

# Logic Programming for Robot Control

David Poole\*

Department of Computer Science  
University of British Columbia  
2366 Main Mall

Vancouver, B.C., Canada V6T 1Z4

email: poole@cs.ubc.ca

<http://www.cs.ubc.ca/spider/poole>

## Abstract

This paper proposes logic programs as a specification for robot control. These provide a formal specification of what an agent should do depending on what it senses, and its previous sensory inputs and actions. We show how to axiomatise reactive agents, events as an interface between continuous and discrete time, and persistence, as well as axiomatising integration and differentiation over time (in terms of the limit of sums and differences). This specification need not be evaluated as a Prolog program; we use the fact that it will be evaluated in time to get a more efficient agent. We give a detailed example of a nonholonomic maze travelling robot, where we use the same language to model both the agent and the environment. One of the main motivations for this work is that there is a clean interface between the logic programs here and the model of uncertainty embedded in probabilistic Horn abduction. This is one step towards building a decision-theoretic planning system where the output of the planner is a plan suitable for actually controlling a robot.

## 1 Introduction

Since Shakey and STRIPS [Fikes and Nilsson, 1971], logic and robotics have had a tumultuous history together. While there is still much interest in the use of logic for high-level robotics (e.g., [Lesperance *et al.*, 1994; Caines and Wang, 1995]), there seems to be an assumption that low-level 'reactive' control is inherently alogical. This paper challenges this assumption.

This paper investigates the idea of using logic programs as a representation for the control of autonomous robots. This should be seen as logic programming in the sense of logic + control [Kowalski, 1979]; we use a logic program to specify what to do at each time, and use an execution mechanism that exploits a derived notion of 'state\*' in order to make it practical.

The main highlights of this approach are:

1. An agent can be seen as a transduction: a function from inputs (sensor values) into outputs (action attempts or

\*Scholar, Canadian Institute for Advanced Research

actuator settings). These are 'causal' in the sense that the output can only depend on current inputs and previous inputs and outputs. This function will be represented as a logic program specifying how the output at any time is implied by current and previous inputs. The causality ensures that we have acyclic rules.

2. The logic programs are axiomatised in *phase space* [Dean and Wellman, 1991] (the product of space and time, i.e., the predicates refer to times as part of the axiomatisation) in a similar manner to the event calculus [Kowalski and Sergot, 1986]. This allows us to axiomatise persistence as well as accumulation (integration) over time and differentiation with respect to time.
3. The notion of 'state' is a derived concept; the state is what needs to be remembered about the past in order for the agent to operate in the present. The axiomatisation is in terms of how the 'current' action depends on current inputs and past inputs and other values; the state is derived so that the output, instead of being a function of the current inputs and all past history, is a function of the current inputs and the state.
4. Although the specification of what to do looks like a Prolog program, it is not evaluated as a Prolog program. Instead we exploit the fact that the agent exists in time; that inputs are received in sequence, and that all previous inputs have already been received (and no subsequent inputs have been received) when the agent makes a decision. Instead of treating this as a logic program that may need to do arbitrary computation reasoning about the past, we actively maintain a state. The reasoning about what to do at any time depends only on the current inputs and the remembered state.

This perspective is useful for a number of reasons:

1. It provides for a representation for an agent's behaviour in a language with a well defined semantics (see [Apt and Bezem, 1991]).
2. It lets us model both the robot and the environment within the same language. The robot axioms can be evaluated in two modes. In the 'situated' mode, the agent gets sensor values directly from the environment, and acts in the environment. In simulation mode, we also have a model of the environment, and can run the models together as a simulation of the integrated system.

3. There is a clean way to integrate this with models of uncertainty (e.g., for noisy sensors and sloppy and unreliable actuators). The logic programs here are of the form that can be used within a probabilistic Horn abduction system [Poole, 1993]. One of the aims of this work is to produce a representation for robot behaviour that is both suitable for controlling a real robot and also can be the output of a decision-theoretic planning system.
4. The logic programs form an executable specification of what an agent should do. Although they can be evaluated reasonably quickly using current logic programming technology, it may be possible to compile these specifications into circuits for robots (in a manner similar to [Gaboury, 1990]).
5. It shows how two traditions in AI (namely logic-based AI and robot programming), seemingly at odds, can be unified. Whether we are successful in this remains to be seen. In particular, this paper should be seen as a proposal and an initial feasibility study — there is still much work that remains to be done before this is a competitor for programming robots.
6. Inspired by constraint nets [Zhang and Mackworth, 1995], this work shows how to model hybrid continuous-discrete systems. The axioms will all be true (in the limit) for continuous time. We derive discrete events from continuous time.

## 2 Representation

The problem that we are trying to solve is to represent, simulate and build an agent that senses and acts in the world. The agent receives a sequence (trace) of inputs (percepts or sensor values) and outputs a sequence (trace) of outputs (actions or actuator settings).

