Planning under Uncertainty as GOLOG Programs

Jorge A. Baier[*] and Javier A. Pinto
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Casilla 306 - Santiago 22 - Chile
Phone: +(56) 2 686 4440 FAX: +(56) 2 686 4444
email: jabaier@ing.puc.cl

Abstract

A number of logical languages have been proposed to represent the dynamics of the world. Among these languages, the Situation Calculus (McCarthy and Hayes 1969) has gained great popularity. The GOLOG programming language (Levesque et al. 1997, Giacomo et al. 2000) has been proposed as a high-level agent programming language whose semantics is based on the Situation Calculus. For efficiency reasons, high-level agent programming privileges programs over plans; therefore, GOLOG programs do not consider planning. In this article we present algorithms that generate conditional GOLOG programs in a Situation Calculus extended with uncertainty of the effects of actions and complete observability of the world. Planning for contingencies is accomplished through two kinds of plan refinement techniques. The refinement process successively increments the probability of achievement of candidate plans. Plans with loops are generated under certain conditions.

Keywords: Cognitive robotics, planning, loop induction, uncertainty.

1 Introduction and motivation

Several logical languages have been proposed to represent the dynamics of the World (Kowalski and Sergot 1986, Allen and Hayes 1985, McDermott 1982, Sandewall 1998, Thielscher 1998). Among these languages, the Situation Calculus (McCarthy and Hayes 1969), a second-order language, has gained great popularity, especially after a simple solution to the frame problem was proposed by Reiter (1991). The original language regards actions as the only entity that may change the world, i.e., change from one situation to another. These actions are atomic, instantaneous, and have deterministic effects. Since these characteristics are somewhat inadequate for real world applications, a number of extensions to the language have been made, such as concurrent actions (Lin and Shoham 1992, Reiter 1996), actions with durations (Baier and Pinto 1998), and continuous change (Pinto 1998, Reiter 1996). Given a Situation Calculus theory, an agent can determine how the world changes after she executes actions in the sense that she can evaluate the truth value of logical formulae after executing an arbitrary sequence of actions.

Stemming from this research, the GOLOG agent programming language (Levesque et al. 1997, Giacomo et al. 2000) has been proposed. GOLOG is a logic-based, interpreted, agent programming language designed to model agents on dynamic worlds at a high abstraction level. The language can be used for diverse applications, such as high-level robot control (Burgard et al. 1998, Lespérance et al. 1999, Boutilier et al. 2000) and intelligent software agents programming (Ruman 1996). The GOLOG language is similar to

[*]Corresponding author.

structured programming languages; the main differences are that the effects and execution of the program can be determined by the agent through a Situation Calculus axiomatisation, and that its execution semantics is defined in logic.

The approach of programming agents — sponsored by the cognitive robotics area — is completely orthogonal to the planning approach. Planning is the process by which an agent looks for a *course of action* to accomplish some *goal*. A goal is something the agent desires to make true. The planning problem has been addressed in different ways. The first works on this area gave rise to the so-called *classical planning*, which relies on several assumptions. Among them is the deterministic effects of actions assumtion, that states that every action has a deterministic effect, which can be determined from the conditions that hold upon the execution of actions. Other assumtions state that the world is perfectly observable, i.e., agent can determine all conditions that hold on states, and that the agent is alone in the world. Variations of the classical planning problem arise by relaxing some of those assumptions. The planning problem is computationally intractable; in fact, classical planning is P-SPACE complete (Bylander 1994). For this reason, the cognitive robotics area favours high-level programming over planning. However, for agents acting in worlds with actions with uncertain effects, the benefits of this approach are in doubt. Programmed agents will certainly be more efficient but somewhat less flexible. Flexibility is a desired property of intelligent agents (Wooldridge 2002).

Classical planning has been subject of study since the beginnings of AI. Nevertheless, the problem of planning in the context of cognitive robotics has had minor development. In a work by Reiter (2001), a simple but efficient classical GOLOG planner is proposed. Plans are generated by a GOLOG program and optimised using search-space pruning as suggested by Bacchus and Kabanza (2000). Boutilier et al. (2000) propose an extension to GOLOG that generates plans as an optimisation of sequences of actions. In this version, actions are regarded to have probabilistic outcomes and the optimisation is made through considering a Markov Decision Process for actions. Using only the Situation Calculus as framework, Finzi et al. (2000) have proposed an open-world classical planner.

In this article, we are concerned with the problem of integrating planning under uncertainty into GOLOG programs. Planning under uncertainty is obtained by allowing actions to have non-deterministic effects. There exist two variants of planning under uncertainty depending on whether or not probability distributions are associated with outcomes of actions. When probabilities are considered, we talk about probabilistic planning under uncertainty.

We are interested in the problem of planning under uncertainty since in many domains the notion of uncertainty of effects of actions is central. Imagine a robot whose objective is to lift a dozen of glasses in order to carry them from one place to another. The action of lifting a glass involves the execution of an extra-logical procedure which, if executed over a particular glass, will have the effect of lifting the glass. Nevertheless, many external conditions may influence upon the real effects of the ‘lifting action’. With a positive probability p, the action could fail because of excess of humidity on the surface of the glass or because the force applied over it was not enough due to automatic control level problems. One could give many different examples in which the notion of uncertainty of effects of actions is crucial; therefore, we think that integrating planning under uncertainty into GOLOG is an important issue which must not be left unexplored.

The planning problem under uncertainty has been widely treated under other formalisms. Most of these planners are based on the STRIPS (Fikes and Nilsson 1971) action representation. The first developed, CNLP (Peot and Smith 1992), considers uncertainty without probabilities. Based on a similar action representation, the BURIDAN (Kushmerick et al. 1995) and C-BURIDAN (Draper et al. 1994) algorithms are proposed for probabilistic planning. The CASSANDRA algorithm (Pryor and Collins 1996) is also based on STRIPS and does not consider probabilities. The Weaver architecture (Blythe 1998) uses the STRIPS action representation to generate conditional plans. It is based upon the PRODIGY (Veloso et al. 1995) classical planning architecture.

Decision-theoretic planning is also a way of doing planning under uncertainty. In this view, Markov Decision Processes (Boutilier et al. 1999) are used to generate plans which are optimal *policies*. A policy is a function that returns the action that the agent must execute given the current state of the world in order to maximise the final utility (reward) of the agent.

With the exception of the work by Boutilier et al. (2000), none of this planners directly generate programs — which could be integrated into GOLOG— nor are they based in a first-order logic representation of the world. In this article we propose algorithms to fill this gap. We will discuss the advantages and limitations of our approach in relation to that of Boutilier et al. (2000) and other approaches in a later section.

This article is structured as follows. In section 2 we describe the Probabilistic Situation Calculus. In section 3 we define an extended version of the GOLOG semantics to handle uncertainty. In section 4 we describe the problem of planning under uncertainty and show two plan refinement strategies for accomplishing planning under uncertainty. We show cases in which programs with loops can be induced with a great impact on the probability of achievement of the goal. In section 5 we discuss limitations of our approach. In section 6 we show related work. We conclude in section 7, sketching directions for extending this work to partial observability of the world.

2 Theoretical framework

In this section we present the Probabilistic Situation Calculus, a variation of the Situation Calculus. The original Situation Calculus (McCarthy and Hayes 1969) is useful to represent worlds in which effects of actions are deterministic. On the other hand, the Probabilistic Situation Calculus gives an account of non-deterministic effects of actions and the probabilities associated to them. The Probabilistic Situation Calculus we present here is a variation of the original by Pinto et al. (2000). For more details on the Situation Calculus, refer to Pirri and Reiter (1999).

2.1 Elements of the language

The Probabilistic Situation Calculus is a many-sorted, first-order logical language extended with induction. We use the sorts $\mathcal{S}$, $\mathcal{I}$, $\mathcal{E}$, $\mathcal{A}$, $\mathcal{F}$, and $\mathcal{D}$ for situations, input actions, outcome actions, actions, fluents, and domain objects, respectively.

Situations represent a snapshot of the world in a given moment plus a history of the evolution of the world. There is a distinguished initial situation S_0, which describes the world before anything has occurred. The world changes from one situation to another as the result of the execution of actions. Once an action is executed over the world, another situation is generated. The term $do(a,s)$ is used to denote the situation which results from the execution of action a in situation s.

