ALLEGRO: Belief-Based Programming in Stochastic Dynamical Domains

ALLEGRO: Belief-based Programming in Stochastic Dynamical Domains

Vaishak Belle*
Dept. of Computer Science
KU Leuven
Belgium
vaishak@cs.kuleuven.be

Hector J. Levesque
Dept. of Computer Science
University of Toronto
Canada
hector@cs.toronto.edu

Abstract

High-level programming languages are an influential control paradigm for building agents that are purposeful in an incompletely known world. GOLOG, for example, allows us to write programs, with loops, whose constructs refer to an explicit world model axiomatized in the expressive language of the situation calculus. Over the years, GOLOG has been extended to deal with many other features, the claim being that these would be useful in robotic applications. Unfortunately, when robots are actually deployed, effectors and sensors are noisy, typically characterized over continuous probability distributions, none of which is supported in GOLOG, its dialects or its cousins. This paper presents ALLEGRO, a belief-based programming language for stochastic domains, that refashions GOLOG to allow for discrete and continuous initial uncertainty and noise. It is fully implemented and experiments demonstrate that ALLEGRO could be the basis for bridging high-level programming and probabilistic robotics technologies in a general way.

1 Introduction

High-level program execution, as seen in GOLOG [Levesque *et al.*, 1997] and FLUX [Thielscher, 2005] among others, offers an attractive alternative to automated planning for the control of artificial agents. This is especially true for the dynamic worlds with incomplete information seen in robotics, where planning would have to generate complex behavior that includes branches and loops. One of research goals of the area of *cognitive robotics* [Lakemeyer and Levesque, 2007] is to understand what sorts of high-level behavior specifications would be both useful and tractable in settings like these.

A major criticism leveled at much of this work, however, is that it makes unrealistic assumptions about the robots themselves, whose sensors and effectors are invariably noisy and best characterized by continuous probability distributions [Thrun *et al.*, 2005]. When a robot is executing a GOLOG program involving some condition ϕ about the world, for example, it is unreasonable to expect that it will either know ϕ to be true, know it to be false, or know nothing about it. To remedy this, we [Belle and Levesque, 2013a] proposed a logical account for reasoning about *probabilistic degrees of belief* against *noise* in a *continuous* setting, building on an earlier discrete model [Bacchus *et al.*, 1999]. In a more recent paper [Belle and Levesque, 2014b], we then presented a system called PREGO that is able to effectively calculate the degree of belief in ϕ as a function of the initial beliefs of the robot, the noisy physical actions that have taken place, and the noisy sensing results that have been obtained. From a logical point of view, PREGO's handling of probabilistic features goes beyond the capabilities of popular knowledge representation action languages. From a probability point of view, PREGO allows successor state and sensing axioms that can be arbitrarily complex [Reiter, 2001a], making it significantly more expressive than standard probabilistic formalisms.

The PREGO system, however, is not a programming formalism. First, it must be given the actual sequence of physical actions and sensing results, so these must be known ahead of time. Second, it works by regression: to calculate belief in ϕ after the actions and sensing, it regresses ϕ to a formula ϕ' about the initial state (using the machinery of the situation calculus), and then calculates the initial belief in ϕ'. This regression to the initial state leads to an expression involving the integration of a large number of variables (one for each fluent, and one for each noisy action), and so is simply not feasible for an iterative program that has to evaluate belief in ϕ repeatedly after thousands or even millions of actions.

In this paper, we propose a new *programming formalism* called ALLEGRO (= ALGOL in PREGO) that is intended as an alternative to GOLOG for high-level control in robotic applications. Following PREGO, the user provides a *basic action theory* (or BAT) [Reiter, 2001a] to describe the domain in terms of fluents, actions and sensors, possibly characterized by discrete and continuous distributions. Given such a BAT, ALLEGRO additionally allows the user to specify iterative programs that appeal to the robot's beliefs, and is equipped with an efficient methodology for program execution. Overall, the proposal incorporates the following advances in this line of work:

- Unlike PREGO, ALLEGRO is not just a projection system: it interprets standard GOLOG constructs (*e.g.,* while-loops) while adapting GOLOG's primitive programs for noise and outcome unknowns. An implementation-independent semantic characterization of programs is provided.
- Unlike PREGO, which can be used only in an *offline* manner (*i.e.,* knowing sensed outcomes in advance), ALLE-

*Partially funded by the FWO project on Data Cleaning and KU Leuven's GOA on Declarative Modeling for Mining and Learning.

GRO can be used in offline, *online* (*i.e.,* sense as the robot is acting in an unknown environment) and *network* (*e.g.,* controlling a robot over TCP) modes.

- To handle iterative programs involving noisy actions, ALLEGRO is realized in terms of an efficient interpreter based on *sampling* and *progression*. Inspired by particle filtering systems [Thrun *et al.*, 2005], the interpreter maintains a database of sampled states corresponding to the agent's initial uncertainty and updates these as the program executes. For one thing, this allows belief-level queries in programs to be evaluated efficiently in an online setting. For another, sampling admits a simple strategy for handling the nondeterminism in noisy actions. Nonetheless, the interpreter is argued to be correct using limits. To our knowledge, no other GOLOG implementation embodies such techniques, and such a correctness notion is new to the high-level programming literature.
- Empirical evaluations are then discussed that demonstrate the promise of the system.

In terms of organization, we first introduce ALLEGRO, before turning to its foundations and empirical behavior.

2 The ALLEGRO System

The ALLEGRO system is a programming language with a simple LISP-like syntax.[1] In this section, we (very briefly) recap domain axiomatizations, introduce the grammar of ALLEGRO programs, and discuss how ALLEGRO is used.

