Reinventing Shakey

Murray Shanahan

Department of Computer Science,
Queen Mary and Westfield College,
Mile End Road,
London E1 4NS,
England.
mps@dcs.qmw.ac.uk

Abstract

This paper presents analogous logical characterisations of perception and planning in a mobile robot, and describes their implementation via an abductive logic programming meta-interpreter. In addition, the incorporation of the perception and planning processes in a sense-plan-act cycle is described.

Introduction

In the late Sixties, when the Shakey project started at SRI, 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 but *ad hoc* 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.

Let's briefly review the Shakey project [Nilsson, 1984] and Brooks's critique of it [Brooks, 1991]. Shakey worked by building an internal model of the world, using a logic-based representational formalism. This model was used by the STRIPS planner to find a sequence of actions for achieving a desired goal. Having constructed such a plan, Shakey proceeded to execute it via a plan execution module, which possessed some ability to recover from action failure and had a limited re-planning capacity.

According to Brooks, robots constructed in the style of Shakey are handicapped by the need for all information from the sensors to pass through the modelling and planning modules before having any effect on the robot's motors. Hence Shakey's success was confined to highly engineered, static environments. The Shakey approach, so the argument goes, is bound to fail in a more realistic, dynamic environment, in which there isn't time to construct a plan in response to ongoing, unexpected events.

On the other hand, the Shakey style of architecture, having an overtly deliberative component, seems to offer researchers a direct path to robots with high-level cognitive skills, such as planning, reasoning about other agents, and possibly language. Accordingly, many roboticists have attempted to reconcile the deliberative aspect of the sense-model-plan-act architecture with the sort of reactive architecture favoured by Brooks and his followers.

Other researchers have instigated a more thoroughgoing Shakey revival, and are aiming to achieve robots with the sort of high-level cognitive skills listed above by using logic as a representational medium [Lespérance, *et al.*, 1994]. Robots whose design is based on logical representation have the virtue of producing behaviour which can be accounted for in terms of *correct reasoning* and *correct representation*. Moreover, where the Shakey project compromised on the representation of the effects of actions, using the STRIPS language instead of logic, contemporary researchers, armed with modern solutions to the frame problem [Shanahan, 1997a], expect to go one better than their predecessors and use logic throughout.

The work reported here shares the aims of both the above groups of researchers. In particular, this paper presents correctly implemented logical accounts of

- planning,
- perception, and
- the interleaving of sensing, planning and acting.

The third bullet is critical for achieving the hoped-for combination of deliberation and reactivity.

A logic programming approach is taken to the problem of correctly implementing the logical accounts in question, since it offers a suitable compromise between the unattainable ideal of directly using a general purpose theorem prover, and the use of efficient special purpose algoithms whose intermediate computational states may have no declarative meaning.

1 Representing Action

To supply the required logical accounts of perception and planning, a formalism for reasoning about action is needed. The formalism presented here is based on the circumscriptive event calculus [Shanahan, 1997a]. Because this material 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(f,t) represents that fluent f holds at time t. Throughout the paper, all variables are universally quantified with maximum scope, unless otherwise indicated.

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

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

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

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

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

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

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

A particular *domain* is described in terms of Initiates, Terminates, and Releases formulae. Initiates(a,f,t) represents that fluent f starts to hold after action a at time t. Conversely, Terminates(a,f,t) represents that f starts not to hold after action a at t. Releases(a,f,t) represents that fluent f is no longer subject to the common sense law of inertia after action a at t.

A particular *narrative* of events is described in terms of Happens and Initially formulae. The formulae $\text{Initially}_P(f)$ and $\text{Initially}_N(f)$ respectively represent that fluent f holds at time 0 and does not hold at time 0. Happens(a,t1,t2) represents that action or event a occurs, starting at time t1 and ending at time t2.

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

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

As we'll see in Section 5, similar formulae can be used to define compound events.

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

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

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 all 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.

A detailed account of this formalism and the accompanying solution to the frame problem can be found in [Shanahan, 1997a].

2 A Logical Account of Planning

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. Given Σ, Ω and Δ as above, we're interested in HoldsAt formulae Γ such that,

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

Conversely, as first pointed out by Eshghi [1988], 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

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

This logical characterisation of event calculus planning is analogous to Green's logical characterisation of situation calculus planning [Green, 1969].

2.1 An Example Planning Problem

Figure 1 shows the layout of the rooms and doorways in the environment used for the robot experiments that are being carried out to vindicate the ideas of this paper. We'll consider the problem of planning a sequence of "go through door" actions to lead from the robot's initial location to a goal location. This problem can, of course, be recast as a trivial graph search, but here it will be used as an example of general purpose planning.

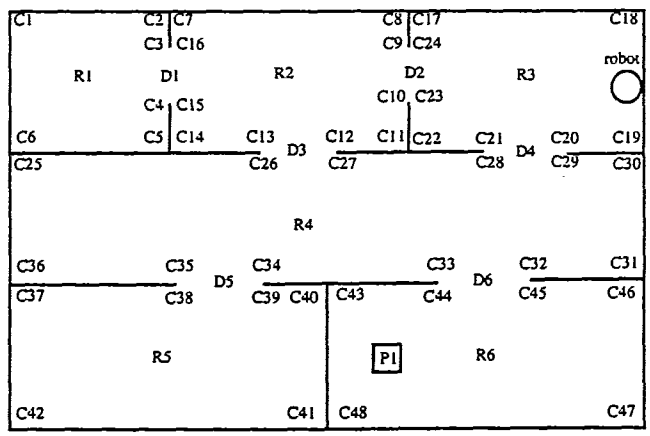

Figure 1: The Robot's Pen

The following formulae capture the connectivity of the rooms.

Connects(D1,R1,R2) (W2.1)

Connects(D2,R2,R3) (W2.2)

Connects(D3,R2,R4) (W2.3)

Connects(D4,R3,R4) (W2.4)

Connects(D5,R4,R5) (W2.5)

Conects(D6,R4,R6) (W2.6)

Connects(d,r1,r2) ← Connects(d,r2,r1) (W2.7)

The robot can perform only one action, which is to go through a specified door d, denoted by the term GoThrough(d). The only fluent in the domain is InRoom(r) representing that the robot is in room r. We have the following Initiates and Terminates formulae.