Input actions are the set of actions that agents execute in the world, i.e., grabbing an object, tossing a coin, etc. On the other hand, *outcomes* are natural actions — executed by nature — which complement the effects of an input action. An outcome is always associated with an input action. Finally, an action is an ordered pair $\langle i,e\rangle$ consisting of an input action i and an outcome e. To exemplify the notion of action consider the input action ‘flip a coin’. This action has a deterministic effect, ‘coin on the floor’, and two non-deterministic effects: ‘coin landed tails’ and ‘coin landed heads’. In our framework, this action could be modelled by two actions, each having the same input, $\langle flip, tails\rangle$ and $\langle flip, heads\rangle$, where *tails* is an outcome whose effect is ‘coin landed tails’ and *heads* is an outcome whose effect is ‘coin landed heads’. Deterministic actions are modelled by an input with only one possible outcome.

Fluents are functional terms of the language that serve to describe properties of the world. These properties are static within situations but may change between different situations. We use the special predicate $holds \subseteq \mathcal{F} \times \mathcal{S}$ to state that a fluent is true during a situation. For example, the sentence $holds(on(A,B),S_0)$

may be used to establish that object A is over object B in situation S_0 [1]. The *holds* predicate can be straightforwardly extended to *formulae of fluents,* which are like first-order formulae where terms can only be fluents, e.g. $(\forall x)\neg on(x,A)$. Fluents cannot have situation terms as arguments.

The binary predicate symbol $Poss\_i \subseteq I \times \mathcal{S}$ states when an input action is possible in a situation, i.e. $Poss\_i(i,s)$ is true if and only if input action i is possible in situation s. The function $Outcome : I \times \mathcal{E} \times \mathcal{S} \to [0,1]$[2] is such that $Outcome(i,e,s) = p$ if and only if p is the probability that input action i will have outcome e in situation s.

2.2 Foundational axioms

The following axioms are necessary to precisely define the structure of the situations as well as to formalise precisely the notion of probability.

$$(\forall P).(P(S_0) \wedge (\forall a,s)[P(s) \supset P(do(a,s))]) \supset (\forall s)P(s), \tag{1}$$
$$(\forall a_1,a_2,s_1,s_2).do(a_1,s_1) = do(a_2,s_2) \supset a_1 = a_2 \wedge s_1 = s_2, \tag{2}$$
$$(\forall b,b',e,e').\langle b,e\rangle = \langle b',e'\rangle \supset b = b' \wedge e = e'. \tag{3}$$

In short, axioms (1) and (2) force the structure of situations to be a tree. Axiom (3) is a unique names axiom for actions.

The following axioms constrain the *Outcome* predicate to correctly reflect the notion of probability within the language.

$$(\forall i,s).\neg Poss\_i(i,s) \supset (\forall e)\, Outcome(i,e,s) = 0, \tag{4}$$
$$(\forall i,s).\, Poss\_i(i,s) \supset \sum_{e\in\mathcal{E}} Outcome(i,e,s) = 1. \tag{5}$$

Axiom (4) forces that no outcome of an impossible input action will have a positive probability. On the other side, axiom (5) obliges that the sum of probabilities of outcomes of a possible input action is 1. Notice that axiom (5) is not a proper formula of the language, nevertheless, it is intended to represent the standard notion of summation in the real numbers.

The $Poss \subseteq \mathcal{A} \times \mathcal{S}$ predicate is used to specify when an action is possible. It is defined by:

$$(\forall i,e,s).\, Poss(\langle i,e\rangle,s) \overset{\text{def}}{=} Poss\_i(i,s) \wedge Outcome(i,e,s) > 0, \tag{6}$$

thus $Poss(\langle i,e\rangle,s)$ is true, if and only if input action i has an associated outcome e with a positive probability.

We define also the following macros as syntactic sugar:

$$in(a) = i \overset{\text{def}}{=} (\exists e)\, a = \langle i,e\rangle \qquad out(a) = e \overset{\text{def}}{=} (\exists i)\, a = \langle i,e\rangle$$

2.3 Theories of action

When modelling a certain domain it is necessary to construct a theory of action. We illustrate how to construct one by modelling this sample domain:

> “In the world of coins and tables there are two coins and one table. There is one agent in the world who may execute two actions: grab a coin from the table or the floor, and drop the coin causing it to fall on the floor.”

[1]This view of fluents differs from that of Reiter (1991), since this version is reified.

[2]We use $[0,1]$ to refer to the interval of real numbers between 0 and 1.

First, we define the actions and fluents of the domain. We use the terms $grab(x)$ and $drop(x)$ to denote, respectively, the input action of grabbing and dropping coin x. Since we regard $grab(x)$ as a deterministic action, the outcome for $grab(x)$ is $grab(x)$, and the possible outcomes for $drop(x)$ are $tails(x)$ and $heads(x)$. The language constants C_1 and C_2 denote the coins.

The fluent $headsUp(x)$ denotes that x is heads up. The fluents $onfloor(x)$, $ontable(x)$ and $holding(x)$ denote, respectively, that coin x is over the floor, over the table or held by the agent.

A theory of actions Σ is formed by the union of the following sets of axioms:

1. An initial description set, Σ_{S_0}. In this case we suppose both coins are tails up and over the table.

$$holds(ontable(x), S_0) \equiv x = C_1 \lor x = C_2, \tag{7}$$

$$\neg holds(headsUp(x), S_0) \equiv x = C_1 \lor x = C_2, \tag{8}$$

$$\neg holds(onfloor(C_1), S_0) \land \neg holds(onfloor(C_2), S_0). \tag{9}$$

2. A set of precondition axioms, Σ_{prec}, for actions. These axioms define all the necessary and sufficient conditions by which an input action may be executed. In our example, input action $grab(x)$ is possible if x is either on the table or on the floor. Input action $drop(x)$ is possible if the agent is holding coin x.

$$Poss\_i(grab(x), s) \equiv holds(ontable(x), s) \lor holds(onfloor(x), s)$$
$$Poss\_i(drop(x), s) \equiv holds(holding(x), s)$$

3. A set of successor state axioms, Σ_{ssa}, one for each fluent, compiled from effect axioms. These axioms establish all the conditions under which a property holds in a given situation, giving a solution to the *frame problem*. We refer the reader to Reiter (1991) for more details.

   In our example, the successor state axioms are the following:

$$\begin{aligned}
&Poss(a,s) \supset [holds(headsUp(x), do(a,s)) \equiv \\
&\qquad out(a) = heads(x) \lor (holds(headsUp(x), s) \land \neg out(a) = tails(x))], \\
&Poss(a,s) \supset [holds(ontable(x), do(a,s)) \equiv \\
&\qquad holds(ontable(x), s) \land \neg in(a) = grab(x)], \\
&Poss(a,s) \supset [holds(onfloor(x), s) \equiv \\
&\qquad in(a) = drop(x) \lor (holds(holding(x), s) \land \neg in(a) = grab(x))], \\
&Poss(a,s) \supset [holds(holding(x), s) \equiv \\
&\qquad in(a) = grab(x) \lor (holds(holding(x), s) \land \neg in(a) = drop(x))].
\end{aligned}$$

   The first axiom states that a coin x is heads up in situation s if s is a result of executing an action with $heads(x)$ as outcome or if s is not a result of an action with $tails(x)$ as outcome. The reading of the remaining axioms is analogous.

4. A set of axioms for outcomes Σ_{prob}. In our example,

$$Poss\_i(grab(x), s) \supset Outcome(grab(x), grab(x), s) = 1,$$
$$Poss\_i(drop(x), s) \supset Outcome(drop(x), tails(x), s) = 0.5,$$
$$Poss\_i(drop(x), s) \supset Outcome(drop(x), heads(x), s) = 0.5.$$

   The first axiom says that action $grab(x)$ is deterministic. The last two axioms state that $tails(x)$ and $heads(x)$ are equally probable after executing $drop(x)$.

5. Unique name axioms set Σ_{una} for domain objects, fluents, and actions. These axioms are used to distinguish actions and domain objects. In our example:

$$C_1 \neq C_2$$

The rest of the axioms say that actions with different names are in fact different. Note that we distinguish actions belonging to the same sort.

$$heads(x) \neq grab(y) \wedge tails(x) \neq heads(y) \wedge tails(x) \neq grab(y),$$
$$grab(x) \neq drop(y),$$
$$ontable(x) \neq holding(y) \wedge ontable(x) \neq headsUp(y) \wedge headsUp(x) \neq holding(y)$$

3 U-GOLOG: a non-deterministic GOLOG

The GOLOG language is a logic-based, interpreted, robot programming language designed to model dynamic worlds at a high abstraction level. The language has been successfully used in high-level robot control and intelligent software agents programming.

