tree is built to level $2k$: the value of of any state is given by the maximum among all values of its successor actions, and the value of an action is given by the expected value of its successor states.[2] Search-based approaches to solving MDPs can use heuristics, learning, sampling and pruning to improve their efficiency [3, 7, 8, 9, 10]. 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 by Parr and Russell [14]: they use a finite state machine to model a partial policy and devise an algorithm to find the optimal policy consistent with the constraints imposed by the FSM. In Section 4 we develop the DTGolog interpreter to capture similar intuitions. We will marry the MDP model with the Golog programming language [11].

## 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 [18], one of the arguments to such an action function symbol—typically, its last argument—is the time of the action's occurrence. For example, $startGo(l, l', 3.1)$ might denote the action of a robot starting to move from location $l$ to $l'$ at time 3.1. Following Reiter

[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] States at the leaves are assigned their reward value.

[18], all actions are instantaneous (i.e, with zero duration).[3]

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 $\alpha$ to the sequence $s$. The special constant $S_0$ denotes the *initial situation*, namely the empty action sequence. Therefore, the situation term

$$do(endGo(l, l', 7.3), do(startGrasp(o, 2), do(startGo(l, l', 2), S_0)))$$

denotes the following sequence of actions: $startGo(l, l', 2)$, $startGrasp(o, 2)$, $endGo(l, l', 7.3)$. Foundational axioms for situations without time are given in [15]. Axioms for situations with time are given in [19]. We refer to these papers for further details.

Relations or functions whose values vary from state to state are called *fluents*, and are denoted by predicate or function symbols whose last argument is a situation term. For 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 (see [15] for details):

**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 ([20]). There are similar axioms for functional fluents, but we shall not be using these in this paper, so we omit them.

**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.

**Example** The following are action precondition and successor state axioms for a blocks world. To keep the example short, we suppose that blocks may only be moved onto other blocks. The axioms appeal to a process fluent $moving(x, y, t, t', s)$, meaning that block $x$ is in the process of moving to $y$, and $t$ and $t'$ are the initiation and termination times of this process. The process has its own instantaneous initiating and terminating actions, $startMove(x, y, t)$ and $endMove(x, y, t)$, with the obvious meanings.

**Action Precondition Axioms**

$$Poss(startMove(x, y, t), s) \equiv clear(x, s) \wedge clear(y, s) \wedge x \neq y \wedge t = start(s),$$

$$Poss(endMove(x, y, t), s) \equiv (\exists t')moving(x, y, t', t, s).$$

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

**Successor State Axioms**

$$clear(x, do(a, s)) \equiv (\exists y, z, t)\{on(y, x, s) \wedge a = startMove(y, z, t)\} \vee clear(x, s) \wedge \neg(\exists y, t)a = endMove(y, x, t),$$

$$on(x, y, do(a, s)) \equiv (\exists t)a = endMove(x, y, t) \vee on(x, y, s) \wedge \neg(\exists z, t)a = startMove(x, z, t),$$

$$onTable(x, do(a, s)) \equiv onTable(x, s) \wedge \neg(\exists y, t)a = startMove(x, y, t).$$

$$moving(x, y, t, t', do(a, s)) \equiv a = startMove(x, y, t) \wedge t' = t + moveDuration(x, y, s) \vee moving(x, y, t, t', s) \wedge a \neq endMove(x, y, t').$$

Golog [11] 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 $\alpha$, followed by action $\beta$.
2. *Test actions:* $p?$ Test the truth value of expression $p$ in the current situation.
3. *Nondeterministic action choice:* $\alpha \mid \beta$. Do $\alpha$ or $\beta$.
4. *Nondeterministic choice of arguments:* $(\pi\ x)\alpha$. Nondeterministically pick a value for $x$, and for that value of $x$, do action $\alpha$.
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 $\delta$ beginning in situation $s$. Given a program $\delta$, 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 $\delta$. A Golog interpreter for the situation calculus with time, implemented in Prolog, is described in [19].

