This paper was selected by a process of
anonymous peer reviewing for presentation at

COMMONSENSE 2007

8th International Symposium on Logical Formalizations of Commonsense Reasoning

Part of the AAAI Spring Symposium Series, March 26-28 2007,
Stanford University, California

Further information, including follow-up notes for some of the
selected papers, can be found at:

www.ucl.ac.uk/commonsense07

Resolving Non-Determinism in Programs for Complex Task Planning with Search Control

Alfredo Gabaldon
National ICT Australia,
School of CS and Eng., UNSW
Sydney, Australia,
alfredo@cse.unsw.edu.au

Abstract

We consider the problem of planning in complex domains where actions are stochastic, non-instantaneous, may occur concurrently, and time is represented explicitly. Our approach is based on the situation calculus based language Golog. Instead of general search for a sequence of actions, as in classical planning, we consider the problem of computing a deterministic, sequential program (with stochastic actions as primitives) from an underspecified, non-deterministic, concurrent program. Similar to the search for a plan, the process of obtaining a deterministic program from a non-deterministic one is carried out offline, with the deterministic program obtained by this process then being available for online execution. We then describe a form of domain-dependent search control that can be used to provide some degree of goal-directedness to the search for solutions in this setting, and also show how a simple programming construct for probabilistic tests can be used for further prunning of the search space.

Introduction

We consider the problem of planning in complex domains where actions have stochastic outcomes, have a non-instantaneous duration, may execute concurrently, and time is represented explicitly. Planning in such domains is a challenge to the current state-of-the-art in planning (Bresina *et al.* 2002). Recent developments in response to this challenge have been reported. Mausam & Weld (Mausam & Weld 2004) extend Markov Decision Processes (MDPs) to allow concurrency. Little et al. (Little, Aberdeen, & Thiébaux 2005) present a planner for actions with probabilistic effects and non-instantaneous actions. Little & Thiébaux (Little & Thiébaux 2006) extend the Graphplan approach for probabilistic, concurrent (but instantaneous) actions. Mausam & Weld (Mausam & Weld 2005) extend concurrent MDPs with durative actions.

In this paper we consider this problem in the framework of the situation calculus Basic Action Theories of (Reiter 1991) and the complex action programming languages Golog (Levesque *et al.* 1997), ConGolog (De Giacomo, Lesperance, & Levesque 2000), and their variants. These action theories have a higher expressive power than the above planning frameworks. For instance, in these theories the set of outcomes of a stochastic action and their corresponding probabilities may vary according to the state where the action occurs. As far as we know, in all of the probabilistic planning frameworks mentioned above, the possible set of outcomes and their probabilities are fixed and context independent. On the other hand, a more restricted class of problems allows some planning systems to gain computationally.

Moreover, we consider the problem from the more general perspective of solving the following: given a high-level non-deterministic program δ, a formula ϕ and a probability lower-bound p, compute a sequence of stochastic actions such that 1) it represents one of the possible ways in which the execution of δ may proceed, and 2) its execution will result in a state where formula ϕ holds with probability at least p. If the program δ used is a planning procedure, then we will be solving a planning problem.

The motivation behind taking the above perspective, however, is to be able to take advantage of certain types of control knowledge that can be used to inform the search for solutions. Control knowledge has been successfully used in some classical planning systems, e.g., (Bacchus & Kabanza 2000; Kvarnström & Doherty 2000). These two systems in particular use declarative control knowledge in the form of linear-temporal logic formulas to prune the search space of their forward-chaining algorithms. A similar approach has also been applied with Golog-based planners (Reiter 2001; Gabaldon 2003). In this work, however, we consider a form of control that allows one to engage some of the goal-directedness typical of regression-based planners, while still building solutions in a forward-chaining manner. This, however, does not preclude the possibility of combining this form of control with that used in the previous approaches.

Preliminaries

We briefly review the main components of Reiter's Basic Action Theories and of action programming languages.

