Turning High-Level Plans into Robot Programs in Uncertain Domains

Henrik Grosskreutz and Gerhard Lakemeyer

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-defintions.

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. High-level plans are like ordinary GOLOG programs except that during projection the names of low-level processes are replaced by their pGOLOG-defintions.

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 ~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 ~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, for which it can guarantee that after execution of the program the goal PA ∧ PR ∧ ~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
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 $F(x, do(a, s))$ 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.

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 $\sigma_1$ with probability $p$, and $\sigma_2$ 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 pGOLOG together with the new prob-instruction:

$$\alpha$$ primitive action
$$\phi$$ wait/test action
$$seq(\sigma_1, \sigma_2)$$ sequence
$$i f(\phi, \sigma_1, \sigma_2)$$ 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 $\text{proc}(\beta) = \sigma$ to express that there is a procedure named $\beta$ whose body consists of the pGOLOG program $\sigma$. 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 procedures and restrict to procedures that take no arguments.

Using the proc 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

$$\text{proc(paint)} =$$
$$i f(\langle PR, setER, \text{prob}(0.95, \text{seqPA, clipBL})\rangle)$$

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 $\text{transPr}(\sigma, s, \delta, \delta')$ which, roughly, yields the transition probability associated with a given program $\sigma$ and situation $s$ as well as a new situation $\delta'$ that results from executing $\sigma$‘s first primitive action in $s$, and a new program $\delta$ that represents what remains of $\sigma$ after having performed that action. Let nil be the empty program, $a$ a primitive action, and $\beta$ a procedure name. Throughout the paper we assume that free variables are universally quantified, unless stated otherwise.

$$\text{transPr}(\text{nil}, s, \delta, \delta') = 0$$
$$\text{transPr}(\alpha, s, \delta, \delta') =$$
$$\text{if } Poss(a, s) \land \delta = \text{nil} \land \delta' = \text{do}(\alpha, s) \text{ then } 1 \text{ else } 0$$
$$\text{transPr}(\phi, s, \delta, \delta') =$$
$$\text{if } \phi(s) \land \delta = \text{nil} \land \delta' = s \text{ then } 1 \text{ else } 0$$
$$\text{transPr}(\text{seq}(\sigma_1, \sigma_2), s, \delta, \delta') =$$
$$\text{transPr}(\text{if}(\phi, \sigma_1, \sigma_2), s, \delta, \delta') =$$
$$\text{transPr}(\langle PR, setER, \text{prob}(0.95, \text{seqPA, clipBL})\rangle)$$

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

$6$ 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.

$7$ We write prob($p, a$) as a shorthand for prob($p, a, \text{nil}$). Similarly, we write $i f(\phi, \alpha)$ for $i f(\phi, \alpha, \text{nil})$ and $\text{seq}(\alpha, \beta, \gamma)$ for $\text{seq}(\alpha, \text{seq}(\beta, \gamma))$. 

---

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.
transPr(while(φ, σ), s, δ, s') =
    if φ(s) ∧ δ = seq(δ', while(φ, σ))
    then transPr(σ, s, δ', s') else 0
transPr(prob(p, σ1, σ2), s, δ, s') =
    if δ = σ1 ∧ δ' = do( tossHead, s)
    then p else
    if δ = σ2 ∧ δ' = 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 if α is executable. The execution of seq(σ1, σ2) in s may result in any successor situation that could be reached by the execution of σ1, with a remaining program seq(δ', σ2), where δ' is what remains of σ1; or if σ1 is final, it just corresponds to the execution of σ2. A procedure name β is simply replaced by its body, which is the value of proc(β). Finally, the execution of prob(p, σ1, σ2) results in the execution of a dummy action tossHead or tossTail with probability p resp. 1 − p with remaining program σ1, resp. σ2.

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

Final(α, s) ≡ FALSE
Final(nil, s) ≡ TRUE
Final(prob(p, σ1, σ2), s) ≡ FALSE
Final(while(φ, σ), s) ≡ φ(s) ∧ Final(σ, s) ∨ ¬φ(s)

So far, we have only defined which successor configurations can be reached through a single transition. Next, we define transpr(δ, δ', s'), which represents the probability to reach a configurations (δ', s') by a sequence of transitions, starting in configuration (δ, s), that is, the transitive closure of transPr.

transPr*(δ, δ', s') = p = Var[t] . ∃ t(δ, δ', s') = p ∨ p = 0 ∧ ¬∃p[ Var[t] . ∃ t(δ, δ', s') = p ' ]

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

(1)
 t(δ, δ', s') = 1

(2)
 t(δ, δ', s', s) = p1 ∧ transPr(δ', s*, δ', s') = p1 ∧ p1, p2 > 0 ∨ t(δ, δ', s') = p1 ∗ s2

Basically, this formula says that i) if there is a path of nonzero transitions from {δ, s} to {δ', s'}, then transPr*(δ, δ', 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 (δ, s) to (δ', s'). If there is no path without nonzero transitions, then one can always find a function t1 which satisfies the ellipsis such that t1(δ, δ', s') = 0. Hence transPr*(δ, δ', s') = 0.