2.1 Domain Axiomatization

As noted above, a ALLEGRO domain is modeled as in PREGO. As a small example, Figure 2 shows a BAT for the domain illustrated in Figure 1, which can be read as follows:[2]

1. The domain has a single fluent `h`, the distance to the wall, whose initial value is taken by the agent to be drawn from a (continuous) uniform distribution on [2, 12].
2. The successor state axiom for `h` says that it is affected only by the `nfwd` action. The first argument of `nfwd` is the amount the agent intends to move, and the second is the amount that is actually moved (which determines how much the value of `h` is to be reduced).
3. The `alt` axiom is used to say that if an action of the form `(nfwd 2 2.79)` occurs, the agent will only know that `(nfwd 2 z)` happened for some value of `z`.

[1]Like PREGO, ALLEGRO is realized in the RACKET dialect of the SCHEME family (`racket-lang.org`). We use RACKET arithmetic freely, such as the `max` function, as well as any other function that can be defined in RACKET, like `GAUSSIAN`. However, the technical development does not hinge on any feature unique to that language.

[2]For ease of exposition, we limit discussions in the following ways. First, we omit any mention of action preconditions here. Second, fluents are assumed to be real-valued, and moreover, only basic features of the language are explained in the paper. In general [Belle and Levesque, 2014b], fluent values can range over any set, BATs are not limited to any specific family of discrete/continuous distributions, and the language supports notions not usually seen in standard probabilistic formalisms, such as *contextual* likelihood axioms. (For example, if the floor is slippery, then the noise of a move action may be amplified.) All of these are fully realized in PREGO and ALLEGRO.

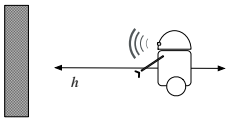


Figure 1: Robot moving towards a wall.

```
(define-fluents h)

(define-ini-p-expr '(UNIFORM h 2 12))

(define-ss-exprs h
    (nfwd x y) '(max 0 (- h ,y))

(define-alts
    (nfwd x y) (lambda (z) '(nfwd ,x ,z)))

(define-l-exprs
    (nfwd x y) '(GAUSSIAN ,y ,x 1.0)
    (sonar z) '(GAUSSIAN ,z h 4.0))
```

Figure 2: A BAT for the simple robot domain.

4. The likelihood axiom for `nfwd` says that the actual amount moved by `nfwd` will be expected by the agent to be centered (wrt a normal distribution) around the intended amount with a standard deviation of `1.0`.
5. The likelihood axiom for `sonar` says that sonar readings are noisy but will be expected to be centered around the true value of `h` with a standard deviation of `4.0`.

The idea here is that starting with some initial beliefs, executing `(sonar 5)` does not guarantee that the agent is actually `5` units from the wall, although it should serve to increase the robot's confidence in that fact. Analogously, performing a noisy move with an intended argument `2` means that the robot may end up moving (say) `2.79` units. Nevertheless, its degree of belief that it is closer to the wall should increase.

2.2 Belief-based Programs

The BAT specification describes the noise in actions and sensors wrt the actual outcomes and values observed. But a robot executing a program need not know these outcomes and values. For this reason, the *primitive programs* of ALLEGRO are actions that suppress these parameters. So for the actions `(nfwd x y)` and `(sonar z)` appearing in a BAT, the primitive programs will be `(nfwd x)` and `(sonar)`.

The basic ALLEGRO language uses five program constructs:

prim	primitive programs;
(`begin` $prog_1 \ldots prog_n$)	sequence;
(`if` *form* $prog_1$ $prog_2$)	conditional;
(`let` ((var_1 $term_1$) $\ldots$ (var_n $term_n$)) *prog*)	assignments;
(`until` *form prog*)	until loop.

Here *form* stands for formulas built from this grammar:

$$form ::= (\circ\ term_1\ term_2) \mid (\bullet\ form_1\ form_2) \mid (\texttt{not}\ form)$$

where $\circ \in \{<, >, =\}$ and $\bullet \in \{\texttt{and}, \texttt{or}\}$. Here *term* stands for terms built from the following grammar:

$$term ::= (\texttt{exp}\ term) \mid number \mid fluent \mid var \mid (\diamond\ term_1\ term_2) \mid (\texttt{if}\ form\ term_1\ term_2)$$

where $\diamond$ is any arithmetic operator (*e.g.,* + and -). The primary "epistemic" operator in ALLEGRO is `exp`: `(exp` *term*`)` refers to the *expected value* of *term*. (Reasoning about the expected value about a fluent allows the robot to monitor how it changes with sensing, for example.) The *degree of belief* in a formula *form* can then be defined in terms of `exp` as follows:

$$\texttt{(bel } \textit{form}\texttt{)} \doteq \texttt{(exp (if } \textit{form}\ \texttt{1.0 0.0))}$$

For our purposes, it is convenient to also introduce `(conf` *term number*`)`, standing for the *degree of confidence* in the value of a term, as an abbreviation for:

$$\texttt{(bel (> } \textit{number}\ \texttt{(abs (- } \textit{term}\ \texttt{(exp } \textit{term}\texttt{)))))}$$

For example, given a fluent `f` that is normally distributed, `(conf f .1)` is higher when the curve is narrower.

2.3 Usage

The ALLEGRO system allows programs to be used in three ways: in *online* mode – the system displays each primitive program as it occurs, and prompts the user to enter the sensing results; in *network* mode – the system is connected to a robot over TCP, the system sends primitive programs to the robot for execution, and the robot sends back the sensing data it obtains; and finally, in *offline* mode – the system generates ersatz sensing data according to the given error model. In all cases, the system begins in an initial belief state, and updates this belief state as it runs the program.