Basic Action Theories

These theories are dynamic domain axiomatizations in the situation calculus (sitcalc) (McCarthy & Hayes 1969). Actions, situations, and objects are first-class citizens with corresponding sorts. Fluents, actions and situations, are the main ingredients of a sitcalc formalization of a dynamic domain. Intuitively, fluents are the properties of the domain

that change when actions occur, and situations are sequences of actions that represent possible ways in which the domain may evolve. A special constant S_0 denotes the initial situation. Other situations can be constructed as terms formed with the reserved function symbol do.

A Basic Action Theory $\mathcal{D}$ consists of the following set of axioms (lower-case letters denote variables, which are implicitly prenex universally quantified when they appear free. $\vec{x}$ denotes a vector of variables $x_1, \ldots, x_n$):

1. For each action function $A(\vec{x})$, an **Action Precondition Axiom** of the form: $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$, where s is the only term of sort situation in formula $\Pi_A(\vec{x}, s)$.
2. For each fluent $F(\vec{x}, s)$, a **Successor State Axiom** of the form: $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where s is the only term of sort situation in formula $\Phi_F(\vec{x}, a, s)$.
3. A set of axioms describing the initial sitate of the world.
4. The **Foundational Axioms** Σ which axiomatize situations in terms of the constant S_0 and the function do.

This set of axioms is for deterministic actions. Stochastic actions are formalized in this setting by defining them in terms of deterministic ones representing the possible outcomes. The outcomes are thus standard actions and are first-class citizens in the sitcalc language, while stochastic actions are defined through abbreviations as described below.

The possible outcomes of a stochastic action $A(\vec{x})$ are described by an abbreviation of the form:

$$choice(A(\vec{x}), c) \stackrel{\text{def}}{=} c = O_1(\vec{x}) \vee \ldots \vee c = O_k(\vec{x}).$$

The probability that executing a stochastic action $A(\vec{x})$ results in the execution of outcome $O(\vec{x})$ is given by a sentence denoted by $prob_0(O(\vec{x}), A(\vec{x}), s)$. Note that the probability that executing a stochastic action result in a particular outcome depends on the situation where the stochastic action is executed. This means that the probabilities may vary according to the circumstance at the time of execution. For full details on representing stochastic actions and extending actions theories with explicit time, as we shall use them here, we refer the reader to (Reiter 2001).

Action Programming Languages

The action programming languages Golog (Levesque *et al.* 1997), ConGolog (De Giacomo, Lesperance, & Levesque 2000) and its variants provide Algol-like programming constructs for defining complex actions on top of an underlying basic action theory as defined above. Without going into the formal details, we briefly review some of the constructs available in these languages.

The following are some of the constructs provided in Golog:

- Test action: $\phi?$. Test whether ϕ is true in the current situation.
- Sequence: $\delta_1; \delta_2$. Execute program δ_1 followed by program δ_2.
- Nondeterministic action choice: $\delta_1 | \delta_2$. Nondeterministically choose between executing δ_1 or executing δ_2.
- Nondeterministic choice of arguments: $(\pi x)\delta$. Choose a value for x and execute δ with such a value for x.
- Procedure definitions: **proc** $P(\vec{x})$ δ **endProc** . $P(\vec{x})$ being the name of the procedure, $\vec{x}$ its parameters, and program δ its body.

ConGolog extends Golog with a number of constructs. The most relevant for our purposes is the following:

- concurrent execution: $\delta_1 \| \delta_2$.

Intuitively, the execution of a complex action $\delta_1 \| \delta_2$ results in the execution of one of the possible interleavings of the primitive actions that result from δ_1 with those from δ_2. If there is no interleaving that is executable (the preconditions of each action are satisfied at the appropriate moment), then the execution of $\delta_1 \| \delta_2$ fails.

A semantics for these languages is captured through a relation $Do(\delta, s, s')$, meaning that executing the program δ in situation s results in situation s'. When stochastic actions are included, an additional argument p is included denoting the probability of arriving at situation s'. Given a background theory $\mathcal{D}$ including an axiomatization of this relation, executing a program δ in a situation s is defined as the problem of finding a sequence of actions $\vec{\alpha}$ such that

$$\mathcal{D} \models Do(\delta, s, do(\vec{\alpha}, s)).$$

Representing Tasks and Plans

As mentioned above, we are interested in domains that involve actions that are stochastic, have a duration, and may execute in parallel. In this section we describe our formalization of such actions. Since these actions are not primitive constructs in the framework, we will refer to them as "tasks" to distinguish them from the primitive, deterministic actions.

