Turning High-Level Plans into Robot Programs in Uncertain Domains

Henrik Grosskreutz and **Gerhard Lakemeyer**[1]

Abstract. The actions of a robot like lifting an object are often best thought of as low-level processes with uncertain outcome. A high-level robot plan can be seen as a description of a task which combines these processes in an appropriate way and which may involve nondeterminism in order to increase a plan's generality. In a given situation, a robot needs to turn a given plan into an executable program for which it can establish, through some form of projection, that it satisfies a given goal with some probability. In this paper we will show how this can be achieved in a logical framework. In particular, low-level processes are modelled as programs in pGOLOG, a probabilistic variant of the action language GOLOG. High-level plans are like ordinary GOLOG programs except that during projection the names of low-level processes are replaced by their pGOLOG-definitions.

1 Introduction

The actions of a robot like lifting an object are often best thought of as low-level processes with uncertain outcome. For example, the lifting action may only succeed 80% of the time. A high-level robot plan can be seen as a description of a task which combines these processes in an appropriate way. An elegant way to obtain plans which are applicable in many circumstances is to allow (a limited amount of) nondeterminism such as "either do this or do that." For a particular circumstance, it is then up to the robot to turn such a plan into a suitable executable program. By suitable we mean that the robot is able, through some form of projection, to determine that executing the program will satisfy a given goal with a sufficient degree of probability. In this paper we will show how this can be done in a logical framework, in particular, by suitably modifying the action language GOLOG [10], which has many desirable features such as nondeterminism and control structures familiar from conventional programming languages, yet does not address actions with uncertain outcomes.

To get a better feel for what we are aiming at, let us consider the following *ship/reject*-example (adapted from [3]), which we follow throughout the paper: We are given a manufacturing robot with the goal of having a widget painted (PA) and processed (PR). Processing widgets is accomplished by rejecting parts that are flawed (FL) or shipping parts that are not flawed. The robot also has an action paint that usually makes PA true. Initially, all widgets are flawed iff they are blemished (BL), and the probability of being flawed is 0.3.

Although the robot cannot tell directly if the widget is flawed, the action *inspect* can be used to determine whether or not it is blemished. *inspect* reports $\neg OK$ if the widget is blemished and *OK* if not. The inspect action can be used to decide whether or not the widget is flawed because the two are initially perfectly correlated. The use of inspect is complicated by two things, however. (1) *inspect* is not perfect: if the widget is blemished then 90% of the time it reports $\neg OK$, but 10% of the time it erroneously reports *OK*. If the widget is not blemished, however, *inspect* always reports *OK*. (2) Painting the widget removes a blemish but not a flaw, so executing *inspect* after the widget has been painted no longer conveys information about whether it is flawed.

All actions are always possible, but may result in different effects. *paint* makes PA *true* (and BL *false*) with probability 0.95 if the widget was not already processed. Otherwise, it causes an execution error (ER). *ship* and *reject* always make PR *true*, *ship* makes ER true if FL holds, and *reject* makes ER *true* if FL does not hold.

In this example, *paint, ship, reject*, and *inspect* are considered low-level processes which we assume the robot is able to perform, subject to the uncertainties as outlined above. Also, during execution we assume the robot has direct access to the value of OK, which is set by *inspect*. We call OK *directly observable*. Suppose we hand the robot the following nondeterministic, high-level plan: *For an arbitrary number of times either paint or inspect; if OK holds afterwards then ship else reject.* The question we want to answer is the following: how can the robot turn this plan into a program, which we take to be a deterministic variant of the plan[2], for which it can guarantee that after execution of the program the goal $\mathrm{PA} \wedge \mathrm{PR} \wedge \neg\mathrm{ER}$ holds with probability 0.95?

To attack this problem, we first model the low-level processes by means of procedures in a probabilistic action language, which we call pGOLOG. In a nutshell, pGOLOG is the deterministic fragment of GOLOG augmented with a new construct, which allows us to express that a program is executed only with a certain probability. Given a faithful characterization of the low-level processes in terms of pGOLOG procedures, we can then *project* the effect of the activation of these processes using their corresponding pGOLOG models. Moreover, this projection mechanism allows us to assess the degree of belief in sentences like the above goal after the execution of a pGOLOG program.