Initiates(GoThrough(d),InRoom(r1),t) ← Connects(d,r2,r1) ∧ HoldsAt(InRoom(r2),t) (R2.1)

Terminates(GoThrough(d),InRoom(r),t) ← HoldsAt(InRoom(r),t) (R2.2)

Since there is only one action and only one fluent, this example doesn't require any uniqueness-of-names axioms.

Suppose the robot is initially in room R3.

Initiallyp(InRoom(R3)) (N2.1)

The goal is to get the robot to room R6.

HoldsAt(InRoom(R6),T) (G2.1)

Clearly one plan for achieving the goal is to go through D4 then go through D6.

Happens(GoThrough(D4),T1) (P2.1)

Happens(GoThrough(D6),T2) (P2.2)

T1 < T2 (P2.3)

T2 < T (P2.4)

The fact that this is a plan according to the abductive definition is expressed in the following proposition. Let

- Σ be the conjunction of (R2.1) and (R2.2)
- Δ_N be formula (N2.1),
- Δ_P be the conjunction of (P2.1) to (P2.4),
- Φ be the conjunction of (W2.1) to (W2.7), and
- Γ be formula (G2.1).

Proposition 2.1.

CIRC[Σ ; Initiates, Terminates, Releases] ∧
CIRC[$\Delta_N \wedge \Delta_P$; Happens] ∧
EC ∧ $\Phi \models \Gamma$. □

More complex examples of event calculus planning are to be found in [Shanahan, 1997c], as well as in Section 5 of this paper.

3 Abductive Event Calculus Through Logic Programming

This section presents a logic programming implementation of abductive event calculus planning. The computation carried out by the resulting implementation is very similar to that of a partial-order planning algorithm, such as UCPOP [Penberthy & Weld, 1992]. But, as we'll see, essentially the same logic program also implements both abductive sensor data assimilation and hierarchical planning.

The implementation is in essence a resolution based abductive theorem prover, coded as a Prolog meta-interpreter. But this theorem prover is tailored for the event calculus by compiling the event calculus axioms into the meta-level, resulting in an efficient implementation.

In what follows, I will assume some knowledge of logic programming concepts and terminology. Following convention logic program variables will begin with upper-case letters, while constants and function symbols will begin with lower-case letters.

Meta-interpreters are a standard part of the logic programmer's toolkit. For example, the following "vanilla" meta-interpreter, when executed by Prolog, will mimic Prolog's own execution strategy.

```
demo([]).
demo([G|Gs1]) :-
  axiom(G,Gs2), append(Gs2,Gs1,Gs3),
  demo(Gs3).
demo([not(G)|Gs]) :- not demo([G]), demo(Gs).
```

The formula `demo(Gs)` holds if `Gs` follows from the object-level program. If Π is a list of Prolog literals $[\lambda_1, \ldots, \lambda_n]$, then the formula `axiom`$(\lambda_0, \Pi)$ holds if there is a clause of the following form in the object-level program.

λ_0 :- $\lambda_1, \ldots, \lambda_n$

One of the tricks we'll employ here is to compile object-level clauses into the meta-level. For example, the above clause can be compiled into the definition of `demo` through the addition of the following clause.

```
demo([λ0|Gs1]) :-
  axiom(λ1,Gs2),
  append(Gs2,[λ2, ..., λn|Gs1],Gs3),
  demo(Gs3).
```

The resulting behaviour is equivalent to that of the vanilla meta-interpreter with the object-level clause. Now consider the following object-level clause, which corresponds to Axiom (EC2) of Section 1.

```
holds_at(F,T3) :-
  happens(A,T1,T2), T2 < T3,
  initiates(A,F,T1), not clipped(T1,F,T2).
```

This can be compiled into the following meta-level clause, in which the predicate `before` is used to represent temporal ordering.

```
demo([holds_at(F,T3)|Gs1]) :-
  axiom(initiates(A,F,T1),Gs2),
  axiom(happens(A,T1,T2),Gs3),
  axiom(before(T2,T3),[]),
  demo([not clipped(T1,F,T3)]),
  append(Gs3,Gs2,Gs4), append(Gs4,Gs1,Gs5),
  demo(Gs5).
```

To represent Axiom (EC5), which isn't in Horn clause form, we introduce the function neg. Throughout our logic program, we replace the classical predicate calculus formula ¬ HoldsAt(f,t) with `holds_at(neg(F),T)`. So we obtain the following object-level clause.

```
holds_at(neg(F),T3) :-
  happens(A,T1,T2), T2 < T3,
  terminates(A,F,T1),
  not declipped(T1,F,T2).
```

This compiles into the following meta-level clause.

```
demo([holds_at(neg(F),T3)|Gs1]) :-
  axiom(terminates(A,F,T1),Gs2),
  axiom(happens(A,T1,T2),Gs3),
  axiom(before(T2,T3),[]),
  demo([not declipped(T1,F,T3)]),
  append(Gs3,Gs2,Gs4), append(Gs4,Gs1,Gs5),
  demo(Gs5).
```

The Prolog execution of these two meta-level clauses doesn't mimic precisely the Prolog execution of the corresponding object-level clause. This is because we have taken advantage of the extra degree of control available at the meta-level, and adjusted the order in which the sub-goals of `holds_at` are solved. For example, in the above clause, although we resolve on `terminates` immediately, we postpone further work on the sub-goals of `terminates` until after we've resolved on `happens` and `before`. This manoeuvre is required to prevent looping.

The job of an abductive meta-interpreter is to construct a *residue* of *abducible* literals that can't be proved from the object-level program. In the case of the event calculus, the abducibles will be `happens` and `before` literals. Here's a "vanilla" abductive meta-interpreter, without negation-as-failure.

```
abdemo([],R,R).

abdemo([G|Gs],R1,R2) :-
  abducible(G), abdemo(Gs,[G|R1],R2).

abdemo([G|Gs1],R1,R2) :-
  axiom(G,Gs2), append(Gs2,Gs1,Gs3),
  abdemo(Gs3,R1,R2).
```

The formula `abdemo(Gs,R1,R2)` holds if `Gs` follows from the conjunction of `R2` with the object-level program. (`R1` is the input residue and `R2` is the output residue.) Abducible literals are declared via the `abducible` predicate. In top-level calls to `abdemo`, the second argument will usually be `[]`.