The description of a task consists of several items. As a *name* for a task, we will use a term $Tsk(\vec{x})$. The name of a task thus includes a vector $\vec{x}$ of parameters similar to the arguments of a standard primitive action. Since we will formalize tasks in the situation calculus, we will formalize them as complex actions defined in terms of primitive ones. Thus, the description includes a set of primitive actions $oTsk_i(\vec{x})$ called *outcomes*. The preconditions and effects of a task are determined and vary according to each outcome. For each outcome o of a task tsk, we must provide an action precondition axiom, plus a probability specified by means of sentences $prob_0(o, tsk, s)$. The duration of the task is specified by means of macro:

$$duration(tsk, s) = d \stackrel{\text{def}}{=} \Delta(tsk, d, s).$$

Given a description of a task, the next step is to specify how tasks execute. The standard approach to modeling durative actions in the situation calculus is to model them as *processes*. This is done by means of a start action, an end action and a fluent that holds only while the process is occurring, i.e., during the time between the occurrence of the start and the end actions.

Since we use terms of the situation calculus as names of tasks, we can use a single start and end action for all tasks by including the task name as an argument. A second argument for these actions will be the occurrence time: the action term $sT(tsk, t)$ denotes the start action and the term $eT(tsk, t)$

the end action, where tsk is a task, and t is the temporal argument.

The start action $sT(tsk, t)$ is stochastic: it results in the execution of one of the outcomes of tsk. The end action we will make deterministic here, but it could be made stochastic if necessary by providing another set of outcomes. So we specify the start action for each task $Tsk(\vec{x})$ by means of a $choice$ macro:

$$choice(sT(Tsk(\vec{x}), t), c) \stackrel{\text{def}}{=} c = oTsk_1(\vec{x}, t) \vee \ldots \vee c = oTsk_k(\vec{x}, t).$$

The "plans" we intend to compute consist of sequences of task start and end actions. Since the start and possibly also the end action is not primitive, instead of sequences of primitive actions, i.e., situations, we will compute sequential Golog programs.

Definition 1 A *sequence of actions* (or simply, *sequence*) is a Golog program $\alpha_1; \alpha_2; \ldots; \alpha_n$, where each α_i is a start action $sT(Tsk(\vec{x}), t)$ or an end action $eT(Tsk(\vec{x}), t)$.

Clearly, several different situations (sequences of primitive, deterministic outcome actions) can result from executing a sequence, each with a corresponding probability.

The probability that a formula ϕ holds after executing a sequence σ can be defined as follows:

$$probF(\phi, \sigma) \stackrel{\text{def}}{=} \sum_{\{(p,s')|Do(\sigma,p,S_0,s') \wedge \phi[s']\}} p$$

Programming and Planning with Tasks

The encoding of tasks and the definition of sequence in hand, we can intuitively describe the problem we are interested in solving as follows: given a non-deterministic program, a goal and a number p, compute a sequence that achieves the goal with probability at least p and such that the sequence corresponds to one of the possible ways in which the non-determinism in the program may be resolved. The following definitions make this precise.

Definition 2 A *task execution problem* is a triple (δ_N, ϕ, p) where δ_N is a Congolog program whose primitive actions are of the form $sT(tsk, t)$ and $eT(tsk, t)$, ϕ is a situation suppressed formula, and p a real number s.t. $0 < p \leq 1$.

The following definition describes when a sequence is a solution to a task execution problem.

Definition 3 A sequence δ_D is a *solution* for the task execution problem (δ_N, ϕ, p), iff

1. $\mathcal{D} \models (\exists s').Do(\delta_D, S_0, s')$, i.e. δ_D is executable;
2. $\mathcal{D} \models (\forall s').Do(\delta_D, S_0, s') \supset Do(\delta_N, S_0, s')$;
3. $probF(\phi, \delta_D) \geq p$.

