Event Calculus Planning Revisited

Murray Shanahan

Department of Computer Science,
Queen Mary and Westfield College,
Mile End Road,
London E1 4NS,
England.

Email: mps@dcs.qmw.ac.uk
Web: http://www.dcs.qmw.ac.uk/~mps

May 1997 DRAFT

Abstract

In 1969 Cordell Green presented his seminal description of planning as theorem proving with the situation calculus. The most pleasing feature of Green's account was the negligible gap between high-level logical specification and practical implementation. This paper attempts to reinstate the ideal of planning via theorem proving in a modern guise. In particular, I will show that if we adopt the event calculus as our logical formalism and employ abductive logic programming as our theorem proving technique, then the computation performed mirrors closely that of a hand-coded partial order planning algorithm. Furthermore, if we extend the event calculus in a natural way to accommodate compound actions, then using exactly the same abductive theorem prover we obtain a hierarchical planner. All this is a striking vindication of Kowalski's slogan “Algorithm = Logic + Control”.

Introduction

In 1969, Green offered a logical characterisation of planning couched in terms of the situation calculus, in addition to an implementation based on a resolution theorem prover. What makes Green’s treatment so attractive is the close correspondence between implementation and specification. The very same axioms that feature in the formal description of the planning task form the basis of the representation deployed by the implemented planner, and each computation step performed by the planner is a step in the construction of a proof that a suitable plan exists.

However, Green’s seminal work, though much admired, has had little impact on subsequent work in planning, owing to the widespread belief that a theorem prover cannot form the basis of a practical planning system. The following quote from [Russell & Norvig, 1995] exemplifies the widely held belief that planning via theorem proving is impractical.

> Unfortunately a good theoretical solution does not guarantee a good practical solution. . . . To make planning practical we need to do two things: (1) Restrict the language with which we define problems. . . . (2) Use a special purpose algorithm . . . rather than a general-purpose theorem prover to search for a solution. The two go hand in hand: every time we define a new problem-description language, we need a new planning algorithm to process the language. . . . The idea is that the algorithm can be designed to process the restricted language more efficiently than a resolution theorem prover. [Russell & Norvig, 1995, page 342]

The aim of the present paper is to demonstrate that a good theoretical solution can indeed co-exist with a good practical solution, through the provision of a logical account of partial order and hierarchical planning in the spirit of Green’s work. However, where Green’s account was based on the formalism of the situation calculus [McCarthy & Hayes, 1969], the present paper adopts the event calculus [Kowalski & Sergot, 1986], [Shanahan, 1997a]. Furthermore, while Green regarded planning as a deductive process, planning with the event calculus is most naturally considered as an abductive process. When event calculus formulae are submitted to a suitably tailored resolution based abductive theorem prover, the result is a purely logical planning system whose computations mirror closely those of a hand-coded planning algorithm.

The developments reported in this paper are part of an ongoing research programme in Cognitive Robotics [Lespérance, *et al.*, 1994]. Researchers in Cognitive Robotics aim to design and build robots whose architectures are based on the manipulation of sentences of logic, with a view to endowing them with high-level cognitive skills such as the ability to reason about their own goals and knowledge and the goals and knowledge of others. The ideals of this research are close to those of the designers of Shakey in the late Sixties and early Seventies [Nilsson, 1984], and Cognitive Robotics can be thought of as a revival of a research programme largely abandoned after the Shakey project.

But hindsight has exposed a number of conceptual flaws in the style of robotics promoted by the Shakey project. In particular, Shakey was obliged to cease any activity in the world while it planned, and once it had found a plan it would commit to the execution of that plan come what may.[1] A major concern here, as in mainstream robotics and planning, is to avoid these problems by smoothly interleaving planning, sensing and acting. As we shall see, this is one motivation for the study of hierarchical planning. Theorem proving approaches to planning usually employ regression, which generates actions in reverse order, which is useless if we want to start executing a plan before we've finished constructing it. Hierarchical decomposition, on the other hand, can be adjusted to generate actions in progression order (first action first).