Things start to get tricky when we incorporate negation-as-failure. The difficulty here is that when we add to the residue, previously proved negated goals may no longer be provable. So negated goals (such as negated `clipped` goals) have to be recorded and re-checked each time the residue is modified. Here's a version of `abdemo` which handles negation-as-failure.

```
abdemo([],R,R,N).                                   (A3.1)

abdemo([G|Gs],R1,R3,N) :-                           (A3.2)
  abducible(G), abdemo_nafs(N,[G|R1],R2),
  abdemo(Gs,R2,R3,N).

abdemo([G|Gs1],R1,R2,N) :-                          (A3.3)
  axiom(G,Gs2), append(Gs2,Gs1,Gs3),
  abdemo(Gs3,R1,R2,N).

abdemo([not(G)|Gs],R1,R3,N) :-                      (A3.4)
  abdemo_naf([G],R1,R2),
  abdemo(Gs,R2,R3,[[G]|N]).
```

The last argument of the `abdemo` predicate is a list of negated goal lists, which is recorded for subsequent checking (in Clause (A3.2)). If N = $[\gamma_{1,1} \cdots \gamma_{1,n_1}] \cdots [\gamma_{m,1} \cdots \gamma_{m,n_m}]$] is such a list, then its meaning, assuming a completion semantics for our object-level logic program, is,

$$\neg(\gamma_{1,1} \wedge \ldots \wedge \gamma_{1,n_1}) \wedge \neg(\gamma_{m,1} \wedge \ldots \wedge \gamma_{m,n_m}).$$

The formula `abdemo_nafs(N,R1,R2)` holds if the above formula is provable from the (completion of the) conjunction of `R2` with the object-level program. (In the vanilla version, `abdemo_nafs` doesn't add to the residue. However, we will eventually require a version which does, as we'll see shortly.)

`abdemo_nafs(N,R1,R2)` applies `abdemo_naf` to each list of goals in N. `abdemo_naf` is defined in terms of Prolog's `findall`, as follows.

```
abdemo_naf([G|Gs1],R,R) :-
  not resolve(G,R,Gs2).

abdemo_naf([G1|Gs1],R1,R2) :-
  findall(Gs2,(resolve(G1,R1,Gs3),
    append(Gs3,Gs1,Gs2)),Gss),
  abdemo_nafs(Gss,R1,R2).

resolve(G,R,Gs) :- member(G,R).

resolve(G,R,Gs) :- axiom(G,Gs).
```

The logical justification for these clauses is as follows. In order to show, $\neg(\gamma_1 \wedge \ldots \wedge \gamma_n)$, we have to show that, for every object-level clause λ :- $\lambda_1 \ldots \lambda_m$ which resolves with γ_1, $\neg(\lambda_1 \wedge \ldots \wedge \lambda_m, \gamma_2 \wedge \ldots \wedge \gamma_n)$. If no clause resolves with γ_1 then, under a completion semantics, $\neg\gamma_1$ follows, and therefore so does $\neg(\gamma_1 \wedge \ldots \wedge \gamma_n)$.

However, in the context of incomplete information about a predicate we don't wish to assume that predicate's completion, and we cannot therefore legitimately use negation-as-failure to prove negated goals for that predicate.

The way around this is to trap negated goals for such predicates at the meta-level, and give them special treatment. In general, if we know $\neg\phi \leftarrow \psi$, then in order to prove $\neg\phi$, it's sufficient to prove ψ. Similarly, if we know $\neg\phi \leftrightarrow \psi$, then in order to prove $\neg\phi$, it's both necessary and sufficient to prove ψ.

In the present case, we have incomplete information about the `before` predicate. Accordingly, when the meta-interpreter encounters a goal of the form `not before(X,Y)`, which it will when it comes to prove (or re-prove) a negated `clipped` goal, it attempts to prove `before(Y,X)`. One way to achieve this is to add `before(Y,X)` to the residue, first checking that the resulting residue is consistent.

Similar considerations affect the treatment of the `holds_at` predicate, which inherits the incompleteness of `before`. When the meta-interpreter encounters a `not holds_at(F,T)` goal, where F is a ground term, it attempts to prove `holds_at(neg(F),T)`, and conversely, when it encounters `not holds_at(neg(F),T)`, it attempts to prove `holds_at(F,T)`. In both cases, this can result in further additions to the residue.

Note that these techniques for dealing with negation in the context of incomplete information are general in scope. They're generic theorem proving techniques, and their use isn't confined to the event calculus. For further details of the implementation of `abdemo_naf`, the reader should consult the (electronic) appendix.

As with the `demo` predicate, we can compile the event calculus axioms into the definition of `abdemo` and `abdemo_naf` via the addition of some extra clauses, giving us a finer degree of control over the resolution process. Here's an example.

```
abdemo([holds_at(F,T3)|Gs1],R1,R4,N) :-   (A3.5)
  axiom(initiates(A,F,T1),Gs2),
  abdemo_nafs(N,
    [happens(A,T1,T2),before(T2,T3)|R1],R2),
  abdemo_nafs([clipped(T1,F,T3)],R2,R3),
  append(Gs2,Gs1,Gs3),
  demo(Gs3,R3,R4,[clipped(T1,F,T3)|N]).
```

Now, to solve a planning problem, we simply describe the effects of actions directly as Prolog `initiates`, `terminates` and `releases` clauses, we present a list of `holds_at` goals to `abdemo`, and the returned residue, comprising `happens` and `before` literals, is a plan.

Soundness and completeness results for this implementation are presented elsewhere, but they should come as no surprise since, as shown in [Shanahan, 1997c], the computation performed by the planner is in essence the same as that carried out by UCPOP, for which such results are well-known [Penberthy & Weld, 1992].

4 A Logical Account of Perception

This section offers a logical account of sensor data assimilation (perception) which mirrors the logical account of planning in Section 2. 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.

The methodology for supplying the required logical account is as follows. First, using a suitable formalism for reasoning about actions, 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. Second, consider sensor data assimilation as abduction with this theory. Roughly speaking, 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,

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

In event calculus terms, 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 which have modified that configuration.