If the program δ_N happens to implement a planning algorithm, then the problem (δ_N, ϕ, p) is a probabilistic planning problem similar to how (Kushmerick, Hanks, & Weld 1995) defines it. It is in fact not difficult to write a simple planner for tasks using a subset of the Golog constructs. Figure 1 shows one such planner. It takes a bound n on the number of tasks a plan may contain and performs a depth-first search.

proc $dfplan(n)$
  $G?$ |
  $[n > 0?; (\pi\ tsk)sT(tsk, timeNow); dfplan(n-1)]$ |
  $[(\pi\ tsk, t_{end})(t_{end} = duration(tsk) + timeNow)?$;
    $eT(tsk, t_{end})$; $dfplan(n)]$
endProc

Figure 1: A simple bounded forward-search temporal task planner in stGolog.

The planner can choose between starting a task, which is followed by a recursive call with a decreased bound, or ending one of the tasks currently executing, which is followed by a recursive call with the same bound. Parameter n bounds task usage, not plan length, so the planner will consider all possible interleavings of n tasks: plans where the execution of all n tasks overlaps (provided preconditions allow that, of course), plans where the execution is completely sequential, and all the possibilities in between. Parameter n can of course be used to conduct a search in a manner similar to iterative-deepening.

Conducting a blind search for a solution in this manner is computationally too onerous, especially considering that tasks are allowed to execute in parallel. In classical planning, one alternative that has been employed with great success is the use of declarative search control (Bacchus & Kabanza 2000). This approach has been used in the situation calculus with the simple Golog planning procedures in (Reiter 2001) and it would be easy to use it with the planner in Fig. 1. We do not explore this alternative in this paper though. Instead, in a later section we consider two additional and complementary ways to exert search control.

Resolving Non-Determinism in Programs

We turn now to the question of computing a sequence that solves a problem (δ_N, ϕ, p). This task is similar to the task carried out by the interpreters of Golog and its relatives: given a program, the interpreters compute a situation that corresponds to a successful execution trace. In our case, instead of a sequence of primitive actions, we want to compute a sequence as in Definition 1. Thus we define an operator below that takes a non-deterministic program δ_N and computes sequences that are deterministic instances of δ_N. In the following subsection we discuss another difference regarding test actions.

Tests

Given a task execution problem, we want to use the probabilities to make all decisions offline while computing a solution. Although the extensions of Golog that allow stochastic actions include tests, it seems useful to reconsider how tests are handled. In stGolog, for example, the semantics of programs is defined in such a way that every execution trace either satisfies all tests, or if a test fails, the program terminates without executing any remaining actions. For the problem we are interested on, this seems to be too strong, as

it can lead to the elimination of programs that actually are likely enough to be successful, and in some cases can lead to too pessimistic estimates of the probability of success.

We thus introduce a new construct for testing that a formula holds with a lower bound probability: $lbp(\phi, p, \sigma)$, where σ is a sequence and $0 \leq p \leq 1$. The definition of this probabilistic test is as follows:

$$lbp(\phi, p, \sigma) \stackrel{\text{def}}{=} probF(\phi, \sigma) \geq p.$$

Similar to situation suppressed formulas, when $lbp(\phi, p, \sigma)$ appears in programs, it will do so with parameter σ suppressed, which will be replaced with the "current sequence" when computing a solution, as shown below. We still use standard tests $\phi?$ but only for situation independent ϕ.

Computing a Sequence

Given a non-deterministic program δ_N, the task is to obtain a deterministic program δ_D for online execution. We define such a computation by means of a relation $Det(\delta_N, \delta_D)$ that intuitively holds if δ_D is one of the deterministic versions of δ_N (Theorem 1 below makes this precise). We define this relation in terms of a slightly more involved relation that intuitively defines the computation step-by-step in terms of program pairs: $Det(\delta_N, \delta_D, \delta'_N, \delta'_D)$ intuitively means that the pair (δ'_N, δ'_D) is one of the possible results of applying one step of the computation to the pair (δ_N, δ_D). The computation would normally start with $\delta_D = nil$ and terminate with a $\delta_N = nil$. We use this approach for two reasons: first, in order to resolve a test in δ_N, we need access to the deterministic δ_D that corresponds to the segment of δ_N that precedes the test. Second, the Congolog construct $\|$ seems to require this kind of transition semantics. Relation $Det(\delta_N, \delta_D, \delta'_N, \delta'_D)$ is inductively defined as follows:

$$
\begin{array}{l}
Det(\alpha, \delta_D, \delta'_N, \delta'_D) \equiv (\delta'_N = nil) \wedge (\delta'_D = \delta_D; \alpha), \\
Det(\phi?, \delta_D, \delta'_N, \delta'_D) \equiv (\delta'_N = nil) \wedge (\delta'_D = \delta_D) \wedge \phi, \\
Det(lbp(\phi, p)?, \delta_D, \delta'_N, \delta'_D) \equiv \\
\quad (\delta'_N = nil) \wedge (\delta'_D = \delta_D) \wedge lbp(\phi, p, \delta_D), \\
Det(\delta_1; \delta_2, \delta_D, \delta'_N, \delta'_D) \equiv \\
\quad (\delta_1 = nil) \wedge (\delta'_N = \delta_2) \wedge (\delta'_D = \delta_D) \vee \\
\quad Det(\delta_1, \delta_D, \delta'_1, \delta'_D) \wedge (\delta'_N = \delta'_1; \delta_2), \\
Det((\pi\, x : R)\delta, \delta_D, \delta'_N, \delta'_D) \equiv Det(\delta|^{\vec{t} \in R}_x, \delta_D, \delta'_N, \delta'_D), \\
Det(\delta_1|\delta_2, \delta_D, \delta'_N, \delta'_D) \equiv \\
\quad Det(\delta_1, \delta_D, \delta'_N, \delta'_D) \vee Det(\delta_2, \delta_D, \delta'_N, \delta'_D), \\
Det(\delta_1 \| \delta_2, \delta_D, \delta'_N, \delta'_D) \equiv \\
\quad (\delta_1 = nil) \wedge Det(\delta_2, \delta_D, \delta'_N, \delta'_D) \vee \\
\quad (\delta_2 = nil) \wedge Det(\delta_1, \delta_D, \delta'_N, \delta'_D) \vee \\
\quad [(\delta_1 \neq nil) \wedge Det(\delta_1, \delta_D, \delta'_1, \delta^*_D) \wedge (\delta'_N = (\delta'_1 \| \delta_2)) \vee \\
\quad\ (\delta_2 \neq nil) \wedge Det(\delta_2, \delta_D, \delta'_2, \delta^*_D) \wedge (\delta'_N = (\delta_1 \| \delta'_2))\ ] \\
\quad \wedge (\delta'_D = \delta_D; \delta^*_D), \\
Det(P(\vec{t}), \delta_D, \delta'_N, \delta'_D) \equiv Det(\delta|^{\vec{x}}_{\vec{t}}, \delta_D, \delta'_N, \delta'_D) \\
\quad \text{where } P \text{ is a procedure defined by} \\
\quad \textbf{proc } P(\vec{x})\ \delta\ \textbf{endProc}\ .
\end{array}
$$

We have modified the construct $(\pi x)\delta$ by including a "range" for variable x, in order to instantiate it with a ground term $\vec{t}$ during the computation of the sequence. This is necessary since it is not possible to make probability of success guarantees while delaying that choice until execution time. One way to implement this is to provide a range for the variable x in the form of a relation $range(Type, V)$ where V is one of a finite number of possible values in the range $Type$. For instance, a set of facts $range(Block, B1)$ could be used describe the range of variables on blocks. This is also the approach used in DTGolog (Boutilier *et al.* 2000), where the same issue arises with the construct π in decision theoretic Golog.

