Robot Navigation and Map Building with the Event Calculus

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.

Introduction

In the late Sixties, when the Shakey project started [Nilsson, 1984], the vision of robot design based on logical representation seemed both attractive and attainable. Through the Seventies and early Eighties, however, the desire to build working robots led researchers away from logic to more practical approaches to representation. This movement away from logical representation reached its apogee in the late Eighties and early Nineties when Brooks jettisoned the whole idea of representation, along with the so-called sense-model-plan-act architecture epitomised by Shakey [Brooks, 1991].

However, the Shakey style of architecture, having an overtly logic-based deliberative component, seems to offer researchers a direct path to robots with high-level cognitive skills, such as planning, communication, and reasoning about other agents. Accordingly, a number of researchers have instigated a Shakey revival. Armed with modern solutions to the frame problem, work in so-called *cognitive robotics* aims to build robots with high-level cognitive skills using logic as a representational medium [Lespérance, *et al.*, 1994].

This paper presents an implemented logic-based, high-level robot control system. 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. Both these programs have been deployed and tested on actual robots.

The paper is organised as follows. After an informal presentation of the behaviour of these application programs, the theoretical underpinnings of the controller are outlined — the event calculus is described, and abductive accounts of planning and perception are sketched. The implementation of the sense-plan-act cycle is then discussed, and finally, the two application programs — navigation and map building — are presented.

1 What the Robot Can Do

The robotic platform for the experiments reported in this paper is a Khepera, a miniature robot with two drive wheels and a suite of eight infra-red proximity sensors around its circumference. The robot inhabits a miniaturised office-like environment, depicted in Figure 1.

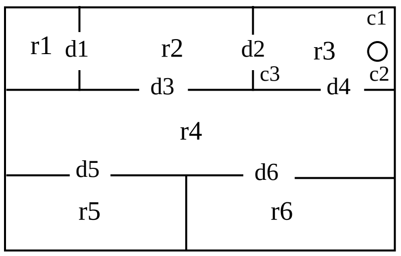

Figure 1: The Robot's Environment

The robot has a simple repertoire of low-level actions, executed on-board, which includes wall following, turning into doorways, and turning around corners. Using these, the high-level, off-board controller manoeuvres the robot around its environment.

Let's take a look at a navigation example to see how the high-level controller functions. Suppose the robot is initially between corners c1 and c2, as shown, and suppose it has the goal of retrieving a package from room r6. Informally, this is how the robot's high-level controller achieves the goal.

First, the robot plans a route. As soon as it finds a complete (though perhaps not fully decomposed) plan with an executable first action, the robot starts carrying out that plan. In this case, the first action is to go through door d4, so the robot sets out along the wall until it reaches corner c2. It then turns the corner, and heads off towards door d4.

Suppose someone now closes door d4. Unfortunately, because of its poor sensors, the robot cannot detect closed doors, which are indistinguishable from walls. So the robot continues wall following until it reaches corner c3.

Up to this point, the assimilation of the robot's sensor data has been a trivial matter. The sensor events it receives are exactly what it would expect given what it has done and what it believes about its environment. So the explanations of those sensor events are trivial. But this encounter with a corner requires a non-empty explanation.

Using abduction, the robot constructs an explanation of its encounter with the corner — door d4 must have been closed, and it must now be at corner c3. But this new piece of information conflicts with the assumptions underlying the plan it's executing. So the robot is forced to replan. It now finds a new route to room r6, via doors d2, d3, and d6, which it successfully executes and retrieves the package.

Now let's consider a map building example.

Where the navigation task takes for granted the prior availability of a map of the robot's environment, the map building task starts with a complete *tabula rasa*. Nothing is initially known of the layout of rooms, doors and corners, and the robot's job is to explore its environment, building a map as it goes along. This is done by abductively explaining the robot's sensor data, exactly as in navigation, the chief difference being that explanations are now bits of map rather than door closing events. (We assume all the doors stay open during map building.)

Suppose it starts off in corner c1. As the robot doesn't know what the next corner from c1 is, it starts wall following. After travelling a certain distance, its front sensors go high, an event which is explained by postulating an inner (concave) corner which is the next one along from c1. So the robot names the corner and records its relationship to and distance from c1.