To illustrate this, we'll pursue the mail delivery domain used in Section 2. But we must begin with a look at the sensory capabilities of the Khepera robots which are being used to test the ideas presented in this paper.

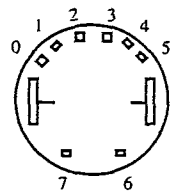

Figure 2: The Khepera Robot

The Khepera robot is a miniature robot platform with two drive wheels and a suite of eight infra-red proximity sensors around its circumference. The infra-red sensors are arranged in pairs, as shown in Figure 2.

The Khepera can be straightforwardly programmed to navigate around the environment of Figure 1. Using its proximity sensors, it can follow walls and detect inner and outer corners. If all the doors are open, the GoThrough action of Section 2 can be executed, assuming the robot's initial location is known, by counting inner and outer corners until the robot reaches the required door, then passing through it.

If any of the doors is closed, however, this approach to executing the GoThrough action will fail, because the infra-red proximity sensors cannot detect a closed door, which looks to them just like a continuation of the wall.

This, in an extreme form, is the predicament facing any perceptual system. Inference must be carried out on raw sensor data in order to produce knowledge. In this case, the robot can abduce the fact that a door is closed as the only possible explanation of its unexpected arrival at an inner corner instead of the outer corner of the doorway.

Our aim here is to give a formal account of this sort of inference that gels with the formal account of planning already supplied. Indeed, as we'll see, in the final system, the same knowledge, expressed using the same formalism, is used for both planning and sensor data assimilation. Furthermore, as already emphasised, both planning and sensor data assimilation are viewed as abductive tasks with a very similar character. This means that the same logic programming implementation techniques, indeed the very same abductive meta-interpreter, can be applied to both tasks.

4.1 The Robot's Environment

Returning to the example at hand, the representation of the robot's environment, as depicted in Figure 1, now needs to include corners, which were neglected in the planning

example. The formula NextCorner(r,c1,c2) represents that corner c2 is the next inneror outer corner in room r after corner c1, in a clockwise direction. For room R1 alone, we have the following formulae.

NextCorner(R1,C1,C2)

NextCorner(R1,C2,C3)

NextCorner(R1,C3,C4)

NextCorner(R1,C4,C5)

NextCorner(R1,C5,C6)

NextCorner(R1,C6,C1)

In addition, the formula Door(d,c1,c2) represents that there is a doorway between the two corners c1 and c2. For each door, there will be a pair of such formulae. Here they are for door D1.

Door(D1,C3,C4)

Door(D1,C15,C16)

Finally, the formulae Inner(c) and Outer(c) represent respectively that c is an inner corner and c is an outer corner. Again confining our attention to room R1, we have the following.

Inner(C1)	Inner(C2)
Outer(C3)	Outer(C4)
Inner(C5)	Inner(C6)

Each of these predicates will need to be minimised using circumscription, so that their completions are formed.

4.2 The Robot's Effect on the World

Now we can formalise the effects of the robot's actions on the world. To simplify the example, the following formulae assume the robot always hugs the left wall, although parameters are provided which allow for it to hug the right wall as well.

Again, a finer grain of detail is required than for the planning example. Instead of a single GoThrough action, the robot's repertoire now comprises three actions: FollowWall, Turn(s), and GoStraight, where s is either Left or Right. These actions affect three fluents. The fluent AtCorner(c,s) holds if the robot is at (inner or outer) corner c, with c in direction s, where s is Left or Right. The fluent BesideWall(w,s) holds if the robot is adjacent to wall w in direction s, where s is Left or Right. The fluent InDoorway(d,r) holds if the robot is in doorway d, with its back to room r. (By convention, the three fluents are mutually exclusive.)

Let's formalise the effects of the three actions in turn. Each action is assumed to be instantaneous, an assumption which has no practical implications in the present example. The term Wall(c1,c2) denotes the wall between corners c1 and c2. First, if the robot follows a wall, it ends up at the next visible corner.

Initiates(FollowWall,AtCorner(c3,Left),t) ← (K4.1)
  HoldsAt(BesideWall(Wall(c1,c2),Left),t) ∧
    NextVisibleCorner(c1,c3,Left,t)

Terminates(FollowWall,BesideWall(w,s),t) (K4.2)

The formulae NextVisibleCorner(c1,c2,s,t) means that, at time t, c2 is the next visible corner after c1, where the wall in question is in direction s. The corner of a doorway whose door is closed is invisible.

NextVisibleCorner(c1,c2,Left,t) ← (K4.3)
  NextCorner(r,c1,c2) ∧ ¬ InvisibleCorner(c2,t)

NextVisibleCorner(c1,c3,Left,t) ← (K4.4)
  NextCorner(r,c1,c2) ∧ InvisibleCorner(c2,t) ∧
    NextVisibleCorner(c2,c3,Left,t)

NextVisibleCorner(c1,c2,s,t) ∧ (K4.5)
  NextVisibleCorner(c1,c3,s,t) → c2 = c3

InvisibleCorner(c1,t) ↔ (K4.6)
  ∃ d,c2 [[Door(d,c1,c2) ∨ Door(d,c2,c1)] ∧
    ¬ HoldsAt(DoorOpen(d),t)]

Next we have the GoStraight action, which the robot executes to bypass a doorway, travelling in a straight line from the near corner of the doorway and coming to rest when it detects the far corner.

Initiates(GoStraight, (K4.7)
  BesideWall(Wall(c2,c3),Left),t) ←
    HoldsAt(AtCorner(c1,Left),t) ∧
      Door(d,c1,c2) ∧ NextCorner(r,c2,c3)

Initiates(GoStraight, (K4.8)
  BesideWall(Wall(c2,c1),Right),t) ←
    HoldsAt(AtCorner(c3,Right),t) ∧
      Door(d,c2,c3) ∧ NextCorner(r,c2,c1)

Terminates(GoStraight,AtCorner(c,s),t) (K4.9)

Finally we have the Turn action. Since the robot has to hug the left wall, it always turns left (or goes straight) at outer corners, and always turns right at inner corners. If it turns left at the corner of a doorway, it ends up in the doorway.