Next we introduce the language mGOLOG, which allows us to formulate nondeterministic high-level plans such as the one above. The syntax of mGOLOG is very similar to the original GOLOG, with the names of low-level processes modelled in pGOLOG taking on the role of primitive actions. A robot who wants to achieve a certain goal with a given plan considers deterministic variants P of the plan, which are pGOLOG programs, and does the following: (1) using projection it determines whether the goal is achievable with sufficiently high probability; (2) in case this succeeds use P as the

[1] Department of Computer Science, Aachen University of Technology, D-52056 Aachen, Germany, {grosskreutz,gerhard}@cs.rwth-aachen.de

[2] One deterministic variant is to *inspect*, then *paint*, followed by the conditional.

program to be executed, otherwise consider a different P. Note that the resulting P, if it exists, only mentions processes which we assume the robot is able to initiate like *paint*. P may also contain conditionals like *if OK then ship else reject*. We require that the condition is *directly observable* by the robot, as is the case for OK, but not for BL, for example. (We remark that our approach captures a restricted form of sensing. In the example, sensing happens through the activation of the *inspect* process, which has the effect of providing the execution system with an *OK* or ¬*OK* answer.)

The rest of the paper is organized as follows. After a very brief introduction to the situation calculus, we define pGOLOG and show, starting from a probabilistic model of what the world looks like initially, how projection works in pGOLOG. Next we introduce mGOLOG and the mapping from a nondeterministic mGOLOG plan into an appropriate deterministic program. After briefly touching on experimental results, the paper ends with a discussion of related work and concluding remarks.

2 The Situation Calculus

One increasingly popular language for representing and reasoning about the effects of actions is the situation calculus [13]. We will only go over the language briefly here: all terms in the language are of sort ordinary objects, actions, situations, or reals.[3] There is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation of s resulting from performing action a in s; relations whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

Within this language, we can formulate theories which describe how the world changes as the result of the available actions. One possibility is a *basic action theory* of the following form [11]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as a function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [15].
- Domain closure and unique names axioms for the primitive actions, as well as unique names axioms for situations.

3 pGOLOG - modelling low-level processes.

Most processes in real-world applications need to be described at a level of detail involving many atomic actions interacting in complicated ways. To describe such processes, we introduce pGOLOG, a probabilistic descendant of the high-level programming language GOLOG [10]. GOLOG is a special action programming language which offers constructs such as sequences, iterations and recursive procedures to define complex actions. Most importantly, it is entirely based on the situation calculus, which allows us to project the outcome of a program, that is, reason about how the world evolves when a program is executed.

[3] While the reals are not normally part of the situation calculus, we need them to represent probabilities. For simplicity, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations.

In order to specify that processes like *paint* may result in different possible outcomes, pGOLOG provides a new probabilistic branching instruction, that did not exist in GOLOG: $prob(p, \sigma_1, \sigma_2)$. Its intended meaning is to execute program σ_1 with probability p, and σ_2 with probability $1 - p$. This allows us to specify a probabilistic process as a pGOLOG program, where the different probabilistic branches of the program correspond to different outcomes of the process. We only consider the following deterministic fragment of GOLOG together with the new $prob$-instruction.

α	primitive action
$\phi?$	wait/test action
$seq(\sigma_1, \sigma_2)$	sequence
$if(\phi, \sigma_1, \sigma_2)$	conditional
$while(\phi, \sigma)$	loop
$prob(p, \sigma_1, \sigma_2)$	probabilistic execution

Besides these instructions, we provide a restricted notion of procedures in pGOLOG, where procedure names can be used like atomic actions. To do so, we use a special function symbol $proc$ and write axioms of the form $proc(\beta) = \sigma$ to express that there is a procedure named β whose body consists of the pGOLOG program σ. Note that this necessitates the reification of programs as first order terms in the language, an issue we gloss over completely here.[4] For the purpose of this paper, we do not allow (recursive) procedure calls within procedure bodies and restrict to procedures that take no arguments.

