Planning and Programming with First-Order Markov Decision Processes: Insights and Challenges

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

1 Introduction

Markov decision processes (MDPs) have become the *de facto* standard model for decision-theoretic planning problems. However, classic dynamic programming algorithms for MDPs [22] require explicit state and action enumeration. For example, the classical representation of a value function is a table or vector associating a value with each system state; such value functions are produced by iterating over the state space. Since state spaces grow exponentially with the number of domain features, the direct application of these models to AI planning problems is limited. Furthermore, for infinite and continuous spaces, such methods cannot be used without special knowledge of the form of the value function or optimal control policy.

As a consequence, much MDP research in AI has focussed on representations and algorithms that allow complex planning problems to be specified concisely and solved effectively. Techniques such as function approximation [2] and state aggregation [3] have proven reasonably effective at solving MDPs with very large state spaces. These methods can be viewed as either imposing or automatically discovering regularities in the value function or control policy.

Another class of approaches investigated in the AI community can be viewed as integrating planning with programming [28, 18, 7]. These models provide a system designer with the ability to specify partial control programs or constraints on behavior that circumscribe the decision problem faced by an agent in such a way as to make it tractable. Intuitively, programming paradigms for MDPs allow a system designer to encode *a priori* knowledge about the structure of an optimal policy easily, while allowing the agent to fill in the missing pieces of the policy that the designer cannot predict.

Unfortunately, most methods investigated to date are designed to work with unstructured representations or *propositional* representations of MDPs. Specifically, little work has addressed *first-order structure* in decision-theoretic planning problems. Many (if not most) realistic planning domains and decision problems are best represented in first-order terms, exploiting the existence of domain objects, relations over those objects, and the ability to express objectives and action effects using quantification. Existing MDP algorithms can only be applied to these problems by grounding or "propositionalizing" the domain.[1] Unfortunately such an approach is impractical: the number of propositions grows very quickly with the number of domain objects and relations, and even relatively simple domains can generate incredibly large numbers of propositions when grounded. Furthermore, expressing objectives and system dynamics using such propositionalized representations is generally unwieldy and unnatural. For instance, the fact that there should be a part of type A at Plant 27 is most naturally expressed in first-order terms: $\exists p.loc(p, Plant27) \wedge type(p, A)$. In contrast, the propositional version is rather obscure and cumbersome: $loc(P_1, Plant27) \vee loc(P_2, Plant27) \vee \ldots \vee loc(P_n, Plant27)$, where $P_1, \ldots, P_n$ range over the parts of type A.

[1] This assumes a *finite domain*: if the domain is infinite, these algorithms cannot generally be made to work.

In this extended abstract, I describe a line of research that attempts to generalize existing techniques for both automated planning and the integration of planning with programming to representations of MDPs that take advantage of the first-order structure of state and action spaces. After some brief background on MDPs in Section 2, I begin by describing one approach the first-order representation of MDPs in Section 3. I then describe in Section 4 how this representation can be used in classic dynamic programming algorithms to obviate the need for explicit state and action space enumeration when constructing optimal policies and value functions. This technique can be viewed as a decision-theoretic extension of standard first-order regression. In Section 5, I describe a simple programming language for first-order MDPs (FOMDPs) that allows a system designer to describe various aspects of a control policy: these may include constraints on the form of a (possibly, though not necessarily) optimal policy, or components (subroutines) that can be pieced together to form such a policy. The aspects of the policy left unspecified (through a nondeterministic choice operator) are filled in optimally by the control agent through the usual decision-theoretic techniques. Specifically, I describe a simple search procedure used to construct the optimal *completion* of the program. Finally, I will discuss a number of interesting challenges facing the use of this paradigm for designing intelligent agents, including ongoing research on the integration of dynamic programming techniques with programs, and the use of dynamic programming techniques to "compile" program fragments that are reused.