Initiates(Turn(Left),InDoorway(d,r),t) ← (K4.10)
  HoldsAt(AtCorner(c1,Left),t) ∧ Door(d,c1,c2) ∧
    HoldsAt(DoorOpen(d),t) ∧ NextCorner(r,c1,c2)

If the robot turns left when in a doorway, it ends up alongside a wall in the next room.

Initiates(Turn(Left), (K4.11)
  BesideWall(Wall(c2,c3),Left),t) ←
    HoldsAt(InDoorway(d,r1),t) ∧ Connects(d,r1,r2) ∧
      Door(d,c1,c2) ∧ NextCorner(r2,c1,c2)

If the robot turns right at an inner corner, it ends up next to a new wall.

Initiates(Turn(Right), (K4.12)
  BesideWall(Wall(c1,c2),Left),t) ←
    HoldsAt(AtCorner(c1),t) ∧
      Inner(c2) ∧ NextCorner(r,c1,c2)

The mutual exclusivity of the AtCorner, InDoorway and BesideWall fluents is preserved by the following formulae.

Terminates(Turn(s1),AtCorner(c,s2),t) (K4.13)

Terminates(Turn(s),InDoorway(d,r),t) ← (K4.14)
  HoldsAt(InDoorway(d,r),t)

4.3 The Effect of the World on the Robot

Having axiomatised the effects of the robot's actions on the world, now we need to formalise the impact the world has on the robot's sensors. For this purpose, we introduce two new types of event. The event GoesHigh(s) occurs if the average value of the two sensors in direction s exceeds a threshold $\delta 1$, where s is Left, Right or Front. Similarly the event GoesLow(s) occurs if the average value of the two sensors in direction s goes below a threshold $\delta 1$. (By making $\delta 1 > \delta 2$, we avoid a chatter of GoesHigh and GoesLow events when the robot approaches an obstacle.)

Happens(GoesHigh(Front),t) ← (S4.1)
  Happens(FollowWall,t) ∧
    Initiates(FollowWall,AtCorner(c,s),t) ∧ Inner(c)

Happens(GoesLow(Front),t) ← (S4.2)
  HoldsAt(AtCorner(c,Left),t) ∧
    Inner(c) ∧ Happens(Turn(Right),t)

Happens(GoesHigh(s),t) ← (S4.3)
  HoldsAt(AtCorner(c,s),t) ∧ Outer(c) ∧
    [Happens(GoStraight,t) ∨ Happens(Turn(s),t)]

Happens(GoesLow(s),t) ← (S4.4)
  Happens(FollowWall,t) ∧
    Initiates(FollowWall,AtCorner(c,s),t) ∧ Outer(c)

Our overall aim, of course, is to use abduction to explain the occurrence of GoesHigh and GoesLow events. In the present example, if the doors are all initially open and never subsequently closed, every sensor event is predicted by the theory as it stands, so no explanation is required. The interesting case is where there are sensor events which can only be explained by a closed door. Accordingly, we need to introduce the events OpenDoor(d) and CloseDoor(d), with the obvious meanings.

Initiates(OpenDoor(d),DoorOpen(d),t) (K4.15)

Terminates(CloseDoor(d),DoorOpen(d),t) (K4.16)

Finally, we need some uniqueness-of-names axioms.

UNA[FollowWall, GoStraight, Turn, GoesHigh, GoesLow, OpenDoor, CloseDoor] (U4.3)

UNA[BesideWall, AtCorner, InDoorway, DoorOpen] (U4.4)

4.4 An Example Narrative

Now let's examine a particular narrative of robot actions and sensor events. The following formulae describe the initial situation.

Initiallyp(DoorOpen(d)) (N4.1)

Initiallyp(BesideWall(w,s)) ↔ (N4.2)
  w = Wall(C18,C19) ∧ s = Left

Initially_N(AtCorner(c,s)) (N4.3)

Initially_N(InDoorway(d,r)) (N4.4)

The robot follows the wall to its left until it arrives at corner C19, where it turns right and follows the wall to its left again.

Happens(FollowWall,T1) (N4.5)

Happens(Turn(Right),T2) (N4.6)

Happens(FollowWall,T3) (N4.7)

T1 < T2 (N4.8)

T2 < T3 (N4.9)

Now let's suppose someone closes door D4 shortly after the robot sets out, and consider the incoming sensor events. The robot's front sensors go high at time T1, when it arrives at corner C19. (Recall that the FollowWall action is considered instantaneous.) They go low when it turns, then (unexpectedly) go high again, when it arrives at corner C22, having bypassed door D4.

Happens(GoesHigh(Front),T1) (D4.1)

Happens(GoesLow(Front),T2) (D4.2)

Happens(GoesHigh(Front),T3) (D4.3)

The above formulae only describe the sensor events that *do* occur. But in general, we want explanations of sensor data to exclude those sensor events that have *not* occurred. Hence we have the following definition, which captures the completion of the Happens predicate for sensor events.

Definition 4.1.

$$\text{COMP}[\Psi] \equiv_{def} [\text{Happens}(a,t) \wedge [a = \text{GoesHigh}(s) \vee a = \text{GoesLow}(s)]] \rightarrow \bigvee_{\langle\alpha,\tau\rangle \in \Pi} [a = \alpha \wedge t = \tau]$$

where $\Pi = \{\langle\alpha,\tau\rangle \mid \text{Happens}(\alpha,\tau) \in \Psi\}$ □

The following formula is one possible explanation of the above sensor events.

Happens(CloseDoor(D4),t) ∧ 0 ≤ t < T30 (E4.1)

This is expressed by the following proposition. Let

- Σ be the conjunction of (K4.1) to (K4.16),
- Δ_N be the conjunction of (N4.1) to (N4.9),
- Δ_T be the conjunction of (S4.1) to (S4.4),
- Δ_E be formula (E4.1),
- Φ be the conjunction of the formulae representing the robot's environment, as described in Section 4.1,
- Ω be the conjunction of (U4.1) and (U4.2), and
- Γ be the conjunction of (D4.1) to (D4.3).