The transitive closure Det^* of Det is defined by a second-order axiom as follows:

$$Det^*(\delta_N, \delta_D, \delta'_N, \delta'_D) \stackrel{\text{def}}{=} (\forall P)[(*) \supset P(\delta_N, \delta_D, \delta'_N, \delta'_D)]$$

where the (*) stands for the conjunction of the implications

$$
\begin{array}{l}
True \supset P(nil, \delta, nil, \delta), \\
Det(\delta_N, \delta_D, \delta''_N, \delta''_D) \wedge P(\delta''_N, \delta''_D, \delta'_N, \delta'_D) \supset \\
\qquad P(\delta_N, \delta_D, \delta'_N, \delta'_D).
\end{array}
$$

Finally, what we are really after can be defined using the following macro:

$$Det!(\delta_N, \delta_D) \stackrel{\text{def}}{=} Det^*(\delta_N, nil, nil, \delta_D).$$

Theorem 1 Let δ_N be a program as in Definition 2. For all δ_D such that $Det!(\delta_N, \delta_D)$, we have that:

1. δ_D is a sequence as in Definition 1,
2. Condition 2 of Definition 3 holds.

So given a task execution problem (δ_N, ϕ, p) all sequences δ_D such that $Det!(\delta_N, \delta_D)$ and $probF(\phi, \delta_D, S_0) \geq p$ are solutions.

We remark that since programs δ_N may contain procedures, which can be recursive, it is possible to write a δ_N that is non-terminating. For such a program there is no δ_D such that $Det!(\delta_N, \delta_D)$.

Search Control

Pruning with Probability Tests

The probability tests $lbp(\phi, p)?$ can of course be used to prune out sequences just as standard tests $\phi?$ prune out some of the situations that can be reached by a Golog program. Probability tests are useful for expressing domain-dependent control information such as "if the probability that ϕ holds is at least p, execute δ_1, otherwise execute δ_2." When this form of control knowledge is available, this construct allows pruning out sequence prefixes early in the search and can thus lead to substantial speed ups. It can potentially lead to exponential savings, as the following theorem shows. Let $P(\delta) = \{\sigma | (\exists \delta') Det^*(\delta, nil, \delta', \sigma)\}$, intuitively, the set of sequence prefixes σ that are valid according to operator Det.

Theorem 2 There exist programs δ_1, δ_2 that are syntactically the same except for the appearance of $lbp(\phi, p)$ tests in δ_1, and such that $P(\delta_1)$ is exponentially smaller than $P(\delta_2)$.

Consider the programs

$$\delta_2 = (A_1|B_1)\ ;\ (A_2|B_2)\ ; \ldots ; (A_n|B_n)$$

$$\delta_1 = (A_1|B_1);\ lbp(F_1, p_1)?; \ldots ; (A_n|B_n);\ lbp(F_n, p_n)?$$

and suppose that each F_i is true after A_i or B_i but not both and that all A_i, B_i are atomic actions. It is easy to see that the size of $P(\delta_2)$ is exponential in n while $P(\delta_1)$ contains n prefixes.

Domain Dependent Search Control

The use of declarative domain-dependent search control has been shown to be a successful approach to making planning more feasible. The planners described in (Bacchus & Kabanza 2000; Kvarnström & Doherty 2000), for instance, use search control expressed in linear temporal logic to control forward search in the space of plan prefixes. This type of control knowledge has also been used for planning in Golog (Reiter 2001; Gabaldon 2003) and can be used in our framework. Here, however, we explore a different and complementary form of search control that is readily available in the real world military operations planning domain we are considering.

The control knowledge we will use comes in two forms. The first one is used to specify that a task tsk *may* achieve a goal ϕ, with the intention that during a search for a sequence that achieves ϕ, the task tsk should be considered before tasks that have not been labeled "may achieve ϕ." We declared this by a statement of the form:

$$mayAchieve(tsk, \phi).$$

The second form of search control statement specifies for a task tsk that it requires a condition ϕ to be established before it executes:

$$requires(tsk, \phi).$$

Given a set of facts of the above forms, we can then define non-deterministic programs for achieving a goal that utilize this control knowledge. We define two procedures: $achieve(\phi, p)$ and $execTsk(tsk, p)$. The first one is invoked to find a sequence to achieve an atomic goal ϕ with probability of success at least p, and is defined as follows:

proc $achieve(\phi, p)$
  $lbp(\phi, p)?$ |
  $(\pi tsk)\{mayAchieve(tsk, \phi)?$;
    $execTsk(tsk, p)$;
    $lbp(\phi, p)?\ \}$ |
  $plan(\phi, p)$
endProc