1 A Circumscriptive Event Calculus

The formalism for reasoning about action used in this paper is derived originally from Kowalski and Sergot's event calculus [Kowalski & Sergot, 1986], but is based on many-sorted first-order predicate calculus augmented with circumscription [Shanahan, 1997a].[2] This section presents the bare outlines of the formalism. An example of the use of the formalism, which should make things clearer to those unfamiliar with it, appears in the next section. For a more thorough treatment, consult [Shanahan, 1997a].

Table 1 presents the essentials of the language of the calculus, which includes sorts for fluents, actions (events), and time points.

Formula	**Meaning**
Initiates(α,β,τ)	Fluent β holds after action α at time τ
Terminates(α,β,τ)	Fluent β does not hold after action α at time τ
Releases(α,β,τ)	Fluent β is not subject to the common sense law of inertia after action α at time τ
$\text{Initially}_P(\beta)$	Fluent β holds from time 0
$\text{Initially}_N(\beta)$	Fluent β does not hold from time 0
Happens(α,τ_1,τ_2)	Action α starts at time τ_1 and ends at time τ_2
HoldsAt(β,τ)	Fluent β holds at time τ

Table 1: The Language of the Event Calculus

We have the following axioms, whose conjunction is denoted EC.[3]

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

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

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

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

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

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

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

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

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

The frame problem is overcome through circumscription. Given a conjunction Σ of Initiates, Terminates, and Releases formulae describing the effects of actions (a *domain description*), 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 all the cases we're interested in, Σ and Δ will be conjunctions of Horn clauses, and the circumscriptions will reduce to predicate completions. This result will come in handy when we come to implement the event calculus as a logic program.

2 Planning as Abduction

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,

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

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_0 of Initially formulae describing the initial

[1] This isn't quite true. If plan execution went extremely badly, Shakey would eventually abandon its current plan and re-plan from scratch.

[2] Here we will confine our attention to relatively simple domains, but [Shanahan, 1997a] shows how the calculus can be used to handle domain constraints, continuous change, and non-deterministic effects. Indeed, the planner described in this paper can handle many types of domain constraint without further modification.

[3] Variables begin with lower-case letters, while function and predicate symbols begin with upper-case letters. All variables are universally quantified with maximum possible scope unless otherwise indicated.

situation, a *plan* is a consistent conjunction Δ of Happens and temporal ordering formulae such that,

$$\text{CIRC}[\Sigma \,;\, \text{Initiates, Terminates, Releases}] \wedge \text{CIRC}[\Delta_0 \wedge \Delta \,;\, \text{Happens}] \wedge \text{EC} \wedge \Omega \vDash \Gamma.$$

As suggested by the title of Levesque's Green-inspired paper, "What is planning in the presence of sensing?" [Levesque, 1996], logical characterisations such as this aim to settle the question of the underlying nature of one or other type of planning. Levesque's answer, echoing Green's 1969 paper, is based on the situation calculus. In the situation calculus, a plan is expressed using the Result function, which maps an action and a situation onto a new situation. The Result function does not facilitate the representation of narratives of events whose order is incompletely known.[4] By contrast, since the narrative of actions described by Δ above doesn't have to be totally ordered, the event calculus seems a natural candidate for answering the question "What is partial order planning?".

As an example, let's formalise the shopping trip domain from [Russell & Norvig, 1995]. The domain comprises just two actions and two fluents. The term Go(x) denotes the action of going to x, and the term Buy(x) denotes the action of buying x. The fluent At(x) holds if the agent is at location x, and the fluent Have(x) holds if the agent possesses item x. Let Σ be the conjunction of the following Initiates, Terminates and sundry formulae.