Proposition 4.1.

$$\text{CIRC}[\Sigma ; \text{Initiates, Terminates, Releases}] \wedge \text{CIRC}[\Delta_N \wedge \Delta_T \wedge \Delta_E ; \text{Happens}] \wedge \text{EC} \wedge \Omega \wedge \Phi \models \text{COMP}[\Gamma].$$ □

In general, a collection of sensor data can have many explanations. Explanations can be ordered using a preferrce criterion, such as one which favours explanations with few events. But there can still be many mimimal explanations. In these circumstances, the robot cannot do any better than simply to proceed on the assumption that the first explanation it finds is the true explanation. It's reasonable to expect that, if the explanation is indeed false,

the processing of subsequent sensor data will reveal this. But obviously this topic merits further inverstigation.

For discussion of a more complex example of abductive sensor data assimilation, involving both sensor and motor noise, see [Shanahan, 1997b], which shows how the same approach can be taken to map-building, where the incompleteness of the robot's knowledge of its environment is more radical than in the example used here.

Let me finish this section with a few words on implementation. The strong similarities between the abductive account of planning and the abductive account of perception, means that the meta-interpreter of Section 3 can be used to implement both. Furthermore, the same knowledge, in particular the formulae describing the effects of actions, is used by both tasks.

5 Interleaving Sensing, Planning, and Acting

The impression given by the foregoing accounts is that perception and planning are isolated tasks which run to completion, uninterrupted and in their own time. Of course, this is not true. To meet the criticisms of Brooks, outlined in the opening section, we need to show how perception, planning and plan execution can be smoothly integrated in a robot in such a way that the robot performs in real-time and reacts in a timely fashion to ongoing events.

For this reason, the robot cannot be allowed to devote an arbitrary amount of computation to planning, or to assimilating old sensor data, before it consults its sensors again, in case the world is presenting it with an urgent threat or opportunity. Rather, sensing, planning and acting have to be interleaved. The robot' sensors are read, then a bounded amount of time is allotted to processing sensor data before the robot moves on to planning. Then a bounded amount of time is given over to planning before the robot proceeds to act. Having acted, the robot consults its sensors again, and the cycle is repeated.

To realise this idea, however, the planner described in Sections 2 and 3 must be augmented. As it stands, the planner produces actions in regression order. That is to say, the last action to be carried out is generated first. This means that, if interrupted, the planner's partial results are useless. What we require instead is a progression planner — one that generates the earliest action of a plan first. If a progression planner is interrupted, its partially constructed plan will contain actions that can be executed immediately.

One way to generate plans in progression order is via *hierarchical planning*. This is the approach adopted here. It's a surprisingly straightforward matter to extend the foregoing logical treatment of partial order planning to planning via hierarchical decomposition. The representation of compound actions and events in the event calculus is very natural, and is best illustrated by example. The following formulae axiomatise a robot mail delivery domain that is an extension of the example from Section 2.

5.1 An Example of Hierarchical Planning

First we formalise the effects of the primitive actions. The term Pickup(p) denotes the action of picking up package p, the term PutDown(p) denotes the action of putting down package p. The fluent Got(p) holds if the robot is carrying the package p, and the fluent In(p,r) holds if package p is in room r.

$$\text{Initiates}(\text{Pickup}(p),\text{Got}(p),t) \leftarrow p \neq \text{Robot} \wedge \text{HoldsAt}(\text{InRoom}(r),t) \wedge \text{HoldsAt}(\text{In}(p,r),t) \quad \text{(R5.1)}$$

$$\text{Releases}(\text{Pickup}(p),\text{In}(p,r),t) \leftarrow p \neq \text{Robot} \wedge \text{HoldsAt}(\text{InRoom}(r),t) \wedge \text{HoldsAt}(\text{In}(p,r),t) \quad \text{(R5.2)}$$

$$\text{Initiates}(\text{PutDown}(p),\text{In}(p,r),t) \leftarrow p \neq \text{Robot} \wedge \text{HoldsAt}(\text{Got}(p),t) \wedge \text{HoldsAt}(\text{InRoom}(r),t) \quad \text{(R5.3)}$$

Formulae (R2.1) and (R2.2), which deal with the GoThrough action and InRoom fluent, are inherited from Section 2.

Next we have our first example of a compound action definition. Compound actions have duration, while primitive actions will usually be represented as instantaneous. The term ShiftPack(p,r) denotes the action of retrieving and delivering package p to room r. It comprises a RetrievePack action and a DeliverPack action, both of which are themselves compound actions. The term RetrievePack(p,r) denotes the action of retrieving package p from room r, and the term DeliverPack(p,r) denotes the action of delivering package p to room r.

$$\text{Happens}(\text{ShiftPack}(p,r1,r2),t1,t4) \leftarrow \text{Happens}(\text{RetrievePack}(p,r1),t1,t2) \wedge t2 < t3 \wedge \text{Happens}(\text{DeliverPack}(p,r1,r2),t3,t4) \wedge \neg\,\text{Clipped}(t2,\text{Got}(p),t3) \quad \text{(H5.1)}$$

$$\text{Initiates}(\text{ShiftPack}(p,r1,r2),\text{In}(p,r2),t) \leftarrow \text{HoldsAt}(\text{In}(p,r1),t) \quad \text{(R5.4)}$$

Note the need for the final ¬ Clipped condition to ensure the correctness of the definition in (H5.1). The RetrievePack and DeliverPack actions are defined in terms of Pickup, PutDown and GoToRoom actions. The GoToRoom action is another compound action.

$$\text{Happens}(\text{RetrievePack}(p,r1),t1,t3) \leftarrow \text{HoldsAt}(\text{InRoom}(r2),t1) \wedge \text{Happens}(\text{GoToRoom}(r2,r1),t1,t2) \wedge \text{Happens}(\text{Pickup}(p),t3) \wedge t2 < t3 \wedge \neg\,\text{Clipped}(t2,\text{InRoom}(r1),t3) \quad \text{(H5.2)}$$

$$\text{Initiates}(\text{RetrievePack}(p,r),\text{Got}(p),t) \leftarrow \text{HoldsAt}(\text{In}(p,r),t) \quad \text{(R5.5)}$$

$$\text{Happens}(\text{DeliverPack}(p,r1,r2),t1,t3) \leftarrow \text{Happens}(\text{GoToRoom}(r1,r2),t1,t2) \wedge t2 < t3 \wedge \text{Happens}(\text{PutDown}(p),t3) \wedge \neg\,\text{Clipped}(t2,\text{InRoom}(r2),t3) \quad \text{(H5.3)}$$

$$\text{Initiates}(\text{DeliverPack}(p,r1,r2),\text{In}(p,r2),t) \leftarrow \text{HoldsAt}(\text{InRoom}(r1),t) \wedge \text{HoldsAt}(\text{Got}(p),t) \quad \text{(R5.6)}$$

The effects of compound actions should follow from the effects of their sub-actions, as can be verified in this case by inspection. Next we have the definition of the GoToRoom action, where the term GoToRoom(r1,r2) denotes the action of going from room r1 to room r2.