We assume a time structure  $T$  that is totally ordered and has a metric over intervals.  $T$  can either be continuous or discrete. A trace is a function from  $T$  into some domain  $A$ .

A transduction is a function from (input) traces into (output) traces that is 'causal' in the sense that the output at time  $t$  can only depend in inputs at times  $t'$  where  $t' < t$ . An agent will be a specification of a transduction.

Transductions form a general abstraction of dynamic systems [Zhang, 1994; Zhang and Mackworth, 1995; Rosen-schein and Kaelbling, 1995]. The problem that we consider is to use logic programs to specify transductions.

The language that we use is that of acyclic logic programs [Apt and Bezem, 1991], with a limited repertoire of predicates that explicitly refer to time. We assume that the acyclicity corresponds to temporal ordering (if time  $t_1$  is before time  $t_2$  then predicates referring to time  $t_1$  will be lower in the acyclic indexing than those referring to time  $t_2$ ). We will use negation as failure — for those who do not like this, we mean the completion of the program (which forms a sound and complete semantics for acyclic programs [Apt and Bezem, 1991]). The axioms below assume a limited form of arithmetic constraints.

A fluent [McCarthy and Hayes, 1969] is a function that depends on time. Each fluent has an associated set called the range of the fluent. A propositional fluent is a fluent with range  $\{true\ false\}$ . Syntactically a fluent is a term in our language.

**Definition 2.1** An agent specification module is a tuple  $(I, O, R, L, A)$  where

- $J$  is a set of fluents called the inputs. The inputs specify what sensor values will be available at various times. The range of the input trace is the cross product of the ranges of the fluents in the inputs. Atom sense  $(F, Val, T)$  is true if input fluent  $F$  has value  $Val$  at time  $T$ .
- $O$  is a set of fluents called the outputs. An output is a propositional fluent that specifies actuator settings at various times. These can also be seen as the actions of the agent (in particular, action attempts). The atom  $do(F, Val, T)$  is true if the agent sets actuator  $F$  to value  $Val$  at time  $T$ , or to 'do' action  $F = Val$  at time  $T$ .
- $R$  is a set of fluents called the recallable fluents. These are fluents whose previous value can be recalled. Recallable fluents will be used to model persistence as well as integration and differentiation.
- $L$  is a set of fluents called the local fluents. These are fluents that are neither inputs, outputs nor recallable. The predicate  $val(F, Val, T)$  is true if local fluent  $F$  has value  $Val$  at time  $T$ .
- $A$  is an acyclic logic program.  $A$  specifies how the outputs are implied by the inputs, and perhaps previous values of the recallable fluents, using local fluents, arithmetic constraints and other (non-temporal) relations as intermediaries.

**The Interface of agent specification module  $(I, O, R, L, A)$  is the pair  $(I, O)$ .**

Each rule in the logic program has a 'current' time, to which the predicates refer. There are a restricted facilities for referring to the past (i.e., to values of fluents in 'previous' times). In particular, an agent cannot recall what it hasn't remembered. There is one facility for specifying what needs to be remembered, and two predicates for recalling a value (these provide the predicates for use with recallable fluents):

$set(F, Val, T)$  is a user-defined predicate that specifies that recallable fluent  $F$  has value  $Val$  at time  $T$ . Only fluents that are specified in this way can be referred to in the future.

$was(F, Val, T_1, T)$  is a predicate that specifies that recallable fluent  $F$  was assigned value  $Val$  at time  $T_1$  and this was the latest time it was given a value before time  $T$  (so  $F$  had value  $Val$  immediately before time  $T$ ). It is axiomatised in a manner similar to the event calculus [Kowalski and Sergot, 1986] (see Section 4.5):

$$\begin{aligned} was(F, Val, T_1, T) \leftarrow \\ & time(T_1) \wedge \\ & T_1 < T \wedge \\ & set(F, Val, T_1) \wedge \\ & \sim reset\_before(F, T_1, T). \end{aligned}$$

where  $reset\_before(F, T_1, T)$  is true if fluent  $F$  was assigned a value in the interval  $(T_1, T)$ :

$$\begin{aligned} reset\_before(F, T_1, T) \leftarrow \\ & time(T_2) \\ & T_1 < T_2 \wedge \\ & T_2 < T \wedge \\ & set(F, V_2, T_2). \end{aligned}$$

The predicate  $time(T)$  is defined in Section 2.4, where we consider the problem of representing dense time.

$now(FI, Val, T)$  is a predicate that specifies that recallable fluent  $FI$  has value  $Val$  at time  $T$ . It can be axiomatised as follows:

$$\begin{aligned} now(FI, Val, T) \leftarrow \\ set(FI, Val, T). \\ now(FI, Val, T) \leftarrow \\ \sim \exists V_1 set(FI, V_1, T) \wedge \\ was(FI, Val, T_1, T). \end{aligned}$$

In user programs, the following restrictions apply:

- *set* cannot appear in the *body* of any (user) clause. It only appears in the *body* of the rules given above. It does however appear in the *head* of user clauses.
- *now* or *was* (or *reset* or *reset\_before*) do not appear in the *head* of any user clause. They are defined by the clauses above.
- All times referred to in clauses must be to the same time (the 'current time' of the clause), with the exception of the third parameter of *was*, and the predicate *now* can refer to times before the current time of the clause<sup>1</sup>, and arithmetic comparisons can be used on times.