Initiates(Go(x),At(x),t)

Terminates(Go(x),At(y),t) $\leftarrow$ x $\neq$ y

Initiates(Buy(x),Have(x),t) $\leftarrow$
HoldsAt(At(y),t) $\wedge$ Sells(y,x)

Sells(DIYShop,Drill)

Sells(Supermarket,Banana)

Sells(Supermarket,Milk)

Let Ω be the conjunction of the following uniqueness-of-names axioms.

UNA[Go, Buy]

UNA[At, Have]

Our desired goal state is to have a banana, some milk, and a drill. Let Γ be the following conjunction of HoldsAt formulae.

HoldsAt(Have(Banana),T) $\wedge$ HoldsAt(Have(Milk),T) $\wedge$
HoldsAt(Have(Drill),T)

Let Δ be the conjunction of the following Happens and temporal ordering formulae.

Happens(Go(Supermarket),T0)

Happens(Buy(Banana),T1)

Happens(Buy(Milk),T2)

Happens(Go(DIYShop),T3)

Happens(Buy(Drill),T4)

T0 < T1	T0 < T2
T1 < T3	T2 < T3
T3 < T4	T4 < T

Note that Δ is not committed to any particular ordering of the Buy(Banana) and Buy(Milk) actions. As we would expect, according to the definition above, Δ is indeed a plan for Γ. In other words,we have,

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

The provision of a logical characterisation of the planning task is all very well. But for a complete picture, and to address the issues raised in the Russell and Norvig quote in the introduction, we need to look at computational matters. These are the focus of the sequel.

3 Partial Order Planning = Event Calculus + Abduction

The title of this section deliberately echoes Kowalski's slogan "Algorithm = Logic + Control" [Kowalski, 1974]. The aim of the section is to sketch the use of logic programming techniques to render the previous section's logical specification of partial order planning into a practical implementation.[5] The basis of this implementation will be a resolution based abductive theorem prover, coded as a Prolog meta-interpreter. This theorem prover is tailored for the event calculus by compiling the event calculus axioms into the meta-level, resulting in an efficient implementation.

As pointed out by [Missiaen, *et al.*, 1995], the event calculus axioms, in particular (EC1) to (EC6), can be likened to Chapman's "modal truth criterion" (but stripped of the modalities) [Chapman, 1987]. The logic programming approach to planning advocated in this paper can be thought of as *directly executing* the modal truth criterion. The event calculus Initiates, Terminates and Releases formulae that constitute a purely logical description of the effects of actions in a particular domain are used directly as the domain description in the implemented planner.

Many of the computational concepts central to the literature on partial order planning, such as threats, protected links, promotions and demotions [Chapman, 1987], [Penberthy & Weld, 1992], turn out to have direct counterparts in the theorem proving process. It's interesting to note that these features of the logic programming implementation weren't designed in. Rather, they are naturally arising features of the theorem prover's search for a proof. So our attempt to provide a mathematically respectable answer to the question "What is partial order planning?" inadvertantly offers similar answers to questions like "What are protected links?". To see all this we need to delve into the details of the meta-interpreter. In what follows, I will assume some

[4] But see [Miller & Shanahan, 1994].

[5] A full listing of the Prolog implementation is given in the (electronic) appendix.

knowledge of logic programming concepts and terminology.

3.1 An Abductive Meta-Interpreter for the Event Calculus

Meta-interpreters are a standard part of the logic programmer's toolkit [Sterling & Shapiro, 1986, Chapter 19]. For example, the following "vanilla" meta-interpreter, when executed by Prolog, will mimic Prolog's own execution strategy.[6]

```
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 [λ_1, ..., λ_n], then the formula axiom(λ_0,Π) holds if there is a clause of the following form in the object-level program.

λ_0 :- λ_1, ..., λ_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).
```

[6] Throughout the paper, I use standard Edinburgh syntax for Prolog. Variables begin with upper-case letters, while predicate and function symbols with lower-case letters, which is the opposite convention to that used for predicate calculus.

