High-Level Robot Control Through Logic

Murray Shanahan and Mark Witkowski

Department of Electrical and Electronic Engineering,
Imperial College,
Exhibition Road,
London SW7 2BT,
England.
`m.shanahan@ic.ac.uk, m.witkowski@ic.ac.uk`

Abstract. This paper presents a programmable logic-based agent control system that interleaves planning, plan execution and perception. In this system, a program is a collection of logical formulae describing the agent's relationship to its environment. Two such programs for a mobile robot are described — one for navigation and one for map building — that share much of their code. The map building program incorporates a rudimentary approach to the formalisation of epistemic fluents, knowledge goals, and knowledge producing actions.

Introduction

Contemporary work in *cognitive robotics* has demonstrated the viability of logic-based high-level robot control [Lespérance, *et al.*, 1994], [De Giacomo, *et al.*, 1997], [Baral & Tran, 1998], [Shanahan, 2000b]. Building on the progress reported in [Shanahan, 2000b], this paper describes an implemented logic-based, high-level robot control system in the cognitive robotics style. The controller is programmed directly in logic, specifically in the event calculus, an established formalism for reasoning about action. The controller's underlying computational model is a sense-plan-act cycle, in which both planning and sensor data assimilation are abductive theorem proving tasks.

Two small application programs written in this language are described in detail, one for navigation and one for map building. In navigation, the abductive processing of sensor data results in knowledge of the status (open or closed) of doors, while in map building (during which all doors are assumed to be open), it results in knowledge of the layout of rooms and doorways.

Both these programs have been deployed and tested on a Khepera robot. The Khepera is a miniature robotic platform with two drive wheels and a suite of eight infra-red proximity sensors around its circumference. The robot inhabits a miniaturised office-like environment comprising six rooms connected by doors which can be either open or closed (Figure 1). The robot cannot distinguish a closed door from a wall using its sensors alone, but has to use a combination of sensor data and abductive reasoning.

High-level control of the robot is the responsibility of an off-board computer, that communicates with the robot via an RS232 port. The high-level controller can initiate low-level routines that are executed on-board, such as wall following, corner turning, and so on. Upon termination, these low-level routines communicate the status of the robot's sensors back to the high-level controller, which then decides how to proceed. The implementation details of the low-level actions are outside the scope of this paper, whose aim is to present the high-level controller.

It should be noted that the aim of the paper is not to present advances in any particular sub-area of AI, such as planning, knowledge representation, or robotics, but rather to show how techniques from these areas can be synthesised and integrated into an agent architecture, using logic as a representational medium and theorem proving as a means of computation.

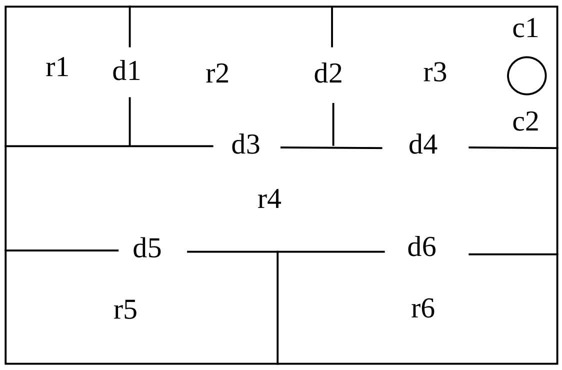

Figure 1: The Robot's Environment

1 Theoretical Background

A high-level control program in our system is a set of logical formulae describing actions and their effects, those of both the robot and other agents. The formalism used to represent actions and their effects is the event calculus, and the frame problem is overcome using circumscription, as set out in [Shanahan, 1997a].