As a simple illustration, imagine the robot from Figure 2 would like to get within 2 and 6 units from the wall. It might proceed as follows: sharpen belief about current position (by sensing), (intend to) move by an appropriate amount, adjust beliefs for the noise of the move, and repeat these steps until the goal is achieved. This intuition is given as a program in Figure 3: here, `conf` is used to first become confident about `h` and then `exp` is used to determine the distance for getting midway between 2 and 6 units. (An arbitrary threshold of `.8` is used everywhere wrt the robot's beliefs.) For an online execution of this program `prog` in ALLEGRO, we would do:

```
> (online-do prog)
Execute action: (sonar)
Enter sensed value: 4.1
Enter sensed value: 3.4
Execute action: (nfwd 1.0)
Enter sensed value: 3.9
Enter sensed value: 4.2
Execute action: (nfwd 0.0)
```

We see the robot first applying the sonar sensor, for which the user reports a value of `4.1`. After updating its beliefs for this observation, the robot is not as confident as required by `prog`, and so, a second sensing action is executed for which `3.4` is read. Then the robot attempts a (noisy) move, but its confidence degrades as a result. So these steps are repeated once more, after which the program terminates. On termination, the required property can be tested in ALLEGRO using:

```
> (bel (and (>= h 2) (<= h 6)))
0.8094620133032484
```

```
(until (> (bel (and (>= h 2) (<= h 6))) .8)
    (until (> (conf h .4) .8) (sonar))
    (let ((diff (- (exp h) 4)))
       (nfwd diff)))
```

Figure 3: A program to get between 2 and 6 units from the wall.

3 Mathematical Foundations

The formal foundation of ALLEGRO is based on the *situation calculus* [Reiter, 2001a]. Basically, the interpretation of PREGO-style BATs is as in [Belle and Levesque, 2014b], which we can only summarize here. The interpretation of programs is based on the online version of GOLOG [Sardina *et al.*, 2004].

3.1 BAT Semantics

In a nutshell, the situation calculus is a many-sorted dialect of predicate logic, with sorts for *actions*, *situations* and *objects*. Situations are basically *histories* in that initial situations correspond to the ways the world might be initially, and successor situations are obtained by applying a distinguished binary symbol *do*. That is, $do(a_1 \cdots a_k, s)$ denotes the situation resulting from performing actions a_1 through a_k at situation s. We let the constant S_0 denote the actual initial situation.

Fluents capture changing properties about the world. Here we assume that nullary functions $f_1, \ldots, f_k$ are all the fluents in the language, and that these take no arguments other than a single situation term. When writing formulas, we often suppress the situation argument in a logical expression ϕ or use a distinguished variable *now*, and we let $\phi[s]$ denote the formula with that variable replaced by s. In this sense, the BAT syntax of PREGO (and ALLEGRO) is to be seen as a situation-suppressed fragment of the situation calculus language.[3]

The situation calculus has been extended over the years, and here, we appeal to our prior account on reasoning about probabilistic degrees of belief and noise [Belle and Levesque, 2013a; 2014b]. It is based on three distinguished fluents: p, l and *alt*. Using these, a BAT $\mathcal{D}$ consists of (free variables are implicitly assumed to be quantified from the outside):

- *Initial theory* $\mathcal{D}_0$: an axiom of the form $p(s, S_0) = \Theta[s]$ to specify the agent's initial uncertainty, where Θ is a situation-suppressed expression using the fluents. Roughly, the p fluent acts as a numeric version of the accessibility relation in modal logic [Bacchus *et al.*, 1999].
- *Likelihood and alternate actions axioms* of the form $l(a(\vec{x}), s) = \text{LH}_a(\vec{x})[s]$ and $alt(a, u) = a'$. These formalize the noise and outcomes of noisy sensors and effectors.
- *Successor state axioms* of the form $\forall a, s.\ f(do(a, s)) = \text{SSA}_f(a)[s]$: the values of fluents may change after actions as determined by these axioms, which appeal to Reiter's [2001a] solution to the frame problem. A domain-independent successor state axiom is provided for the p fluent that functions as follows: the p-value of the successor situation $do(a, s)$ is obtained by multiplying the p-value of s with the likelihood of action a.

[3] Non-logical symbols, such as fluent and action symbols, in the situation calculus are italicized; *e.g.,* fluent `h` in PREGO/ALLEGRO is h in the language of the situation calculus.

Example 1: The sonar's noise model, move action's nondeterminism and initial beliefs of the robot from Figure 2 can be mapped to the following situation calculus formulas: $l(sonar(z), s) = Gaussian(z; h, 4)[s]$, $alt(nfwd(x, y), z) = nfwd(x, z)$ and $p(s, S_0) = Uniform(h; 2, 12)[s]$.

Given a BAT $\mathcal{D}$, the *degree of belief* in any situation-suppressed ϕ at s is defined using:

$$Bel(\phi, s) \doteq \frac{1}{\gamma} \int_{f_1,...,f_k} \int_{u_1,...,u_n} Density(\phi, s^*)$$

where the normalization factor γ is obtained by replacing ϕ with TRUE, and if $s = do(a_1 \cdots a_n, S_0)$ then $s^* = do(alt(a_1, u_1) \cdots alt(a_n, u_n), S_0)$. We do not expand the *Density* term here, except note that it applies the successor state axiom for p to all ground situation terms that are obtained as a result of noisy actions. (That is, possible outcomes for a are given by $alt(a, u)$ wrt the integration variable u.) Given an argument ϕ, then, the *Density* term returns the p-value of the situation if ϕ holds, and 0 otherwise. This definition of *Bel* can be shown to subsume Bayesian conditioning [Belle and Levesque, 2013a]. (Note also that for discrete distributions, we would replace the appropriate integral with a sum.)

Analogously, the *expected value* of a situation-suppressed term t at s can be defined as:

$$Exp(t, s) \doteq \frac{1}{\gamma} \int_{f_1,...,f_k} \int_{u_1,...,u_n} (t \times Density(\text{TRUE}, now))[s^*]$$

where the denominator γ replaces t with 1, and s and s^* are as before. This is to be read as considering the t-values across situations and multiplying them by the p-values of the corresponding situations. So, if a space of situations is uniformly distributed, then we would obtain the average t-value.