## 2.1 Pure Reaction

We can model pure reaction, that is memoryless systems built from combinational networks (as in, e.g., [Agre and Chapman, 1987]), where the output is a function of the immediately available inputs. A logic program (with all fluents referring to the same time) can represent arbitrary combinatorial circuits.

## 12 Persistence

The use of *set*, *now* and *was* allows recallable fluents to persist. Once a value of a fluent is set that value persists until a new value is set. At any time it is always the last value set that we look at (it is this property that allows us to build efficient implementations) — where 'last' means means 'before now\*' for *was* and 'at or before now\*' for *now*.

When setting the value of fluent  $f$  at time  $T$ , we cannot use *now* for that fluent and time in the proof for *set*, as this violates the acyclicity assumption. We can however, use the *was* predicate for that fluent and time.

Persistent values are true in left closed, right open intervals. If fluent  $f$  was set to value  $v$  at time  $t_1$  and was set to a different value at time  $t_2$  (and no other settings for value  $v$  occurred in between), then the fluent  $f$  has value  $v$  in the interval  $t_1, t_2$ . This is the opposite convention to the event calculus [Shanahan, 1990]. We want this convention as robots have internal state so that it can affect what they will do; if a robot realises at time  $t_2$  that it should be doing something different, then it should change what it is doing immediately, and not wait.

This notion of persistence is close to that of the event calculus [Kowalski and Sergot, 1986; Shanahan, 1990] (see Section 4.5).

<sup>1</sup>This is to allow us to model 'transport delays' (see Section 4.2), that are essential for the modelling of analogue systems. In general using this facility means that we have to maintain a history of *set* values and not just a state of *set* values.

## 2.3 Integration and Differentiation

One of the reasons for axiomatising in phase space and making time explicit is that we can integrate over time. We will assume that anything we try to integrate is Riemann integrable.

If we want to produce the value of some accumulative predicate, we can use

$$\begin{aligned} F(t_1) &= F(t_0) + \int_{t_0}^{t_1} f(t) dt \\ &= \lim_{\delta \rightarrow 0} \left( F(t_0) + \sum_{k=1}^{(t_1-t_0)/\delta} f(t_0 + k\delta) \times \delta \right) \end{aligned}$$

where  $f(t) = \frac{dF(t)}{dt}$ . For any fixed  $\delta$  we can compute the sum recursively. The integral is the limit as  $\delta$  approaches zero. We can write the sum that approaches the integral using the following schema (with a rule for each integrable fluent  $f$ ):

$$\begin{aligned} set(f, V_1 + Df \times (T - T_1), T) \leftarrow \\ was(f, V_1, T_1, T) \wedge \\ val(deriv(f), Df, T). \end{aligned}$$

Similarly we can axiomatise the derivative of a fluent with respect to time using the schema for each differentiable fluent  $f$ :<sup>2</sup>

$$\begin{aligned} val(deriv(f), Df, T) \leftarrow \\ now(f, V, T) \wedge \\ V - V_1 = Df \times (T - T_1) \wedge \\ was(f, V_1, T_1, T). \end{aligned}$$

Solving these may involve some constraint solving.

Before we define 'true in the limit', we discuss some issues relevant to the acyclicity restriction. Suppose, the derivative of integrable fluent  $f$  is a function of the value of  $f$  at  $T$  (e.g., the force of a spring is a function of the position of the object). To define the clause for the derivative, we need to determine the value of  $f$  at time  $T$ , but cannot have  $now(f, V, T)$  in the *body* of a rule to prove  $val(deriv(f), Df, T)$ , as this violates the acyclicity constraint. There are two solutions to this problem: the forward Euler and the backward Euler.

Assume the relation  $val(deriv(f, V), DF, T)$  is defined which means that  $DF$  is the derivative of  $f$  at time  $T$ , with  $V$  being  $f$ 's value.

The 'forward Euler' uses the 'previous' value of  $V$  (i.e. the value  $V_1$  in the integration clause above). This is correct as 'in the limit'  $V = V_1$ .

$$\begin{aligned} set(f, V_1 + Df \times (T - T_1), T) \leftarrow \\ was(f, V_1, T_1, T) \wedge \\ val(deriv(f, V_1), Df, T). \end{aligned}$$

This solution turns out to have problems with the stability of discretizations. The 'backward Euler', which works better in practice, uses the 'future' value for the derivative:

$$\begin{aligned} set(f, V, T) \leftarrow \\ was(f, V_1, T_1, T) \wedge \end{aligned}$$

<sup>2</sup>These rules are schemata as we have to choose whether a fluent is integrable or derivable (or neither). A fluent cannot be both: we cannot use the differences in values to compute derivatives as well as using derivatives to compute differences in values. This violates the acyclicity restriction, and leads to infinite regress.

$$V - V_1 = Df \times (T - T_1) \wedge \\ \text{val}(\text{deriv}(f, V), Df, T).$$

Solving this may involve some constraint solving.