**Example** The following is a nondeterministic Golog program for the example above. $makeOneTower(z)$ creates a single tower of blocks, using as a base the tower whose top block is initially $z$.

**proc** $makeOneTower(z)$
$\neg(\exists y).y \neq z \wedge clear(y)?\ \mid$
$(\pi\, x, t)[startMove(x, z, t)\ ;$
$(\pi\, t')endMove(x, z, t')\ ;\ makeOneTower(x)]$
**endProc**

Like any Golog program, this is executed by proving

$$(\exists s)Do(makeOneTower(z), S_0, s),$$

in our case, using background axioms above. We start with $S_0$ as the current situation. In general, if $\sigma$ is the current situation, $makeOneTower(z)$ terminates in situation $\sigma$ if $\neg(\exists y).y \neq z \wedge clear(y, \sigma)$ holds. Otherwise it nondeterministically selects a block $x$ and time $t$, and "performs" $startMove(x, z, t)$, meaning it makes $do(startMove(x, z, t), \sigma)$ the current situation; then it picks a time $t'$ and "performs" $endMove(x, z, t')$, making $do(endMove(x, z, t'), do(startMove(x, z, t), \sigma))$ the current situation; then it calls itself recursively. On termination, the current situation is returned as a side effect of the computation; this is an execution trace of the program. It is important to understand that this is an offline computation; the resulting trace is intended to be passed to some execution module for the online execution of the primitive actions in the program trace, in our example, to physically build the tower.

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.

# 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 [5], little work has gone into the development of first-order languages for specifying MDPs (see [1, 16] 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., [12, 14, 22]) fails to provide a language for imposing such constraints, and certainly offers no tools for "programming" agent behavior.

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, 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 [6].

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. Such 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 typically 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 important 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 $stochastic(a, s, n)$ relates a stochastic agent action $a$ to one of nature's action $n$ in a situation $s$, and $prob(n, p, s)$ denotes the probability with which $n$ is chosen in $s$. Deterministic agent's actions are axiomatized in 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 $startGo(l_1, l_2, t)$; but the terminating action $endGo(l_1, l_2, t)$ is stochastic (e.g., the robot may end up in some location other than $l_2$, say, the hallway). We give nature two choices, $endGoS(l_1, l_2, t)$ (successful arrival) and $endGoF(l_1, Hall, t)$ (end with failure), and include axioms such as $stochastic(endGo(l_1, l_2, t), s, endGoS(l_1, l_2, t))$ and $prob(endGoS(l_1, l_2, t), 0.9, s)$ (i.e., successful movement occurs with probability 0.9 in any situation). Let $going(l_1, l_2, s)$ 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 $robotLoc(l, s)$ 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*:

$$Poss(startGo(l_1, l_2, t), s) \equiv \neg(\exists l, l')going(l, l', s)$$
$$\wedge robotLoc(l_1, s),$$

$$Poss(endGoS(l_1, l_2, t), s) \equiv going(l_1, l_2, s),$$
$$Poss(endGoF(l_1, l_2, t), s) \equiv \exists l'.going(l_1, l', s) \wedge l' \neq l_2,$$

$$\begin{aligned} going(l, l', do(a, s)) \equiv\ & (\exists t)a = startGo(l, l', t) \vee \\ & going(l, l', s) \wedge \neg(\exists t)a = endGoS(l, l', t) \vee \\ & going(l, l', s) \wedge \neg(\exists t, l'')a = endGoF(l, l'', t), \end{aligned}$$

The background action theory also includes a new class of axioms, *sense conditions axioms*, which assert atomic formulae using predicate $senseCond(n, \phi)$: this holds if $\phi$ 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:

$senseCond(endGoS(l_1, l_2, t), robotLoc(l_2))$

$senseCond(endGoF(l_1, l_2, t), robotLoc(Hall))$

A DTGolog optimization theory contains axioms specifying the reward function.[4] In their simplest form, reward axioms use the predicate $reward(r, s)$ 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

$$hasCoffee(Jill, s) \supset reward(6.3, s)$$

Because primitive actions have an explicit temporal argument, we can also describe time-dependent reward functions easily (associated with behaviors that extend over time).[5] 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 a 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 $(A(\vec{x}, t), s)$. For example, suppose the robot receives a reward $r = max(\frac{100-t}{distance(l_1, l_2)})$ for doing the action $endGoS(l_1, l_2, t)$ 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 in inversely proportional to distance).