Much of this abstract describes these techniques at an intuitive level. I refer to the reader to the cited papers to obtain more technical detail.

2 Markov Decision Processes

We begin with the standard state-based formulation of MDPs. We assume that the domain of interest can be modeled as a fully-observable MDP [1, 22] with a finite set of states S and actions A. Actions induce stochastic state transitions, with $\Pr(s, a, t)$ denoting the probability with which state t is reached when action a is executed at state s. We also assume a real-valued reward function R, associating with each state s its immediate utility $R(s)$.[2]

A *stationary policy* $\pi : S \rightarrow A$ describes a particular course of action to be adopted by an agent, with $\pi(s)$ denoting the action to be taken in state s. The decision problem faced by the agent in an MDP is that of forming an *optimal policy* that maximizes expected total accumulated reward over an infinite horizon (i.e., the agent acts indefinitely). We compare policies by adopting *expected total discounted reward* as our optimality criterion, wherein future rewards are discounted at a rate $0 \leq \gamma < 1$, and the *value of a policy* π, denoted $V_\pi(s)$, is given by the expected total discounted reward accrued, that is, $E(\sum_{t=0}^{\infty} \gamma^t R(s^t)|\pi, s)$. Policy π is *optimal* if $V_\pi \geq V_{\pi'}$ for all $s \in S$ and policies π'. The *optimal value function* V^* is the value of any optimal policy.

Value iteration [1] is a simple iterative approximation algorithm for constructing optimal policies. It proceeds by constructing a series of n-stage-to-go *value functions* V^n. Setting $V^0 = R$, we recursively define n-stage-to-go *Q-functions*:

$$Q^n(a, s) = R(s) + \left\{ \gamma \sum_{t \in S} \Pr(s, a, t) \cdot V^{n-1}(t) \right\} \tag{1}$$

and value functions:

$$V^n(s) = \max_a Q^n(a, s) \tag{2}$$

The Q-function $Q^n(a, s)$ denotes the expected value of performing action a at state s with n stages to go and acting optimally thereafter. The sequence of value functions V^n produced by value iteration converges linearly to V^*. For some finite n, the actions that maximize Eq. (2) form an optimal policy, and V^n approximates its value. We refer to Puterman [22] for a discussion of stopping criteria.

[2] We ignore actions costs for ease of exposition. These impose no additional complications on our model.

The definition of a Q-function can be based on any value function. We define $Q^V(a, s)$ exactly as in Eq. (1), but with arbitrary value function V replacing V^{n-1} on the right-hand side. $Q^V(a, s)$ denotes the value of performing a at state s, then acting in such a way as to obtain value V subsequently.

MDPs provide a very general model for decision-theoretic planning problems in AI (as well as for many other types of stochastic sequential decision problems in OR, economics, etc.). However, as we see above the classic presentation of MDPs requires that one explicitly enumerate state space, providing an $|S| \times |S|$ transition matrix for each action a, and a reward vector of dimension $|S|$. Furthermore, standard solution techniques, such as value iteration, require $O(|S|)$ expected value and maximization calculations at each step of the algorithm. Since we are often interested in planning problems whose state space is defined by a set of propositional variables, S will generally be too large to enumerate explicitly, causing problems for both the representation and effective solution of MDPs.

3 A First-order Representation of MDPs

Several representations for propositionally-factored MDPs have been proposed [13, 9, 3], which allow one to take advantage of the fact that actions often have independent effects on different state variables, and that these effects do not vary at each state, but instead depend only on the values of a small number of state variables. In happy circumstances, these representations allow us to specify MDPs in size linear in the number of state variables (thus in size $O(\log |S|)$), and are arguably quite natural. Unfortunately, they are unable to handle first-order concepts, which essentially induce large numbers of propositions through the use of predicates and domain objects.