Using the $prob$ instruction, it is possible to model processes with uncertain effects as pGOLOG *procedures*. The following procedure models the *paint* process informally described in the introduction.[5]

$$proc(paint) = \\ if(PR, setER, prob(0.95, seq(setPA, clipBL)))^6$$

Formal Semantics The semantics of pGOLOG is defined using a so-called transition semantics similar to ConGolog [5]. It is based on defining single steps of computation and, as we use a probabilistic framework, their relative probability. We define a function $transPr(\sigma, s, \delta, s')$ which, roughly, yields the transition probability associated with a given program σ and situation s as well as a new situation s' that results from executing σ's first primitive action in s, and a new program δ that represents what remains of σ after having performed that action. Let nil be the empty program, α a primitive action, and β a procedure name. Throughout the paper we assume that free variables are universally quantified, unless stated otherwise.

$$transPr(nil, s, \delta, s') = 0$$
$$transPr(\alpha, s, \delta, s') = \\ \quad \textbf{if } Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s) \textbf{ then } 1 \textbf{ else } 0$$
$$transPr(\phi?, s, \delta, s') = \\ \quad \textbf{if } \phi(s) \wedge \delta = nil \wedge s' = s \textbf{ then } 1 \textbf{ else } 0^7$$
$$transPr(if(\phi, \sigma_1, \sigma_2), s, \delta, s') = \\ \quad \textbf{if } \phi(s) \textbf{ then } transPr(\sigma_1, s, \delta, s') \textbf{ else } transPr(\sigma_2, s, \delta, s')$$
$$transPr(seq(\sigma_1, \sigma_2), s, \delta, s') = \\ \quad \textbf{if } \delta = seq(\delta', \sigma_2) \textbf{ then } transPr(\sigma_1, s, \delta', s') \\ \quad \textbf{else if } Final(\sigma_1, s) \textbf{ then } transPr(\sigma_2, s, \delta, s') \textbf{ else } 0$$
$$transPr(\beta, s, \delta, s') = transPr(proc(\beta), s, \delta, s')$$

[4] See [5] for details. The reification of pGOLOG programs is also necessary for the definition of the semantics of pGOLOG as done below.

[5] We assume successor state axioms that ensure that the truth value of PA is only affected by the primitive actions *setPA* and *clipPA*, whose effect is to make it *true* resp. *false*. Similarly for the other fluents.

[6] We write $prob(p, \alpha)$ as a shorthand for $prob(p, \alpha, nil)$. Similarly, we write $if(\phi, \alpha)$ for $if(\phi, \alpha, nil)$ and $seq(\alpha, \beta, \gamma)$ for $seq(\alpha, seq(\beta, \gamma))$.

$transPr(while(\phi,\sigma),s,\delta,s') =$
  if $\phi(s) \wedge \delta = seq(\delta', while(\phi,\sigma))$
    then $transPr(\sigma,s,\delta',s')$ **else** 0
$transPr(prob(p,\sigma_1,\sigma_2),s,\delta,s') =$
  if $\delta = \sigma_1 \wedge s' = do(tossHead, s)$ **then** p **else**
    if $\delta = \sigma_2 \wedge s' = do(tossTail(start,s)$ **then** $1-p$ **else** 0

Intuitively, a program that consists of a single atomic action α results in the execution of α and an empty remaining program with probability 1 iff α is executable. The execution of $seq(\sigma_1,\sigma_2)$ in s may result in any successor situation that could be reached by the execution of σ_1, with a remaining program $seq(\delta',\sigma_2)$, where δ' is what remains of σ_1; or, if σ_1 is final, it just corresponds to the execution of σ_2. A procedure name β is simply replaced by its body, which is the value of $proc(\beta)$. Finally, the execution of $prob(p,\sigma_1,\sigma_2)$ results in the execution of a dummy action[8] $tossHead$ or $tossTail$ with probability p resp. $1-p$ with remaining program σ_1, resp. σ_2.[9]

Besides the specification of which transitions are possible, we have to define which configurations $\langle\sigma,s\rangle$ are final, meaning that the computation can be considered completed when a final configuration is reached. This is denoted by the predicate $Final(\sigma,s)$. Here we only consider some of the definitions, where α is a primitive action.