Our representation is related somewhat to the representations proposed in [1, 8, 16].

## 4.2 DTGolog: Interpreter

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", with probabilities of occurrence labeling each branch. Furthermore, when the stochastic primitive action dictated by a program is executed, one action outcome may preclude continuation of the program, while another may lead to successful completion. In general, a program "trace" will be associated with a probability of *successful completion* of the program. If an agent enters a situation where the next step of its program is impossible, we halt execution of the program by having the agent execute a zero-cost *Stop* action that takes it to a zero-cost absorbing state.[6]

A second difference has to do with how a "legal trace" is defined. In the original Golog framework, no criteria are used to distinguish one legal execution trace from another. In our decision-theoretic setting, we want nondeterministic choices to be made in a way that maximizes value. 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. Intuitively, then, the semantics of a DTGolog program will be given by the *optimal execution of that program*, where optimality is defined in much the same way as for MDPs. Viewing the program as advice—or a constraint on the set of policies we are allowed to consider—the DTGolog interpreter provides us with an optimal execution trace of $\delta$. Specifically, every nondeterministic choice point will be *grounded* with the selection of an optimal choice.

We 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 (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. We use a predicate $\leq$ that compares pairs of the form $\langle p, v\rangle$, where $p$ is a success probability and $v$ is an expected value.[7]

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

[5] These can be dealt with in the interpreter because of our use of situation terms rather than states.

[6] This can be viewed as having an agent simply give up its attempt to run the program and await further instruction.

[7] 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 $\langle p_1, v_1\rangle < \langle p_2, v_2\rangle$ 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 forms of multiattribute preference 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 any criteria we consider here and is not violated by our lexicographic preference.

The semantics of a DTGolog program is defined by a predicate *BestDo*$(E, S, \pi, V, P, k)$, where $E$ is a program, $S$ is an initial situation, $\pi$ is the optimal conditional policy or completion of program $E$ in situation $S$, $V$ is the expected value of that policy, $P$ is the probability of successful execution of $E$ under $\pi$, and $k$ is the horizon of interest. Generally, an intepreter implementing this definition will be called with a program $E$ and situation $S$, with arguments $\pi$, $V$ and $P$ instantiated by the interpreter. The policy returned by the interpreter is a DTGolog program composed only of: agent actions, *senseEffect*$(A)$ actions (which serve to identify nature's choices), and tests specified by sense conditions, where these are composed sequentially; and the nondeterministic choice operator. Nondeterministic choices are used *only* to connect alternative program branches after a *senseEffect* action, implementing a conditional plan. Each program branch is headed by a test condition determining which of nature's outcomes was implemented. Thus the "choice" is actually *dictated* by nature (not the agent).

We illustrate the structure of the definition of *BestDo* by examining some of the important Prolog clauses in the implementation of our interpreter.[8] Intuitively, the interpreter builds an analog to an MDP search tree of some suitable depth, and evaluates nondeterministic choices in the usual way. Note that unlike a full MDP search tree, many nodes at action levels do *not* involve choices; rather they have only one possible action: the action dictated by the program.