First-order representations have been proposed for MDPs, including those of Poole [20], and Geffner and Bonet [11]. In this paper we describe the first-order, situation calculus MDP representation developed by Reiter [23] and Boutilier *et al.* [7, 6], which has several unique features that make it useful for planning and programming. We provide only an overview of this language and representational methodology, referring to the papers mentioned above for full details.

The situation calculus [15] is a first-order language for axiomatizing dynamic worlds, whose basic ingredients consist of *actions*, *situations* and *fluents*. A *situation* is a first-order term denoting a sequence of actions. These 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. In a blocks world, the situation term

$$do(stack(A, B), do(putTbl(B), do(stack(C, D), S_0)))$$

denotes the sequence of actions

$$[stack(C, D), putTbl(B), stack(A, B)].$$

Relations whose truth values vary from state to state are called *fluents*, and are denoted by predicate symbols whose last argument is a situation term. For example, $BIn(b, Paris, s)$ is a relational fluent meaning that in that state reached by performing the action sequence s, box b is in Paris.

A domain theory is axiomatized in the situation calculus with various types of axioms [19]. Here we describe only *successor states axioms*. There is one such axiom for each 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 [25]. For example, the following axiom:

$$BIn(b, c, do(a, s)) \equiv (\exists t)[TIn(t, c, s) \wedge a = unloadS(b, t)] \vee (BIn(b, c, s) \wedge \neg(\exists t)a = loadS(b, t))$$

asserts that box b is in city c after doing action a in situation s iff a is the action of (successfully) unloading b from a truck t located at c, or if the box was in c prior to a being executed and a is not the action of

successfully loading b onto some truck.

Before moving to the extension of the situation calculus to MDPs, we review a concept that will prove useful a bit later. The *regression* of a formula ψ through an action a is a formula ψ' that holds prior to a being performed iff ψ holds after a. Successor state axioms support regression in a natural way. Suppose that fluent F's successor state axiom is $F(\vec{x}, do(a,s)) \equiv \Phi_F(\vec{x}, a, s)$. We inductively define the regression of a formula whose situation arguments all have the form $do(a,s)$ as follows:

$Regr(F(\vec{x}, do(a,s))) = \Phi_F(\vec{x}, a, s)$

$Regr(\neg\psi) = \neg Regr(\psi)$

$Regr(\psi_1 \wedge \psi_2) = Regr(\psi_1) \wedge Regr(\psi_2)$

$Regr((\exists x)\psi) = (\exists x)Regr(\psi)$ Thus the regression of a formula $\psi(\vec{x}, do(a,s))$ is a formula $\psi'(\vec{x}, s)$ denoting some property of situation s.

The situation calculus can be extended to deal with stochastic actions by adopting the following trick [7, 6, 23]: each stochastic action is decomposed into deterministic primitives under nature's control—nature chooses the deterministic action that actually gets executed, with some specified probability, when an agent performs a stochastic action. We then formulate situation calculus domain axioms using these deterministic choices, as above. The only extra ingredients are the specification of possible nature's choices associated with each (stochastic) agent action, and the probability with which each choice is made in a given situation. For instance, the stochastic *unload* action can succeed (denoted by *unloadS*) or fail (*unloadF*):

$choice(load(b,t), a) \equiv a = unloadS(b,t) \vee a = unloadF(b,t)$

and the probabilities associated with each choice might vary with the weather:

$prob(unloadS(b,t), unload(b,t), s) = p \equiv Rain(s) \wedge p = 0.7 \vee \neg Rain(s) \wedge p = 0.9$

$prob(unloadF(b,t), unload(b,t), s) = 1 - prob(unloadS(b,t), unload(b,t), s)$

Generally speaking, we can view this model as providing a form of state space abstraction: when actions have the same effect at different states, this representation describes that effect only once as a function of the relevant state *properties*. Thus complex state dynamics can be expressed quite concisely. Not surprisingly, reward functions can also be expressed compactly if reward depends on only a small number of (first-order) properties of the state; for instance, we might have:

$R(s) = r \equiv (\exists b)BIn(b, Paris, s) \wedge r = 10 \vee \neg(\exists b)BIn(b, Paris, s) \wedge r = 0$

Obviously, when solving an MDP, one can represent a value function (or a policy) in a similar fashion, associating with a value (or an action choice) with certain first-order representable state *properties* rather than with states directly. We will exploit this fact in the next section when describing how dynamic programming can be used to solve first-order MDPs without explicit state space enumeration.