$Final(\alpha,s) \equiv FALSE$ $Final(nil,s) \equiv TRUE$
$Final(prob(p,\sigma_1,\sigma_2,s)) \equiv FALSE$
$Final(while(\phi,\sigma),s) \equiv \phi(s) \wedge Final(\sigma,s) \vee \neg\phi(s)$

So far, we have only defined which successor configurations can be reached through a single transition. Next, we define $trans\text{-}Pr^*(\delta,s,\delta',s')$, which represents the probability to reach a configurations $\langle\delta',s'\rangle$ by a sequence of transitions, starting in configuration $\langle\delta,s\rangle$, that is, the transitive closure of $transPr$.

$$transPr^*(\delta,s,\delta',s') = p \equiv \forall t[... \supset t(\delta,s,\delta',s') = p] \vee$$
$$p = 0 \wedge \neg\exists p'.\forall t[... \supset t(\delta,s,\delta',s') = p']$$

where the ellipsis stands for the universal closure of the following formulas:

$$t(\delta,s,\delta,s) = 1 \quad (1)$$

$$t(\delta,s,\delta^*,s^*) = p_2 \wedge transPr(\delta^*,s^*,\delta',s') = p_1 \wedge p_1,p_2 > 0 \supset t(\delta,s,\delta',s') = p_1 * p_2 \quad (2)$$

Basically, this formula says that i) if there is a path of nonzero transitions from $\langle\delta,s\rangle$ to $\langle\delta',s'\rangle$, then $transPr^*(\delta,s,\delta',s')$ is equal to the product of the transition probabilities p along this path (which we call its weight), otherwise it is zero; and ii) there are no two paths from one configuration to another with different weights.

If there is a path of nonzero transitions, then (i) obtains, roughly, by "iterating" through Formula 2, making use of the reflexivity of t (Formula 1) for the case where there is exactly one transition from $\langle\delta,s\rangle$ to $\langle\delta',s'\rangle$. If there is no path without nonzero transitions, then one can always find a function t_1 which satisfies the ellipsis such that $t_1(\delta,s,\delta',s') = 0$. Hence $transPr^*(\delta,s,\delta',s') = 0$.

To see why ii) holds, let us assume that there are two paths with different weights from $\langle\delta,s\rangle$ to $\langle\delta',s'\rangle$. Then no function t exists that satisfies Formula 2; therefore $\forall t[...]$ is vacuously *true*, and $transPr^*(\delta,s,\delta',s') = p$ for all p, a contradiction. Note that to prevent this from happening when executing a $prob$ even if $\sigma_1 = \sigma_2$, we introduce the dummy actions $tossHead$ and $tossTail$ which ensure that the situations associated with σ_1 and σ_2 are different.

[7] ϕ is a situation calculus formula with all situation arguments suppressed. $\phi(s)$ is obtained from ϕ by restoring s as the situation argument in all fluents of ϕ.

[8] $tossHead$ or $tossTail$ have no effects and are always possible.

[9] The reader familiar with [5] might wonder why we don't define a synchronized version of $prob$. The reason is explained when we define $transPr^*$.

4 Probabilistic projections in pGOLOG

So far the language allows us to talk only about how the actual world evolves, starting in the initial situation S_0. But in scenarios like the ship/reject-example, there is uncertainty about the initial situation. To take this into account, we opt for a probabilistic characterization of an agent's epistemic state. More specifically, we characterize an epistemic state by a *set of situations considered possible*, and the *likelihood* assigned to the different possibilities. We thereby follow [1], who introduce a binary functional fluent $p(s',s)$ which can be read as "in situation s, the agent thinks that s' is possible with probability $p(s',s)$."[10] All weights must be non-negative and situations considered impossible will be given weight 0. Note that we are restricting ourselves to discrete probability distributions. To keep things simple, we additionally require that the probabilities of all situations considered possible in S_0 sum to 1, that is, we need the following axiom:[11]

$$\sum_s p(s,S_0) = 1 \quad (3)$$

As an example, we describe the initial belief in the ship/reject domain. Here, the world is initially in one of two states, $s1$ and $s2$, which occur with probability 0.3 and 0.7, respectively. In this simple scenario, these are the only possibilities, all other situations have likelihood 0. The following axiom makes this precise together with what holds and does not hold in each of the two states.

$$\exists s_1,s_2 \forall s.s \neq s_1 \wedge s \neq s_2 \supset p(s,S_0) = 0 \wedge$$
$$\wedge p(s_1,S_0) = 0.3 \wedge p(s_2,S_0) = 0.7$$
$$\wedge FL(s_1) \wedge BL(s_1) \wedge \neg PA(s_1) \wedge \neg PR(s_1) \wedge \neg ER(s_1)$$
$$\wedge \neg FL(s_2) \wedge \neg BL(s_2) \wedge \neg PA(s_2) \wedge \neg PR(s_2) \wedge \neg ER(s_2)$$

Now that we have defined which situations may result from the execution of a pGOLOG program, and which situations the agent considers possible initially, we turn our attention to another question: how probable, from the point of view of an agent with a probabilistic belief state, is it that ϕ will hold *after* the execution of a pGOLOG program σ? To determine this probability, we project a program ϕ *wrt each situation considered possible*, weighted by the likelihood assigned by p.