How uncertainty is handled is best dealt with by examining how the interpreter deals with the initial agent actions in a program. We start with the case of a deterministic action.[9]

```
bestDo(A : E,S,Pol,V,Prob,k) :-
 agentAction(A), deterministic(A),
 (not poss(A,S), Pol=stop, Prob is 0, reward(V,S);
  poss(A,S), start(S,T1), time(A,T2), T1 $<= T2,
  bestDo(E,do(A,S),RestPol,Vfuture,Prob,k-1),
  reward(R,S),
  V is R + Vfuture,
  (RestPol = nil, Pol = A ;
   not RestPol=nil, Pol = (A : RestPol)
  )
 ).
```

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 $E$ in situation $(A, S)$. Its value is given by the expected value of this continuation plus the reward in $S$ (action cost for $A$ can be included), 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. Notice that if $A$ is *not* possible at $S$, then the policy is simply the *Stop* action, success probability is zero, and the value is simply the reward associated with situation $S$.

The definition is slightly different when $A$ is stochastic:

```
bestDo(A : E,S,Pol,V,Prob,k) :-
 agentAction(A), nondet(A,S,Outcomes),
 backedUpValue(E,Outcomes,S,RestPol,FV,Prob,k),
 reward(R,S),
 ( RestPol = stop, V = R, Pol = (A : stop) ;
   not RestPol=stop, V is R + FV,
   Pol=(A : senseEffect(A) : (RestPol))
 ).
```

[8] This definition is not complete, but does illustrate the most important concepts.

[9] All of our clauses are defined for horizon $k > 0$; the case of $k = 0$ is straightforward.

In this case, $A$ consists of a stochastic choice from a set of nature's actions: the predicate *nondet*$(A, S, \textit{Outcomes})$ has as its last argument the list of all nature's actions related to stochastic action $A$ (this list is computed from the predicate *stochastic*$(A, S, N)$ given in the action theory). Each of nature's actions gives rise to a new situation, which occurs with a given probability. The predicate *BackedUpValue* (defined below) intuitively considers the values of the optimal execution of the remainder of the program $E$ in *each* of those situations, and defines the expected value and success probability of the remainder of the program based on the evaluation of each of these situations. Total expected value is given by adding the current reward to the expected value of $E$, and the optimal policy is given by $A$, followed by the action *senseEffect*$(A)$—which determines which of the possible outcomes was actually realized—together with the optimal policy for $E$.

*BackedUpValue* is straightforward in the case where the list of nature's actions contains a single outcome.

```
backedUpValue(E,[AN],S,Pol,V,Prob,k) :-
 poss(AN,S), start(S,T1), time(AN,T2),
 T1 $<= T2, prob(AN,PA,S),
 bestDo(E,do(AN,S),PolA,VA,ProbA,k-1),
 V is PA * VA, Prob is PA * ProbA,
 senseCondition(AN,W),
  ( PolA = nil, Pol= ?(W) ;
    not PolA=nil, Pol= (?(W) : PolA)
   ).

backedUpValue(E,[AN],S,Pol,V,Prob,k) :-
 not poss(AN,S), Pol=stop, Prob is 0, reward(V,S).
```

The standard recursion over the list *Outcomes* when it contains more than one element yields the policy and allows us to compute the total probability of branches that lead to successful execution and the expected utility of all branches:

```
backedUpValue(E,[A,B | L],S,Pol,V,Prob,k) :-
 backedUpValue(E,[A],S,PolA,VA,ProbA,k),
 backedUpValue(E,[B | L],S,Tree,VT,ProbT,k),
 V is VA + VT, Prob is ProbA + ProbT,
 (PolA=stop, Tree=stop, Pol=stop ;
  (PolA\=stop; Tree\=stop), Pol=(PolA # Tree)
 ).
```

The expected utility and termination probability are determined using the probability of each outcome, and the policy corresponding to each outcome has that outcome's sense condition placed as the first action (essentially guarding against execution of that branch unless that outcome in fact occurred).