3.2 Program Semantics

ALLEGRO implements a deterministic fragment of GOLOG over online executions [Sardina *et al.*, 2004]. In general, GOLOG programs are defined over the following constructs:

$$\alpha \mid \phi? \mid (\delta_1; \delta_2) \mid (\delta_1 \mid \delta_2) \mid (\pi\, x)\delta(x) \mid \delta^*$$

standing for primitive programs, tests, sequence, nondeterministic branch, nondeterministic choice of argument and nondeterministic iteration respectively. The constructs used in ALLEGRO can then be defined in terms of these:

if ϕ **then** δ_1 **else** $\delta_2 \doteq [\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$
until $\phi\ \delta \doteq [\neg\phi?; \delta]^*; \phi?$
let $(x = t)\ \delta \doteq (\pi\, x)[(x = t)?; \delta]$

(Here ϕ would mention *Bel* or *Exp*.) Unlike standard GOLOG, however, the smallest programs in our setting are not atomic actions, but new symbols called *primitive programs* (Section 2.2), which serve as placeholders for actual actions on execution. So for every $a(\vec{x}, y)$ in the logical language, where y corresponds to the actual outcome/observation, there are primitive programs $a(\vec{x})$ that suppress the latter argument.

A semantics for the online execution of ALLEGRO programs over noisy acting and sensing can be given in the very same style as [Sardina *et al.*, 2004]. Two special predicates *Trans* and *Final* are introduced to define a single-step transition semantics for ALLEGRO programs: $Trans(\delta, s, \delta', s')$ means that by executing program δ at s, one can get to s' with an elementary step with the program δ' remaining, whereas $Final(\delta, s)$ holds when δ can successfully terminate in situation s.

The predicates *Trans* and *Final* are defined axiomatically, precisely as in [Sardina *et al.*, 2004], with the exception of the definition for primitive programs:

- if $a(\vec{x})$ is a primitive program in ALLEGRO, and on doing this action we obtain a value c (either an observation or the outcome of a noisy action) then
  $Trans(a(\vec{x}), s, \delta', s') \equiv$
  $Poss(a(\vec{x}, c), s) \wedge \delta' = nil \wedge s' = do(a(\vec{x}, c), s).$

The definitions are lumped here as a set $\mathcal{E}$, along with an encoding of programs as first-order terms for quantificational purposes as a set $\mathcal{F}$ [Sardina *et al.*, 2004]. Putting it all together, we say that the *online execution* of a program δ *successfully terminates* after a sequence of actions σ if

$$\mathcal{D} \cup \mathcal{E} \cup \mathcal{F} \models Do(\delta, S_0, do(\sigma, S_0))$$

where $Do(\delta, s, s') \doteq \exists\delta'\ [Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')]$ and $Trans^*$ denotes the reflexive transitive closure of *Trans*.

4 A Sampling-based Interpreter

We now present pseudo-code for the interpreter of ALLEGRO and argue for its correctness relative to the specification above. The interpreter is based on sampling [Murphy, 2012], and its correctness is defined using limits.

The overall system is described using three definitions: an evaluator of expressions EVAL, an epistemic state e, and an interpreter of programs INT. In what follows, we let a *world state* w be a vector of fluent values (*i.e.,* $w[i]$ is the value of the i-th fluent) and an *epistemic state* e is a finite set of pairs (w, q) where w is a world state and $q \in [0, 1]$. Intuitively, w is a world considered possible by the agent, and q is the probabilistic weight attached to that world.

The Evaluator

The interpreter relies on an evaluator EVAL that takes three arguments: a term or a formula, a world state, an epistemic state, and returns a value (corresponding to the value of the term or the truth value of a formula):

Definition 2: EVAL$[\cdot, w, e]$ is defined inductively over a term t or formula d by

1. EVAL$[u, w, e] = u$, when u is a number or a variable.
2. EVAL$[f, w, e] = w[i]$, when f is the i-th fluent.
3. EVAL$[(\texttt{not}\ d), w, e] = 1 -$ EVAL$[d, w, e]$, and similarly for other logical connectives over formulas.
4. EVAL$[(\texttt{+}\ t_1\ t_2), w, e] =$ EVAL$[t_1, w, e] +$ EVAL$[t_2, w, e]$, and similarly for other arithmetic operators over terms.
5. EVAL$[(\texttt{if}\ d\ t_1\ t_0), w, e] =$ EVAL$[t_i, w, e]$, where we obtain $i =$ EVAL$[d, w, e]$.
6. EVAL$[(\texttt{exp}\ t), w, e] = \sum_{(w',\ q) \in e}$ EVAL$[t, w', e] \times q \Big/ \sum_{(w',\ q) \in e} q.$

We write EVAL$[t, e]$ when w is not used (because all the fluents in t appear in the scope of an `exp`), and EVAL$[t, w]$ when e is not used (because t contains no `exp` terms).

Epistemic State

To relate the evaluator to a BAT specification, we first obtain the initial epistemic state e_0. To get the idea, suppose the values of the fluents f_i were to range over finite sets $dom(f_i)$. Given any $\mathcal{D}_0$ of the form $\forall s(p(s, S_0) = \Theta[s])$, we let:[4]

$$e_0 = \{(w, q) \mid w[i] \in dom(f_i) \text{ and } q = \textsc{eval}[\Theta, w]\}.$$

By construction, e_0 is the set of all possible worlds in a finite domain setting. However, in continuous domains, $dom(f_i)$ is $\mathbb{R}$, leading to infinitely many possible worlds. Nonetheless, this space can be approximated in terms of a finite epistemic state by sampling: assume we can randomly draw elements (and vectors) of $\mathbb{R}$. Suppose the BAT is defined using k fluents, and let n be a large number. Then:

$$e_0 = \{(w_i, q_i) \mid \text{choose } w_1, \ldots, w_n \in \mathbb{R}^k, \text{ and } q_i = \textsc{eval}[\Theta, w_i]\}.$$

When actions occur, the interpreter below also depends on a *progression* procedure PROG that takes an epistemic state and an action instance as arguments and returns a new state:

Definition 3: Suppose $a(c)$ is a noisy sensor. Then
$\textsc{prog}[e, a(c)] =$
$\{(w', q') \mid (w, q) \in e,$
$w'[j] = \textsc{eval}[\textsc{ssa}_{f_j}(a(c)), w] \text{ and } q' = q \times \textsc{eval}[\textsc{lh}_a(c), w]\}$
where SSA and LH refer to the RHS of successor state and likelihood axioms in $\mathcal{D}$ instantiated for $a(c)$.