Apart from the natural high-level control use, GOLOG can do *plan verification*. Given a program σ and formula of fluents φ, it can be determined whether φ holds during or after the execution of the program.

Up to the moment, we have knowledge of three extensions to GOLOG in order to represent actions with uncertain effects (Baier 2000, Reiter 2001, Grosskreutz and Lakemeyer 2001). Grosskreutz and Lakemeyer (2001) treat uncertainty explicitly within GOLOG programs and not in the Situation Calculus ontology, i.e., programs simulate action's non-deterministic effects. We describe the extension by Baier (2000) which is very similar to the one by Reiter (2001).

3.1 The GOLOG language definition

The semantics of the language is based on the Situation Calculus, and defines which situations are the result of the execution of a U-GOLOG program. Since situations represent histories of the world, the semantics defines the sequence of actions that may result from the execution of a program.

In a U-GOLOG program we distinguish the following elements:

Primitive actions and test conditions

- Primitive actions, denoted by α (possibly with subscripts). U-GOLOG primitive actions differ from those of the original GOLOG in that primitive actions are not actual actions but inputs executed by the agent. The special action *NoOp* is a primitive action that has no effects over the world.

- ϕ?: test conditions. Here ϕ is a fluent formula. These formulae play the same role in U-GOLOG as boolean expressions in traditional imperative languages. In GOLOG, test conditions can be executed in a particular situation s if and only if the condition holds in the situation situation s. In case a test condition is attempted to be executed in a situation where the condition does not hold, the program fails, i.e., it cannot be logically entailed that the agent will be in any situation after executing that action.

Complex actions They are denoted by the letters σ and δ and can be inductively defined as:

- α, a primitive action, is also a complex action.
- If σ_1 and σ_2 are complex actions, then the following are also complex actions:
  - $(\sigma_1;\sigma_2)$, is a *sequence of actions*. The execution of the sequence corresponds to the execution of σ_1 followed by the execution of σ_2.
  - $\textbf{if}\,\phi\,\textbf{then}\,\sigma_1\,\textbf{else}\,\sigma_2\,\textbf{endIf}$, is a *conditional sentence*. σ_1 is executed if ϕ holds, otherwise, σ_2 is executed.
  - $\textbf{while}\,\phi\,\textbf{do}\,\sigma\,\textbf{endWhile}$: *while loops*. σ is executed while ϕ holds.

It is also possible to define procedures, which should be understood in the same way as in structured programming languages. However, for simplicity, we omit them from our simplified language.

Definition 1 *A* U-GOLOG *program is a complex action* σ*, which corresponds to the main program.*

In the original GOLOG, actions are deterministic and, therefore, the agent has complete control over the effects that her actions have in the world. Under non-deterministic effects of actions, however, the agent has complete control on what she decides to do but incomplete control on the effects of the actions she executes. Thus, the primitive actions of our U-GOLOG programs (i.e., the simple actions denoted by α) are input actions. The real action executed over the world will depend on the outcomes associated to every particular input action.

The semantics of U-GOLOG given by Baier (2000) is strongly based on the CONGOLOG semantics (Giacomo et al. 2000) which defines, primarily, what situations are generated in a single step of execution and, ultimately, what situations are the result of the execution of a program. The ternary predicate *Do* is defined such that $Do(\sigma, s, s')$ is true if and only if program σ may terminate in situation s' if started at situation s. If a program σ cannot be executed in a particular situation s, then $Do(\sigma, s, s')$ is false for all situations s'.

Example 1 Consider the theory of coins and tables, Σ, of section 2.3. Let δ be the U-GOLOG program defined by

$$\delta \overset{\text{def}}{=}$$

$$grab(C_1); drop(C_1); \quad \textbf{if}\,\neg headsUp(C_1)\,\textbf{then}\,grab(C_1); drop(C_1); \textbf{else}\,NoOp$$

Given the semantics of U-GOLOG, δ may end in any of the following situations[3]:

$$S_1 = do([\langle grab(C_1), grab(C_1)\rangle, \langle drop(C_1), heads(C_1)\rangle], S_0)$$
$$S_2 = do([\langle grab(C_1), grab(C_1)\rangle, \langle drop(C_1), tails(C_1)\rangle, \langle grab(C_1), grab(C_1)\rangle, \langle drop(C_1), heads(C_1)\rangle], S_0)$$
$$S_3 = do([\langle grab(C_1), grab(C_1)\rangle, \langle drop(C_1), tails(C_1)\rangle, \langle grab(C_1), grab(C_1)\rangle, \langle drop(C_1), tails(C_1)\rangle], S_0)$$

[3]We use the abbreviation $do([a_1, \ldots, a_n], s)$ to refer to $do(a_n, \ldots, do(a_1, s) \ldots)$.

3.2 Computing probabilities

In probabilistic planning, it is of a central relevance to be able to compute the probability that a given property holds after the execution of a program. This task is essential to achieve the so-called *plan assessment*. Plan assessment consists of computing the probability that a goal holds after the execution of a plan.

In example 1, it would be interesting to be able to compute the probability that $headsUp(C_1)$ holds after the execution of program δ. In order to be able to compute this probability, and any of this kind, we introduce the following:

- The ternary function $ProbS_G$ is such that $ProbS_G(\sigma, s, s')$ is the probability that, if the agent is in situation s, after the execution of σ, it gets to situation s'.

  The following axioms define $ProbS_G$ for primitive actions:

$$\begin{aligned}
&ProbS_G(\alpha, s, s') = Outcome(\alpha, e, s) \equiv Poss(\langle \alpha, e\rangle, s) \wedge s' = do(\langle \alpha, e\rangle, s),\\
&ProbS_G(\phi?, s, s') = 1 \equiv holds(\phi, s) \wedge s = s',\\
&ProbS_G(\phi?, s, s') = 0 \equiv \neg holds(\phi, s) \wedge s = s'.
\end{aligned}$$

  For complex actions we define the following.

$$\begin{aligned}
&ProbS_G((\sigma_1;\sigma_2), s, s') =\\
&\qquad ProbS_G(\sigma_1, s, s'') \times ProbS_G(\sigma_2, s'', s') \equiv Do(\sigma_1, s, s'') \wedge Do(\sigma_2, s'', s')
\end{aligned} \tag{10}$$

$$ProbS_G(\mathbf{if}\,\phi\ \mathbf{then}\,\sigma_1\ \mathbf{else}\,\sigma_2\,\mathbf{endIf}, s, s') = \begin{cases} ProbS_G(\sigma_1, s, s') & \text{iff } holds(\phi, s)\\ ProbS_G(\sigma_2, s, s') & \text{otherwise}\end{cases}$$

$$ProbS_G(\mathbf{while}\,\phi\ \mathbf{do}\,\sigma\ , s, s') = \begin{cases} 1 & \text{iff } \neg holds(\phi, s)\\ ProbS_G(\sigma; \mathbf{while}\,\phi\ \mathbf{do}\,\sigma\ , s, s') & \text{otherwise}\end{cases}$$

- We define the predicate $Prob_G$ such that $Prob_G(g, \sigma, s)$ is the probability that fluent formula g holds after executing program σ in s.

$$Prob_G(g, \sigma, s) = \sum_{s' \in \{s'' | Do(\sigma, s, s'')\}} ProbS_G(\sigma, s, s') \times \mathtt{holds}(g, s'). \tag{11}$$

  where $\mathtt{holds}(g, s')$ is 1 when $holds(g, s')$ is true and 0 otherwise.

Remark 1 *Let δ be defined as in example 1. From axioms for $ProbS_G$, $Prob_G$, it follows that*

$$Prob_G(headsUp(C_1), \delta, S_0) = 0.75$$

A straightforward implementation of a U-GOLOG interpreter from the semantics axioms is given by Baier (2000).

4 Planning under uncertainty

For us, planning is the process by which an agent looks for a *course of action* to accomplish some *goal*. A goal is something the agent desires to make true. In our framework, a goal is a fluent formula. The course of action sought by the agent is known as a *plan*. In our framework, a plan corresponds to a GOLOG program which specifies the actions the agent should take.