With this in hand, the definition applied to complex program-forming operators can be given:

```
bestDo(?(C) : E,S,Pol,V,Prob,k) :-
 holds(C,S), bestDo(E,S,Pol,V,Prob,k) ;
 holds(-C,S), Prob is 0, Pol=stop, reward(V,S).

/* nondeterministic choice of an argument */
bestDo(pi(X,E1) : E,S,Pol,V,P,k) :-
 sub(X,_,E1,E2),  /*E2 results from substitution of a value for X*/
 bestDo(E2 : E,S,Pol,V,P,k).

/* 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;
        greatereq(V1,P1,V2,P2), Pol=Pol1, P=P1, V=V1).

bestDo(if(C,E1,E2) : E,S,Pol,V,Prob,k) :-
 holds(C,S), bestDo(E1 : E,S,Pol,V,Prob,k) ;
 holds(-C,S), bestDo(E2 : E,S,Pol,V,Prob,k).

bestDo(while(C,E1) : E,S,Pol,V,Prob,k) :-
  holds(-C,S), bestDo(E,S,Pol,V,Prob,k) ;
  holds(C,S),  bestDo(E1 : while(C,E1) : E,S,Pol,V,Prob,k).

bestDo(ProcName : E,S,Pol,V,Prob,k) :- proc(ProcName,Body),
                 bestDo(Body : E,S,Pol,V,Prob,k).
```

Of special interest is the clause involving nondeterministic choice between two complex actions `E1` and `E2`. Given the choice between two subprograms, the optimal policy is given by that subprogram with optimal execution. The $\leq$ predicate is used to compare the expected values and success probabilities of the two alternatives. The standard argument choice construct offered by Golog does not guarantee that an optimal argument will be chosen above. For this reason, in DTGolog we provide a new decision-theoretic construct *pickBest*$(X, F, E)$, whose execution requires that program $E$ be executed using the best argument value from the finite range of values $F$ to instantiate argument $X$:

```
bestDo(pickBest(X,F,E1) : E,S,Pol,V) :-
  range(F,R),  /* R is the list of values */
  bestDoAll(X,R,E1 : E,S,Pol,V).

bestDoAll(X,[D],E,S,Pol_D,V_D) :-
  sub(X,D,E,E_D),  /* Substitute value D for X in E; get E_D */
  bestDo(E_D,S,Pol_D,V_D).

bestDoAll(X,[D1, D2 | R],E,S,Pol,V) :-
  sub(X,D1,E,E1), bestDo(E1,S,Pol1,V1),
  bestDoAll(X,[D2 | R],E,S,Pol2,V2),
  (lesseq(V1,Pol1,V2,Pol2), Pol=Pol2, V=V2 ;
   greatereq(V1,Pol1,V2,Pol2), Pol=Pol1, V=V1).
```

The optimal policy is the policy that corresponds to the best value of the argument varied over a finite number of alternatives; the interpreter finds it using the $\leq$ predicate to compare pairwise the expected values and success probabilities of all alternatives.

Space precludes a detailed discussion, but we can provide a formal account in a fairly abstract setting of the notion of an optimal policy constrained by a program, and show that our definitions correspond to this notion of optimality. Intuitively, we view a policy as a mapping from situation terms into action choices. A program imposes constraints on possible mappings, dictating specific actions except where nondeterministic choices appear. An optimal policy is simply an extension of the mapping dictated by the completion of the program with highest expected value.

## 5 Robot Programming

One of the key advantages 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 in a natural fashion (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 can allow for the scheduling of complex behaviors that are difficult to preprogram. It can also obviate the need to reprogram a robot to adapt its behavior to reflect environmental changes or changes in an objective function. 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 [6, 23], 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 her office, she acknowledges the receipt of 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.

Our robot is provided with the following simple DTGolog program:

**while** $(\exists n \; horizon(n) \wedge n < k)$
  *pickBest*$(p,\ people,$
    $(\neg status(p, Out) \wedge$ $\exists n$ *mailPresent(p,n)* $\wedge$ $\neg$*hasMail*$(p))?$ ;
      $\pi t\,[deliverTo(p,t)]$ )
**endWhile**

Intuitively, this program chooses people from the list `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, delivering 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 in the program so that only people who are not known to be out of the office and who have items to be delivered are actually considered. Interestingly the `status` fluent (denoting whether a person is in their office) allows us to mimic limited partial observability: it can take on the value *unknown* in which case delivery is attempted. If the person is out (delivery failure is sensed), their status becomes out (so no future deliveries are attempted). The robot's prior over the status of a given person (whether in or out) is reflected in the transition probabilities for the *status* fluent.