Intuitively, for $(w, q) \in e$, standing for some current situation and its density, we use the given BAT to compute the fluent values and the density of the successor situation. In particular, for a noisy sensor, the successor density incorporates the likelihood of the observed value c.

Definition 4: Suppose $a(c, c')$ is a noisy action. Then
$\textsc{prog}[e, a(c, c')] =$
$\{(w'_i, q'_i) \mid \text{choose } d_1, \ldots, d_n \in \mathbb{R},$
$(w_i, q_i) \in e,\ w'_i[j] = \textsc{eval}[\textsc{ssa}_{f_j}(a(c, d_i)), w_i]$
$\text{and } q'_i = q_i \times \textsc{eval}[\textsc{lh}_a(c, d_i), w_i]\}.$

The notion of progression is more intricate for noisy actions because for any ground action term $a(c, c')$, the agent only knows that $a(c, y)$ happened for some $y \in \mathbb{R}$. (For ease of exposition, assume $alt(a(c, x), y) = a(c, y)$.) Among other things, this means that the progression would need to account for all possible instantiations of the variable y, which are infinitely many. Sampling, however, permits a simple strategy: assume we can randomly draw n elements of $\mathbb{R}$, once for each world state in e, the idea being that the value drawn is applied as an argument to action a for the corresponding world. Consequently, for any e and a, $|e| = |\textsc{prog}[e, a(c, c')]|$, and the sampling limit is still shown to converge. In essence, we needed to appeal to sampling for each occurrence of an integration variable in *Exp*: for every fluent and for the argument of every noisy action.

[4] We can evaluate the RHS of axioms in $\mathcal{D}$ wrt a world state using Definition 2. These are simply formulas (with a single situation term) of the logical language, and in the ALLEGRO syntax, they correspond to expressions mentioning the fluents. For example, evaluating the function `(UNIFORM h 2 12)` at a world state where `h` is `10` would yield `.1`, but in one where it is `15` would yield `0`.

The Interpreter

Finally, we describe the interpreter for ALLEGRO, called INT. It takes as its arguments a program and an initial epistemic state and returns as its value an updated epistemic state:

Definition 5: $\textsc{int}[\delta, e]$ is defined inductively over δ by

1. $\textsc{int}[(a), e] = \textsc{prog}[e, a(c)]$, where c is observed on doing noisy-sensor primitive program a.
2. $\textsc{int}[(a\ t), e] = \textsc{prog}[e, a(c, c)]$, where $c = \textsc{eval}[t, e]$, after doing noisy-action primitive program a for argument c. (Since the second argument of a is irrelevant for e, we simply set it also to c.)
3. $\textsc{int}[(\texttt{begin}\, \delta_1 \ldots \delta_n), e] = \textsc{int}[\delta_n, \textsc{int}[(\texttt{begin}\, \delta_1 \ldots \delta_{n-1}), e]]$.
4. $\textsc{int}[(\texttt{if}\, \phi\, \delta_1\, \delta_0), e] = \textsc{int}[\delta_i, e]$, where $i = \textsc{eval}[\phi, e]$.
5. $\textsc{int}[(\texttt{let}\, ((x\ t))\, \delta), e] = \textsc{int}[\delta^x_v, e]$, where $v = \textsc{eval}[t, e]$.
6. $\textsc{int}[(\texttt{until}\, \phi\, \delta), e] = e$ if $\textsc{eval}[\phi, e] = 1$, and otherwise, $\textsc{int}[(\texttt{until}\, \phi\, \delta), \textsc{int}[\delta, e]]$.

The interpreter's correctness is established by the following:

Theorem 6: *Suppose $\mathcal{D}$ is a BAT with e_0 (of size n) as above, δ is any program, and t is any term. Suppose there is an action sequence σ such that $\mathcal{D} \cup \mathcal{E} \cup \mathcal{F} \models Do(\delta, S_0, do(\sigma, S_0))$. Then,*

$$\mathcal{D} \models Exp(t, do(\sigma, S_0)) = u \text{ iff } \lim_{n \to \infty} \textsc{eval}[(\texttt{exp}\ \mathrm{t}), \textsc{int}[\delta, e_0]] = u.$$

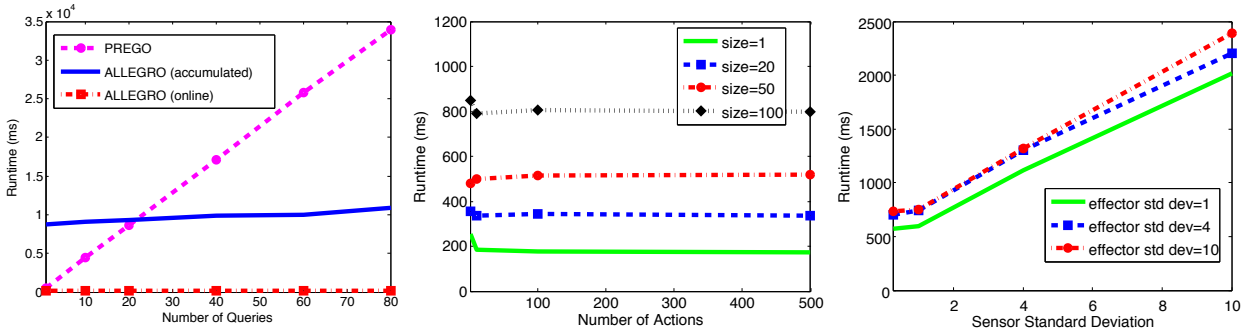
Proof sketch: Proof by induction on δ, and only primitive programs are presented here. Suppose δ is empty. Since $\mathcal{D}_0$ defines a k-variable density function and e_0 draws n samples from $\mathbb{R}^k$, standard notions of convergence readily apply [Murphy, 2012]. Suppose δ is a noisy sensor $a(c)$, in which case the k-variable density function simply takes the likelihood of the sensing into account, and the argument follows easily. Suppose δ is a noisy action $a(c, c')$. Then $Exp(t, do(a(c, c'), S_0))$ is seen to define a $(k+1)$-variable density function. Although our notion of progression considers (only) a single instance of the noisy action for each of the n world states, this is essentially seen as actually drawing n samples from $\mathbb{R}^{k+1}$. Thus, the desired convergence is also obtained. ∎