We are concerned with the problem of planning under uncertainty. This problem arises when disregarding the complete knowledge about the initial state and deterministic effect of actions assumptions of classical planning. Solutions given to the planning problem by classical planners consist, roughly, by linear sequences of actions (linear plans), i.e. programs with no loops or conditionals. When planning under uncertainty these solutions are clearly insufficient. For example, consider the world of coins and tables of section 2.3 and imagine one wants a plan for having coin C_1 heads up with an probability of, at least, 90%. It is easy to see that no linear sequence of action will achieve the goal with a probability over 0.5.

When considering non-deterministic effects of actions, the planner must identify certain key conditions or *contingencies* under which it may take different courses of action. The recognition of contingencies is done be adding if-then-else constructs to programs. Plans which have an account of these contingencies are known in the literature as *contingent plans*. Furthermore, the recognition of certain patterns in the plans allows to generate loops within the plan.

4.1 An algorithm for complete knowledge

In this section we deal with planning under uncertainty under complete knowledge. This implies that we assume our agents will always know the conditions that are true in the world. We show, however, that the algorithm proposed can cope with uncertainty about the initial state too.

As suggested above, good solutions to the planning problem are U-GOLOG programs instead of linear plans. The following definition formalises our notion of plan.

Definition 2 *Given a Probabilistic Situation Calculus theory of action* Σ*, a* U-GOLOG *program* σ *is a plan for goal G with probability threshold P in situation S if and only if,*

$$\Sigma \models Prob_G(G, \sigma, S) \geq P$$

where G is a formula of fluents and P a real number between 0 and 1.

Following ideas from Green (1969), a plan could be generated by a constructive demonstration of the theorem $\Sigma \models Prob_G(G, \sigma, S) \geq P$. Nevertheless, this approach is quite unfeasible since a theorem prover may explore infinitely many programs before getting to a plan.

Given a certain goal, our algorithm starts generating an initial linear sequence of actions that achieves the goal with a certain positive probability threshold given as parameter, i.e., a *satisfiability* solution. Since this it is a linear sequence of actions, this initial plan can be generated using a classical planning search technique.

[insert figure 1 about here]

Suppose that the first step of our algorithm generates the sequence of inputs $i_1; i_2; i_3; i_4$ as a satisfiability sequence to achieve some particular goal G from situation S_0. Furthermore, suppose that the execution of this sequence leads to the situations shown in figure 1, where leaves labelled with a $\surd$ satisfy the goal, whereas the ones labelled with a $\times$ do not. If the outcomes associated to each input are equally probable, the sequence achieves the goal with a probability of 0.5. Notice that when the program is executing, it may reach some situations in which the remaining sequence of actions will never achieve the goal. All situations labelled with a black-filled circle are in the same case. These are bad situations in the sense that if one executes the remaining sequence in that situation, the goal cannot be reached. Formally, a bad situation is defined as follows.

Definition 3 (Bad Situations for a Goal and a Plan) *A situation s is bad for a goal formula G and a complex action σ if the probability of achieving the goal after executing σ in s is 0, i.e.,*

$$Bad(s,g,\sigma) \overset{def}{=} Prob_G(g,\sigma,s) = 0$$

In the example of figure 1, if we get to S_4 after performing the sequence $i_1;i_2$ it is completely impossible to achieve the goal using the remaining sequence $i_3;i_4$. Therefore, S_4 is bad for goal G and plan $i_3;i_4$, and S_5 is bad for goal G and plan i_4.

The refinement of the linear sequence of actions $i_1;i_2;i_3;i_4$ is done in the following steps. Initially, the agent is in S_0, thus, we say that the set of *current situations* is $\{S_0\}$. This set is used to simulate situations in which the agent may be while executing the actions in the plan.

The algorithm finds no bad situations for plan $i_1;i_2;i_3;i_4$ in the set of current situations ($\{S_0\}$). Then, it generates the set of successor situations for input action i_1, obtaining a new current set with one situation, $\{S_1\}$. Again, since the current set contains no bad situations, the algorithm generates three new current situations resulting from the execution of i_2 in S_1, $\{S_2,S_3,S_4\}$.

The algorithm then determines that there is one bad situation (S_4 in figure 1) in the current set. At this point, the algorithm has detected there is a contingency in which the plan does not work. Here is where the refinement is done. If we are in situation S_4 then we should not continue with the remaining plan $i_3;i_4$. Then, to improve the plan, the agent should identify whether it is in situation S_4 and, if that is the case, a new plan should be generated for the goal. Thus, the algorithm modifies the candidate plan by adding an *if-then-else* construct. The refined plan looks like the following:

```
$i_1$; $i_2$;
if in situation $S_4$ then new plan for $G$ in $S_4$
else recursive refinement of $i_3;i_4$ for the rest of current set
```

We cannot have a condition 'in situation S_4' directly in a GOLOG program, since conditions in a GOLOG program can only be fluent formulae, which do not have any situation terms. Certainly, the properties that hold in S_4 are different from the ones that hold in S_2 and S_3, otherwise the plan $i_3;i_4$ would not fail in S_4. Therefore, there must exist a property (or condition) which makes S_4 different from the others. We call this condition a *discriminating condition* which should be a fluent formula that uniquely identifies situation S_4 from the other situations in the current set. Since the discriminating condition is, in general, an arbitrary formula, the problem of finding such formula is quite complex. For this reason, our algorithms are restricted to find a discriminating *fluent* instead of a formula. Thus, in our example, the algorithm will search for a fluent that is true in S_4 and false in the rest of the current set or vice versa. If fluent F is found to be true in S_4 and false in the rest of situations, then the condition used in the *if-then-else* construct is F.

The 'new plan for G in S_4' is generated by re-invoking the plan refiner in S_4, that is, we do replanning for goal G. In each reinvocation, the refiner is called with a higher *depth level*. This depth level is used to stop replanning and represents how many recursive invocations have been already done. When this variable has exceeded certain top level — given as a parameter — replanning is ignored, and, in turn, recursion is stopped.

[insert figure 2 about here]

For the generation of linear, satisfiability sequences of actions to achieve the goal, we use the FindSeq procedure, shown in figure 2. The version we present in figure 2 is the simplest version one can think of. It uses a breadth-first procedure which stops as soon as it has found a sequence that achieves the goal with the given threshold. See section 5 for a discussion of this.

Figure 2 also shows pseudo-code[4] for a simple, conditional-generating, refinement algorithm, CRefine. Arguments should be understood as input/output arguments as in a PROLOG program. We now describe the algorithm.

CandPlan, an input argument, is a linear sequence of actions, which corresponds to a candidate plan for the goal. It is initialised as an empty plan. *FinalPlan* is an output argument, where the final plan is returned. *CurSits* is the set of *current situations*. The planner assumes it can be in any of the situations in this set. T is a probability threshold used to generate linear sequences of actions to achieve the goal. *Level* is the depth level, used to stop replanning. *Top* is the maximum depth level the algorithm can reach.

When CRefine is invoked, it partitions the set of current situations *CurSits* into two sets: *BadSits* and *GoodSits*. *BadSits* is a set which contains all bad situations in *CurSits*. If *BadSits* is empty, it means that our candidate plan is not bad (in the sense of definition 3) for any of the current situations. Thus, it generates the situations resulting from executing the first action of the sequence *CandPlan* and recursively invokes itself with this new set and the remaining sequence of actions.

If *BadSits* is not empty, it means we can be in bad situations. In this case, if *GoodSits* is not empty, it means that *CandPlan* is good for some situations but bad for others. The algorithm looks for a fluent literal[5] l which is true in *GoodSits* and false in *BadSits*. It then returns as final plan

$$\textbf{if}\, l\, \textbf{then}\, \mathit{PlanForGoods}\, \textbf{else}\, \mathit{PlanForBads},$$

where *PlanForGoods* is the plan returned by a recursive call to CRefine over *BadSits* and *PlanForBads* is obtained from a recursive call to CRefine over *BadSits* and *GoodSits* with *Level* incremented in one.

Observe that *CurSits* is initialised to the *set* of initial situations. Since we may use several situations as the initial current-situations set, the algorithm can do planning with uncertainty of the initial state of the world.