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, although we resolve on `initiates` immediately, we postpone further work on the sub-goals of `initiates` 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 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).                                  (A1)
abdemo([G|Gs],R1,R3,N) :-                 (A2)
  abducible(G),
  abdemo_nafs(N,[G|R1],R2),
  abdemo(Gs,R2,R3,N).
```

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

```
abdemo([not(G)|Gs],R1,R3,N) :-            (A4)
  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 (A2)). If `N` = $[\gamma_{1,1} ... \gamma_{1,n_1}] ... [\gamma_{m,1} ... \gamma_{m,n_m}]$ is such a list, then its meaning, assuming a completion semantics for our object-level logic program, is,

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

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, $\neg (\gamma_1 \wedge ... \wedge \gamma_n)$, we have to show that, for every object-level clause λ `:-` $\lambda_1 ... \lambda_m$ which resolves with γ_1, $\neg (\lambda_1 \wedge ... \wedge \lambda_m, \gamma_2 \wedge ... \wedge \gamma_n)$. If no clause resolves with γ_1 then, under a completion semantics, $\neg \gamma_1$ follows, and therefore so does $\neg (\gamma_1 \wedge ... \wedge \gamma_n)$.

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 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.[7]

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, 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],              (A5)
  R1,R4,N) :-
    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. Notice that, since the sub-goals of `initiates` are solved abductively, actions with context-dependent effects are handled correctly, unlike the implementation described in [Missiaen, *et al.*, 1995].

Further details of the implementation are relegated to the (electronic) appendix, which presents a full program listing. But with this sketch, we're already in a position to compare the behaviour of an abductive theorem prover applied to the event calculus to that of a hand-coded partial order planning algorithm.

[7] In [Shanahan, 1989] and [Missiaen, *et al.*, 1995], this problem is tackled via the use of nested negations-as-failure at the object level. The approach of the present paper is more principled.

3.2 Protected Links, Threats, Promotions and Demotions

The algorithm below, which is very similar to UCPOP [Penberthy & Weld, 1992], illustrates the style of algorithm commonly found in the literature on partial order planning. It constructs a partially ordered plan given a goal list. A goal list is a list of pairs ⟨F,T⟩ where F is a fluent and T is a time point. A plan is a list of pairs ⟨A,T⟩ where A is an action (more properly called an operator in planning terminology) and T is a time point.

The key idea in the algorithm is the maintenance of a list of *protected links*. This is a list of triples ⟨T1,F,T2⟩, where T1 and T2 are time points and F is a fluent. The purpose of this list is to ensure that, once a goal has been achieved by the addition of a suitable action to the plan, that goal isn't "clobbered" by a subsequent addition to the plan. Accordingly, each addition to the plan is followed by a check to see whether it constitutes a *threat* to any protected link. An action ⟨A,T1⟩ threatens a protected link ⟨T2,F,T3⟩ if the ordering constraint T2 < T1 < T3 is consistent with the plan and one of the effects of A is to make F false. By *promoting* or *demoting* the new action, in other words by constraining its time of occurrence to fall either before T2 or after T3, we eliminate the threat.

```
1  while goal list non-empty
2    choose a goal <F1,T1> from goal list
3    choose an action <A,T2> whose effects
       include F1
4    for each precondition F2 of A add <F2,T2>
       to goal list
5    add <A,T2> to plan
6    add T2 < T1 to plan
7    add <T2,F1,T1> to protected links
8    for each <A,T3> in plan that threatens
       some <T4,F3,T5> in protected links
9      choose either
10       promotion: add T3 < T4 to plan
11       demotion: add T5 < T3
12   end for
13 end while
```

Since the algorithm is non-deterministic, it has to be combined with a suitable search strategy. With some minor modifications, the algorithm can be turned into UCPOP, which is both sound and complete, assuming a breadth-first or iterative deepening search strategy [Penberthy & Weld, 1992]. Unlike the above algorithm, but like the abductive meta-interpreter of the last section, UCPOP can also handle actions with context-dependent effects.