In a similar fashion, the robot discovers the near and far corners of door d4, postulates a door, and goes on its way. It continues like this until it has been right around the room and arrives at a corner it already knows about. (The robot maintains a rough idea of its co-ordinates, and knows there is a minimum distance between two distinct corners. Therefore it can tell when it is back where it started.) Then, since there is a door in the room leading to somewhere unexplored, the robot navigates to that door, goes through it, and repeats the process for the next room. Eventually all rooms are explored, and a complete map is built.

2 Event Calculus Basics

The formalism used throughout this paper is based on the circumscriptive event calculus [Shanahan, 1997a]. Because the event calculus is presented in considerable detail elsewhere, the description here will be kept fairly brief.

A many sorted language is assumed, with variables for *fluents*, *actions* (or *events*), and *time points*. We have the following axioms, whose conjunction will be denoted EC. Their main purpose is to constrain the predicate HoldsAt. HoldsAt(β,τ) represents that fluent β holds at time τ. Throughout the paper, all variables are universally quantified with maximum scope, unless otherwise indicated.

$$\text{HoldsAt}(f,t) \leftarrow \text{Initially}_P(f) \wedge \neg\, \text{Clipped}(0,f,t) \qquad \text{(EC1)}$$

$$\begin{aligned}&\text{HoldsAt}(f,t3) \leftarrow \qquad \text{(EC2)}\\ &\quad \text{Happens}(a,t1,t2) \wedge \text{Initiates}(a,f,t1) \wedge\\ &\qquad t2 < t3 \wedge \neg\, \text{Clipped}(t1,f,t3)\end{aligned}$$

$$\begin{aligned}&\text{Clipped}(t1,f,t4) \leftrightarrow \qquad \text{(EC3)}\\ &\quad \exists\, a,t2,t3\, [\text{Happens}(a,t2,t3) \wedge t1 < t3 \wedge t2 < t4 \wedge\\ &\qquad [\text{Terminates}(a,f,t2) \vee \text{Releases}(a,f,t2)]]\end{aligned}$$

$$\begin{aligned}&\neg\, \text{HoldsAt}(f,t) \leftarrow \qquad \text{(EC4)}\\ &\quad \text{Initially}_N(f) \wedge \neg\, \text{Declipped}(0,f,t)\end{aligned}$$

$$\begin{aligned}&\neg\, \text{HoldsAt}(f,t3) \leftarrow \qquad \text{(EC5)}\\ &\quad \text{Happens}(a,t1,t2) \wedge \text{Terminates}(a,f,t1) \wedge\\ &\qquad t2 < t3 \wedge \neg\, \text{Declipped}(t1,f,t3)\end{aligned}$$

$$\begin{aligned}&\text{Declipped}(t1,f,t4) \leftrightarrow \qquad \text{(EC6)}\\ &\quad \exists\, a,t2,t3\, [\text{Happens}(a,t2,t3) \wedge t1 < t3 \wedge t2 < t4 \wedge\\ &\qquad [\text{Initiates}(a,f,t2) \vee \text{Releases}(a,f,t2)]]\end{aligned}$$

$$\text{Happens}(a,t1,t2) \rightarrow t1 \leq t2 \qquad \text{(EC7)}$$

A particular *domain* is described in terms of Initiates, Terminates, and Releases 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. Releases(α,β,τ) represents that fluent β is no longer subject to the common sense law of inertia 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(α,τ1,τ2) represents that action or event α occurs, starting at time τ1 and ending at time τ2.

A two-argument version of Happens is defined as follows.

$$\text{Happens}(a,t) \equiv_{def} \text{Happens}(a,t,t)$$

Formulae describing triggered events are allowed, and will generally have the form,

$$\text{Happens}(\alpha,\tau) \leftarrow \Pi$$

where Π can be any formula. As we'll see, similar formulae, in "Happens if Happens" form, can be used to define high-level, compound actions in terms of more primitive ones.

The frame problem is overcome through circumscription. Given a conjunction Σ of Initiates, Terminates, and Releases formulae describing the effects of actions, a conjunction Δ of Initially, Happens and temporal ordering formulae describing a narrative of actions and events, and a conjunction Ω of uniqueness-of-names axioms for actions and fluents, we're interested in,

$$\begin{aligned}&\text{CIRC}[\Sigma\, ;\, \text{Initiates, Terminates, Releases}] \wedge\\ &\quad \text{CIRC}[\Delta\, ;\, \text{Happens}] \wedge \text{EC} \wedge \Omega.\end{aligned}$$

By minimising Initiates, Terminates and Releases we assume that actions have no unexpected effects, and by minimising Happens we assume that there are no unexpected event occurrences. In most of the cases we're interested in, Σ and Δ will be conjunctions of Horn clauses, and the circumscriptions will reduce to predicate completions.