Formally, we make use of the special situation term now. Let $\phi[now]$ be a formula whose only term of sort situation is now. We write $Bel(\phi[now],s,\sigma)$ to denote the belief that ϕ holds after the execution of σ in situation s. Note that this is merely a projection of the effects of σ, no action actually gets executed. $Bel(\phi[now],s,\sigma)$ is an abbreviation for the following term expressible in second-order logic.

$$\Sigma_{\{s'',\delta'' \mid Final(\delta'',s'') \wedge \phi[now|s'']\}} p(s',s) * transPr^*(\sigma,s',\delta'',s'')$$

$Bel(\phi[now],s,\sigma)$ is defined to be the weight of all paths that reach a final configuration $\langle\delta'',s''\rangle$ that fulfills $\phi[now|s'']$ (= ϕ with now replaced by s''), starting from a possible initial configuration $\langle\sigma,s'\rangle$, weighted by the agent's belief in s'. Note that for all situations s'', there is at most one final configuration reachable by a path with positive weight. Through this definition we are restricting ourselves to discrete probability distributions, where the probability of a set can be computed as the sum of the probabilities of the elements of the set.

For example, let us calculate the belief that the widget is painted, processed and no execution error occurred ($\equiv \phi_r$) after the execution of $\sigma_{robby1} = seq(paint, ship)$. Here is the specification of the *ship* process as a pGOLOG procedure:

[10] Having more than one initial situation means that Reiter's induction axiom for situations [11] no longer holds, just as in [1].

[11] See [1] for how to characterize such equations in second-order logic.

$$proc(ship) = seq(if(FL, \mathit{setER}), \mathit{setPR}))$$

Let AX be the set of foundational axioms of Section 2 together with the definitions of $transPr$, $Final$, $transPr^*$ and Axiom 3. Further, let Γ be the set of axioms AX together with successor state axiom for the fluents, precondition axioms stating that all set and clip actions are always possible, the definitions of all pGOLOG-$proc$'s used and the above axiom describing the initial situations. Then,

$$\Gamma \models Bel(\phi_r[now], S_0, \sigma_{robby1}) = 0.665$$

This is determined as follows:

If the world is as described in s_1, the only final configurations that can be reached along a path of transitions with positive weight consist of the situations $[tossHead, \mathit{setPA}, \mathit{clipBL}, \mathit{setER}, \mathit{setPR}, s_1]$[12] or $[tossTail, \mathit{setER}, \mathit{setPR}, s_1]$ with remaining program nil. If the world is as described in s_2, the possible results are $[tossHead, \mathit{setPA}, \mathit{clipBL}, \mathit{setPR}, s_2]$ ($= s_{ok}$) or $[tossTail, \mathit{setPR}, s_2]$, again with remaining program nil. The situation s_{ok} is the only one that fulfills ϕ_r, and $transPr^*(\sigma_{robby1}, S_0, nil, s_{ok})$ is equal to $0.95 * 0.7 = 0.665$.

Theorem 1 *For all* $\phi[now]$ *and* σ*:* $\mathrm{AX} \models Bel(\phi[now], S_0, \sigma) \leq 1$.

Proof: The proof relies on the fact that $\Sigma_{\langle s', \delta \rangle} transPr(\sigma, s, \delta, s') \leq 1$, i.e. for each configuration the set of directly reachable configurations has a total probability that is no more than 1. Additionally, if a configuration is final it has no successor configuration.

5 Nondeterministic high-level plans

One of the key features of high-level programming is the ability to make use of nondeterministic instructions. It is then the task of an interpreter to determine the appropriate actions to perform, thereby making reasoned decisions. To this end, we define the nondeterministic high-level plan language mGOLOG. Although an mGOLOG plan looks like a GOLOG plan, there are differences. First, while a GOLOG program is made up of atomic actions, in mGOLOG the names of low-level processes take their role. Second, the fluents mentioned in an mGOLOG program are restricted, as we will explain below.