The close correspondence between the behaviour of this algorithm and that of the abductive theorem prover of the previous section can be established by inspection. In particular, consider Clause (A5). Line 3 of the algorithm (choosing an action) corresponds to the first sub-goal of (A5) (resolving on `initiates`). Line 4 (adding new preconditions to the goal list) corresponds to the fourth sub-goal. The effect of Lines 5 and 6 (adding the new action to the plan) is achieved in (A5) by the second sub-goal. Line 7 (adding the new protected link) and the for loop of Lines 8 to 12 are matched by the third sub-goal of (A5), which adds a new `clipped` literal to the list of negations. Promotion and demotion (Lines 11 and 12) are achieved in the theorem prover by `abdemo_nafs` which, as explained in the previous section, will add further `before` literals to the residue if necessary.

Like the non-deterministic hand-coded algorithm, the search space defined by Clauses (A1) to (A5) can be explored with a variety of strategies. If executed by Prolog, a depth-first search strategy would result, but a breadth-first or iterative deepening strategy is also possible.

To summarise, the concepts of a protected link, of a threat, and of promotion and demotion, rather than being special to partial order planning, turn out to be instances of general concepts in theorem proving when applied to general purpose axioms for representing the effects of actions. In particular,

- A protected link is a negated `clipped` goal which, like any negated goal in abduction with negation-as-failure, is preserved for subsequent checking when new literals are added to the residue,
- A threat is an addition to the residue which, without further additions, would undermine the proof of a previously solved negated `clipped` goal.
- Promotion and demotion are additions to the residue which preserve the proof of a previously solved negated `clipped` goal.

4 Soundness, Completeness and Complexity

The aim of this section is to establish soundness and completeness results for the abductive implementation of Section 3.1 for a certain class of planning problems as defined by the logical characterisation of Section 2, and to demonstrate formally that the complexity of the implementation is comparable to that of a hand-coded partial order planning algorithm such as the one presented in Section 3.2.

The soundness result is particularly straightforward to obtain because the trace of a successful program execution is itself a proof that the goals follow from the returned plan.

TO BE COMPLETED

5 Hierarchical Planning

It's a surprisingly straightforward matter to extend the foregoing logical treatment of partial order planning to planning via hierarchical decomposition, as first described in detail by Sacerdoti [1974]. 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.

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, and the term GoThrough(d) denotes the action of going through door d. The fluent Got(p) holds if the robot is carrying the package p, and the fluent In(x,r) holds if object x is in room r. The formula Connects(d,r1,r2) represents that door d connects rooms r1 and r2.

```
Initiates(Pickup(p),Got(p),t) ← 
    p ≠ Robot ∧ HoldsAt(In(Robot,r),t) ∧ 
        HoldsAt(In(p,r),t)

Releases(Pickup(p),In(p,r),t) ← 
    p ≠ Robot ∧ HoldsAt(In(Robot,r),t) ∧ 
        HoldsAt(In(p,r),t)

Initiates(PutDown(p),In(p,r),t) ← 
    p ≠ Robot ∧ HoldsAt(Got(p),t) ∧ 
        HoldsAt(In(Robot,r),t)

Initiates(GoThrough(d),In(Robot,r1),t) ← 
    HoldsAt(In(Robot,r2),t) ∧ Connects(d,r2,r1)

Terminates(GoThrough(d),In(Robot,r),t) ← 
    HoldsAt(In(Robot,r),t)
```