5 Evaluations

We consider the empirical behavior of ALLEGRO in this section. Mainly, we argue for progression rather than regression, test its scalability, and investigate the effect of noise on program termination tasks. We set the sample size $|e_0| = 100000$. Experiments were run on Mac OS X 10.9 using a system with 1.4 GHz Intel Core 2 Duo processor, 2 GB RAM, and RACKET v6.1. CPU time is measured in milliseconds (ms).

Regression and Progression

The PREGO system was shown in [Belle and Levesque, 2014b] to exhibit reasonable empirical behavior on projection tasks. Here, we are interested in contrasting the two systems. As discussed, regressing to the initial state leads to an expression involving the integration of a large number of variables, and calculating probabilities with such expressions would simply not be feasible in iterative programs over many actions. (We note that the regression simplification itself is unproblematic,



(a) PREGO and ALLEGRO (b) scalability (c) program termination

Figure 4: Empirical evaluations on the ALLEGRO system.

but the integration is challenging.) So, progression seems like a better choice for the interpreter in long-lived agents.

We test this intuition as follows. Imagine the robot from Figure 1 has performed 50 noisy actions: `(nfwd 2 2.1)(sonar 5)(nfwd 1 2.1)` ... , after which we query the robot's beliefs, both in the current situation and future ones, *e.g.,* `(expt h)`, `(bel (< h 2))`, `(nfwd 3 3.05)(expt h)`, `(nfwd 3 3.05)(bel (< h 2))`, and so on. In other words, we test static and projection queries against PREGO and ALLEGRO in Figure 4a. (For ALLEGRO, we consider its *online behavior* where given an action sequence σ, and the time τ for computing PROG$[e_0, \sigma]$, we plot $\tau/|\sigma|$.) We observe that evaluation by regression grows exponentially in the number of queries considered in the experiments. Thus, even purely sequential programs over hundreds of actions will be difficult to handle. In contrast, with progression, the increased effort for more queries is insignificant (as needed for iterative programs), mainly because the states are updated after actions and queries are evaluated in the current situation. Moreover, in a separate evaluation for scaling behavior in Figure 4b, ALLEGRO is tested for action sequences of length up to 500; PREGO runs out of memory on sequences of that length.

We can further contrast the online performance of ALLEGRO to the *total accumulated time* for progressing e_0 wrt the 50 actions, also plotted in Figure 4a. In this context, however, we see progressing the database is more expensive than regression provided the number of queries is small. In other words, when the agent simply needs to test a property of the future (*e.g.,* in planning), regression is preferable. In sum, in continuous domains, regression is the right choice for testing plan validity, and progression for long-running programs.

Scalability

Here, ALLEGRO's online performance is tested for domains of increasing sizes over long action sequences. The BAT from Figure 2 is a domain of size 1, and for size n, we axiomatize domains with n fluents and n noisy actions.

Taking into account the increased effort needed to maintain larger domains (that is, the world states correspond to lengthier vectors), we observe in Figure 4b that the implemented ALLEGRO interpreter performs exactly as one would require for prolonged sequences of actions: the progression effort is constant. In sum, the implementation scales well relative to domain size in long-lived systems.

From Noise To Termination

For our final evaluation, we test the sensitivity of the termination of the program from Figure 3 to the *standard deviation* in the noise of the robot's effectors and sensors. (Basically, a lower standard deviation means a more accurate device.) The plot in Figure 4c gives an indication of what one should expect regarding the program's time for success wrt this noise. At one extreme, an accurate sensor with an inaccurate effector would mean that the robot gets confident about h quickly but takes longer to move to the desired position. At the other extreme, an accurate effector would mean that reaching the desired position is easily accomplished, but only after getting confident about h that takes longer with an inaccurate sensor.

6 Related Work and Discussions

We first relate ALLEGRO to high-level agent programming proposals, and then to planning proposals.

The ALLEGRO system is a programming model based on the situation calculus, and so is a new addition to the GOLOG family of high-level programming languages [Levesque *et al.*, 1997; Sardina *et al.*, 2004]. In particular, it follows in the tradition of *knowledge-based* GOLOG with sensing in an online context [Reiter, 2001b; Claßen and Lakemeyer, 2006; Fan *et al.*, 2012], but generalizes this in the sense of handling degrees of belief and probabilistic noise in the action model.

The GOLOG family has been previously extended for probabilistic nondeterminism, but there are significant differences. For example, in the PGOLOG model [Grosskreutz and Lakemeyer, 2003], the outcome of a nondeterminism action is immediately *observable* after doing the action, and continuous distributions are not handled. This is also true of the notable DTGOLOG approach [Boutilier *et al.*, 2000] and its variants [Ferrein and Lakemeyer, 2008]. In this sense, the ALLEGRO model is more general where the agent cannot observe the outcomes and sensors are noisy. Moreover, these proposals do not represent beliefs explicitly, and so do not include a query language for reasoning about nested belief expressions.