The ontology of the event calculus includes *fluents*, *actions* (or *events*), and *time points*. The standard axioms of the event calculus (whose conjunction is denoted EC) serve to constrain the predicate HoldsAt, where HoldsAt(β,τ) represents that fluent β holds at time τ. Here are two examples of these axioms.

$$
\begin{aligned}
&\text{HoldsAt}(f,t3) \leftarrow \\
&\quad \text{Happens}(a,t1,t2) \wedge \text{Initiates}(a,f,t1) \wedge \\
&\qquad t2 < t3 \wedge \neg\, \text{Clipped}(t1,f,t3)
\end{aligned}
$$

$\text{Clipped}(t1,f,t4) \leftrightarrow$
$\quad \exists\, a,t2,t3\ [\text{Happens}(a,t2,t3) \wedge t1 < t3 \wedge t2 < t4 \wedge$
$\quad\quad \text{Terminates}(a,f,t2)]$

A particular *domain* is described in terms of Initiates and Terminates formulae. Initiates(α,β,τ) represents that fluent β starts to hold after action α at time τ. Conversely, Terminates(α,β,τ) represents that β starts not to hold after action α at τ.

A particular *narrative* of events is described in terms of Happens and Initially formulae. The formulae $\text{Initially}_P(\beta)$ and $\text{Initially}_N(\beta)$ respectively represent that fluent β holds at time 0 and does not hold at time 0. Happens($\alpha,\tau1,\tau2$) represents that action or event α occurs, starting at time $\tau1$ and ending at time $\tau2$. Happens(α,τ) is equivalent to Happens(α,τ,τ).

Both planning and sensor data assimilation can be viewed as abductive tasks with respect to the event calculus [Shanahan, 1997a], [Shanahan, 2000a]. First, we need to construct a theory Σ of the effects of the robot's actions on the world and the impact of the world on the robot's sensors. Then, roughly speaking (omitting details of the circumscriptions), given a conjunction Γ of goals (HoldsAt formulae), and a conjunction Δ_I of formulae describing the initial state, a plan is a consistent conjunction Δ_P of Happens and temporal ordering formulae such that,

$$\Sigma \wedge \Delta_I \wedge \Delta_P \wedge EC \vDash \Gamma.$$

In order to interleave planning, sensing and acting effectively, we need to carry out hierarchical planning. The logical story for hierarchical planning is more or less the same, with the addition to Σ of Happens formulae describing how a compound action decomposes into its constituent actions. For more details, see [Shanahan, 2000a].

A similar abductive account of sensor data assimilation (SDA) can be constructed. The need for such an account arises from the fact that sensors do not deliver facts directly into the robot's model of the world. Rather, they provide raw data from which facts can only be inferred. Given a conjunction Δ_N of Happens and temporal ordering formulae describing the actions already carried out by the robot, and a description Γ of the sensor data received, an explanation of that sensor data is a consistent Ψ such that,

$$\Sigma \wedge \Delta_N \wedge \Psi \wedge EC \vDash \Gamma.$$

The predicates allowed in Ψ depend on the task at hand, either map building or navigation.

In the present system, both these abductive tasks — planning and SDA — are implemented by a single logic programming meta-interpreter. This meta-interpreter is sound and complete for a large class of domain theories. For more details, see [Shanahan, 2000a].

2 Robot Programming in the Event Calculus

This section describes the robot's control system in more detail. In essence, it is a general purpose high-level agent control system, programmable directly in the event calculus. Although the focus of the present discussion is on robotics, the technology is applicable to other types of agent as well.

The system executes a sense-plan-act cycle. The execution of this cycle has the following features.

- Planning and sensor data assimilation are both resolution-based *abductive* theorem proving processes, working on collections of event calculus formulae. These processes conform to the logical specifications sketched in the previous section.
- Planning and SDA are both *resource-bounded* processes. They are subject to constant suspension to allow the interleaving of sensing, planning and acting. The abductive meta-interpreter is made resource-bounded using techniques similar to those described in [Kowalski, 1995].
- To permit reactivity, planning is *hierarchical*. This facilitates planning in progression order, which promotes the rapid generation of a first executable action.
- The results of sensor data assimilation can expose conflicts with the current plan, thus precipitating *replanning*.

An event calculus robot program comprises the following five parts.

A. A set of Initiates and Terminates formulae describing of the effects of the robot's primitive, low-level actions.

B. A set of Happens formulae describing the causes of robot sensor events.

C. A set of Initiates and Terminates formulae describing the effects of high-level, compound actions.

D. A set of Happens formulae defining high-level, compound actions in terms of more primitive ones. These definitions can include sequence, choice, and recursion.