4 Decision-theoretic Planning with First-order MDPs

The first-order representation of MDPs in the situation calculus can be exploited to great effect when solving an MDP using value iteration. Logical descriptions exploiting regularities in value functions (and policies) can be very compact. Specifically, we consider logical representations of a value function V in which the state space is partitioned using a collection of logical formulae $\{\beta_1(s), \cdots, \beta_M(s)\}$, each formula β_i associated with a specific value v_i. Intuitively, any situation s satisfying condition β_i has value $V(s) = v_i$.

An FOMDP can be solved very effectively if the logical structure of value functions can be discovered through inference using the the logical MDP specification, with expected value computations performed once per abstract state (i.e., property β_i) instead of once per state. Thus a dynamic programming algorithm that works directly with logical representations of value functions offers great potential computational ben-

efit. In this section, I provide an intuitive overview of the notion of decision-theoretic regression (DTR) [5], and the first-order version of DTR developed in [6], forming the basis of a *first-order value iteration* algorithm. I refer to [6] for a detailed description of the approach as well a detailed logical derivations that prove the soundness of the technique.

Suppose we are given a logical description of value function V of the form:

$$V(s) = v \equiv \bigvee_{i \leq M} \{\beta_i(s) \wedge v = v_i\}$$

We assume that the β_i logically partition state space, and contain as their only situation term variable s. Each β_i can be viewed as an *abstract state.* The *first-order decision theoretic regression (FODTR)* of V through action type $A(\vec{x})$ is a logical description of the Q-function $Q^V(A(\vec{x}), s)$. In other words, given a set of abstract states corresponding to regions of state space where V is constant, we wish to produce a corresponding abstraction for $Q^V(A(\vec{x}), s)$. We will now describe prescribe precisely how to obtain such a logical description from the logical description of $V(s)$ together with the axiomatization of the FOMDP.

Let $A(\vec{x})$ be a stochastic action with corresponding nature's choices $\{n_j(\vec{x}) : j \leq N\}$. $Q^V(A(\vec{x}), s)$ is defined classically as

$$Q^V(A(\vec{x}), s) = R(s) + \gamma \cdot \sum_{t \in S} \Pr(s, A(\vec{x}), t) \cdot V(t)$$

Since different successor states arise only through different nature's choices, the situation calculus analog of this is:

$$Q^V(A(\vec{x}), s) = R(s) + \gamma \cdot \sum_j \mathit{prob}(n_j(\vec{x}), A(\vec{x}), s) \cdot V(\mathit{do}(n_j(\vec{x}), s))$$

Now assume that the functions $R(s)$ and $\mathit{prob}(n, A(\vec{x}), s)$ are all described in the logical form described above. Specifically, let:

$$R(s) = r \equiv \bigvee_{i \leq R} \{\xi_i(s) \wedge r = r_i\}$$

where the ξ_i partition state space, and

$$\mathit{prob}(n_j(\vec{x}), A(\vec{x}), s) = p \equiv \bigvee_{i \leq P} \{\phi_i(s) \wedge p = p_j^i\}$$

where the ϕ_i partition state space. Intuitively, in any situation satisfying ϕ_i, the probability with which nature chooses $n_j(\vec{x})$ (when $A(\vec{x})$ is executed by the agent) is p_j^i.

Given this information, the Q-function $Q^V(A(\vec{x}), s)$ can be constructed logically by making the following observations.