## 2.4 Truth in the limit

The axioms given for *was* and *now* are incomplete — they do not specify the structure of time.

If time is discrete<sup>3</sup>, then there are no interpretation problems with the axioms above. In particular, we make the predicate *time(ti)* true for each 'time' *t*. The predicate *was* always refers to the previous time point (or to the last time point when the value was set), and there is always some finite duration of each time interval.

If time is continuous, there are semantic difficulties in interpreting these sentences (in particular the integration and differentiation formulae that allow for the setting of values at each time point). We cannot interpret the integration and differentiation axioms 'in the limit', as in the limit,  $V = V_1$  and  $T = T_1$ ; the integration axioms become cyclic (and tautologies), and the differentiation axioms provide no constraints on the value of  $Df$ .

In order to be able to interpret the above sentences we have to consider the limit as finite discretizations become finer and finer (in the same way that integration is defined). The axioms will talk about what is true for each discretization. The values that '*was*' refers to will be well defined for each of these discretizations. The meaning for the continuous case will be what is true in the limit.

To define the limit, consider a uniform discretization with time interval  $dt > 0$ . For each  $dt$  we consider the discretization that consists of the time points  $k \times dt$  for some integer  $k$ .

**Definition 2.2** Axioms  $\mathcal{A}$  entails fluent  $f$  has value  $v$  under discretization  $dt > 0$  at time  $t$ , written  $\mathcal{A} \models_{dt} \text{now}(f, v, t)$  if

$$\mathcal{A} \cup \{\text{time}(K \times dt) \leftarrow \text{integer}(K)\} \models \text{now}(f, v, t)$$

where  $\models$  denotes truth under Clark's completion, or in the (unique) stable model, or one of the other equivalent semantics for acyclic logic programs [Apt and Bezem, 1991], of the axioms together with axioms defining arithmetic.

**Definition 2.3** Axioms  $\mathcal{A}$  entails fluent  $f$  has value  $v$  in the limit at time  $t$ , written  $\mathcal{A} \models_{\infty} \text{now}(f, v, t)$ , if for every  $\epsilon > 0$  there exists  $\delta > 0$  such that if  $0 < dt < \delta$ , then  $\mathcal{A} \models_{dt} \text{now}(f, v', t)$  where  $|v - v'| < \epsilon$ .

One subtlety should be noted here. We can ask about  $\text{now}(f, v, t)$  for any  $t$  even if  $\text{time}(t)$  is not true — if this were not the case these definitions would not work, as most values of  $dt$  'miss' any particular time (i.e., for most times  $t$  and increments  $dt$  there does not exist an integer  $k$  such that  $t = k \times dt$ ). Being an official 'time' only constrains what past values can be referred to.

Note that we never consider the theory with the infinite partition.

<sup>3</sup>By discrete I mean that there are no sequences of different values which get closer and closer (Cauchy sequences — see e.g. [Zhang and Mackworth, 1995]). Informally this means that there are only finitely many time points between any two time points.

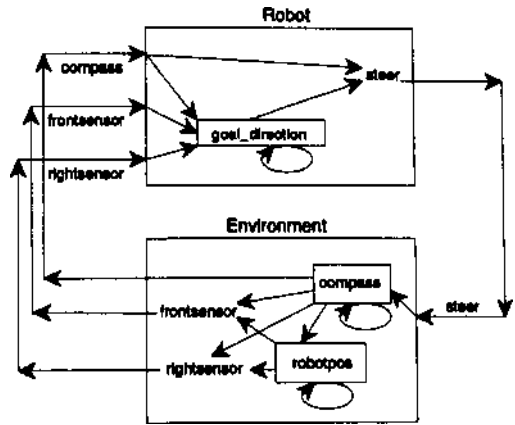


Figure 1: Coupled Robot and Environment

## 3 An Example in Detail

We demonstrate the representation using an example of modelling a robot and an environment. These two axiomatisations will highlight different features; the robot model will highlight reactive systems with remembered events; the environment model will highlight integration over time.

The example is of a maze travelling robot that is continuously trying to go East (i.e., at  $0^\circ$  orientation), but may have to avoid obstacles. The robot can sense obstacles and its direction of travel, but only has control over its direction of steering.

### 3.1 Robot Model

We assume that the robot can sense which direction it is travelling in, has a sensor on the front of the robot and a sensor on the right that can detect obstacles. The only control that the agent has is to change the steering angle — we assume that the agent can instantaneously change from steering left, right or straight (but steering takes time to achieve the desired effect). This example is adapted from [Zhang, 1994].

For the robot specification, we axiomatise what the steering should be depending on current, and perhaps previous sensor values.