The following remark gives an account of an execution:

Remark 2 *Given the theory of action of section 2.3, the plan returned in FinalPlan by the execution of*

$$\mathsf{CRefine}(\mathit{headsUp}(C_1), \{\}, \mathit{FinalPlan}, \{S_0\}, 0.4, 0, 2)$$

is

```
$grab(C_1); drop(C_1);$
if $headsUp(C_1)$ then $NoOp$; else
  $grab(C_1); drop(C_1);$
  if $headsUp(C_1)$ then $NoOp$; else
    $grab(C_1); drop(C_1);$
  endIf
endIf
```

which achieves the goal with probability 0.875.

The version of CRefine presented is the simplest one can think of. In fact, it does not have any account of probabilities. A probability-aware version has also been implemented. This version, when generating a *PlanForBads*, chooses the most probable bad situation, among situations in *BadSits*, for replanning. In this way, it minimises the probability of failure of the new plan in *BadSits*. This version stops when the situations in *BadSits* have a probability smaller than a parameter. An incremental-refinement algorithm has also been implemented. This version receives a probability threshold as parameter an generates, if possible, a plan that satisfies the threshold by successively refining a candidate plan using the CRefine strategy.

[4]The pseudo-prolog code assumes the procedural semantics of prolog, i.e., when there is no possible demonstration for a sub-goal, the underlying theorem prover backtracks.

[5]A fluent literal l is a fluent or a negation of a fluent.

4.2 Loop induction

From the examples of the previous section it should be obvious that the refinement operators of CRefine will never solve problems such as that of remark 2 in a completely satisfactory way, since they will never return a plan like

$$\textbf{while } \neg headsUp(C_1) \textbf{ do } grab(C_1); drop(C_1) \textbf{ endWhile},$$

which achieves the goal with probability 1. This plan shows that in certain cases loops dramatically increase the probability of achievement of plans.

The plan returned by CRefine in remark 2 suggests that loops could be induced when repeated sequences of if-then-else conditionals appear involving the same body, e.g., in that remark, the $grab(C_1); drop(C_1)$ sequence appears repeatedly. In fact, if CRefine is invoked on the same arguments but replacing Top by 3 we obtain the following program:

```
$grab(C_1); drop(C_1)$;
**if** $headsUp(C_1)$ **then** $NoOp$; **else**
 $grab(C_1); drop(C_1)$;
 **if** $headsUp(C_1)$ **then** $NoOp$; **else**
   $grab(C_1); drop(C_1)$;
   **if** $headsUp(C_1)$ **then** $NoOp$; **else**
     $grab(C_1); drop(C_1)$;
   **endIf**
 **endIf**
**endIf**,
```

which achieves the goal with probability 0.9375. A loop with a body $grab(C_1); drop(C_1)$ can be induced because, along the execution of the program, the same conditions hold before every execution of the body. This body can be seen as a *retry sequence* since its repetition increases the probability that the goal is achieved. Each time the body is executed two things can occur: the body achieves certain key condition, after which the program may continue (positive case) or the body needs to be re-executed (negative case). For inducing loops from the detection of this pattern, it is necessary to assure that the state of the world in the negative cases is always the same. In our example, this means that all conditions must be the same after any execution of $grab(C_1); drop(C_1)$, in negative cases. Therefore, before specifying loop induction we have to precisely define the notion of *state*.

Although the concept of state is inherent in most other approaches that address the planning problem, it is not part of the ontology of the Situation Calculus. A state is the set of properties that hold in a given situation. A state can be properly defined if and only if the theory of action is *definitional* for the predicate *holds*, i.e., a theory that assumes the closed world assumption on the *holds* predicate. The theory of action of section 2.3 can be made definitional if axioms (7)-(9) are replaced by:

$$holds(f, S_0) \equiv f = ontable(x) \wedge (x = C_1 \vee x = C_2),$$

which defines all the cases in which *holds* is true in the initial situation. This condition essentially serves to assure that the predicate *holds* will have the same extent in all models of the theory.

Definition 4 *The state of a situation s is*

$$\mathsf{state}(s) = \{f \mid holds(f, s)\}$$

We now define the refinement operator, which takes as input a program and a situation and induces a program with loops[6].

$$
\begin{aligned}
\mathsf{LRefine}(\delta,s,\delta') \overset{\text{def}}{=} \\
&\delta = \sigma_p;\sigma_\ell;\textbf{if } l \textbf{ then } \sigma_t \textbf{ else } \sigma_\ell;\sigma_t \,\textbf{endIf} \wedge \\
&\delta' = \sigma_p;\sigma_\ell;\ \textbf{while } \neg l \textbf{ do } \sigma_\ell \textbf{ endWhile};\sigma_t \wedge \\
&(\forall s')\,(Do(\sigma_p,s,s') \supset ((\forall s'')\,Do(\sigma_\ell;\neg l?,s',s'') \supset \mathsf{state}(s') = \mathsf{state}(s''))) \\
&\qquad\qquad\qquad \vee \\
&\delta = \sigma_p;\sigma_\ell;\textbf{if } l \textbf{ then } \sigma_t \textbf{ else } \sigma_\ell;\textbf{while } \neg l \textbf{ do } \sigma_\ell \textbf{ endWhile};\sigma_t \,\textbf{endIf} \wedge \\
&\delta' = \sigma_p;\sigma_\ell;\textbf{while } \neg l \textbf{ do } \sigma_\ell \textbf{ endWhile};\sigma_t \wedge \qquad (12)\\
&(\forall s')\,(Do(\sigma_p,s,s') \supset ((\forall s'')\,Do(\sigma_\ell;\neg l?,s',s'') \supset \mathsf{state}(s') = \mathsf{state}(s''))) \\
&\qquad\qquad\qquad \vee \\
&\delta = \sigma_p;\ \textbf{if}\, l\, \textbf{then}\, \sigma_1;\sigma_t\, \textbf{else}\, \sigma_2;\sigma_t \,\textbf{endIf} \wedge \\
&\delta' = \sigma_p;\ \textbf{if } l \textbf{ then } \sigma_1 \textbf{ else } \sigma_2 \,\textbf{endIf};\sigma_t \\
&\qquad\qquad\qquad \vee \\
&\delta = \delta'.
\end{aligned}
$$

The operator defines four cases[7]. The first case is a simple loop induction from a conditional construct. If a conditional $\sigma_p;\sigma_\ell;$**if** l **then** σ_t **else** $\sigma_\ell;\sigma_t$ **endIf** appears in a program, then it was generated by a CRefine refinement on the sequence $\sigma_p;\sigma_\ell;\sigma_t$. This means that the probability of success of the plan is increased when σ_ℓ is repeated when $\neg l$ holds after executing $\sigma_p;\sigma_\ell$. If, moreover, the state of the world after executing σ_p is the same that the state of situations in which l does not hold and that result after executing $\sigma_p;\sigma_\ell$ (negative case), then a loop can be induced.

The second case, induces a loop from a loop nested within a conditional construct. This is useful when loops have already been induced inside a conditional. The third case is used to join different branches of the program. Finally, the last case states that a refinement may leave a program unchanged.

We say that the LRefine operator is *state conservative* in the sense that after the execution of the loop body, the world is always in the same state if the loop condition does not hold.

The following theorem supports the benefits of loop induction.

Theorem 1 *Let s be any situation and $\sigma = \sigma_p;\sigma_l;\sigma_t$ a plan found by* CRefine *for a goal g in s such that g succeeds with a positive probability p if f holds after the execution of $\sigma_p;\sigma_l$, and fails otherwise. Furthermore, suppose that every situation s^- that results from the execution of $\sigma_p;\sigma_l$ is such that when fluent f does not hold* $\mathsf{state}(s^-) = \mathsf{state}(s)$. *Then the refinement of σ, $\sigma^+ = \sigma_p;\sigma_l;$***while** $\neg f$ **do** σ_l **endWhile**;σ_t *achieves goal g with probability p, which is an upper bound to any* CRefine *refinement for σ.*

PROOF: Let r ($0 < r < 1$) be the probability that f holds after executing $\sigma_p;\sigma_l$. Observe that upon the first refinement of CRefine, the program $\sigma_1 = \sigma_p;\sigma_l;$**if** f **then** σ_t **else** $\sigma_l;\sigma_t$ is generated. By definition, in each execution path of the program, the state resulting after executing both $\sigma_p;\sigma_l$ and σ_1 are the same when f does not hold after their execution. Thus, the probability that goal g is true after σ_1 is $rp + (1-r)rp$. It is easy to show by induction that the probability p_j that goal holds after a j^{th} refinement of σ is $p_j = rp\sum_{i=0}^{j}(1-r)^i$.