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 number of sub-actions: two GoToRoom actions, a Pickup action and a PutDown action. A GoToRoom action is itself a compound action, to be defined shortly.

```
Happens(ShiftPack(p,r1),t1,t6) ← 
    HoldsAt(In(p,r2),t1) ∧ 
        Happens(GoToRoom(r2),t1,t2) ∧ t2 < t3 ∧ 
            Happens(Pickup(p),t3) ∧ t3 < t4 ∧ 
                Happens(GoToRoom(r1),t4,t5) ∧ 
                    t5 < t6 ∧ Happens(PutDown(p),t6)

Initiates(ShiftPack(p,r),In(p,r),t)
```

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 a GoToRoom action.

```
Happens(GoToRoom(r),t,t) ← HoldsAt(In(Robot,r),t)

Happens(GoToRoom(r1),t1,t3) ← 
    HoldsAt(In(Robot,r2),t1) ∧ Connects(d,r2,r3) ∧ 
        Happens(GoThrough(d),t1) ∧ t1 < t2 ∧ 
            Happens(GoToRoom(r1),t2,t3)

Initiates(GoToRoom(r),In(Robot,r),t)
```

This illustrates 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], a programming language built on a different logic-based action formalism, namely the situation calculus. Note, however, that we can freely mix direct programming with planning from first principles.

Once again, the effects of the compound action should follow from the effects of its components. This property is made more precise below.

Let Ω denote the conjunction of the following uniqueness-of-names axioms.

```
UNA[Pickup, PutDown, GoThrough, 
    ShiftPack, GoToRoom]

UNA[Got, In]
```

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.

Now let's take a look at a particular mail delivery task. Let Σ_P be the conjunction of the above Initiates, Terminates and Releases formulae for primitive actions, and let Σ_C be the conjunction of the above Initiates formulae for compound actions. Let Δ_C be the conjunction of the above compound event definitions.

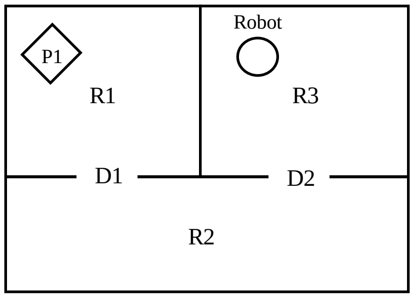

Figure 1: A Mail Delivery Domain

The conjunction Φ of the following Connects formulae represents the layout of rooms illustrated in Figure 1.

```
Connects(D1,R1,R2)

Connects(D1,R2,R1)

Connects(D2,R2,R3)

Connects(D2,R3,R2)
```

Let Δ_0 denote the conjunction of the following formulae representing the initial situation depicted in Figure 1.

```
Initially(In(Robot,R3))

Initially(In(P1,R1))
```

Let Γ denote the following HoldsAt formula, which is our goal state.

```
HoldsAt(In(P1,R2),T)
```

Consider the following narrative of actions Δ_p.

Happens(GoThrough(D2),T0)

Happens(GoThrough(D1),T1)

Happens(Pickup(P1),T2)

Happens(GoThrough(D1),T3)

Happens(PutDown(P1),T4)

$T0 < T1 \qquad T1 < T2$

$T2 < T3 \qquad T3 < T4$

$T4 < T$

Now we have, for example,

$$\mathrm{CIRC}[\Sigma_p \wedge \Sigma_c \text{ ; Initiates, Terminates, Releases}] \wedge \mathrm{CIRC}[\Delta_0 \wedge \Delta_p \wedge \Delta_c \text{ ; Happens}] \wedge \mathrm{EC} \wedge \Omega \wedge \Phi \vDash \mathrm{Happens(ShiftPack(P1,R2),T0,T4)}.$$

We also have,

$$\mathrm{CIRC}[\Sigma_p \wedge \Sigma_c \text{ ; Initiates, Terminates, Releases}] \wedge \mathrm{CIRC}[\Delta_0 \wedge \Delta_p \wedge \Delta_c \text{ ; Happens}] \wedge \mathrm{EC} \wedge \Omega \wedge \Phi \vDash \Gamma.$$