We use the following predicates defining the inputs at different times:

*sense(compass, C, T)* means that the robot is sensing that it is heading in direction  $C$  at time  $T$ . All directions are in degrees anticlockwise from East (i.e., the standard directions in an  $x$ - $y$  graph).

*sense(front, blocked, T)* means that the front sensor is detecting an obstacle at time  $T$ .

*sense(right, blocked, T)* means that the right sensor is on at time  $T$ .

and the output:

*do(steer, D, T)* means the robot should steer  $D$ -wards at time  $T$  where  $D \in \{\text{left, right, straight}\}$ .

The following clauses axiomatise a 'bang-bang' controller that specifies which direction to steer based on the compass reading and the current desired direction of the robot. In these clauses,  $\text{now}(\text{goal\_direction}, G, T)$  is true if the robot wants to go in direction  $G$  at time  $T$ :

```

do(steer, left, T) ←
  now(goal_direction, G, T) ∧
  sense(compass, C, T) ∧
  (G - C + 540) mod 360 - 180 > 5.
do(steer, straight, T) ←
  now(goal_direction, G, T) ∧
  sense(compass, C, T) ∧
  |(G - C + 540) mod 360 - 180| ≤ 5.
do(steer, right, T) ←
  now(goal_direction, G, T) ∧
  sense(compass, C, T) ∧
  (G - C + 540) mod 360 - 180 < -5.

```

$5^\circ$  is an arbitrarily chosen threshold.

The goal direction depends on current and previous sensor values. The robot changes the goal direction when it is travelling in the previous desired direction<sup>4</sup> and it finds its way blocked, in which case it now wants to travel at  $90^\circ$  to the left of where it was travelling:

```

set(goal_direction, D + 90, T) ←
  sense(compass, C, T) ∧
  was(goal_direction, D, ¬T) ∧
  same(C, D) ∧
  sense(front, blocked, T).

```

Alternatively the robot changes its desired direction by  $90^\circ$  to the right if it is travelling in the previous desired direction, it is not blocked on the right and is not going in its ultimate desired direction (which is  $0^\circ$ ). Note that sometimes the desired direction is  $360^\circ$  — in this case it still wants to turn right to follow the right wall — this will enable the robot to get out of maze traps (such as the maze in Figure 2).

```

set(goal_direction, D - 90, T) ←
  sense(compass, C, T) ∧
  was(goal_direction, D, ¬T) ∧
  same(C, D) ∧
  D ≠ 0 ∧
  ~ sense(front, blocked, T) ∧
  ~ sense(right, blocked, T).

```

In order to allow for noise and sampling error the robot needs only the desired direction to be within some margin of error to the current direction. The predicate  $\text{same}(D_1, D_2)$  is true if directions  $D_1$  and  $D_2$  are the same up to this error.

```

same(A1, A2) ←
  |A1 - A2| mod 360 ≤ 10.

```

With this model, we can derive what to do at any time based on what sensor values it has received.

<sup>4</sup>This means that the robot designer (and modeller) does not need to consider how the sensors work in the middle of a turn. This was chosen to show how 'events' can be derived from continuous change.

Such a logic program could be run as a situated robot that gets sensor values from the environment and acts in the environment (see Section 3.3). In this paper, in order to (a) show the generality of the model, and (b) to show how integration over time can work, we will use the same language to model the environment. The environment model together with the agent model can be used to simulate the system.

### 3.2 Environment Model

We assume that the environment is another agent like the robot. It has inputs (from the output of the robot) and has outputs which can be sensed by the robot. The two models can be joined together to form a feedback control system. The main constraint is that the conjoined logic programs are acyclic [Apt and Bezem, 1991] with the indexing due to acyclicity constrained to be temporal.

Whether the front sensor is on depends on the position of the robot, the direction of the robot and whether there is a wall close to the robot in front of the robot. Here  $\text{robotpos}(X, Y, T)$  means that the robot is at position  $(X, Y)$  at time  $T$ .  $\text{seeblock}(X, Y, D)$  is true if the robot can detect a wall in direction  $D$  from position  $(X, Y)$  — this does not depend on the time, but only on the position and direction of the robot.

```

sense(front, blocked, T) ←
  now(compass, D, T) ∧
  now(robotpos, (X, Y), T) ∧
  seeblock(X, Y, D).

```

Similarly we can axiomatise  $\text{sense}(\text{right}, \text{blocked}, T)$ , and  $\text{sense}(\text{compass}, D, T)$  and can axiomatise the maze using  $\text{seeblock}$ .

The position of the robot is the integral of the velocities over time:

```

set(compass, (C + CD × (T - T1) + 360)
  mod 360, T) ←
  was(compass, C, T1, T) ∧
  val(deriv(compass), CD, T1).