On the other hand σ^+ is equivalent to the program generated by infinitely many CRefine refinements; therefore, it achieves the goal with probability $p^+ = rp\sum_{i=0}^{\infty}(1-r)^i = p$. Furthermore, it is easy to see that $p_j < p$, for all natural j. □

[6]In this definition l stands for a fluent literal.

[7]We have omitted some cases which are permutations of the then and else bodies.

The direct consequence of this theorem is that loop induction always yields better plans in case retry sequences appear within the plan.

One important property to be determined about a plan is the probability that the goal holds after its execution. According to axiom (11), in order to determine the probability that any formula holds after the execution of a program with loops it would be necessary to generate all possible executions of the program and evaluate the truth value of the goal in each terminal situation. However, programs with loops of this nature have infinitely many different executions, making that approach infeasible; even more, considering that loops could be induced nested inside conditionals. Another alternative is to do Montecarlo simulation of the program. Once again, this solution is expensive since to obtain reasonable precision one needs several hundreds simulations. The following corollary to theorem 1 is of essential relevance for plan assessment, since it gives a fast way to compute the probability of achievement of plans that contain loops.

Corollary 1 *Let* $\sigma = \sigma_l; \textbf{while} \neg f \textbf{ do} \sigma_l \textbf{ endWhile}; \sigma_t$ *be a program generated by a refinement of* CRefine *and* LRefine *for a goal g in a situation s. Then the probability that goal g holds after the execution of* σ *in s is:*

$$Prob_G(g,\sigma,s) = \frac{Prob_G(g,\sigma_l;\sigma_t,s)}{Prob_G(f,\sigma_l,s)}$$

PROOF: Straightforward from theorem 1. □

Note that both terms of the fraction can be efficiently computed since they are probability of success of sequences of actions. In case σ_t contains a loop, the rule can be recursively applied. This result cannot be applied to other formulae different from the goal formula for it is necessary that goal fails if f does not hold after the execution of σ_l.

We have empirically seen that successive applications of CRefine and LRefine lead to good plans. In order to use CRefine to refine a program with loops it is necessary to make some modifications. The CRefine algorithm simulates the execution of programs but, now, it may encounter loops which can generate infinitely many executions. Since loops generated by LRefine are state conservative, we can determine precisely in which kind of situations the program will be after executing a loop. Therefore CRefine does not compute all situations that result from the execution of a loop but only those that result after a successful termination of the loop, i.e., a loop execution that ends with the loop condition not holding. Furthermore, the CRefine algorithm is extended to handle recursive refinement of if-then-else constructs in a straightforward way.

We have implemented in PROLOG the refinement operator defined by (12). The program returned by successively applying the refinement operator on the program of remark 2:

$$grab(C_1); drop(C_1); \textbf{while } \neg headsUp(C_1) \textbf{ then} grab(C_1); drop(C_1) \textbf{ endWhile}.$$

Furthermore, if the goal is to have both coins heads up and held by the agent, successive interleaving of CRefine and LRefine leads to the plans shown in figure 3. Its last refinement achieves the goal with probability 1 and is computed in 0.87 seconds by our SWI-PROLOG implementation on a Pentium III 1GHz machine running Linux, using a blind breadth-first sequence generator implementation for FindSeq (as in figure 2) and no optimisations of any kind. This is a considerable improvement over CRefine refinements; in fact, a plan without loops and 96% of probability of achievement can be generated by CRefine in over 5 seconds.

[insert figure 3 about here]

5 Limitations and discussion

The most important limitation is time complexity of the algorithms. The time complexity strongly depends on FindSeq's execution time. We have only implemented breadth-first forward-chaining generators of sequences as the one shown in figure 2. Our search is completely blind; however, this can be improved through the application of heuristics appropriate to particular problems. Several well-known algorithms can be used to program this procedure, taking advantage of research in the area. For example, it is possible to use iterative deepening with an incrementable fixed horizon *h*. In this case, the algorithm can search the *most probable* sequence of actions that satisfies the goal, shorter than *h* steps. If no sequence is found to meet the threshold, the horizon can be incremented in more than one unit. The advantage of this view is that, in certain cases, longer sequences of actions can achieve goals with higher probability than shorter ones. This approach would find such sequences. It is also possible to use heuristics-aided search such as A^* or IDA^*. Utility models such as the ones used for MDPs could also be used in this search. Furthermore, we think domain-dependent optimisations to the search such as those by Reiter (2001) — which is, in turn, based on work by Bacchus and Kabanza (2000) — can also be applied to increase efficiency.

Our algorithm does not manage a plan cache for goals and states. This makes it prone to invoke FindSeq with the same goal for situations with equal state. Although this may seem to be a drawback, it is possible to extend FindSeq to remember plans it has previously generated to avoid repeated searches.

The algorithm supposes complete observability of the world. Although this may seem a serious drawback, plans generated by these algorithms can be used in domains where properties of the world can be perfectly sensed. Thus, before every conditional sentence a sensing action can be added. In domains with partial observability of the world, this may not be possible and therefore the search strategy should be changed. Currently we are developing algorithms that can cope with this issue (see section 7).

6 Related work

In this section we analyse the relationship between our work and that developed by the AI community regarding Planning under Uncertainty and Probabilistic Planning. We consider the differences and advantages if any of our approach with respect to three different lines of research that address the problem.

6.1 Classical planners under uncertainty

The first working algorithm for conditional plans is WARPLAN-C(Warren 1976), which constructs conditional plans for *conditional actions* that may have two possible outputs p or $\neg p$. The idea of this planner is similar to that of our CRefine procedure. The CNLP planner(Peot and Smith 1992) assumes incomplete knowledge about the state of the world and generates conditional plans consisting of a set of steps, *context* labels for those steps, and a set of ordering constraints. Its action description is a variant of the STRIPS action notation. Both algorithms do not deal with probabilities.

The work by Kushmerick et al. (1995) was the first attempt of doing planning under uncertainty with probabilities. In this work, an algorithm (called BURIDAN) for constructing satisfiability plans in domains with complete knowledge is given. BURIDAN does not construct conditional plans. In fact, it cannot find a solution to the coins problem for a probability threshold of 0.8. Following the same ideas, the algorithm C-BURIDAN is proposed by Draper et al. (1994). The latter algorithm assumes incomplete knowledge and takes as input a probability distribution over initial states, a goal expression, a threshold and a set of action descriptions. One of the most important differences between them is that C-BURIDAN can generate plans with contingencies, i.e., which simulates a conditional construct. The action representation of both is strongly based on that of CNLP, therefore plans are described in a similar way.

The CASSANDRA algorithm (Pryor and Collins 1996), is another example of a planner whose action representation is based on STRIPS. The algorithm generates conditional plans in the form of *if-then-else* sentences. There is no notion of probabilities in this planner.

The B-PRODIGY algorithm (Blythe 1998), uses the STRIPS action representation to generate conditional plans. It is based upon the PRODIGY classical planner Veloso et al. (1995). In this framework, assessment of plans is rather complicated, since it has to generate a Bayesian belief net.

Blum and Langford (1999), and Weld et al. (1998), have extended the well-known GRAPHLAN (Blum and Langford 1997) algorithm for planning under uncertainty. The first of these extensions, PGRAPHPLAN, uses a forward-chaining strategy to generate finite-horizon, contingent plans for domains with probabilistic actions when the initial state is completely known. On the other hand, the second extension (SGP) can generate contingent plans with information gathering actions. Uncertainty is modelled without probabilities through multiple *possible worlds*. Sensory actions are used to eliminate some of the uncertainty of possible worlds.

The main difference between the algorithms described above and ours is that they are not embedded in any cognitive robotics programming language. Furthermore, to our knowledge, none of these planners are able to generate plans with loops.

6.2 Decision-theoretic planning

A different approach is taken in decision-theoretic planning (Boutilier et al. 1999). Here, Markov Decision Processes (MDPs) are used to generate plans that are optimal *policies*. A policy is a function that returns the action that the agent must execute given the current state of the world. Moreover, they have a *finite horizon* in the sense that they explore the search space up to a fixed number of states away from the initial state. Goals are represented through a utility model, such that each state has an associated utility. Thus, an optimal policy is such that maximises the utility (reward) of the agent.