Care must be taken when domain constraints and triggered events are included. The former must be conjoined to EC, while the latter are conjoined to Δ.

3 Planning and Perception as Abduction

Logically speaking, both planning and sensor data assimilation can be considered as abduction, the reverse of deduction, which in the present context means reasoning from effects to causes. Let's consider planning first.

Planning can be thought of as the inverse operation to temporal projection, and temporal projection in the event calculus is naturally cast as a deductive task in the following way. Given Σ, Ω and Δ as in Section 2, we're interested in HoldsAt formulae Γ such that,

$$\text{CIRC}[\Sigma \,;\, \text{Initiates, Terminates, Releases}] \wedge \text{CIRC}[\Delta \,;\, \text{Happens}] \wedge \text{EC} \wedge \Omega \vDash \Gamma.$$

Conversely, planning in the event calculus can be considered as an abductive task. Given a domain description Σ, a conjunction Γ of goals (HoldsAt formulae), and a conjunction Δ_N of Initially_P and Initially_N formulae describing the initial situation, a *plan* is a consistent conjunction Δ_P of Happens and temporal ordering formulae such that,

$$\text{CIRC}[\Sigma \,;\, \text{Initiates, Terminates, Releases}] \wedge \text{CIRC}[\Delta_N \wedge \Delta_P \,;\, \text{Happens}] \wedge \text{EC} \wedge \Omega \vDash \Gamma.$$

In order to interleave planning, sensing and acting in a respectable way, we need to carry out *hierarchical* planning. The logical story for hierarchical planning is the same, except that we add a conjunction of "Happens if Happens" formulae to $\Delta_N \wedge \Delta_P$.

Now let's take a look at the topic of perception (sensor data assimilation). An abductive logical account of sensor data assimilation (SDA) can be constructed which mirrors the above account of planning. 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 be inferred. Perception involves finding explanations of raw sensor data, hence the need for abduction.

The methodology for supplying the required logical account is as follows [Shanahan, 1997b]. First, using a suitable formalism for reasoning about actions, 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, sensor data assimilation can be considered as abduction with this theory. Roughly speaking (omitting details of the circumscriptions), given a narrative Δ of the robot's actions, and a description Γ of the robot's sensor data, the robot needs to find some Ψ such that,

$$\Sigma \wedge \Delta \wedge \Psi \vDash \Gamma.$$

Γ might comprise Happens and/or HoldsAt formulae describing sensor events or values, and Ψ might comprise Initially_N and Initially_P formulae describing the environment's initial configuration and/or Happens formulae describing the intervening actions of other agents who have modified that configuration.

There is, of course, no guarantee that a *unique* Ψ exists to explain any given collection of sensor data. So some strategy needs to be adopted for dealing with multiple explanations. This can involve imposing a preference ordering on explanations, perhaps resulting in the adoption of the "simplest" explanation, in some sense. The whole issue of multiple explanations merits further study.

Since most of the formulae we're using are in extended Horn clause form, these accounts of planning and perception can be implemented through abductive logic programming, as described in [Shanahan, 1999]. The present implementation is a meta-interpreter with built-in facilities for handling the axioms of the event calculus. The same meta-interpreter is used for both planning and sensor data assimilation.

The computation carried out by this system strongly resembles that of a hand-coded partial-order planning algorithm, such as UCPOP [Penberthy & Weld, 1992], as shown in [Shanahan, 1999]. In particular, the implementation has to record the negated Clipped and Declipped formulae it has proved, and these are treated in much the same way as *protected links* in a partial-order planner. They also play a vital role in deciding when to replan.

Throughout the sequel, when discussing the actual implementation rather than the logic, the event calculus syntax employed by the meta-interpreter will be used instead of predicate calculus syntax. This is much like standard Prolog syntax, with predicate and function symbols starting with lower case letters, and variables starting with upper case letters. For example, the predicate calculus formula,

$$\text{Initiates}(\text{Move}(x,y),\text{On}(x,y),t) \leftarrow \text{HoldsAt}(\text{Free}(x),t) \wedge \text{HoldsAt}(\text{Free}(y),t)$$

in meta-interpreter syntax becomes,

```
initiates(move(X,Y),on(X,Y),T) :-
  holds_at(free(X),T), holds_at(free(Y),t).
```

This serves to emphasise the distinction between specification and implementation. Indeed, the relationship between pure event calculus theories and the event calculus programs they correspond to is one that needs to be carefully policed. Ideally, we would like, not only a trivial translation from one to the other, but also an implementation that is both sound and complete with respect to the logic.