```

The derivatives are axiomatised as follows: If robot is steering left,  $DC/DT = 10$  (i.e., 10 degrees per time unit). If robot is steering straight,  $DC/DT = 0$ . If robot is steering right,  $DC/DT = -10$ .

```

val(deriv(compass), 10, T) ←
  do(steer, left, T).
val(deriv(compass), 0, T) ←
  do(steer, straight, T).
val(deriv(compass), -10, T) ←
  do(steer, right, T).

```

The position of the robot is the integral of velocities:

```

set(robotpos, (X1 + DX × (T - T1),
  Y1 + DY × (T - T1)), T) ←
  was(robotpos, (X1, Y1), T1, T) ∧
  val(deriv(robotpos), (DX, DY), T).

```

The following axioms define the  $x$  derivatives and the  $y$  derivatives of the position with respect to time. We are assuming that the speed is 1 and that cos and sin use degrees (as

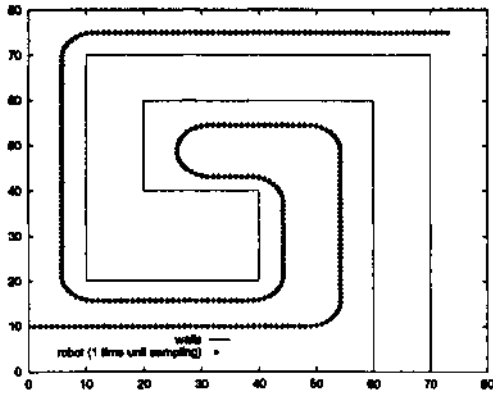


Figure 2: Simulation of the robot in a maze.

opposed to, say, radians).

```

val(deriv(robotpos), (cos(D), sin(D)), T) ←
now(compass, D, T).

```

### 3.3 Computation

If we were to run the above axiomatisation as a Prolog program, the code is hopelessly inefficient. The problem is that we have to consider all previous times to check whether an event occurred (at least all previous times where inputs arrived). Moreover to check whether an event occurred, we have to check all previous times to check whether a previous event occurred. As you can imagine, such computation is hopelessly slow.

In order to make this efficient, we take advantage of the fact that we are evaluating in time: at each time all previous observations have arrived and no subsequent observations have arrived. We exploit the fact that all of the references to the past are in terms of *was*. Instead of using the axioms defining *was* explicitly, we actively maintain a state, always remembering the latest values that were *set*. The predicate *was* can be implemented by looking up the latest *set* values. In other words, the *set* values are remembered forming the state of the agent. The logic program is evaluated by proving the output from the current inputs and the current state.

Figure 2 shows a simulation of the robot in the maze. It is simulated by discretising time with one time unit intervals. Other discretizations, as long as they are not too coarse give similar results.

The above simulation (of both the robot and the environment), ran faster than 20 steps per second, on a 68040 running Sicstus Prolog. Partial evaluation should be able to speed this up, and it seems that it should be possible to compile the logic program specification into hardware (as does [Gaboury, 1990]). Thus it seems as though the logical specification of robot action is not impractical from an efficiency point of view.

## 4 Discussion and Comparison

This paper is not intended to just define yet another robot programming language. Let us take the very general view of an agent as a 'causal' function from input history to outputs

[Zhang and Mackworth, 1995; Rosenschein and Kaelbling, 1995]. Suppose we want to use logic as a formal specification for the actions of the robot, for example in order to prove theorems about the robot behaviour. If we are to treat values of the inputs at various times and values of the outputs at various times as propositions, then the constraint imposed by the robot function is that it can't be the case that the inputs have certain values and the output is not the appropriate function of the inputs; but this is exactly the definition of a definite clause: inputs imply the outputs.

It might be argued that the logic here is too weak to represent what we want to, for example it cannot represent disjunction. We have to be careful here; if a robot is to do something, it cannot be unsure about its own actions. It must commit to one action in order to carry it out. A robot cannot 'do' action  $a \vee a2$  without doing one of them. This does not mean that the agent cannot be ignorant (or unsure) of what other agents will do, or be unsure about what values it will receive from the environment. For a general discussion of these issues, and a way to handle them within the logic presented here (by allowing independent 'choices' made by different agents and nature) see [Poole, 1995].

### 4.1 Noisy sensors and actuators

The above axiomatisation showed how to model partial information about the environment (the agent had very limited sensing ability). In this section we sketch a way to model noisy sensors and actuators using a continuous version of probabilistic Horn abduction [Poole, 1993; 1995].

The general idea of probabilistic Horn abduction is that there is a probability distribution over possible world generated by unconditionally independent random variables. A logic program gives the consequences of the random choices for each world. Formally, a possible world selects one value from each alternative (disjoint set); what is true in the possible world is defined by the unique stable model of the selection and the acyclic logic program [Poole, 1995]. The probability of the world is the product of the probabilities of the values selected by the world. In this framework, the logic programs can still be interpreted logically, and the resulting framework, although based on independent random variables, can represent any probability distribution [Poole, 1993].