- Notice that $Q^V(A(\vec{x}), s)$ can vary with the conditions $\xi_i(s)$, since for the different $\xi_i(s)$, the reward obtained can vary.
- Notice that the probability with which a specific nature's choice occurs varies with the conditions ϕ_i, which can also cause $Q^V(A(\vec{x}), s)$ to vary.
- Suppose $A(\vec{x})$ has been executed by the agent. Once a specific nature's choice $n_j(\vec{x})$ is made, the effect of $n_j(\vec{x})$ is deterministic and leads with certainty to a specific successor state satisfying exactly one of the formulae β_k determining future value. However, the specific β_k that results from the execution of $n_j(\vec{x})$ depends, as expected, on properties of the state in which $n_j(\vec{x})$ is executed. Specifically, by the definition of regression, we know that $\beta_k(\mathit{do}(n_j(\vec{x}), s)) \equiv \mathit{Regr}(\beta_k(\mathit{do}(n_j(\vec{x}), s)))$.

  Here $\mathit{Regr}(\beta_k(\mathit{do}(n_j(\vec{x}), s)))$ is a formula that refers only to properties of s (the situation prior to the action being performed).

At any situation, if $Regr(\beta_k(do(n_j(\vec{x}), s)))$ holds, then should nature choose $n_j(\vec{x})$ when $A(\vec{x})$ is executed, we know β_k will hold in the resulting situation. Let $\delta : \{1 \ldots N\} \mapsto \{1 \ldots M\}$ be a mapping associating with each nature's choice n_j ($j \leq N$) a value-determining formula β_k ($k \leq M$). Let Δ be the set of all such mappings. For any $\delta \in \Delta$, the formula

$$\bigwedge_{j \leq N} Regr(\beta_{\delta(j)}(do(n_j(\vec{x}), s)))$$

uniquely determines the value function formula that will hold after the choice of *any* n_j; specifically, if n_j is chosen by nature, $\beta_{\delta(j)}$ will hold after its execution.

Putting these components together, we can show that for any $\xi_i(s)$, $\phi_k(s)$, and $\delta \in \Delta$, if

$$\xi_i(s) \wedge \phi_k(s) \bigwedge_{j \leq N} Regr(\beta_{\delta(j)}(do(n_j(\vec{x}), s)))$$

holds, the the Q-value of action $A(\vec{x})$ is

$$r_i(s) + \gamma \sum_{j \leq N} p_j^k v_{\delta(j)}$$

Thus if we consider all combinations probability formulae, reward formulae, and conditions under which nature's choices lead to different value formulae (in the representation of $V(s)$), we obtain all of the logical conditions under which $Q^V(A(\vec{x}), s)$ may vary. As a result, we can produce a logical description of $Q^V(A(\vec{x}), s)$ having the form

$$Q^V(A(\vec{x}), s) = q \equiv \bigvee_{i \leq Q} \{\alpha_i(s) \wedge q = q_i\}$$

for some set of formulae $\alpha_i(s)$ that partition state space and corresponding Q-values q_i.

The number of formulae resulting from this repartitioning may be quite large; however, many of these formulae will be inconsistent and can be eliminated from the description of the Q-function. Practical implementation of this scheme requires nontrivial logical simplification as well [6].

As an example consider value function

$$V(s) = v \equiv \exists b.BIn(b, Rome, s) \wedge v = 10 \vee \neg\exists t.BIn(b, Rome, s) \wedge v = 0$$

That is, if some box b is in *Rome*, value is 10; otherwise value is 0. Suppose that reward R is identical to V and our discount rate is 0.9. We use the *unload*(b, t) action, described above, to illustrate FODTR. The regression of V^0 through *unload*(b, t) results in a formula (after simplification) denoting $Q^V(unload(b, t), q, s)$ with four elements:

$\alpha_1(b, t, s) \equiv \exists b' BIn(b', Rome, s)$

$\alpha_2(b, t, s) \equiv Rain(s) \wedge TIn(t, Rome, s) \wedge On(b, t, s) \wedge \neg\exists b' BIn(b', Rome, s)$

$\alpha_3(b, t, s) \equiv \neg Rain(s) \wedge TIn(t, Rome, s) \wedge On(b, t, s) \wedge \neg\exists b' BIn(b', Rome, s)$

$\alpha_4(b, t, s) \equiv (\neg TIn(t, Rome, s) \vee \neg On(b, t, s)) \wedge \exists b' BIn(b', Rome, s)$

and the associated Q-values: $q_1 = 19$; $q_2 = 6.3$; $q_3 = 8.1$; $q_4 = 0$. Notice that this method produces a Q-function for *all instances* of the action (i.e., for any instantiation of arguments b, t). Thus we obviate the need for explicit action enumeration as well (though all action *types* must be dealt with individually).

First-order DTR can be used as the basis of a first-order value iteration algorithm. Intuitively, one uses the technique described above to produce logical descriptions of Q-functions for each action. Then a logical description of the maximization over all action types is produced to form a value function for the next stage. We refer to [6] for details. To illustrate the form of a value function produced in this way, we show the optimal value function for a fully specified logistics example involving some of the components described above:

$\exists b BIn(Paris, b, s) : 10$

$\neg Rain(s) \wedge \exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) \wedge \neg\exists b BIn(Paris, b, s) : 5.56$

$Rain(s) \wedge \exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) \wedge \neg\exists b.BIn(Paris, b, s) : 4.29$

$\neg Rain(s) \wedge \exists b, tOn(b, t, s) \wedge \neg\exists b BIn(Paris, b, s) \wedge \neg\exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) : 2.53$

$\neg Rain(s) \wedge \exists b, t, c(BIn(c, s) \wedge TIn(c, s)) \wedge \neg\exists b, tOn(b, t, s) \wedge \neg\exists b BIn(Paris, b, s) : 1.52$

$Rain(s) \wedge \exists b, tOn(b, t, s) \wedge \neg\exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) \wedge \neg\exists b BIn(Paris, b, s) : 1.26$

$\neg\exists b BIn(Paris, b, s) \wedge \neg\exists b, tOn(b, t, s) \wedge [Rain(s) \vee \neg\exists b, t, c(BIn(c, s) \wedge TIn(c, s)) : 0.0$

We emphasize again that this value function applies no matter how many domain objects there are. Thus it avoids the problem of state space enumeration, producing a compact description of the value function in a computationally tractable fashion, and providing an intuitive description of value function as well. Though we don't provide details, we note that extracting an optimal policy from this value function can also be effected by logical means.

5 Integrating Planning and Programming

Another way in which the complexity of decision problems is sometimes managed is by restricting the choices available to the decision making agent. In AI, in fact, many "intelligent agents" do not make decisions at all! In robotics, for example, it is generally the case that a robot's behavior is explicitly programmed by its designer, with no planning or decision making to left to the agent itself. The difficulty of designing sophisticated controllers for such agents is alleviated by the provision of 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. In a sense, this viewpoint transforms the computational burden of decision making for the agent into a difficult conceptual chore faced by the agent's designer.

Recent research in the AI community has focussed on bringing together the planning and programming paradigms. Within the MDP framework, this research often takes the guise of *macroactions* or *local policies* that can be provided to an agent. These local policies dictate behavior (leaving no decisions to the agent) in a local fashion (for example, over some bounded spatial or temporal region), but allow the agent to decide which of these local behaviors to invoke at any given time. Thus the system designer provides partial programs (local policies) and the agent does some planning (piecing together these programs). A number of interesting questions arise in such a model. For instance, if an agent is to make decisions regarding the selection of these component behaviors, it must have a model of the effects and rewards associated with them: how does one construct such a model efficiently? If an agent is to act (close to) optimally, the space of possible local behaviors must not be so restrictive as to rule out optimal global behavior. How one designs a set of local policies that is small enough to make the agent's decision problem manageable and to permit tractable construction of local models, but large enough to achieve the desired flexibility of behaviors is a fundamental question. A number of researchers have begin to provide answers to these and other questions [26, 12, 8, 30, 21, 29, 16, 17, 10]

In this section, I describe the DTGolog framework [7], a model that combines the planning and programming perspectives in FOMDPs, 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, DTGolog allows 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.