E. A set of declarations, specifying, for example, which predicates are abducible.

The formulae in A to D figure in the sense-plan-act cycle in the following way. Initially, the system has an empty plan, and is presented with a goal Γ in the form of a HoldsAt formula. Using resolution against formulae in C, the planning process identifies a high-level action α that will achieve Γ. (If no such action is available, the planner uses the formulae in A to plan from first principles.) The planning process then decomposes α using resolution against formulae in D. This decomposition may yield any combination of the following.

- Further sub-goals to be achieved (HoldsAt formulae).
- Further sub-actions to be decomposed (Happens formulae).
- Executable, primitive actions to be added to the plan (Happens formulae).
- Negated Clipped formulae, analogous to protected links in partial-order planning, whose validity must be preserved throughout subsequent processing.

As soon as a complete but possibly not fully decomposed plan with an executable first action is generated, the robot can act.

Meanwhile, the SDA process is also underway. This receives incoming sensor events in the form of Happens formulae. Using resolution against formulae in B, the

SDA process starts trying to find an explanation for these sensor events. This may yield any combination of Happens, HoldsAt and negated Clipped formulae, which are subject to further abductive processing through resolution against formulae in A, taking into account the actions the robot itself has performed.

In many tasks, such as navigation, the SDA process ultimately generates a set of abduced Happens formulae describing external actions (actions not carried out by the robot itself) that explain the incoming sensor data. Using resolution against formulae in A, it can be determined whether these external events threaten the validity of the negated Clipped formulae (protected links) recorded by the planning process. If they do, the system replans from scratch.

In the context of this sense-plan-act cycle, the event calculus can be regarded as a logic programming language for agents. Accordingly, event calculus programs have both a *declarative* meaning, as collections of sentences of logic, and a *procedural* meaning, given by the execution model outlined here. The following sections present two robotic applications written as event calculus programs, namely navigation and map building.

Although neither of the robot programs presented here exhibits much reactivity, the system does facilitate the construction of highly reactive control programs. The key to achieving reactivity is to ensure that the program includes high-level compound actions that quickly decompose, in as many situations as possible, to a first executable action. Although each unexpected event will precipitate replanning from scratch, this replanning process then very rapidly results in an appropriate new action to be executed.

3 A Navigation Program

Appendices A and C of the full paper contain (almost) the complete text of a working event calculus program for robot navigation. This section describes the program's construction and operation. (Throughout the sequel, fragments of code will be written using a Prolog-like syntax, while purely logical formulae will retain their usual syntax.)

The robot's environment (Figure 1) is represented in the following way. The formula `connects(D,R1,R2)` means that door `D` connects rooms `R1` and `R2`, `inner(C)` means that corner `C` is a concave corner, `door(D,C1,C2)` means corners `C1` and `C2` are door `D`'s doorposts, and `next_corner(R,C1,C2)` means that `C2` is the next corner from `C1` in room `R` in a clockwise direction, where `C1` and `C2` can each be either convex or concave. A set of such formulae (a *map*) describing the room layout is a required background theory for the navigation application, but is not given in the appendices.

The robot has a repertoire of three primitive actions: `follow_wall`, whereby it proceeds along the wall to the next visible corner, `turn(S)`, whereby it turns a corner in direction `S` (either `left` or `right`), and `go_straight`, whereby it crosses a doorway. For simplicity, we'll assume the robot only proceeds in a clockwise direction around a room, hugging the wall to its left. The navigation domain comprises just two fluents. The term `in(R)` denotes that the robot is in room `R`, while the term `loc(corner(C),S)` denotes that the robot is in corner `C`. The `S` parameter, whose value is either `ahead` or `behind`, indicates the relative orientation of the robot to `C`.