Non-atomic goals are handled by simple programs: $[achieve(\phi_1, p) \| achieve(\phi_2, p)]$; $(\phi_1 \wedge \phi_2)?$ for a conjunction and $[achieve(\phi_1, p) | achieve(\phi_2, p)]$ for disjunction. Procedure $achieve$ calls $execTsk(tsk, p)$ which is defined as follows:

proc $execTsk(tsk, p)$
  $achieveReq(tsk, p)$;
  $sT(tsk, now)$;
  $(\pi d)\{d = duration(tsk)?$;
    $eT(tsk, now + d)\ \}$
endProc

where $achieveReq(tsk, p)$ is an auxiliary procedure that invokes $achieve(\phi_i, p)$ for all ϕ_i required by tsk.

Procedure $achieve$ involves a non-deterministic choice among three subprograms: 1) successfully test that ϕ is currently true with sufficient probability, 2) execute one of the tasks that may achieve ϕ and check afterwards that ϕ was indeed achieved with sufficient probability, and 3) call a general planning procedure with ϕ as the goal and a lower bound probability of p. An implementation should make sure the last alternative is a last resort, since it is a general planning procedure. The actual planning procedure called here could, for example, be one based on the planner shown in Fig. 1.

Finding a plan for goal ϕ and probability p can then be done simply by computing a sequence δ_D s.t. $Det!(achieve(\phi, p), \delta_D)$. Clearly, any such δ_D is a solution to the problem $(achieve(\phi, p), \phi, p)$. Notice that the conditions required by a task are also achieved with probability at least p. This is an instance of pruning using lbp: sequences where conditions required by a task are not achieved with probability at least p are pruned away since a lower probability means the goal itself will not be achievable with sufficient probability.

The two types of control statements are obviously related to the notions of pre/post-conditions of a program and it is possible to define them logically, for example, as follows:

$$mayAchieve(tsk, \phi) \equiv$$
$$(\exists s, s', p).Do(execTsk(tsk, p), s, s') \wedge p > 0 \wedge \neg\phi[s] \wedge \phi[s']$$

$$requires(tsk, \phi) \equiv$$
$$(\forall s, otsk).choice(otsk, sT(tsk)) \wedge Poss(otsk, s) \supset \phi[s].$$

Of course, it would not make sense to use these definitions directly since proving them is at least as hard as planning. If a complete set of formulas ϕ satisfying the above definitions could be pre-computed, we could eliminate the call to a planner in procedure $achieve(\phi, p)$. However, this also seems too hard: since there are many formulas that satisfy the above conditions, we would need to find those ϕ that, for $mayAchieve$, are strongest (i.e., for all ϕ' satisfying the definition, $\phi \supset \phi'$), and, for $requires$, are weakest ($\phi' \supset \phi$ for all other ϕ'). To make things even more complicated, we would also need to define $mayAchieve$ for combinations of tasks: $mayAchieve(\delta_E, \phi)$ if $\delta_E = (\delta_{T_1} \| \ldots \| \delta_{T_k})$ for a minimal set of tasks T_i s.t. $Do(\delta_E, s, s') \wedge \neg\phi[s] \wedge \phi[s']$ with positive probability. This essentially means we would need to pre-compute task programs for all achievable ϕ.

Another reason for not insisting on the above definitions is that weaker, resp. stronger, formulas for $mayAchieve$, resp. $requires$, can be better heuristics. For example, it may be a good idea not to include a $mayAchieve(tsk, \phi)$ if ϕ is only a side effect of tsk. Similarly, we may want to include $requires(tsk, \phi)$ where ϕ is not a precondition of *all* outcomes of the start action of tsk, but only of the "successful" or "good" start outcomes. For these reasons, we only consider user supplied search control statements. As in the case of the planning systems mentioned above that use control expressed in temporal logic, this type of search control is also frequently and readily available.