So Δ_p constitutes a plan. Furthermore, we have,

$$\mathrm{CIRC}[\Sigma_p \text{ ; Initiates, Terminates, Releases}] \wedge \mathrm{CIRC}[\Delta_0 \wedge \Delta_p \text{ ; Happens}] \wedge \mathrm{EC} \wedge \Omega \wedge \Phi \vDash \Gamma.$$

So Δ_p constitutes a plan in the context of only the primitive actions. In general, if we let Δ_p be any narrative description comprising only primitive actions and Φ be any conjunction of Connects formulae, we have the following theorem. For any fluent β and time point τ, HoldsAt(β,τ) follows from,

$$\mathrm{CIRC}[\Sigma_p \wedge \Sigma_c \text{ ; Initiates, Terminates, Releases}] \wedge \mathrm{CIRC}[\Delta_p \wedge \Delta_c \text{ ; Happens}] \wedge \mathrm{EC} \wedge \Omega \wedge \Phi$$

if and only if it follows from,

$$\mathrm{CIRC}[\Sigma_p \text{ ; Initiates, Terminates, Releases}] \wedge \mathrm{CIRC}[\Delta_p \text{ ; Happens}] \wedge \mathrm{EC} \wedge \Omega \wedge \Phi.$$

We should expect such a property to follow from any correctly formulated domain description involving compound actions, since the (chief) purpose of compound actions is to adjust the computation by cutting down on search, and not to increase the set of consequences of the theory. However, the inclusion of compound actions in the logical account gives meaning to partially decomposed plans, which are the intermediate steps in this computation. This is an example of a logical innovation which is highly suggestive of the form the computation should take, and this in turn suggests that the absolute separation of computational and representational issues that's popular in certain quarters isn't always appropriate.

In general we will require our planner to find fully decomposed plans, although it's extremely useful to be able to suspend the planning process before a fully decomposed plan has been found, and still to 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]. Furthermore, the use of hierarchical decomposition facilitiates the generation of plans in progression order (first action first), as opposed to the regression order (last action first) usually found in logic-based planners. The generation of plans in regression order would rule out the possibility of suspending planning in mid-execution and still receiving useful results.[8]

This brings us to the issue of implementation, and one of the most striking results of this paper. What modifications are required to the abductive meta-interpreter of Section 3 to enable it to perform hierarchical decomposition? The answer is remarkable. None whatsoever. When presented with compound event definitions of the above form, it automatically performs hierarchical decomposition. Whenever a `happens` goal is reached for a compound action, its resolution yields further `happens` sub-goals, and this process continues until primitive actions are reached, which are added to the residue.[9]

Section 3 was entitled "Partial Order Planning = Event Calculus + Abduction". Now we've arrived at another instantiation of Kowalski's equation. Hierarchical planning = event calculus with compound events + abduction. Using the methodology of this paper, all we have to do to obtain a hierarchical planner from a partial order planner is represent compound actions in the obvious way.

Concluding Remarks

This paper continues a line of work on event calculus planning begun in [Eshghi, 1988]. Eshghi's techniques were simplified (and applied to temporal explanation) in [Shanahan, 1989]. But neither of these papers described a practical planner. The first usable event calculus planner was developed in Belgium by Missiaen, *et al.* [1995]. Recently, another abductive event calculus planner has been developed at DFKI in Germany [Jung, *et al.*, 1996]. All of these planners are based on similar ideas to those presented in this paper: all use abductive logic programming techniques to generate plans using a similar style of representation via `initiates`, `terminates` and `happens` predicates.

The present paper goes beyond the work of its predecessors in several ways. First, it tackles the issue of hierarchical

8 This observation prompts Kowalski [1995] to abandon traditional effect axioms altogether. Using compound actions, we can preserve traditional event calculus style effect axioms, along with Eshghi's abductive characterisation of event calculus planning with its strong logical relationship between goals and plans.