The program comprises the five parts mentioned in Section 2. To begin with, let's look at the formulae describing high-level, compound actions (parts C and D, according to Section 2). Let's consider the high-level action `go_to_room(R1,R2)`. The effect of this action is given by an `initiates` formula.

```
initiates(go_to_room(R1,R2),in(R2),T) :-                  (A1)
  holds_at(in(R1),T).
```

In other words, `go_to_room(R1,R2)` puts the robot in `R2`, assuming it was in `R1`. The `go_to_room` action is recursively defined in terms of `go_through` actions.

```
happens(go_to_room(R,R),T,T).                             (A2)
happens(go_to_room(R1,R3),T1,T4) :-                       (A3)
  towards(R2,R3,R1), connects(D,R1,R2),
  holds_at(door_open(D),T1),
  happens(go_through(D),T1,T2),
  happens(go_to_room(R2,R3),T3,T4),
  before(T2,T3),
  not(clipped(T2,in(R2),T3)).
```

In other words, `go_to_room(R1,R3)` has no sub-actions if `R1` = `R3`, but otherwise comprises a `go_through` action to take the robot through door `D` into room `R2` followed by another `go_to_room` action to take the robot from `R2` to `R3`. Door `D` must be open. The `towards` predicate supplies heuristic guidance for the selection of the door to go through.

Notice that the action is only guaranteed to have the effect described by the `initiates` formula if the room the robot is in doesn't change between the two sub-actions. Hence the need for the negated `clipped` conjunct. The inclusion of such negated `clipped` conjuncts ensures that the sub-actions of overlapping compound actions cannot interfere with each other.

The `go_through` action itself decomposes further into `follow_wall`, `go_straight` and `turn` actions that the robot can execute directly (see Appendix A).

Now let's consider the formulae describing the effects of these primitive executable actions (part A of the program, according to Section 2). The full set of these formulae is to be found in Appendix C. Here are the formulae describing the `follow_wall` action.

```
initiates(follow_wall,                                    (S1)
    loc(corner(C2),ahead),T) :-
  holds_at(loc(corner(C1),behind),T),
  next_visible_corner(C1,C2,left,T).
terminates(follow_wall,                                   (S2)
  loc(corner(C),behind),T).
```

A follow_wall action takes the robot to the next visible corner in the room, where the next visible corner is the next one that is not part of a doorway whose door is closed. The effects of go_straight and turn are similarly described. The formulae in Appendix C also cover the fluents facing and pos which are used for map building but not for navigation.

Next we'll take a look at the formulae describing the causes of sensor events, which figure prominently in sensor data assimilation (part B of the program, according to Section 2). Three kinds of sensor event can occur: left_and_front, left_gap and left.

The left_and_front event occurs when the robot's left sensors are already high and its front sensors go high, such as when it's following a wall and meets a concave corner. The left_gap event occurs when its left sensors go low, such as when it is following a corner and meets a convex corner such as a doorway. The left event occurs when its front and left sensors are high and the front sensors go low, such as when it turns right in a concave corner.

In the formulae of Appendix C, each of these sensor events has a single parameter, which indicates the distance the robot thinks it has travelled since the last sensor event, according to its on-board odometry. This parameter is used for map building and can be ignored for the present. Here's the formula for left_and_front.

```
happens(left_and_front(X),T,T) :-                                   (S3)
  happens(follow_wall,T,T),
  holds_at(co_ords(P1),T),
  holds_at(facing(W),T),
  holds_at(loc(corner(C1),behind),T),
  next_visible_corner(C1,C2,left,T),
  inner(C2),
  displace(P1,X,W,P2), pos(C2,P2).
```

The second, third, and final conjuncts on the right-hand-side of this formula are again the concern of map building, so we can ignore them for now. The rest of the formula says that a left_and_front event will occur if the robot starts off in corner C1, then follows the wall to a concave corner C2. Similar formulae characterise the occurrence of left and left_gap events (see Appendix C).

4 A Worked Example of Navigation

These formulae, along with their companions in Appendices A and C, are employed by the sense-plan-act cycle in the way described in Section 2. To see this, let's consider an example. The system starts off with an empty plan, and is presented with the initial goal to get to room r6.

```
holds_at(in(r6),T)
```

The planning process resolves this goal against clause (A1), yielding a complete, but not fully decomposed plan, comprising a single go_to_room(r3,r6) action. Resolving against clause (A3), this plan is decomposed into a go_through(d4) action followed by a go_to_room(r4,r6) action. Further decomposition of the `go_through` action yields the plan: `follow_wall`, `go_through(d4)`, then `go_to_room(r4,r6)`. In addition, a number of protected links (negated `clipped` formulae) are recorded for later re-checking, including a formula of the form,

`not(clipped(τ1,door_open(d4),τ2))`.