To model noisy sensors, we add an extra 'noise\*' term to the rules. For example, to represent additive Gaussian noise for the compass sensor, with standard deviation 3, we can use the rule:

```

sense(compass, C + 3 * N, T) ←
now(compass, C, T) ∧
val(c.noise, N, T).

```

Where for all  $T$ ,  $\{val(c.noise, N, T) : N \in \mathbb{R}\}$  is an alternative set.  $val(c.noise, N, T)$  is true in world  $w$  if the compass noise is  $N$  standard deviations from the mean at time  $T$  in world  $w$  (each world has a unique  $N$  that is true for each time).  $N$  is normally distributed with mean 0 and standard deviation 1 (this is usually called the  $z$ -score):

$$\forall T \forall N P(val(c.noise, N, T)) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}N^2}$$

With such a noisy sensor the agent could do dead reckoning; maintaining its own record of its position. However if the

actuator is also unreliable, then the errors explode. Unreliable actuators can be modelled similarly to the noisy sensors, for example,

```

set(compass, (C + (CD + E) × (T - T1) + 360)
      mod 360, T) ←
  was(compass, C, T1, T) ∧
  val(deriv(compass), CD, T1) ∧
  val(a.error, E, T).

```

Where *a.error* is treated analogously to *cjnoise*.

When the dynamics are linear and the noise is Gaussian, the posterior distributions can be solved analytically, as in the Kalman filter (see [Dean and Wellman, 1991]).

## 42 Constraint Nets

Constraint nets [Zhang and Mackworth, 1995] form a modelling language for hybrid systems that combines discrete and continuous time, and discrete and continuous domains into a coherent framework. This is done by abstracting the notion of time so that it covers both discrete and continuous models of time, and using 'events' as the interface between continuous and discrete time.

Constraint nets are built from three basic transductions. Transliterations, which do not depend on the past, are axiomatised here by allowing (acyclic) logic programs to specify how current outputs depend on current inputs. Our *was* predicate corresponds to unit delays. Transport delays of time *r* can be modelled as the atom *now*(*FI*, *V*, *T* - *r*) — to implement these we have to maintain a history of at least *r* long, and not just a state.

## 43 Logic Control

COCOLOG [Caines and Wang, 1995] is a logic for discrete control that shares many features of the discrete form of the logic here. The main difference is that in COCOLOG, the state of the system is an explicit term of the language. The language is also more complicated than the simple Horn clauses used here and the main control loop is extra-logical.

The declarative control of Nerode and Kohn [1994] uses Prolog for control. Their main aim is for a Prolog program to prove that some action is optimal. Their Prolog rules are at a much different level than the simple rules used here, which are impractical when using Prolog's back-chaining search strategy.

## 4.4 GOLOG

GOLOG [Lespérance *et al.*, 1994] is a programming language for robots based on the situation calculus. Unlike the proposal in this paper, GOLOG programs are not sentences in a logic. Rather the logic is at the meta-level providing a semantics for the Algol-like GOLOG language.

One intriguing idea is to use the logic programming approach here to write a low-level controller that interprets GOLOG programs. This could be done by having two state variables, one that is the current action the agent is 'doing' and one is a list of actions 'to do'. The rules could be used to reduce expressions in time, for example to interpret action sequences we can use:

```

set(doing, A, T) ∧
set(todo, [B|C], T) ←

```

```

was(doing, (A; B), T) ∧
was(todo, C, T).

```

Similarly, we can interpret more complicated constructs such as while loops as well as monitoring 'primitive' actions (e.g., we can see setting the goal direction in the above example as a high level action that decomposes into continuous action, and is monitored as to when it is complete). A full discussion of this is beyond the scope of this paper.

## 4.5 Event Calculus

The event calculus [Kowalski and Sergot, 1986; Shanahan, 1990] provides a mechanism to represent persistent properties over intervals from events that make the properties true and false. What is new in this paper is deriving events from changes in continuous properties, having cumulative properties, and exploiting evaluation in time to gain efficiency.

There is quite a simple translation to map the event calculus into the framework of this paper. The event calculus uses the predicates: *happens*(*E*, *T*) is true if event *E* happens at time *T*; *initiates*(*E*, *P*) is true if event *E* makes *P* true; *terminates*(*E*, *P*) is true if event *E* makes *P* no longer true. These can be mapped into the fluent representation used here:

```

set(P, true, T) ←
  initiates(E, P) ∧
  happens(E, T).
set(P, false, T) ←
  terminates(E, P) ∧
  happens(E, T).

```

If we want the convention used in this paper that predicates are true in left closed intervals, we can represent *holds*(*P*, *T*) (meaning predicate *P* holds at time *T*) by:

```

holds(P, T) ←
  now(P, true, T).

```

One main advantage of our representation is that, when we act in time, and all of the *set*'s are done in temporal ordering, and we maintain a state, then we can implement *holds* very fast, by looking up the last value that was assigned to the variable.

Shanahan's notion of 'autotermination' is similar to our deriving events from continuous change.