A rather different approach is taken by the work by Hansen and Zilberstein (2001). An infinite horizon algorithm for MDPs is adapted to generate solutions with loops. A loop in this framework means that the optimal policy considers that the agent could return to an already visited state. Each state is over an optimal path to the goal state, and an optimal action is considered in every possible state. Plans in this approach look like directed graphs, where each node correspond to a state and arcs corresponds to actions.

Since a policy is a mapping of a state to an action, translating policies generated by MDP-style algorithms to GOLOG programs would be rather difficult because it would be necessary to determine the state of the agent after every action execution. Moreover, states are not part of the ontology of the Situation Calculus and, therefore, it would be necessary to make unnatural changes to GOLOG.

As an alternative to the use MDPs, Pollock (2000) proposes to use standard regression planning as a means to do decision-theoretic planning in the OSCAR planner. This approach is similar to ours in the sense that classical planning techniques are used in domains with non-determinsm. Since this type of planning is decision-theoretic, the objetive of the planning problem is to maximise the expected utility rather than to achieve a particular goal. Plans generated by OSCAR are not conditional; however, Pollock (2002) gives a thorough description of cases in which contingencies may be added to a plan. Currently, OSCAR cannot generate conditional plans.

6.3 Decision-theoretic GOLOG

An integration of decision-theoretic planning and GOLOG as been made by Boutilier et al. (2000). In this work (referred to as BRST), given a simple linear program, the DTGOLOG interpreter finds an optimal way to execute the sequence, based on the perceptions of the agent, a utility model and a finite horizon. The program actually executed could be seen as program containing if-then-else constructs such as those generated by CRefine. The optimisation is done on-line and relies on the fact that the effect of stochastic actions can be sensed right after their execution. Although the agent is supposed not to be aware of the truth value of fluents it can be immediately aware after executing a *sense action*, therefore the complete knowledge about the world underlies this approach.

BRST's approach relies on a utility model and, furthermore, on a probabilistic distribution associated to actions. In our approach, although a probabilistic model may exist, it is not vital for our algorithms. DTGOLOG can be extended to do planning if given a null program and an appropriate utility function as input. In this case, the goal cannot be made completely explicit for it must be encoded as the maximum reward for the agent. Planning for different goals implies the use of multiple reward functions, which is not considered in the original DTGOLOG. Also, BRST's approach considers a finite horizon to optimise programs. The problem of choosing horizons is far from being trivial.

The main difference of BRST's approach and ours is that BRST is essentially an on-line optimisation procedure. Our planning algorithms are off-line. In both approaches the problem of partial observability and imperfect sensors has not been addressed. Nevertheless, in the following section we briefly describe ongoing work in this matter.

7 Conclusions and future work

We have presented two algorithms for planning under uncertainty and complete knowledge, which generate GOLOG programs. The first generates a conditional program and the second may induce loops in certain cases. The probability of success of plans is incremented by successive refinements. Although a probabilistic view for non-deterministic effects of actions is not necessary for our algorithms to work, in case such view is taken, the probability of success of plans with conditional plans and plans containing loops can be efficiently computed.

This research is especially relevant for the cognitive robotics area which has traditionally privileged high-level programs over planning, sacrificing flexibility. A U-GOLOG program may now contain — after a slight modification of its semantics — an invocation to planning procedures inside a program. The execution of the plan can then be on-line monitored, recalling the planning procedure if it fails. On the other hand, loop induction is especially relevant for domains containing uncertainty in which repetition of retry sequences of action may be the only way of accomplishing goals with a satisfactory probability.

One of the main drawbacks of our approach is that it assumes perfect knowledge about the world. Currently we are developing algorithms that deal with uncertainty and incomplete observability of the world (Baier 2000). In particular, we are able to model the sensing actions as explicit knowledge acquisition actions, e.g., $sense(f)$ is an action that senses the truth value of f. Conditional constructs can only condition over observable properties. In ongoing work, we are developing algorithms that address the imperfect sensors issue. In this version, actions such as $sense(f)$ are regarded as non-deterministic actions which might have incorrect outcomes. Through plan refinement, an account of these imperfections is reflected in the plans, generating appropriate contingencies that increase the probability of success of them.

Acknowledgements

Jorge Baier dedicates this paper to the memory of his friend and advisor, Javier A. Pinto who unexpectedly left us before this paper was concluded. We are grateful to Alvaro Campos, for his valuable comments on drafts of this paper.

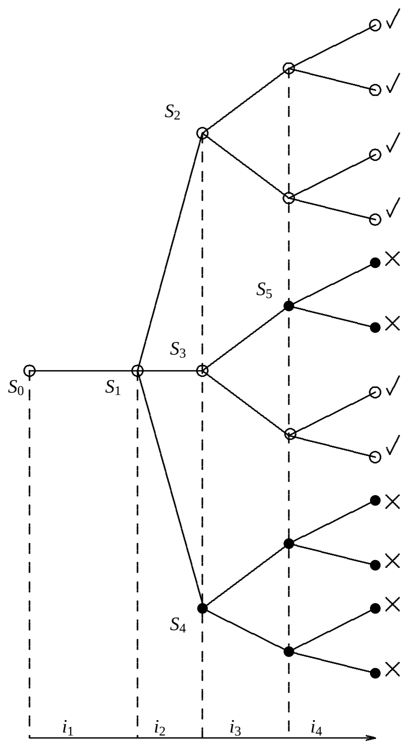

Figure 1: Execution of a sequence of input actions $i_1;i_2;i_3;i_4$, where i_1 is deterministic and the rest are non-deterministic.

```
$\mathsf{CRefine}(Goal, CandPlan, FinalPlan, CurSits, T, Level, Top) \leftarrow$
$BadSits = \{s | s \in CurSits \wedge Bad(s, Goal, CandPlan)\}$
**if** $BadSits = \{\}$ **then**
    **if** $CandPlan = \{\}$ **then**
        $FinalPlan = NoOp$
    **else if** $(\exists \alpha, \sigma) CandPlan = \alpha; \sigma$ **then**
        $NewSits = \{s | (\exists s') s' \in CurSits \wedge Do(\alpha, s', s)\}$
        $\mathsf{CRefine}(Goal, \sigma, \sigma', NewSits, T, Level, Top)$
        $FinalPlan = \alpha; \sigma'$
    **endIf**
**else**
$GoodSits = CurSits - BadSits$
$FirstBad =$ an element of $BadSits$
$\mathsf{FindSeq}(Goal, CandPlanForBads, FirstBad, T)$
**if** $Level < Top$ **then**
    $\mathsf{CRefine}(Goal, CandPlanForBads, PlanForBads, BadSits, T, Level + 1, Top)$
**else**
    $PlanForBads = CandPlanForBads$
**endIf**
**if** $GoodSits = \{\}$ **then**
    $FinalPlan = PlanForBads$
**else**
    $Property =$ fluent literal $l | (\forall s) s \in GoodSits \supset holds(l, s) \wedge$
                                  $(\forall s) s \in BadSits \supset \neg holds(l, s)$
    $\mathsf{CRefine}(Goal, CandPlan, PlanForGoods, GoodSits, T, Level + 1, Top)$
    $FinalPlan =$ **if** $Property$ **then** $PlanForGoods$ **else** $PlanForBads$
**end**
```

```
**proc** $\mathsf{FindSeq}(Goal, Seq, S, T) \leftarrow$
    $\mathsf{BreadthFirstGenerate}(Seq)$
    $do(Seq, S, \_)$,
    $prob_g(Goal, Seq, S, P)$,
    $P >= T$.
```

Figure 2: Pseudo-prolog code for a simple algorithm for planning under uncertainty with complete knowledge

```
grab(c2);
drop(c2);
if headsUp(c2) then
  grab(c1);
  drop(c1);
  grab(c2);
  grab(c1);
  NoOp;
else
  grab(c2);
  drop(c2);
  grab(c1);
  drop(c1);
  grab(c2);
  grab(c1);
  NoOp;
endif

P = 0.375
```

(a)

```
grab(c2);
drop(c2);
while -headsUp(c2) do
  grab(c2);
  drop(c2);
endWhile
grab(c1);
drop(c1);
grab(c2);
grab(c1);
NoOp;

P = 0.5
```

(b)

```
grab(c2);
drop(c2);
while -headsUp(c2) do
  grab(c2);
  drop(c2);
endWhile
grab(c1);
drop(c1);
if headsUp(c1) then
  grab(c2);
  grab(c1);
  NoOp;
else
  grab(c1);
  drop(c1);
  grab(c2);
  grab(c1);
  NoOp;
endif

P = 0.75
```

(c)

```
grab(c2);
drop(c2);
while -headsUp(c2) do
  grab(c2);
  drop(c2);
endWhile
grab(c1);
drop(c1);
while -headsUp(c1) do
  grab(c1);
  drop(c1);
endWhile
grab(c2);
grab(c1);
NoOp;

P = 1
```

(d)

Figure 3: Generation of a plan for $holding(C_1) \wedge holding(C_2) \wedge headsUp(C_1) \wedge headsUp(C_2)$. Plan (a) is a plan generated by CRefine over the goal. Plans (b) and (d) are applications of the LRefine operator over the previous plan. Plan (c) is a refinement by CRefine over (a). The probability of success of the goal is shown below the plans.