The system now possesses a complete, though still not fully decomposed, plan, with an executable first action, namely `follow_wall`. So it proceeds to execute the `follow_wall` action, while continuing to work on the plan. When the `follow_wall` action finishes, a `left_and_front` sensor event occurs, and the SDA process is brought to life. In this case, the sensor event has an empty explanation — it is just what would be expected to occur given the robot's actions.

Similar processing brings about the subsequent execution of a `turn(right)` action then another `follow_wall` action. At the end of this second `follow_wall` action, a `left_and_front` sensor event occurs. This means that a formula of the form,

`happens(left_and_front(δ),τ)`

needs to be explained, where τ is the time of execution of the `follow_wall` action. The SDA process sets about explaining the event in the usual way, which is to resolve this formula against clause (S3). This time, though, an empty explanation will not suffice. Since door d4 was initially open, a `left_gap` event should have occurred instead of a `left_and_front` event.

After a certain amount of work, this particular explanation task boils down to the search for an explanation of the formula,

`next_visible_corner(c2,C,τ), inner(C)`

(The `C` is implicitly existentially quantified.) The explanation found by the SDA process has the following form.

`happens(close_door(d4),τ'), before(τ',τ)`

In other words, an external `close_door` action occurred some time before the robot's `follow_wall` action. Since this `close_door` action terminates the fluent `door_open(d4)`, there is a violation of one of the protected links recorded by the planner (see above). The violation of this protected link causes the system to replan, this time producing a plan to go via doors `d2` and `d3,` which executes successfully.

5 Map Building with Epistemic Fluents

The focus of the rest of this paper is map building. Map building is a more sophisticated task than navigation, and throws up a number of interesting issues, including how to represent and reason with knowledge producing actions and actions with knowledge preconditions, the subject of this section.

During navigation, explanations of sensor data are constructed in terms of open door and close door events, but for map building we require explanations in terms of the relationships between corners and the connectivity of rooms. So the first step in turning our navigation program into a map building program is to declare a different set of abducibles (part E of a robot program, according to Section 2). The abducibles will now include the predicates `next_corner`, `inner`, `door`, and `connects`. Map building then becomes a side effect of the SDA process.

But how are the effects of the robot's actions on its knowledge of these predicates to be represented and reasoned with? The relationship between knowledge and action has received a fair amount of attention in the reasoning about action literature ([Levesque, 1996] is a recent example). All of this work investigates the relationship between knowledge and action on the assumption that knowledge has a privileged role to play in the logic.

In the present paper, the logical difficulties consequent on embarking on such an investigation are to some degree sidestepped by according epistemic fluents, that is to say fluents that concern the state of the robot's knowledge, exactly the same status as other fluents. What follows is in no way intended as a contribution to the literature on the subject of reasoning about knowledge. But it's enough to get us off the ground with logic-based map building.

Before discussing implementation, let's take a closer look at this issue from a logical point of view. To begin with, we'll introduce a generic epistemic fluent Knows. The formula HoldsAt(Knows(ϕ),τ) represents that the formula ϕ follows from the robot's knowledge (or, strictly speaking, from the robot's beliefs) at time τ. (More precisely, to distinguish object- from meta-level, the formula *named by* ϕ follows from the robot's knowledge. To simplify matters, we'll assume every formula is its own name.)

Using epistemic fluents, we can formalise the knowledge producing effects of the robot's repertoire of actions. In the present domain, for example, we have the following.

$$\begin{aligned}&\exists\, \mathrm{r,c2}\ [\mathrm{Initiates(FollowWall,}\\ &\qquad \mathrm{Knows(NextCorner(r,c1,c2)),t)}] \leftarrow\\ &\quad \mathrm{HoldsAt(Loc(Corner(c1),Behind),t)}\end{aligned}$$

In other words, following a wall gives the robot knowledge of the next corner along. This formula is true, given the right set of abducibles, thanks to the abductive treatment of sensor data via clause (S3). In practise, the abductive SDA process gives a new name to that corner, if it's one it hasn't visited before, and records whether or not it's an inner corner.