Outside of the situation calculus, an alternative to GOLOG, called FLUX [Thielscher, 2004; 2005], has been extended for knowledge-based programming with noisy effectors in [Thielscher, 2001; Martin and Thielscher, 2009]; continuous probability distributions and nested belief terms are, however, not handled. The concept of knowledge-based programming was first introduced in [Fagin *et al.*, 1995]. These have been extended for Spohn-style [1988] ordinal functions in [Laverny and Lang, 2005]. For discussions on how high-level programming relates to standard agent programming [Shoham, 1993], see [Sardina and Lespérance, 2010].

Programs, broadly speaking, generalize sequential and tree-like plan structures, but can also be used to limit plan search [Baier *et al.*, 2007]; we refer interested readers to [Lakemeyer and Levesque, 2007] for discussions. There are, of course, many planning approaches in online contexts, including knowledge-based planning [Petrick and Bacchus, 2004; Van Ditmarsch *et al.*, 2007], decision-theoretic proposals [Puterman, 1994; Kaelbling *et al.*, 1998], corresponding relational abstractions [Sanner and Kersting, 2010; Zamani *et al.*, 2012], and belief-based planning [Kaelbling and Lozano-Pérez, 2013]. See [Lang and Zanuttini, 2012] on viewing knowledge-based programs as plans.

In the same vein, let us reiterate that the BAT syntax is based on PREGO, and from the viewpoint of a representation language, see [Belle and Levesque, 2014b] for discussions on its expressiveness relative to other formalisms, such as probabilistic planning languages [Sanner, 2011], action formalisms [Van Benthem *et al.*, 2009; Thielscher, 2001; Iocchi *et al.*, 2009], and related efforts on combining logic and probability [Bacchus, 1990; Richardson and Domingos, 2006]. Notably, its support for continuous distributions and arbitrary successor state axioms makes it a tractable but expressive language.

In the context of probabilistic models, we remark that there are other realizations of program-based probabilistic behavior, such as *probabilistic programming* [Milch *et al.*, 2005; Goodman *et al.*, 2008]. These are formal languages that provide program constructs for probabilistic inference. While they are expressive enough to capture dynamical probabilistic models [Nitti *et al.*, 2013], they belong to a tradition that is different from the GOLOG and FLUX families. For example, atomic programs in GOLOG are actions taken from a basic action theory whereas in [Milch *et al.*, 2005], atomic constructs can be seen as random variables in a Bayesian Network. In other words, in GOLOG, the emphasis is on high-level control, whereas in many probabilistic programming proposals, the emphasis is on inference. So, a direct comparison is difficult; whether these traditions can be combined is an open question.

Before wrapping up, let us emphasize two issues concerning our interpreter. First, regarding Theorem 6, it is worth relating this result to the notion of progression of BATs [Reiter, 2001a]. In prior work [Belle and Levesque, 2014a], we investigated this latter notion of progression for continuous domains. Progression is known to come with strong negative results, and in continuous domains, we noted that further restrictions to *invertible* basic action theories may be necessary. In a nutshell, invertible theories disallow actions such as the *max* function (used in Figure 2) because they have the effect of transforming a uniform continuous distribution on h to one that is no longer purely continuous. (See [Belle and Levesque, 2014a] for examples.) In contrast, no such limitations are needed here: roughly, this is because our notion of an epistemic state is a discrete model of a continuous function, leading to a possible world-based progression notion. Consequently, however, we needed to appeal to limits here, while the progression formulation in that prior work does not [Belle and Levesque, 2014a, Corollary 12].

Second, our discussion on regression and progression in Section 5 assumes an interpreter based entirely on one of these approaches. However, a direction pursued by Erwin *et al.* [2014] maintains the sequence of performed actions and processes these via regression to progress effectively. (For example, the progression of a database wrt an action whose effects are undone by another action in the future can be avoided.) Moreover, interpreters may appeal to regression when some form of plan search is necessary within programs [Lakemeyer and Levesque, 2007]. The consideration of such ideas in the ALLEGRO context would lead to useful extensions, and is left for the future. (For example, if the performed noisy actions are characterized by normal distributions, then their conjugate property can be exploited via regression [Belle and Levesque, 2013b].)

7 Conclusions

This paper proposes an online account of belief-based programs that handles discrete and continuous probability distributions. It is intended as an alternative to GOLOG in stochastic dynamical domains where sensors and effectors are noisy.

Beginning with the expressive logical language of the situation calculus, we presented and implemented ALLEGRO using the BAT syntax of PREGO. This fragment is interesting because it embodies the desirable features of logic-based action languages, such as non-trivial successor state and likelihood axioms. The latter property together with the rich query mechanism and program syntax in ALLEGRO suggests that it could be seen as a basis for relating high-level agent programming and probabilistic robotics [Thrun *et al.*, 2005] in a general way.

There are many interesting avenues for future work, such as relating belief-based programs to decision-theoretic high-level programming [Boutilier *et al.*, 2000], particle filters [Thrun *et al.*, 2005], and probabilistic programming [Goodman *et al.*, 2008], among others. Demonstrating how the ALLEGRO system can lead to robust behavior in robots, in the sense of [Lakemeyer and Levesque, 2007], is also planned for the future.

References

[Bacchus *et al.*, 1999] F. Bacchus, J. Y. Halpern, and H. J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artif. Intell.*, 111(1–2):171 – 208, 1999.

[Bacchus, 1990] F. Bacchus. *Representing and Reasoning with Probabilistic Knowledge*. MIT Press, 1990.

[Baier *et al.*, 2007] J. A. Baier, C. Fritz, and S. A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. ICAPS*, pages 26–33, 2007.

[Belle and Levesque, 2013a] V. Belle and H. J. Levesque. Reasoning about continuous uncertainty in the situation calculus. In *Proc. IJCAI*, 2013.

[Belle and Levesque, 2013b] V. Belle and H. J. Levesque. Reasoning about probabilities in dynamic systems using goal regression. In *Proc. UAI*, 2013.