One of our goals is that an mGOLOG interpreter determines a program that can branch on a sensed value during execution. In contrast, a GOLOG plan is mapped to a fixed sequence of primitive actions. At this point, we have to explain what we mean by sensing. To us, sensing means: activate a sensor. This "activation" has as an effect a sensor reading. In the example, sensing happens through the activation of the *inspect* process, whose effect is to provide an *OK* or $\neg$*OK* answer. This answer is captured by setting the value of the fluent OK. Arguably, there is no uncertainty about the value of this answer. Therefore, we distinguish such fluents from other fluents and call them *directly observable*. Directly observable fluents are such that the agent always has perfect information about them - like the display of one's watch or a fuel gauge in the car.[13]

While, during real execution, the actual low-level $inspect$ process provides the answer, for the task of projection we model the behavior of the sensor by means of a probabilistic program. Here, the effect of $inspect$ is to set the directly observable fluent OK correctly with high probability, as discussed in the introduction.

$$proc(inspect) = if(BL, prob(0.9, \mathit{clipOK}, \mathit{setOK}), \mathit{setOK}))$$

Now that we have explained the restricted form of sensing that we consider, we turn back to the definition of mGOLOG. An mGOLOG program consists of pGOLOG procedure names[14], tests *concerning only directly observable fluents*, sequencing, conditionals and nondeterministic instructions.

β	pGOLOG procedure name
$\phi?$	directly observable test
$seq(\sigma_1, \sigma_2)$	sequence
$if(\phi, \sigma_1, \sigma_2)$	conditional (directly obs.)
$or(\sigma_1, \sigma_2)$	nondeterministic choice
σ^*	nondeterministic iteration
$\pi(v, \sigma)$	nondeterministic argument choice

$or(\sigma_1, \sigma_2)$ may result in the execution of any σ_i. σ^* signifies nondeterministic iteration of σ, i.e. execute σ zero, one or more times. $\pi(v, \sigma)$ means that σ is to be executed with an arbitrary – but fixed – binding for v. As tests, we only allow Boolean combinations of directly observable fluents. That means that during execution, the robot has access to their truth value through appropriate means. Additionally, we require that initially all directly observable fluents are *false*.

The semantics of nondeterministic plans is defined by specifying which determistic pGOLOG programs are legal *deterministic variants* of the plan. To specify this relationship, we use the predicate $det(NP, \sigma)$, meaning that σ is a legal deterministic program wrt plan NP. Note that through the restriction of tests to directly observables all deterministic variants are executable: all tests can be evaluated, they do not mention $prob$ instructions and each β corresponds to the activation of a low-level process.

$$\begin{array}{l}
det(\beta, \beta) \qquad \text{for procedure names and tests} \\
det(seq(\sigma_1, \sigma_2), seq(\delta_1, \delta_2)) \equiv det(\sigma_1, \delta_1) \wedge det(\sigma_2, \delta_2) \\
det(if(\phi, \sigma_1, \sigma_2), if(\phi, \delta_1, \delta_2)) \equiv det(\sigma_1, \delta_1) \wedge det(\sigma_2, \delta_2) \\
det(or(\sigma_1, \sigma_2), \sigma) \equiv det(\sigma_1, \sigma) \vee det(\sigma_2, \sigma) \\
det(\sigma^*, \delta) \equiv \delta = nil \vee det(seq(\sigma^*, \sigma), \delta) \\
det(\pi(v, \sigma), \delta) \equiv \exists x . det(\sigma_x^v, \delta)
\end{array}$$

Using the predicates det and Bel we can now ask for a deterministic variant σ that achieves $GOAL = PA \wedge PR \wedge \neg ER$ with success probability 0.95, using the high-level plan $NP \equiv seq(or(paint, inspect)^*, if(OK, ship, reject))$.[15] To do so, we make use of the predicate $plan$ which is defined as follows:

$$plan(\phi, p, s, NP, \sigma) \equiv det(NP, \sigma) \wedge Bel(\phi, s, \sigma) \geq p$$