Happens(GoToRoom(r,r),t,t) (H5.4)

Happens(GoToRoom(r1,r3),t1,t3) ← (H5.5)
  Connects(d,r1,r2) ∧
    Happens(GoThrough(d),t1) ∧
      Happens(GoToRoom(r2,r3),t2,t3) ∧
        t1 < t2 ∧ ¬ Clipped(t1,InRoom(r2),t2)

Initiates(GoToRoom(r1,r2),InRoom(r2),t) ← (R5.7)
  HoldsAt(InRoom(r1),t)

In effect, when implemented on a meta-interpreter such as that of Section 3, these clauses carry out a forward search, in contrast to the backward search effected by the clauses in Section 2. A more sophisticated set of clauses would incorporate a heuristic to give more direction to the search.

Formulae (H5.4) and (H5.5) illustrate both conditional decomposition and recursive decompostion: a compound action can decompose into different sequences of sub-actions depending on what conditions hold, and a compound action can be decomposed into a sequence of sub-actions that includes a compound action of the same type as itself. A consequence of this is that the event calculus with compound actions is formally as powerful as any programming language. In this respect, it can be used in the same way as GOLOG [Levesque, *et al.*, 1997]. Note, however, that we can freely mix direct programming with planning from first principles.

We need the following uniqueness-of-names axioms.

UNA[Pickup, PutDown, GoThrough, ShiftPack, GoToRoom] (U5.1)

UNA[Got, In, InRoom] (U5.2)

The definition of the planning task from Section 2 is unaffected by the inclusion of compound events. However, it's convenient to distinguish *fully decomposed* plans, comprising only primitive actions, from those that include compound actions.

The following formulae represent the initial situation depicted in Figure 1.

Initially(InRoom(R3)) (N5.1)

Initially(In(P1,R6)) (N5.2)

The following HoldsAt formula represents the goal state.

HoldsAt(In(P1,R4),T) (G5.1)

The following narrative of actions obviously constitutes a plan.

Happens(GoThrough(D4),T1) (P5.1)

Happens(GoThrough(D6),T2) (P5.2)

Happens(Pickup(P1),T3) (P5.3)

Happens(GoThrough(D6),T4) (P5.4)

Happens(PutDown(P1),T5) (P5.5)

T1 < T2 (P5.6)

T2 < T3 (P5.7)

T3 < T4 (P5.8)

T4 < T5 (P5.9)

T5 < T (P5.10)

Let

- Σ_P be the conjunction of (R2.1), (R2.2) and (R5.1) to (R5.3) (the effects of the primitive actions),
- Σ_C be the conjunction of (R5.4) and (R5.7) (the effects of the compound actions),
- Δ_C be the conjunction of (H5.1) to (H5.5),
- Δ_0 be the conjunction of (N5.1) and (N5.2),
- Δ_P be the conjunction of (P5.1) to (P5.10),
- Φ be the conjunction of (W2.1) to (W2.7) from Section 2.
- Ω be the conjunction of (U5.1) and (U5.2), and
- Γ be formula (G5.1).

Now we have. for example, the following proposition.

Proposition 5.1.

CIRC[$\Sigma_P \wedge \Sigma_C$; Initiates, Terminates, Releases] ∧
  CIRC[$\Delta_0 \wedge \Delta_P \wedge \Delta_C$; Happens] ∧ EC ∧ $\Omega \wedge \Phi \models$
    Happens(ShiftPack(P1,R6,R4),T0,T4). □

We also have the following.

Proposition 5.2.

CIRC[$\Sigma_P \wedge \Sigma_C$; Initiates, Terminates, Releases] ∧
  CIRC[$\Delta_0 \wedge \Delta_P \wedge \Delta_C$; Happens] ∧
    EC $\wedge \Omega \wedge \Phi \models \Gamma$. □

So Δ_P constitutes a plan. Furthermore, we have the following.

Proposition 5.3.

CIRC[Σ_P ; Initiates, Terminates, Releases] ∧
  CIRC[$\Delta_0 \wedge \Delta_P$; Happens] ∧ EC $\wedge \Omega \wedge \Phi \models \Gamma$. □

So Δ_P also constitutes a plan in the context of only the primitive actions.

5.2 Suspending the Planning Process

Propositions 5.2 and 5.3 show that, in this example as in many, the introduction of compound events has added little logically. The same set of HoldsAt formulae follows from any given narrative of primitive actions, with or without the compound event definitions. As we'll see shortly, the benefit of introducing compound events is only apparent when we look at implementation issues in the context of the sense-plan-act cycle.

First, note that an abductive meta-interpreter can be built using the principles outlined in Section 3 which, when presented with compound event definitions of the above form, automatically performs hierarchical decomposition. Whenever a `happens` goal is reached for a compound action, that goal is resolved against the clause defining the compound action, yielding further `happens` sub-goals, and this process continues until primitive actions are reached, which are added to the residue. The current

implementation incorporates an iterative deepening search strategy, which avoids looping problems that arise with certain examples of compound actions.

In general we will rarely require our planner to find fully decomposed plans, and we need to be able to suspend the planning process before a fully decomposed plan has been found and still have a useful result in the form of a partially decomposed plan. (The suspension of planning can be achieved in a logic programming implementation with a resource-bounded meta-interpreter such as that described by Kowalski [1995]. The same techniques can be used in a resource-bounded implementation of abductive sensor data assimilation.)