In [Shanahan, 1999], the abductive meta-interpreter is proven to be sound and complete for a certain class of event calculus theories. However, this class does not encompass all the examples looked at in this paper, so further work is required on this issue.

4 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 sets of event calculus formulae. These processes conform to the logical specifications outlined in the previous section.
- Planning and SDA are both *resource-bounded* processes. They are subject to constant suspension to permit the interleaving of sensing, planning and acting.
- To encourage 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 current plan, thus precipitating *replanning*.

An event calculus robot program comprises the following five parts.

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

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

C. A set of Initiates, Terminates and Releases 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.

E. A set of declarations, specifying, for example, what formulae 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 or Declipped 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 and Declipped formulae, which are subject to further abductive processing through resolution against formulae in A, taking into account the actions the robot itself has performed.

Ultimately, the SDA process generates a set of abduced Happens formulae describing external actions (actions not carried out by the robot itself) which 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 and Declipped formulae (protected links) recorded by the planning process. If they do, the system replans from scratch.

In the context of the 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, given by the logic of Sections 2 and 3, and a *procedural* meaning, given by the execution model outlined in this section and at the end of Section 3. The following sections present two robotic applications written as event calculus programs, namely navigation and map building.

5 A Navigation Program

Appendices A and C contain (almost) the full text of a working event calculus program for robot navigation. This section describes the program's construction and operation.

The robot's environment 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 in Figure 1, say, is a required background theory for the navigation application, but is not given in the appendices.

The robot can execute a repertoire of three primitive actions: `follow_wall`, in which case it proceeds along the wall to the next visible corner, `turn(S)`, in which case the robot turns a corner in direction `S` (either left or right), and `go_straight`, in which case the robot crosses a doorway. For simplicity, the current 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`. The term `loc(C,S)` denotes that the robot is in corner `C`. The `S` parameter of the `loc` fluent, whose value is either `ahead` or `behind`, indicates the relative orientation of the robot to the corner `C`.

The program comprises the five parts mentioned in Section 4. To begin with, let's look at the formulae describing high-level, compound actions (parts C and D, according to Section 4). 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 `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 4). 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. It modifies the `loc` fluent, where `loc(corner(C),S)` denotes that the robot is in the corner `C`. `S` is either `ahead` or `behind`, indicating respectively that the robot is facing into the corner and away from it.

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 4). 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, as when it's following a wall and meets a concave corner. The `left_gap` event occurs when its left sensors go low, 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, 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. 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).

6 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 4. To see this, let's return to the navigation example of Section 1. 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` and `declipped` 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.

7 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.

To see why this issue is important, consider the fact that, as a result of building a map of its environment, there may be no physical modification to that environment at all. The robot may even be back in the same location it started in. The only change that has taken place is in the robot's knowledge. The only effects of the robot's actions relevant to map building are their knowledge producing effects. Similarly, the robot's overall goal is a knowledge goal, not a physical one.

The operation of the sense-plan-act cycle, in particular the SDA process, entails that actions do indeed have knowledge producing effects. For instance, in the navigation example, following a wall can result in the robot knowing whether or not a door is open.

In the navigation example, 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 4). 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.

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 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.

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

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 Initially_N and 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, an epistemic fluent KnowsMap is defined as follows.

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

Note the difference between

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

and

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

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 is no analogue to the axiom schema T (reflexivity) in modal logic.)

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.

$$\text{Initiates(Explore,KnowsMap,t)}$$

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 program. Here's an example formula.

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

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.

8 A Map Building Program

Appendices B and C 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,

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

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,

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

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 aim of the ongoing work reported here is to design and build theoretically well-founded, general purpose systems for high-level robot control, in which each computational step is also a step of logical inference, and each computational state has declarative meaning. Needless to say, the ideas presented merit a good deal of further study, and it remains to be seen whether they will scale up to robots with richer sensors in more realistic environments.

Preliminary results are promising, though. In particular, it's encouraging to see that event calculus programs for navigation and map building can be written that are each less than 100 lines long and, moreover, that share more than half their code.

References

[Brooks, 1991] R.A.Brooks, Intelligence Without Reason, *Proceedings IJCAI 91*, pages 569-595.