The DTGolog model is based on the Golog programming language [14]. Golog 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:* α ; β. 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 [24].

As mentioned, one way to circumvent planning complexity in MDPs 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., [18, 28]) 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 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. DTGolog is 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 have found, 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.

We assume that the specification of an FOMDP has been provided using the situation calculus methodology described above. A DTGolog program is written using the standard Golog language. 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.*[3]

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

The interpreter is specified in detail in [7]. Intuitively, the interpreter builds a decision tree rooted at the initial situation, with chance nodes corresponding to each stochastic action (with each branch labeled with a corresponding nature's choice, and its probability of being chosen), and choice nodes corresponding to each choice point in the program itself. The power of the program lies in the fact that it reduces the search tree dramatically by excluding choice points where the behavior of the agent is dictated by the program. As a result, solving an MDP constrained by a DTGolog program is generally much more efficient that solving its unconstrained counterpart.

What makes DTGolog valuable is the fact that it is often very natural to impose partial constraints on agent behavior reflecting a programmers prior knowledge about the structure of an optimal policy or possible component behaviors. The language itself allows the natural specification of such constraints. 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, for example. 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.

In [7], we report on some simple experiments involving an office delivery robot with a program of the form:

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, for example, as people changes their preferences, or have their priorities altered). 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.

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

[3]There is some subtlety involved in the definition of optimal execution. Since program execution can fail (e.g., due to a failed test condition or action precondition), an agent can choose to complete the program in such a way that the program is guaranteed to terminate abnormally, or fail with higher probability than some other completion. To a certain extent, the "advice-giving" interpretation of the program suggests that an agent should not be allowed to cause failure of the program; while the "planning" nature of the agent suggests it should be permitted to maximize expected reward even if this causes the program terminates abnormally. This tradeoff is discussed in detail in [7].

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, as we discuss in the next section).

6 Some Challenges

A number of interesting research directions remain to be explored in the use of FOMDPs and the integration of planning and programming within the FOMDP/DTGolog framework. There are, of course, the usual things one can say about optimizations, improved implementations, and experimentation. Another interesting avenue is the use of various forms of approximation within these models. Some forms of approximation have already been explored in the DTGolog model, for instance, the work of Soutchanski on a bounded-lookahead, online DTGolog interpreter [27]. Other forms of approximation, such as *approximate* decision-theoretic regression [4, 5], can be applied in a relatively straightforward way first-order decision-theoretic regression.

The integration of first-order decision-theoretic regression with DTGolog is something we have been exploring in some detail recently. One problem with the DTGolog interpreter described above is its reliance decision-tree expansion to compute optimal program completions. A dynamic programming approach could provide a much more effective way of computing optimal program completions (without reliance on finite lookahead or a specific initial situation)—as long as state and action space enumeration can be avoided We have begun to develop a model to do just this, exploiting first-order decision theoretic regression. The basic idea is to expand the state space of the MDP so that it includes system states as well as the *program state*. The program state corresponds to the state of a finite-state machine that encodes the program.[4] At any compound state $\langle m, s \rangle$, where m is a machine state and s is a system state, the agent has no choices if the machine state m dictates a specific action. If the program allows the agent to nondeterministically choose an action at that point, the agent makes a choice (a transition in the machine) corresponding to the (system) action it implements. This expanded MDP can be encoded compactly using standard successor state axioms, and thus can be solved using first-order decision-theoretic regression. The resulting policy will ground all choice points in the program, constructing an optimal policy subject to the constraints of the program. The use of an expanded MDP in this way recalls the model proposed by Parr and Russell [18], in which an MDP policy is encoded using a finite-state machine directly. Three critical differences between our model and theirs: (a) they assume that behavior is specified using a finite-state machine directly, whereas our behavior is constrained using an—arguably more natural—Golog program; (b) the nature of our state machine is that machine transitions occur in lockstep with system transitions, which obviates the use of semi-Markov models or related devices to account for asynchrony; and (c) our model can be solved without explicit state space enumeration, exploiting the first-order representation of the original MDP.