Search control expressed in terms of $mayAchieve$ and $requires$ is not very useful in flat domains, such as blocks world with only low level tasks such as $move(x, y)$ and $pickup(x)$. But in domains with a high degree of hierarchical structure among tasks, such as the NASA Rover's domain and the military operations domain described in (Aberdeen, Thiébaux, & Zhang 2004), it can lead to substantial improvement. Experiments in the latter domain show huge payoffs. Our implementation finds an optimal (in terms of prob. of success) plan in 2.5secs (on a PowerBook G4 1.5GHz) terminating the search (without calls to a

general planning procedure from $achieve(\phi, p)$) in 16secs. The LRTDP (Bonet & Geffner 2003) based planner in (Aberdeen, Thiébaux, & Zhang 2004) is reported to have taken 10mins on a cluster of 85 Pentium III 800MHz processors, to find a good policy (optimality cannot be guaranteed). That planner uses domain-dependent heuristics but not search control of the form we discussed. These experiments indicate to us that it is possible and useful to exploit some of the domain structure, even in such complex domains, by means of this form of search control.

Conclusions

In this paper we have described an approach to complex task planning for domains with time, durative actions with probabilistic outcome, and concurrency. Our approach is developed from the point of view of resolving non-determinism in action programs. From this perspective, a problem is specified in the form of a non-deterministic program, a formula, and a probability threshold; and the solutions are deterministic programs in the form of a sequence of stochastic actions. Underlying both types of program is a mathematically solid representation of the background domain in the form of a sitcalc basic action theory. We describe a form of search control that can be used to provide an essentially forward-search planner with some of the goal-directedness typical of regression based planners, and that can result in very substantial improvements. Our approach is related to the work of (Grosskreutz & Lakemeyer 2000), who consider a probabilistic variant of Golog, called pGolog, that includes a construct $prob(p, \delta_1, \delta_2)$, for specifying probabilistic branching between two programs. Their approach is also to compute deterministic versions of non-deterministic programs. Their main goal, however, was to introduce a probabilistic construct for Golog, while here, following (Reiter 2001), we model stochastic actions by using $choice$ in the action theory. This approach is more general since it allows the set of possible outcomes of a stochastic action, and their corresponding probabilities, to depend on the situation where the action is executed, which is not the case in pGolog. (Grosskreutz & Lakemeyer 2000) does not consider search control or probabilistic tests as we have here, but they did consider sensing actions, which we plan to do in future work. The form of search control described above can also be used to prune the search for a policy in DTGolog (Boutilier *et al.* 2000). We plan to experiment with this in the future.

Acknowledgements

We thank Gerhard Lakemeyer for useful discussions on the subject of this paper. National ICT Australia is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

Aberdeen, D.; Thiébaux, S.; and Zhang, L. 2004. Decision-theoretic military operations planning. In *Procs. of ICAPS'04*.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Procs. of ICAPS'03*.

Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. Decision-theoretic, high-level agent programming in the situation calculus. In *Procs. of AAAI'00*, 355–362.

Bresina, J.; Dearden, R.; Meuleau, N.; Smith, D.; and Washington, R. 2002. Planning under continuous time and resource uncertainty: A challenge for AI. In *Procs. of UAI'02*.

De Giacomo, G.; Lesperance, Y.; and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121:109–169.

Gabaldon, A. 2003. Compiling control knowledge into preconditions for planning in the situation calculus. In *Procs. of IJCAI'03*.

Grosskreutz, H., and Lakemeyer, G. 2000. Turning high-level plans into robot programs in uncertain domains. In *Procs. of ECAI'00*

Kushmerick, N.; Hanks, S.; and Weld, D. S. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76:239–286.

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30:119–169.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1–3):59–83.

Little, I., and Thiébaux, S. 2006. Concurrent probabilistic planning in the graphplan framework. In *Procs. of ICAPS'06*

Little, I.; Aberdeen, D.; and Thiébaux, S. 2005. Prottle: A probabilistic temporal planner. In *Procs. of AAAI'05*

Mausam, and Weld, D. S. 2004. Solving concurrent markov decision processes. In *Procs. of AAAI'04*

Mausam, and Weld, D. S. 2005. Concurrent probabilistic temporal planning. In *Procs. of ICAPS'05*

McCarthy, J., and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. Edinburgh University Press. 463–502. Also appears in N. Nilsson and B. Webber (editors), Readings in Artificial Intelligence, Morgan-Kaufmann.

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

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