[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.

[Penberthy & Weld, 1992] J.S.Penberthy and D.S.Weld, UCPOP: A Sound, Complete, Partial Order Planner for ADL, *Proceedings KR 92*, pp. 103–114.

[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, An Abductive Event Calculus Planner, *Journal of Logic Programming*, to appear (provisionally accepted).

Appendix A: Navigation Code

```
/* Navigation Compound Actions */

happens(go_to_room(R,R),T,T).

happens(go_to_room(R1,R3),T1,T4) :-
     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)).

happens(go_to_room(R1,R3),T1,T4) :-
     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)).
```

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

happens(go_through(D),T1,T2) :-
     holds_at(loc(corner(C1),ahead),T1),
     door(D,C1,C2),
     happens(turn(left),T1),
     happens(turn(left),T2),
     before(T1,T2),
     not(clipped(T1,door_open(D),T2)).

happens(go_through(D1),T1,T3) :-
     holds_at(loc(corner(C1,ahead)),T1),
     door(D2,C1,C2), diff(D1,D2),
     holds_at(door_open(D2),T1),
     happens(go_straight,T1),
     happens(go_through(D1),T2,T3),
     before(T1,T2).

happens(go_through(D),T1,T3) :-
     holds_at(loc(corner(C),behind),T1),
     happens(follow_wall,T1),
     happens(go_through(D),T2,T3),
     before(T1,T2),
     not(clipped(T1,door_open(D),T2)).

happens(go_through(D),T1,T3) :-
     holds_at(loc(corner(C),ahead),T1),
     inner(C),
     happens(turn(right),T1),
     happens(go_through(D),T2,T3),
     before(T1,T2),
     not(clipped(T1,door_open(D),T2)).

/* Navigation Heuristics */

towards(R1,R1,R2).

towards(R1,R2,R3) :- connects(D,R1,R2).

towards(R1,R2,R3) :-
     connects(D1,R1,R4), connects(D2,R4,R2).

/* External Actions */

terminates(close_door(D),door_open(D),T).

initiates(open_door(D),door_open(D),T).
```

Appendix B: Map Building Code

```
/* Map Building Compound Actions */

happens(explore,T1,T6) :-
     holds_at(loc(corner(C1),ahead),T1),
     inner(C1),
     unexplored_corner(C1,T1),
     happens(turn(right),T1,T2),
     happens(follow_wall,T3,T4), before(T2,T3),
     happens(explore,T5,T6), before(T4,T5).

happens(explore,T1,T4) :-
     holds_at(loc(corner(C1),ahead),T1),
     not(inner(C1)),
     unexplored_corner(C1,T1),
     happens(go_straight,T1,T2),
     happens(explore,T3,T4), before(T2,T3).

happens(explore,T1,T4) :-
     holds_at(loc(corner(C1),behind),T1),
     unexplored_corner(C1,T1),
     happens(follow_wall,T1,T2),
     happens(explore,T3,T4), before(T2,T3).

happens(explore,T1,T4) :-
     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).

initiates(explore,knows_map,T).

holds_at(knows_map,T) :-
     not(unexplored_door(D,T)),
     not(unexplored_corner(C,T)).

unexplored_corner(C1,T) :-
     pos(C1,P), not(next_corner(R,C1,C2)).

unexplored_door(D,T) :-
     door(D,C1,C2), not(connects(D,R1,R2)).

/* Integrity constraints */

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

inconsistent([next_corner(R,C1,C2),
     next_corner(R,C1,C3), not(eq(C2,C3))]).

inconsistent([next_corner(R1,C1,C2),
     next_corner(R2,C1,C2), not(eq(R1,R2))]).

/* Constraints */

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).
```

Appendix C: Shared Code

```
/* Primitive Actions */

initiates(follow_wall,
          loc(corner(C2),ahead),T) :-
     holds_at(loc(corner(C1),behind),T),
     next_visible_corner(C1,C2,left,T).

terminates(follow_wall,loc(corner(C),behind),T).

next_visible_corner(C1,C2,left,T) :-
     holds_at(in(R),T),
     next_corner(R,C1,C2),
     not(invisible_corner(C2,T)).

next_visible_corner(C1,C3,left,T) :-
     holds_at(in(R),T),
     next_corner(R,C1,C2),
     invisible_corner(C2,T),
     next_visible_corner(C2,C3,left,T).