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.

Note that full MDP planning can be implemented within the DTGolog interpreter by running it with the program that allows *any* (feasible) action to be chosen at *any* time. This

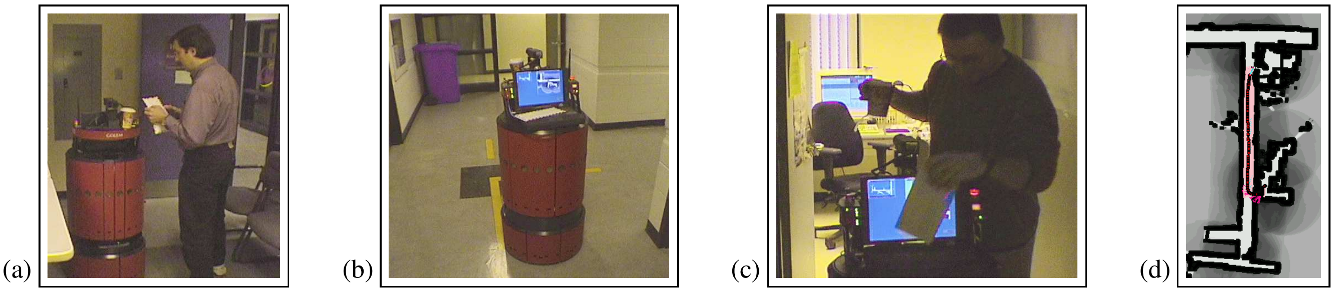

Figure 1: Mail delivery: (a) A person loads mail and coffee onto the robot. (b) DTGolog sends the robot to an office. (c) The recipient accepts the mail and coffee, acknowledging the successful delivery by pressing a button. (d) The map learned by the robot, along with the robot's path (from the main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

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.

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 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 a minimal of 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. These include: integrating efficient algorithms and other techniques for solving MDPs into this framework (dynamic programming, abstraction, sampling, etc.); 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 [21]; and automatically generating sense conditions for stochastic actions.

## References


[1] Fahiem Bacchus, Joseph Y. Halpern, and Hector J. Levesque. Reasoning about noisy sensors in the situation calculus. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1933–1940, Montreal, 1995.

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

[3] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1–2):81–138, 1995.

[4] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[5] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[6] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 1999.

[7] Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89:219–283, 1997.

[8] Hector Geffner and Blai Bonet. High-level planning and control with incomplete information using pomdps. In *Proceedings Fall AAAI Symposium on Cognitive Robotics*, Orlando, FL, 1998.

[9] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, 1999.

[10] Sven Koenig and Reid Simmons. Real-time search in nondeterministic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1660–1667, Montreal, 1995.

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

[12] Shieu-Hong Lin and Thomas Dean. Generating optimal policies for high-level plans with conditional branches and loops. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 187–200. IOS Press, Amsterdam, 1996.

[13] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[14] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In M. Kearns M. Jordan and S. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press, Cambridge, 1998.

[15] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):261–325, 1999.

[16] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):7–56, 1997.

[17] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

[18] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In L.C. Aiello, J. Doyle, and S.C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 2–13. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[19] R. Reiter. Sequential, temporal GOLOG. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 547–556. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

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

[21] Mikhail Soutchanski. Execution monitoring of high–level temporal programs. In *Robot Action Planning, Proceedings of the IJCAI-99 Workshop*, Stockholm, 1999.

[22] Richard S. Sutton. Td models: Modeling the world at a mixture of time scales. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 531–539, Lake Tahoe, Nevada, 1995.

[23] S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A second generation mobile tourguide robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1999.