Similar formulae account for the epistemic effects of the robot's other actions. Then, all we need is to describe the initial state of the robot's knowledge, using the $\mathrm{Initially_N}$ and $\mathrm{Initially_P}$ predicates, and the axioms of the event calculus will take care of the rest, yielding the state of the robot's knowledge at any time.

Epistemic fluents, as well as featuring in the descriptions of the knowledge producing effects of actions, also appear in knowledge goals. In the present example, the overall goal is to know the layout of corners, doors and rooms. Accordingly, a new epistemic fluent KnowsMap is defined as follows.

$$
\begin{array}{l}
\mathrm{HoldsAt(KnowsMap,t)} \leftarrow \\
\quad [\mathrm{Door(d,c1,c2)} \rightarrow \\
\qquad \exists\ \mathrm{r2}\ [\mathrm{HoldsAt(Knows(Connects(d,r1,r2)),t)}]] \wedge \\
\quad [\mathrm{Pos(c1,p)} \rightarrow \\
\qquad \exists\ \mathrm{c2}\ [\mathrm{HoldsAt(Knows(NextCorner(r,c1,c2)),t)}]]
\end{array}
$$

Note the difference between

$\exists$ r2 [HoldsAt(Knows(Connects(d,r1,r2)),t)]]

and

$\exists$ r2 [Connects(d,r1,r2)].

The second formula says that there is a room through door d, while the first formula says that the robot knows what that room is. The robot's knowledge might include the second formula while not including the first. Indeed, if badly programmed, the robot's knowledge could include the first formula while not including the second. (There's no analogue to modal logic's axiom schema T (reflexivity).)

The top-level goal presented to the system will be HoldsAt(KnowsMap,t). Now suppose we have a high-level action Explore, whose effect is to make KnowsMap hold.

Initiates(Explore,KnowsMap,t)

Given the top-level goal HoldsAt(KnowsMap,t), the initial top-level plan the system will come up with comprises the single action Explore. The definition of Explore is, in effect, the description of a map building algorithm. Here's an example formula.

$$
\begin{array}{l}
\mathrm{Happens(Explore,t1,t4)} \leftarrow \\
\quad \mathrm{HoldsAt(In(r1),t)} \wedge \mathrm{HoldsAt(Loc(Corner(c1),s),t1)} \wedge \\
\quad \exists\ \mathrm{c2}\ [\mathrm{HoldsAt(Knows(NextCorner(r1,c1,c2)),t1)}] \wedge \\
\quad \exists\ \mathrm{d, c2, c3}\ [\mathrm{Door(d,c2,c3)} \wedge \\
\qquad \neg \exists\ \mathrm{r2}\ [\mathrm{HoldsAt(Knows(Connects(d,r1,r2)),t1)}]] \wedge \\
\quad \mathrm{Happens(GoThrough(d),t1,t2)} \wedge \\
\quad \mathrm{Happens(Explore,t3,t4)} \wedge \mathrm{Before(t2,t3)}
\end{array}
\tag{L1}
$$

This formula tells the robot to proceed through door d if it's in a corner it already knows about, where d is a door leading to an unknown room. Note the use of epistemic fluents in the third and fourth lines. A number of similar formulae cater for the decomposition of Explore under different circumstances.

6 A Map Building Program

Appendices B and C of the full paper present (almost) the full text of a working event calculus program for map building. This section outlines how it works. The three novel issues that set this program apart from the navigation program already discussed are,

1. the use of epistemic fluents,

2. the need for integrity constraints, and
3. the need for techniques similar to those used in constraint logic programming (CLP).