## 4.6 Other Mixes of Logic and Continuous Time

There have been other proposed mixes of logic and continuous time (e.g., [Sandewall, 1989; Shanahan, 1990; Dean and Siegle, 1990; Trudel, 1991; Pinto and Reiter, 1995]), but in all of these either "during the time span of a situation no fluents change truth values" [Pinto and Reiter, 1995] or the axiomatiser needs to know a priori how properties accumulate (they effectively do integration off-line). For robot control, we do not know how the sensor values will change; the best we can do is to derive (estimate) integrals online. None of these other proposals let us do this.

## 5 Conclusion

This paper has argued the logic programs can be used effectively as a programming language for robot control. The logic program forms an executable specification of what the robot

should do. The same language can be used for modelling the robot and the environment (and also multiple robots). This axiomatisation can be combined with probabilistic Horn abduction [Poole, 1993] to allow for modelling uncertainty in the environment (e.g., exogenous events, noisy sensors and unreliable actuators).

This paper has not described some ideas about improving efficiency by adaptive sampling: by partially evaluating the logic program, we can determine what inputs we must look for in order for an event to occur. When running the robot, we can build into our sensors detectors for these conditions; when detected, we can run the program in a forward direction to derive events. In the simulation, we can query the environment to determine when these events would occur. Such ideas are currently being pursued.

#### Acknowledgements

Thanks to Alan Mackworth and Ying Zhang for interesting discussions on hybrid systems. Thanks to Mike Horsch for comments on a previous draft. This work was supported by Institute for Robotics and Intelligent Systems, Project IC-7 and Natural Sciences and Engineering Research Council of Canada Operating Grant OGP0044121.

#### References

- [Agre and Chapman, 1987] P. E. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In Proc. 6th National Conference on Artificial Intelligence, pages 268-272, Seattle, Washington, 1987.
- [Apt and Bezem, 1991] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3-4):335-363, 1991.
- [Caines and Wang, 1995] P. E. Caines and S. Wang. CO COLOG: A conditional observer and controller logic for finite machines. *SIAM Journal of Control*, to appear, November 1995.
- [Dean and Siegle, 1990] T. Dean and G. Siegle. An approach to reasoning about continuous change for applications in planning. In Proc. 8th National Conference on Artificial Intelligence, pages 132-137, Boston, MA, 1990.
- [Dean and Wellman, 1991] T. L. Dean and M. P. Wellman. *Planning and Control*. Morgan Kaufmann, San Mateo, California, 1991.
- [Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189-208, 1971.
- [Gaboury, 1990] P. Gaboury. VLSI architecture design using predicate logic. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, September 1990.
- [Kowalski and Sergot, 1986] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67-95, 1986.
- [Kowalski, 1979] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424-431, 1979.
- [Lespérance et al., 1994] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. A logical approach to high-level robot programming - a progress report. In B. Kuipers, editor, *Control of the Physical World by Intelligent Systems, Papers from the 1994 AAAI Fall Symposium*, pages 79-85, New Orleans, November 1994.
- [McCarthy and Hayes, 1969] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In M. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463-502. Edinburgh University Press, 1969.
- [Nerode and Kohn, 1994] A. Nerode and W. Kohn. Multiple agent hybrid control architecture. In R. L. Grossman, et al., editor, *Hybrid Systems*, pages 297-316. Springer Verlag, Lecture Notes in Computer Science 736, 1994.
- [Pinto and Reiter, 1995] J. Pinto and R. Reiter. Reasoning about time in the situation calculus. *Annals of Mathematics and Artificial Intelligence*, special festschrift issue in honour of Jack Minker, to appear, 1995.
- [Poole, 1993] D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81-129, 1993.
- [Poole, 1995] D. Poole. Sensing and acting in the independent choice logic. In *Working Notes AAAI Spring Symposium 1995 — Extending Theories of Actions: Formal Theory and Practical Applications*, pages ??-??, ftp://ftp.cs.ubc.ca/ftp/local/poole/papers/actions.ps.gz, 1995.
- [Rosenschein and Kaelbling, 1995] S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73:149-173, 1995.
- [Sandewall, 1989] E. Sandewall. Combining logic and differential equations for describing real-world systems. In Proc. First International Conf. on Principles of Knowledge Representation and Reasoning, pages 412-420, Toronto, 1989.
- [Shanahan, 1990] M. Shanahan. Representing continuous change in the event calculus. In Proc. ECAI-90, pages 598-603, 1990.
- [Trudel, 1991] A. Trudel. The interval representation problem. *International Journal of Intelligent Systems*, 6(5):509-547, 1991.
- [Zhang and Mackworth, 1995] Y. Zhang and A. K. Mackworth. Constraint nets: a semantic model for hybrid dynamic systems. *Theoretical Computer Science*, 138:211-239, 1995.
- [Zhang, 1994] Y. Zhang. A Foundation for the Design and Analysis of Robotic Systems and Behaviour. PhD thesis, Department of Computer Science, University of British Columbia, September 1994.