One final important research direction is the development of techniques to produce "precompiled" models of program fragments. As mentioned earlier, much work has been directed toward the use of local policies in solving MDPs. Part of this work has focussed on the development of "macro models" of local policies that describe the transition probabilities and expected accumulated reward associated with implementing a specific fragment of behavior. With such a model in hand, a local policy can be treated as if it were a single action, and an MDP can be solved using the behavior without reasoning about its "internal structure". This is especially important when this behavior can be invoked repeatedly during the execution of a policy (or reused in multiple problem-solving episodes). The ability to construct such models of DTGolog procedures

[4] In a sense, we can view this is a stripped down version of a program counter in which no machine (counter) transition occurs unless a system state transition occurs as well.

(or other program fragments) would allow completions of DTGolog programs to be solved much more efficiently. Unlike existing models, our aim is to build these models using regression techniques. The goal would be to construct a set of successor state axioms in which procedures and program fragments play the role of actions: thus we would describe their effects on domain fluents rather than on states themselves.

Acknowledgements

The work described in this paper is the product of several collaborations. In particular, the work overviewed in Section 4 is described in further detail in C. Boutilier, R. Reiter, B. Price, "Symbolic Dynamic Programming for First-Order MDPs," *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, Seattle, 2001 (to appear). The work discussed in Section 5 is described in C. Boutilier, R. Reiter, M. Soutchanski, S. Thrun, "Decision-Theoretic, High-level Agent Programming in the Situation Calculus," *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pp.355–362, Austin, TX (2000).

References

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

[2] Dimitri P. Bertsekas and John. N. Tsitsiklis. *Neuro-dynamic Programming*. Athena, Belmont, MA, 1996.

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

[4] Craig Boutilier and Richard Dearden. Approximating value trees in structured dynamic programming. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 54–62, Bari, Italy, 1996.

[5] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121:49–107, 2000.

[6] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, 2001. to appear.

[7] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 355–362, Austin, TX, 2000.

[8] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*. Morgan-Kaufmann, San Mateo, 1993.

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

[10] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomp osition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[11] 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.

[12] Leslie Pack Kaelbling. Hierarchical reinforcement learning: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 167–173, Amherst, MA, 1993.

[13] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1073–1078, Seattle, 1994.

[14] Hector J. Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: a logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[15] 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, pages 410-417.

[16] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 165–172, Madison, WI, 1998.

[17] Ronald Parr. Flexible decomposition algorithms for weakly coupled Markov decision processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 422–430, Madison, WI, 1998.

[18] 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.

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

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

[21] Doina Precup, Richard S. Sutton, and Satinder Singh. Theoretical results on reinforcement learning with temporally abstract behaviors. In *Proceedings of the Tenth European Conference on Machine Learning*, pages 382–393, Chemnitz, Germany, 1998.

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

[23] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 2001.

[24] Ray Reiter. Sequential, temporal GOLOG. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning*, pages 547–556, Trento, 1998.

[25] 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.

[26] Satinder Pal Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[27] Mikhail Soutchanski. On-line decision-theoretic golog interpreter. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, 2001. to appear.

[28] 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.

[29] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. *Artificial Intelligence*, 112:181–211, 1999.

[30] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, Cambridge, MA, 1995. MIT Press.