The first issue was addressed in the previous section. The second two issues, as we'll see shortly, arise from the robot's need to recognise when it's in a corner it has already visited. First, though, let's see how the predicate calculus definition of the Explore action translates into a clause in the actual implementation. Here's the implemented version of formula (L1) at the end of the previous section.

```
happens(explore,T1,T4) :-                                    (B1)
  holds_at(loc(corner(C1),S),T1),
  not(unexplored_corner(C1,T1)),
  unexplored_door(D,T1),
  happens(go_through(D),T1,T2),
  happens(explore,T3,T4), before(T2,T3).
```

Instead of using epistemic fluents explicitly, this clause appeals to two new predicates `unexplored_corner` and `unexplored_door`. These are defined as follows.

```
unexplored_corner(C1,T) :-                                   (B2)
  pos(C1,P), not(next_corner(R,C1,C2)).
unexplored_door(D,T) :-                                      (B3)
  door(D,C1,C2), not(connects(D,R1,R2)).
```

The formula `pos(C,P)` represents that corner `C` is in position `P`, where `P` is a co-ordinate range (see below).

By defining these two predicates, we can simulate the effect of the existential quantifiers in formula (L1) using negation-as-failure. Furthermore, we can use negation-as-failure as a substitute for keeping track of the Knows fluent. (This trick renders the predicates' temporal arguments superfluous, but they're retained for elegance.) Operationally, the formula,

```
not(next_corner(R,C1,C2))
```

serves the same purpose as the predicate calculus formula,

$\neg \exists$ r2 [HoldsAt(Knows(Connects(d,r1,r2)),t1)]].

The first formula uses negation-as-failure to determine what is provable from the robot's knowledge, while the second formula assumes that what is provable is recorded explicitly through the Knows fluent.

The final issue to discuss is how, during its exploration of a room, the robot recognises that it's back in a corner it has already visited, so as to prevent the SDA process from postulating redundant new corners.

Recall that each sensor event has a single argument, which is the estimated distance the robot has travelled since the last sensor event. Using this argument, the robot can keep track of its approximate position. Accordingly, the program includes a suitable set of `initiates` and `terminates` clauses for the `co_ords` fluent, where `co_ords(P)` denotes that `P` is the robot's current position. A *position* is actually a list `[X1,X2,Y1,Y2]`, representing a rectangle, bounded by `X1` and `X2` on the x-axis and `Y1` and `Y2` on the y-axis, within which an object's precise co-ordinates are known to fall.

Using this fluent, explanations of sensor data that postulate redundant new corners can be ruled out using an *integrity constraint*. (In abductive logic programming, the use of integrity constraints to eliminate possible explanations is a standard technique.) Logically speaking, an integrity constraint is a formula of the form,

$\neg [P_1 \wedge P_2 \wedge . . . \wedge P_n]$

where each P_i is an atomic formula. Any abductive explanation must be consistent with this formula. In meta-interpreter syntax, the predicate `inconsistent` is used to represent integrity constraints, and the abductive procedure needs to be modified to take them into account. In the present case, we need the following integrity constraint.

```
inconsistent([pos(C1,P1), pos(C2,P2),                      (B4)
  same_pos(P1,P2),
  room_of(C1,R), room_of(C2,R),
  diff(C1,C2)]).
```

The formula `same_pos(P1,P2)` checks whether the maximum possible distance between `P1` and `P2` is less than a predefined threshold. The formula `diff(X,Y)` represents that `X` $\neq$ `Y`. If the meta-interpreter is trying to prove `not(diff(X,Y))`, it can do so by renaming `X` to `Y`. (Terms that can be renamed in this way have to be declared.) In particular, to preserve consistency in the presence of this integrity constraint, the SDA process will sometimes equate a new corner with an old one, and rename it accordingly.

Having determined, via (B4), that two apparently distinct corners are in fact one and the same, the robot may have two overlapping positions for the same corner. These can be subsumed by a single, more narrowly constrained position combining the range bounds of the two older positions.

This motivates the addition of the final component of the system, namely a rudimentary constraint reduction mechanism along the lines of those found in constraint logic programming languages. This permits the programmer to define simple constraint reduction rules whereby two formulae are replaced by a single formula that implies them both. In the present example, we have the following rule.

```
common_antecedent(pos(C,[X1,X2,Y1,Y2]),
    pos(C,[X3,X4,Y3,Y4]),
    pos(C,[X5,X6,Y5,Y6) :-
  max(X1,X3,X5), min(X2,X4,X6),
  max(Y1,Y3,Y5), min(Y2,Y4,Y6).
```

The formula `common_antecedent(P1,P2,P3)` represents that `P3` implies both `P1` and `P2`, and that any explanation containing both `P1` and `P2` can be simplified by replacing `P1` and `P2` by `P3`.