References

Allen, J. F. and Hayes, P. J., 1985. A Common-Sense Theory of Time. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 528–531 (Los Altos, CA, USA).

Bacchus, F. and Kabanza, F., 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, **116** (1-2): pp. 123–191.

Baier, J., 2000. *Modeling Agents under Uncertainty in the Situation Calculus*. Master's thesis, Pontificia Universidad Católica de Chile, Santiago, Chile. URL = http://www.ing.puc.cl/˜jabaier/articles/mscthesis.ps.zip.

Baier, J. and Pinto, J., 1998. Non-instantaneous Actions and Concurrency in the Situation Calculus (Extended Abstract). In G. de Giacomo and D. Nardi, (eds.) *10th European Summer School in Logic, Language and Information*.

Blum, A. and Langford, J., 1997. Fast Planning through Planning Graph Analysis. *Artificial Intelligence*, **90** (1-2): pp. 281–300.

Blum, A. and Langford, J., 1999. Probabilistic Planning in the Graphplan Framework. In *5th European Conference on Planning (ECP'99)*, pp. 319–332 (Durham, UK: Springer-Verlag).

Blythe, J., 1998. *Planning under uncertainty in dynamic domains*. Ph.D. thesis, Carnegie Mellon University.

Boutilier, C., Dean, T., and Hanks, S., 1999. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of AI Research (JAIR)*, **11**: pp. 1–94.

Boutilier, C., Reiter, R., Soutchanski, M., and Thurn, S., 2000. Decision-Theoretic, High-Level Agent Programming in the Situation Calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*.

Burgard, W., Cremers, A. B., Fox, D., Hhnel, D., Lakemeyer, G., Schulz, D., Steiner, W., and Thrun, S., 1998. The Interactive Museum Tour-Guide Robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pp. 11–18 (Madison, Wisconsin: AAAI Press/MIT Press). Outstanding Paper Award.

Bylander, T., 1994. The computational complexity of propositional STRIPS Planning. *Artificial Intelligence*, **69** (1-2): pp. 165–204.

Draper, D., Hanks, S., and Weld, D., 1994. Probabilistic Planning with Information Gathering and Contingent Execution. In K. Hammond, (ed.) *Proceedings of the Second Conference on AI Planning Systems*, pp. 31–36 (Chicago, IL, USA: AAAI Press).

Fikes, R. and Nilsson, N., 1971. STRIPS: A New Approach to Theorem Proving in Problem Solving. *Artificial Intelligence*, **2**: pp. 189–208.

Finzi, A., Pirri, F., and Reiter, R., 2000. Open world planning in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pp. 196–203 (Austin, TX, USA).

Giacomo, G. D., Lespérance, Y., and Levesque, H., 2000. ConGolog, a concurrent programming language based on the situation calculus: foundations. *Artificial Intelligence*, **121** (1-2): pp. 109–169. URL `http://www.cs.toronto.edu/cogrobo/Papers/ConGologLang.ps.gz`.

Green, C. C., 1969. Applications of Theorem Proving to Problem Solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-69)*, pp. 219–239. Reprinted in *Readings in Artificial Intelligence*, Webber, B. L. and Nilsson, N. J., editors, Tioga Publishing Co., Los Altos, California, pp. 202-222, 1981.

Grosskreutz, H. and Lakemeyer, G., 2001. Belief Updates in the pGolog Framework. In *24th German / 9th Austrian Conference on Artificial Intelligence KI-2001* (Viena, Austria).

Hansen, E. A. and Zilberstein, S., 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, **129** (1-2): pp. 35–62.

Kowalski, R. and Sergot, M., 1986. A Logic-based Calculus of Events. *New Generation Computing*, **4**: pp. 67–95.

Kushmerick, N., Hanks, S., and Weld, D., 1995. An algorithm for probabilistic planning. *Artificial Intelligence*, **76** (1-2): pp. 239–86.

Lespérance, Y., Tam, K., and Jenkin, M., 1999. *Reactivity in a Logic-Based Robot Programming Framework*, pp. 190–212. Logical Foundations for Cognitive Agents (Berlin, Germany: Springer-Verlag).

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B., 1997. GOLOG: A Logic Programming Language for Dynamic Domains. *The Journal of Logic Programming*, **31**: pp. 59–84.

Lin, F. and Shoham, Y., 1992. Concurrent Actions in the Situation Calculus. In *Working Notes of the 4th International Workshop on Nonmonotonic Reasoning*, pp. 133–138 (San Jose, CA, USA).

McCarthy, J. and Hayes, P. J., 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, (eds.) *Machine Intelligence* **4**, pp. 463–502 (Edinburgh, Scotland: Edinburgh University Press).

McDermott, D., 1982. A Temporal Logic for Reasoning About Processes and Plans. *Cognitive Science*, **6**: pp. 101–155.

Peot, M. A. and Smith, D. E., 1992. Conditional Nonlinear Planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pp. 189–197 (Maryland: Springer-Verlag).

Pinto, J., 1998. Integrating Discrete and Continuous Change in a Logical Framework. *Computational Intelligence*, **14** (1): pp. 39–88.

Pinto, J., Sernadas, A., Sernadas, C., and Mateus, P., 2000. Non-determinism and uncertainty in the situation calculus. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, **8** (2): pp. 127–149.

Pirri, F. and Reiter, R., 1999. Some Contributions to the Metatheory of the Situation Calculus. *Journal of the ACM*, **46** (2): pp. 261–325.

Pollock, J. L., 2000. Locally global planning in OSCAR. In *Proceedings of the AIPS2000 Workshop on Decision-Theoretic Planning* (Breckenridge, CO, USA).

Pollock, J. L., 2002. Some logical conundrums from decision-theoretic contingency planning. URL `http://oscarhome.soc- sci.arizona.edu/ftp/PAPERS/DT%20Contingency%20Planning.pdf`, draft.

Pryor, L. and Collins, G., 1996. Planning for contingencies: A decision-based approach. *Journal of AI Research (JAIR)*, **4**: pp. 287–339.

Reiter, R., 1991. *The Frame Problem in the Situation Calculus: A Simple Solution (sometimes) and a completeness result for goal regression*, pp. 359–380. Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy (San Diego, CA: Academic Press).

Reiter, R., 1996. Natural Actions, Concurrency and Continuous Time in the Situation Calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference(KR'96)* (Cambridge, Massachussetts, U.S.A.: Morgan Kaufmann).

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

Ruman, S., 1996. *GOLOG as an Agent Programming Language: Experiments in Developing Banking Applications*. Master's thesis, University of Toronto.

Sandewall, E., 1998. Cognitive robotics logic and its metatheory: Features and fluents revisited. *Electronic Transactions on Artificial Intelligence*, **3** (17). Http://www.ep.liu.se/ea/cis/1998/017/.

Thielscher, M., 1998. Introduction to the fluent calculus. *Linköping Electronic Articles in Computer and Information Science*, **3** (14). Http://www.ep.liu.se/ea/cis/1998/014/.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., and Blythe, J., 1995. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, **7** (1): pp. 81–120.

Warren, D., 1976. Generating conditional plans and programs. In *Proceedings of the Summer Conference of the Society for Articial Intelligence and the Simulation of Behaviour (AISB-76).*, pp. 344–354 (Edinburgh, UK).

Weld, D. S., Anderson, C. R., and Smith, D. E., 1998. Extending Graphplan to Handle Uncertainty & Sensing Actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 897–904 (AAAI press).

Wooldridge, M., 2002. *Introduction to Multiagent Systems* (Chichester, England: Springer-Verlag).