Let Γ be defined as above and AX_{det} be the axiomatization of det. The existence of a feasible pGOLOG program can now be stated as:

$$\Gamma \cup \mathrm{AX}_{det} \models \exists \sigma . plan(GOAL, p, S_0, NP, \sigma)$$

In our example, $seq(inspect, paint, paint, if(OK, ship, reject)$ would be a feasible program σ. Note that this solution, like every program derived from an mGOLOG plan, only mentions directly observable fluents and low-level procedures, and therefore is assumed to be executable. Again, we stress that during the actual execution the procedures ($paint$, $inspect$ etc) are treated as atomic. Indeed, their procedure body cannot be executed, because we have no evidence concerning the value of non-observable fluents like BL. These procedure definitions are part of the agent's model of the world, only intended to project the program σ. During execution the actual low-level processes are activated.

[12] We write $[\alpha_1, ...\alpha_n, s]$ instead of $do(\alpha_n, do(..., do(\alpha_1, s)...)$.

[13] For those familiar with [1], note that we do not model how the epistemic state of the sensing agent, which is characterized by the fluent p, changes. In particular, we have no successor state axiom for p.

[14] We assume that for each low-level process, there is a pGOLOG procedure that models how it affects the world.

[15] Note that, as explained in [3], without making use of some kind of sensing it would be impossible to come up with a plan that has a success probability $\geq$ 0.7.

We have implemented an mGOLOG interpreter in PROLOG, and applied it to some probabilistic domains (see [5] for subtle differences between an implementation and the theory due to PROLOGs closed world assumption). Using this interpreter, we were able to solve the above example in 0.13 seconds. Of course, we have to admit that the amount of nondeterminism that can effectively be handled within our approach is limited. That means that the programmer must carefully consider the use of or, π and $*$ instructions.

6 Conclusions and related work

Within the situation calculus Levesque [9] considers plans with loops and conditionals which are also assumed to be directly executable. Lakemeyer [8] proposes to map nondeterministic plans to conditional action trees, which allows for branching during execution. In both cases, uncertainty is not considered. Acting under uncertainty lies at the heart of POMDPs and they deal with these aspects in a more exhaustive way, but the computational cost is prohibitive already in relatively small domains (e.g. [4]). Note that unlike POMDPs and probabilistic planners like C-Buridan [3] our framework is fully logic based and much more expressive since we are not restricted to propositional representations. Recently, DTGolog [2] has been proposed as a way to integrate the theory of MDPs within the GOLOG framework. The integration of decision theory into the situation calculus has also been investigated in [14].

The work of [1] on noisy sensors and effectors may seem like an alternative to our treatment of probabilistic outcomes. However, the topic of our approach and theirs is different. While they are concerned about how the epistemic state (i.e. the fluent p) changes as a result of the execution of noisy actions and the perception of noisy sensor readings, we completely ignore this aspect. Instead, we model sensors as probabilistic procedures that are activated and whose effect is to set some directly observable fluents. These procedures are intended to be used only for the task of projection. During execution, their activation is replaced by the actual activation of the robots low-level processes. For this task, our approach has the advantage of being simpler than [1].[16] Last but certainly not least, [1] does not even consider projections of programs as in pGOLOG.

As for the connection to probabilistic planning without sensing, we compared our approach with Buridan [7] and MAXPLAN [12] with persuasive results.[17] The comparision with state-of-the-art probabilistic planners that accounts for sensing (cf [6, 16]) is difficult because mGOLOG does not provide means to automatically synthesize branch conditions.

Summarizing, we have proposed pGOLOG, a probabilistic extension of GOLOG. Using pGOLOG, we were able to model low-level processes with uncertain outcome as probabilistic programs. We have then shown how to characterize the epistemic state of an agent and have provided a projection mechanism that allows us to assess how probable it is that a sentence holds after the execution of a pGOLOG program. Having defined pGOLOG and the projection mechanism, we introduced mGOLOG, a high-level plan language that provides nondeterministic instructions. Unlike GOLOG, whose primitive actions are those of the situation-calculus domain theory, the primitive actions of mGOLOG are the names of low-level processes. Additionally, tests in mGOLOG programs are restricted to directly observable fluents. We show that mGOLOG can be used to determine a pGOLOG program that has a sufficient probability to achieve a given goal through projection of the deterministic variants of the mGOLOG plan, whereas the effects of the activation of low-level processes is simulated using the corresponding pGOLOG models. The resulting program is directly executable and branches on the answers of the sensor processes activated.