invisible_corner(C1,T) :-
    door(D,C1,C2),holds_at(neg(door_open(D)),T).
```

```
invisible_corner(C1,T) :-
    door(D,C2,C1),holds_at(neg(door_open(D)),T).

initiates(go_straight,
          loc(corner(C2),behind),T) :-
     holds_at(loc(corner(C1),ahead),T),
     door(D,C1,C2).

terminates(go_straight,
          loc(corner(C1),ahead),T) :-
     holds_at(loc(corner(C1),ahead),T),
     door(D,C1,C2).

initiates(turn(left),loc(door(D),in),T) :-
     holds_at(loc(corner(C1),ahead),T),
     door(D,C1,C2),
     holds_at(door_open(D),T).

terminates(turn(left),
          loc(corner(C1),ahead),T) :-
     holds_at(loc(corner(C1),ahead),T),
     door(D,C1,C2),
     holds_at(door_open(D),T).

initiates(turn(left),
          loc(corner(C2),behind),T) :-
     holds_at(loc(door(D),in),T),
     holds_at(in(R1),T),
     connects(D,R1,R2), door(D,C1,C2),
     next_corner(R2,C1,C2).

terminates(turn(left),loc(door(D),in),T) :-
     holds_at(loc(door(D),in),T).

initiates(turn(left),in(R2),T) :-
     holds_at(loc(door(D),in),T),
     holds_at(in(R1),T),  connects(D,R1,R2).

terminates(turn(left),in(R1),T) :-
     holds_at(loc(door(D),in),T),
     holds_at(in(R1),T).

initiates(turn(right),
          loc(corner(C),behind),T) :-
     holds_at(loc(corner(C),ahead),T),
     inner(C).

terminates(turn(right),
          loc(corner(C),ahead),T) :-
     holds_at(loc(corner(C),ahead),T), inner(C).

initiates(turn(right),facing(W1),T) :-
     holds_at(facing(W2),T), plus_90(W2,W1).

terminates(turn(right),facing(W),T) :-
     holds_at(facing(W),T).

initiates(turn(left),facing(W1),T) :-
     holds_at(facing(W2),T), minus_90(W2,W1).

terminates(turn(left),facing(W),T) :-
     holds_at(facing(W),T).

initiates(follow_wall,co_ords(P),T) :-
     holds_at(loc(corner(C1),behind),T),
     next_visible_corner(C1,C2,left,T),
     pos(C2,P).

terminates(follow_wall,co_ords(P),T) :-
     holds_at(co_ords(P),T).

initiates(go_straight,co_ords(P),T) :-
     holds_at(loc(corner(C1),ahead),T),
     door(D,C1,C2), pos(C2,P).

terminates(go_straight,co_ords(P),T) :-
     holds_at(co_ords(P),T).

initiates(turn(left),co_ords(P),T) :-
     holds_at(loc(door(D),in),T),
     holds_at(in(R1),T),
     connects(D,R1,R2),
     door(D,C1,C2), next_corner(R2,C1,C2),
     pos(C2,P).

terminates(turn(left),co_ords(P),T) :-
     holds_at(loc(door(D),in),T),
     holds_at(co_ords(P),T).

/* Sensor events */

happens(left_and_front(X),T,T) :-
     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).

happens(left(X),T,T) :-
     happens(turn(right),T,T),
     holds_at(loc(corner(C),ahead),T), inner(C).

happens(left(X),T,T) :-
     happens(turn(left),T,T),
     holds_at(loc(door(D),in),T),
     holds_at(in(R1),T),
     connects(D,R1,R2), connects(D,R2,R1),
     holds_at(co_ords(P1),T),
     holds_at(facing(W1),T),
     next_corner(R2,C3,C2), door(D,C3,C2),
     wall_thickness(Y1), displace(P1,Y1,W1,P2),
     pos(C2,P2), door_width(Y2), plus_90(W1,W2),
     displace(P2,Y2,W2,P3), pos(C3,P3).

happens(left(X),T,T) :-
     happens(go_straight,T,T),
     holds_at(co_ords(P1),T),
     holds_at(facing(W),T),
     holds_at(loc(corner(C1),ahead),T),
     holds_at(in(R),T),
     next_corner(R,C1,C2), door(D,C1,C2),
     displace(P1,X,W,P2), pos(C2,P2).

happens(left_gap(X),T,T) :-
     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),
     not(inner(C2)),
     displace(P1,X,W,P2), pos(C2,P2).
```