Concluding Remarks

The work reported in this paper and its companion [Shanahan, 2000b] is part of an ongoing attempt to develop robot architectures in which logic is the medium of representation, and theorem proving is the means of computation. The hope is that such architectures, having a deliberative component at their core, are a step closer to robots that can reason, act, and communicate in a way that mimics more closely human high-level cognition [Shanahan, 1999]. The preliminary results reported here are promising. In particular, it's encouraging to see that event calculus programs for navigation and map building can be written and deployed on real robots that are each less than 100 lines long and, moreover, that share more than half their code.

However, the work presented has many shortcomings that need to be addressed in future work. Most important is the issue of scaling up. The robot environment used in the experiments described here is simple, highly engineered (compared to a real office) and static, and the robots themselves have extremely simple sensors. It remains to be seen, for example, what new representational techniques will be required to accommodate the proposed theory of sensor data assimilation to complex, dynamic environments and rich sensors, such as vision.

In addition, a number of logical issues remain outstanding. First, as reported in [Shanahan, 2000a], the formal properties of the event calculus with compound actions are poorly understood. Second, the topic of knowledge producing actions in the event calculus needs to be properly addressed, and the limitations of the naive approach employed here need to be assessed.

Acknowledgements

This work was carried out as part of EPSRC project GR/N13104 "Cognitive Robotics II".

References

[Baral & Tran, 1998] C.Baral and S.C.Tran, Relating Theories of Actions and Reactive Control, *Linköping Electronic Articles in Computer and Information Science*, vol. 3 (1998), no. 9.

[De Giacomo, *et al*., 1997] G. De Giacomo, Y.Lespérance, H.Levesque, Reasoning about Concurrent Execution, Prioritized Interrupts, and Exogenous Actions in the Situation Calculus, *Proceedings 1997 International Joint Conference on Artificial Intelligence (IJCAI 97)*, Morgan Kaufmann, pp. 1221–1226.

[Kowalski, 1995] R.A.Kowalski, Using Meta-Logic to Reconcile Reactive with Rational Agents, in *Meta-Logics and Logic Programming*, ed. K.R.Apt and F.Turini, MIT Press (1995), pp. 227–242.

[Lespérance, *et al.*, 1994] Y.Lespérance, H.J.Levesque, F.Lin, D.Marcu, R.Reiter, and R.B.Scherl, A Logical Approach to High-Level Robot Programming: A Progress Report, in *Control of the Physical World by Intelligent Systems: Papers from the 1994 AAAI Fall Symposium*, ed. B.Kuipers, New Orleans (1994), pp. 79–85.

[Levesque, 1996] H.Levesque, What Is Planning in the Presence of Sensing? *Proceedings AAAI 96*, pp. 1139–1146.

[Nilsson, 1984] N.J.Nilsson, ed., *Shakey the Robot*, SRI Technical Note no. 323 (1984), SRI, Menlo Park, California.

[Shanahan, 1997a] M.P.Shanahan, *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press (1997).

[Shanahan, 1997b] M.P.Shanahan, Noise, Non-Determinism and Spatial Uncertainty, *Proceedings AAAI 97*, pp. 153–158.

[Shanahan, 1999] M.P.Shanahan, What Sort of Computation Mediates Best Between Perception and Action?, in *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, eds. H.J.Levesque & F.Pirri, Springer-Verlag (1999), pages 352-368.

[Shanahan, 2000a] M.P.Shanahan, An Abductive Event Calculus Planner, *The Journal of Logic Programming*, vol. 44 (2000), pp. 207–239.

[Shanahan, 2000b] M.P.Shanahan, Reinventing Shakey, in *Logic Based Artificial Intelligence*, ed. J.Minker, Kluwer, to appear.

Appendices

This paper has three appendices, which are available electronically from,

`http://www.ee.ic.ac.uk/~mpsha/pubs.html`

Look under the “Robotics” heading.