9 Furthermore, if we make Connects abducible in the mail delivery example instead of Happens, we can use exactly the same meta-interpreter to determine room connectivity given a narrative of actions and a conjunction of formulae of the form HoldsAt(In(Robot,ρ),τ). This further underlines the generic nature of the techniques being applied here.

planning. Second, the event calculus formalism used is not just a logic program, but is specified in first-order predicate calculus augmented with circumscription. Third, the paper exposes close correspondences with existing planning algorithms. Since the planner is simply the result of applying general purpose theorem proving techniques to a general purpose action formalism, it can be argued that this illuminates the nature of several commonly deployed concepts in the planning literature. Fourth, unlike the planners in [Missiaen, *et al.*, 1995] and [Jung, *et al.*, 1996], the planner of the present paper can handle actions with context-dependent effects. Finally, since it uses abduction to solve `initiates` and `terminates` goals, the planner is both sound and complete, and performs correctly on a number of potentially anomalous examples described in [Missiaen, *et al.*, 1995].

In a series of papers related to this one (culminating in [Shanahan, 1997b]), an abductive characterisation of sensor data assimilation is provided. According to this account, sensor data is explained by hypothesising the existence of suitably shaped objects in appropriate locations. The role of the present paper is to supply a complementary account of planning. The present aim of the EPSRC Cognitive Robotics project at Queen Mary and Westfield College is to integrate these accounts into a single system implemented and deployed on our small fleet of miniature Khepera robots.

Acknowledgments

Thanks to Rob Miller. This work was carried out as part of the EPSRC funded project GR/L20023 “Cognitive Robotics”.

References

[Chapman, 1987] D.Chapman, Planning for Conjunctive Goals, *Artificial Intelligence*, vol. 32 (1987), pp. 333–377.

[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.

[Jung, *et al.*, 1996] C.G.Jung, K.Fischer and A.Burt, *Multi-Agent Planning Using an Abductive Event Calculus*, DFKI Report RR-96-04 (1996), DFKI, Germany.

[Kowalski, 1979] R.A.Kowalski, Algorithm = Logic + Control, Communications of the ACM, vol. 22, pp. 424–436.

[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.

[Kowalski & Sergot, 1986] R.A.Kowalski and M.J.Sergot, A Logic-Based Calculus of Events, *New Generation Computing*, vol 4 (1986), pp. 67–95.

[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.

[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* (1997), to appear.

[McCarthy & Hayes, 1969] J.McCarthy and P.J.Hayes, Some Philosophical Problems from the Standpoint of Artificial Intelligence, in *Machine Intelligence 4*, ed. D.Michie and B.Meltzer, Edinburgh University Press (1969), pp. 463–502.

[Miller & Shanahan, 1994] R.S.Miller and M.P.Shanahan, Narratives in the Situation Calculus, *The Journal of Logic and Computation*, vol. 4, no. 5 (1994), pp. 513–530.

[Missiaen, *et al.*, 1995] L.Missiaen, M.Bruynooghe and M.Denecker, CHICA, A Planning System Based on Event Calculus, *The Journal of Logic and Computation*, vol. 5, no. 5 (1995), pp. 579–602.

[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.

[Sacerdoti, 1974] E.D.Sacerdoti, Planning in a Hierarchy of Abstraction Spaces, *Artificial Intelligence*, vol. 5 (1974), pp. 115–135.

[Shanahan, 1989] M.P.Shanahan, Prediction Is Deduction but Explanation Is Abduction, *Proceedings IJCAI 89*, pp. 1055–1060.

[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*, to appear.

[Sterling & Shapiro, 1986] L.Sterling and E.Shapiro, *The Art of Prolog*, MIT Press (1986).

Appendix

A fully commented program listing of the latest version of the planner is available electronically from http://www.dcs.qmw.ac.uk/~mps/planner.txt. The planner is written in LPA MacProlog 32, but should be easy to port to other Prolog systems. Stripped of comments, the current version of the planner is only 150 lines long.