Let's see how this works with the above example. The meta-interpreter reduces the initial goal list

```
[holds_at(in(p1,r4,t)]
```

to

```
[happens(go_to_room(r6),t1,t2),
  happens(pickup(p1),t3), t2 < t3, t3 < t4,
    happens(deliver_pack(p1,r6,r4),t4,t)]
```

via the `shift_pack` compound event. Then the `go_to_room` action is decomposed, yielding

```
[happens(go_through(d4),t1),
  happens(go_through(d6),t2), t1 < t2,
    happens(pickup(p1),t3), t2 < t3, t3 < t4,
      happens(deliver_pack(p1,r6,r4),t4,t)].
```

At this point, although a fully decomposed plan hasn't been produced, the computation has yielded an action, namely `go_through(d4)`, that can be immediately executed, since the following criteria have been met.

- The goal list comprises only `happens` and temporal ordering goals.
- There is a primitive action in the goal list which is temporally before all other actions.

In the following section, we'll see how action execution is integrated with planning and sensor data assimilation, how events abduced from incoming sensor data can conflict with a plan, and how this precipitates replanning.

5.3 The Sense-Plan-Act Cycle

The motivation for suspending the planning process and acting on its partial results is the need for reactivity. If the hierarchy of compound actions is appropriately defined, the robot will be able to quickly generate an executable action in circumstances of imminent threat or opportunity. Where no high-level compound action is defined to achieve a given goal, the robot has to resort to search to find a plan. So it's up to the robot's builder to ensure a judicious mix of direct programming (via compound actions) and search.

In order to detect imminent threat or opportunity, planning is embedded in a tight feedback loop, the sense-plan-act cyle. Within this cycle, abductive planning and abductive sensor data assimilation behave like co-routines. But it's not just the need to respond to threats and opportunities that gives rise to the need for this feedback loop. Often in the course of executing a plan, it becomes apparent that certain assumptions on which the plan was based are false.

We can see this with a mail delivery example that combines the examples of Sections 4.4 and 5.1. As in Section 5.1's example of hierarchical planning, let's suppose the robot is initially in room R3, and wants to move package P1 from room R6 to room R4 (see Figure 1).

Using hierarchical planning as described above, the robot starts to form the obvious plan, represented by formulae (P5.1) to (P5.10). To flesh out this example fully would require Happens formulae defining GoThrough as a compound action comprising a number of FollowWall, GoStraight and Turn actions. Let's take these formulae for granted, and suppose the robot decomposes the GoThrough(D4) action into the appropriate sequence of sub-actions, beginning with FollowWall.

As soon as it comes up with this FollowWall action, the robot can start executing the plan. It sets out along Wall(C18,C19), then turns corner C18 and proceeds along Wall(C19,C20), all the while continuing to fill out the plan by decomposing remaining compound actions, and all the while continuing to process incoming sensor data.

Now, as in Section 4.4's example of sensor data assimilation, suppose someone closes door D3 before the robot reaches the doorway. With the the doorway now invisible to its sensors, the robot carries on until it reaches corner C22. Until it reaches corner C22, the processing of incoming sensor data has been a trivial task, as none of it has required an explanation. But the only way to explain the GoesHigh(Front) event that occurs when corner C22 is encountered is with a CloseDoor action, as in formula (E4.1).

This leads us to the question of how conflicts with plans are detected, and how they're overcome. Let's reconsider the internal mechanics of the planning process when implemented via an abductive meta-interpreter, as described in Section 3. Recall that, as actions are added to the residue (represented as a set of `happens` and `before` formulae), they must be checked against negated `clipped` goals that have been previously proved and recorded. Now, as exemplified by formula (E4.1), the ultimate product of the sensor data assimilation process, when similarly implemented, is also a set of `happens` and `before` formulae. So checking whether incoming sensor data indicates a conflict with the current plan is the same as re-proving negated `clipped` goals when new actions are added to the residue during planning itself, and the same mechanism can be re-used.

So, to return to the example, the robot reaches corner C22 at, say time T', and detects a conflict between the CloseDoor action it has abduced to explain its unexpected encounter with corner C22 and the formula ¬ Clipped(0,DoorOpen(D4),T') on which the success of its plan depends.

The robot's designer can handle this sort of situation in two ways. The subtler option is to incorporate some sort of plan repair mechanism, which with the least possible work

transforms the old failed plan into a new working one. But the present implementation opts for the simpler method of replanning from scratch whenever such a conflict is detected.

The rationale for this is as follows. The assumption behind the adoption of hierarchical planning was that compound action definitions can be written in such a way as to ensure that the planning process generates a first executable action quickly, whenever immediate action is required. If we accept this assumption, there is no need for a clever plan repair mechanism. Replanning from scratch is viable.

In the present example, the robot responds to the detected conflict by forming a new plan. The obvious plan takes it through door D2 into room R2, then through door D3 into room R4, then through door D6 into room R6 where it can pick up the package.

Concluding Remarks

It's important to see how different the robot described here is from the image of Shakey caricatured by Brooks and others, in which the robot sits motionless constructing a plan for minutes on end, then doggedly executes that plan, clinging to it come what may. In the present work, reactivity is achieved through the use of compound events and hierarchical planning, which permit the use of direct programming as well as planning, and also enable the robot to quickly generate an immediately executable action when it needs to. But at the same time, the logical integrity of the design has been preserved.

There are obviously many important areas for further investigation. Two particularly significant topics are *maintenance goals*, that is to say fluents which have to be always kept true, and so-called *knowledge producing actions*. In the present account, although any action can in effect be knowledge producing, as it can lead to the acquisition of new sensor data which in turn leads to an increase in the robot's knowledge, there is no way to *plan* to increase knowledge, since there is no representation of an action's knowledge increasing effects.

References

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

[Eshghi, 1988] K.Eshghi, Abductive Planning with Event Calculus, *Proceedings of the Fifth International Conference on Logic Programming* (1988), pp. 562–579.

[Green 1969] C.Green, Applications of Theorem Proving to Problem Solving, *Proceedings IJCAI 69*, pp. 219–240.

[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, *et al.*, 1997] H.Levesque, R.Reiter, Y.Lespérance, F.Lin and R.B.Scherl, GOLOG: A Logic Programming Language for Dynamic Domains, *The Journal of Logic Programming*, vol. 31 (1997), pp. 59–83.

[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, 1997c] M.P.Shanahan, Event Calculus Planning Revisited, *Proceedings 4th European Conference on Planning (ECP 97)*, Springer Lecture Notes in Artificial Intelligence no. 1348 (1997), pp. 390–402.