[Belle and Levesque, 2014a] V. Belle and H. J. Levesque. How to progress beliefs in continuous domains. In *Proc. KR*, 2014.

[Belle and Levesque, 2014b] V. Belle and H. J. Levesque. PREGO: An Action Language for Belief-Based Cognitive Robotics in Continuous Domains. In *Proc. AAAI*, 2014.

[Boutilier *et al.*, 2000] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proc. AAAI*, pages 355–362, 2000.

[Claßen and Lakemeyer, 2006] J. Claßen and G. Lakemeyer. Foundations for knowledge-based programs using ES. In *Proc. KR*, pages 318–328, 2006.

[Erwin *et al.*, 2014] C. Erwin, A. Pearce and S. Vassos. Transforming Situation Calculus Action Theories for Optimised Reasoning. In *Proc. KR*, 2014.

[Fagin *et al.*, 1995] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[Fan *et al.*, 2012] Y. Fan, M. Cai, N. Li, and Y. Liu. A first-order interpreter for knowledge-based golog with sensing based on exact progression and limited reasoning. In *Proc. AAAI*, 2012.

[Ferrein and Lakemeyer, 2008] A. Ferrein and G. Lakemeyer. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56(11):980–991, 2008.

[Goodman *et al.*, 2008] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proc. UAI*, pages 220–229, 2008.

[Grosskreutz and Lakemeyer, 2003] H. Grosskreutz and G. Lakemeyer. Probabilistic complex actions in golog. *Fundam. Inform.*, 57(2-4):167–192, 2003.

[Iocchi *et al.*, 2009] L. Iocchi, T. Lukasiewicz, D. Nardi, and R. Rosati. Reasoning about actions with sensing under qualitative and probabilistic uncertainty. *ACM TOCL*, 10:5, 2009.

[Kaelbling and Lozano-Pérez, 2013] L. P. Kaelbling and T. Lozano-Pérez. Integrated task and motion planning in belief space. *I. J. Robotic Res.*, 32(9-10):1194–1227, 2013.

[Kaelbling *et al.*, 1998] Leslie P. Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1–2), 1998.

[Lakemeyer and Levesque, 2007] G. Lakemeyer and H. J. Levesque. Cognitive robotics. In *Handbook of Knowledge Representation*, pages 869–886. Elsevier, 2007.

[Lang and Zanuttini, 2012] J. Lang and B. Zanuttini. Knowledge-based programs as plans - the complexity of plan verification. In *Proc. ECAI*, pages 504–509, 2012.

[Laverny and Lang, 2005] N. Laverny and J. Lang. From knowledge-based programs to graded belief-based programs, part i: On-line reasoning. *Synthese*, 147(2):277–321, 2005.

[Levesque *et al.*, 1997] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming*, 31:59–84, 1997.

[Martin and Thielscher, 2009] Y. Martin and M. Thielscher. Integrating reasoning about actions and Bayesian networks. In *Proc. ICAART*, 2009.

[Milch *et al.*, 2005] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Proc. IJCAI*, pages 1352–1359, 2005.

[Murphy, 2012] K. Murphy. *Machine learning: a probabilistic perspective*. The MIT Press, 2012.

[Nitti *et al.*, 2013] D. Nitti, T. De Laet, and L. De Raedt. A particle filter for hybrid relational domains. In *Proc. IROS*, 2013.

[Petrick and Bacchus, 2004] R.P.A. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. ICAPS*, pages 2–11, 2004.

[Puterman, 1994] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[Reiter, 2001a] R. Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT Press, 2001.

[Reiter, 2001b] R. Reiter. On knowledge-based programming with sensing in the situation calculus. *ACM TOCL*, 2(4), 2001.

[Richardson and Domingos, 2006] M. Richardson and P. Domingos. Markov logic networks. *Machine learning*, 62(1), 2006.

[Sanner and Kersting, 2010] S. Sanner and K. Kersting. Symbolic dynamic programming for first-order pomdps. In *Proc. AAAI*, pages 1140–1146, 2010.

[Sanner, 2011] S. Sanner. Relational dynamic influence diagram language (rddl): Language description. Technical report, Australian National University, 2011.

[Sardina and Lespérance, 2010] S. Sardina and Y. Lespérance. Golog speaks the BDI language. In *Programming Multi-Agent Systems*, Springer Berlin Heidelberg, 2010.

[Sardina *et al.*, 2004] S. Sardina, G. De Giacomo, Y. Lespérance, and H. J Levesque. On the semantics of deliberation in indigolog—from theory to implementation. *Annals of Mathematics and Artif. Intell.*, 41(2-4):259–299, 2004.

[Shoham, 1993] Y. Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.

[Spohn, 1988] W. Spohn. Ordinal conditional functions: A dynamic theory of epistemic states. In *Causation in Decision, Belief Change, and Statistics*. Springer Netherlands, 1988.

[Thielscher, 2001] M. Thielscher. Planning with noisy actions (preliminary report). In *Proc. Australian Joint Conference on Artificial Intelligence*, pages 27–45, 2001.

[Thielscher, 2004] M. Thielscher. Logic-based agents and the frame problem: A case for progression. In *First-Order Logic Revisited*, pages 323–336, Berlin, Germany, 2004. Logos.

[Thielscher, 2005] M. Thielscher. Flux: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.

[Thrun *et al.*, 2005] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

[Van Benthem *et al.*, 2009] J. Van Benthem, J. Gerbrandy, and B. Kooi. Dynamic update with probabilities. *Studia Logica*, 93(1):67–96, 2009.

[Van Ditmarsch *et al.*, 2007] H. Van Ditmarsch, A. Herzig, and T. De Lima. Optimal regression for reasoning about knowledge and actions. In *Proc. AAAI*, pages 1070–1075, 2007.

[Zamani *et al.*, 2012] Z. Zamani, S. Sanner, P. Poupart, and K. Kersting. Symbolic dynamic programming for continuous state and observation POMDPs. In *NIPS*, pages 1403–1411, 2012.