Finally, a promising property of our framework is that it is easily amenable to Monte-Carlo methods for the estimation of the success probability of a pGOLOG program (unless, of course, exact assessment is required). In a nutshell, Monte-Carlo simulation can be achieved by pursuing only one of the branches of a $prob$ instruction depending on the outcome of a random number toss. The appealing property of Monte-Carlo methods is that the number of samples to be considered depends only on the desired precision of the estimate, not on the length of the program.

[16] While [1] makes use of nondeterministic π-instructions, action-likelihood axioms (l) and observation-indistinguishability axioms (OI) in order to deal with noisy sensors and effectors, pGOLOG manages solely with the $prob$ instruction.

[17] We used the BOMB/TOILET and SLIP. GRIPPER scenarios to compare the implementations on Pentium III 500 Mhz Linux workstation. Buridan solved the problems in 0.21 to 41 seconds (depending on the assessment algorithm used) resp. 0.41 to 682 seconds, while MAXPLAN took 0.4 resp. 0.3 seconds to solve the problems. Even though we didn't make use of any domain knowledge and used $or(action_1, ..., action_n)^*$ as mGOLOG plans, our implementation outperformed the other implementations by an order of magnitude: it solved the problems in 0.022 resp. 0.0097 seconds.

REFERENCES

[1] F. Bacchus, J.Y. Halpern, and H. Levesque, 'Reasoning about noisy sensors and effectors in the situation calculus', *Artificial Intelligence 111(1-2)*, (1999).

[2] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun, 'Decision-theoretic, high-level agent programming in the situation calculus', in *AAAI'2000*, (2000).

[3] D. Draper, S. Hanks, and D. Weld, 'Probabilistic planning with information gathering and contingent execution', in *Proc. of AIPS'94*, (1994).

[4] H. Geffner and B. Bonet, 'High-level planning and control with incomplete information using pomdps', in *Proc. Fall AAAI Symposium on Cognitive Robotics*, (1998).

[5] Guiseppe De Giacomo, Yves Lesperance, and Hector J Levesque, 'Congolog, a concurrent programming language based on the situation calculus: foundations', Technical report, University of Toronto, `http://www.cs.toronto.edu/cogrobo/`, (1999).

[6] Emmanuel Guere and Rachid Alami, 'A possibilistic planner that deals with non-determinism and contingency', in *IJCAI'99*, (1999).

[7] N. Kushmerick, S. Hanks, and D. Weld, 'An algorithm for probabilistic planning', *Artificial Intelligence*, **76**, 239–286, (1995).

[8] G. Lakemeyer, 'On sensing and off-line interpreting in golog', in *Logical Foundations for Cognitive Agents*, eds., H. Levesque and F. Pirri, Springer, (1999).

[9] H. J. Levesque, 'What is planning in the presence of sensing', in *AAAI'96*, (1996).

[10] Hector J. Levesque, Raymond Reiter, Yves Lesprance, Fangzhen Lin, and Richard Scherl, 'Golog: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**, 59–84, (1997).

[11] F. Lin and R. Reiter, 'State constraints revisited', *Journal of logic and computation*, **4**(5), 655–678, (1994).

[12] Stephen M. Majercik and Michael L. Littman, 'Maxplan: A new approach to probabilistic planning', in *AIPS 98*, (1998).

[13] J. McCarthy, 'Situations, actions and causal laws', Technical report, Stanford University. Reprinted 1968 in Semantic Information Processing (M.Minske ed.), MIT Press, (1963).

[14] David Poole, 'Decision theory, the situation calculus and conditional plans', *Linköping Electronic Articles in Computer and Information Science*, **3**(8), (1998). URL: http://www.ep.liu.se/ea/cis/1998/008/.

[15] Ray Reiter, 'The frame problem in the situation calculus: a simple solution (sometimes) and a ccompleteness result for goal regression.', in *In Artificial Intelligence and Mathematic Theory of Computation: Papers in Honor of John McCarthy*, (1991).

[16] D. Weld, C. Anderson, and D. Smith, 'Extending graphplan to handle uncertainty and sensing actions', in *AAAI'98*, (1998).