To see why ii) holds, let us assume that there are two paths with different weights from (δ, s) to (δ', s'). Then no function t exists that satisfies Formula 2; therefore Var[t] is vacuously true, and transPr*(δ, δ', s') = p for all p, a contradiction. Note that to prevent this from happening when executing a prob even if σ1 = σ2, we introduce the dummy actions tossHead and tossTail which ensure that the situations associated with σ1 and σ2 are different.

7 φ is a situation calculation formula with all situation arguments suppressed. φ(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 S0. 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(δ, s) which can be read as "in situation s, the agent thinks that δ is possible with probability p(δ, s)." 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 S0 sum to 1, that is, we need the following axiom:

(3)
 \[ \sum_{s} p(s, S_0) = 1 \]

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.

(3)
\text{proc}(\text{ship}) = \text{seq}(i f (\text{BL}, \text{prob}(0.9), \text{clipOK}, \text{setOK}), \text{setOK})$

Let AX be the set of foundational axioms of Section 2 together with the definitions of \text{transPr}. \text{Final}, \text{transPr}^*$ and Axiom 3. Further, let \( \Gamma \) 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 \text{pGOLOG}-\text{proc}'s used and the above axiom describing the initial situations. Then,

\[
\Gamma \models \text{Bel}(\phi, [\text{now}], S_0, \sigma_{\text{robby}}) = 0.665
\]

This is determined as follows:

If the world is as described in \( n \), the only final configurations that can be reached along a path of transitions with positive weight consist of the situations \([\text{tossHead}, \text{setPA}, \text{clipBL}, \text{setER}, \text{setPR}, s_1]\) or \([\text{tossTail}, \text{setER}, \text{setPR}, s_1]\) with remaining program nil. If the world is as described in \( s_2 \), the possible results are \([\text{tossHead}, \text{setPA}, \text{clipBL}, \text{setPR}, s_2]\) (\(= s_{\text{ok}} \)) or \([\text{tossTail}, \text{setPR}, s_2]\), again with remaining program nil. The situation \( s_{\text{ok}} \) is the only one that fulfills \( \phi \), and \text{transPr}^*(\sigma_{\text{robby}} 1, S_0, \text{nil}, s_{\text{ok}}) \) is equal to 0.95 \( \times \) 0.7 = 0.665.

\textbf{Theorem 1} For all \( \phi[\text{now}] \) and \( \sigma \): \( AX \models \text{Bel}(\phi[\text{now}], S_0, \sigma) \leq 1 \).

\textbf{Proof:} The proof relies on the fact that \( \Sigma_{\sigma, \delta}^{\text{transPr}}(\sigma, s, \delta, s') \leq 1 \), 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.

\section{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 \text{mGOLOG}. Although an \text{mGOLOG} plan looks like a \text{GOLOG} plan, there are differences. First, while a \text{GOLOG} program is made up of atomic actions, in \text{mGOLOG} the names of low-level processes take their role. Second, the fluents mentioned in an \text{mGOLOG} program are restricted, as we will explain below.

One of our goals is that an \text{mGOLOG} interpreter determines a program that can branch on a sensed value during execution. In contrast, a \text{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 \text{inspect} process, whose effect is to provide an \text{OK} or \text{~OK} answer. This answer is captured by setting the value of the fluent \text{OK}. Arguably, there is no uncertainty about the value of this answer. Therefore, we distinguish such fluents from other fluents and call them \text{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 \text{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 \text{inspect} is to set the directly observable fluent \text{OK} correctly with high probability, as discussed in the introduction.

\[
\text{proc}(\text{inspect}) = i f (\text{BL}, \text{prob}(0.9), \text{clipOK}, \text{setOK}, \text{setOK})
\]

\(\text{12}\) We write \(\sigma_1, \ldots, \sigma_n, s\) instead of \(\text{do}(\alpha_0, \ldots, \text{do}(\text{do}(\ldots (\text{do}(\alpha_1, s) \ldots) \ldots)\ldots)\).

\(\text{13}\) 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 \( p \).

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

\[
\begin{align*}
\beta & \quad \text{pGOLOG procedure name} \\
p & \quad \text{directly observable test} \\
\sigma & \quad \text{sequence} \\
\sigma & \quad \text{conditionally (directly obs.)} \\
\sigma^* & \quad \text{nondeterministic choice} \\
\pi & \quad \text{nondeterministic iteration} \\
\end{align*}
\]

\(\text{14}\) We assume that for each low-level process, there is a \text{pGOLOG} procedure that models how it affects the world.

\(\text{15}\) 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 comparison 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.

REFERENCES