An Abductive Event Calculus Planner

Murray Shanahan

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

May 1998
Revised April 1999
2nd revision September 1999

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, the paper shows 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. Soundness and completeness results for this logic programming implementation are given. Finally the paper shows that, if we extend the event calculus in a natural way to accommodate compound actions, then using the same abductive theorem proving techniques we can obtain a hierarchical planner.

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 [Green, 1969]. 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, 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 sound and complete purely logical planning system whose computations mirror closely those of a hand-coded planning algorithm.

The paper is organised as follows. Section 1 presents the event calculus using full first-order predicate calculus with circumscription. Section 2 presents a logical definition of event calculus planning. Section 3 describes a number of techniques for rendering this specification into a logic programming implementation via an abductive meta-interpreter, and draws attention to the correspondences to existing partial-order planning algorithms. Section 4 offers soundness and completeness results for a simple version of the planner. Section 5 shows how the preceding material can be extended to cover hierarchical planning. Finally Section 7 addresses efficiency issues and presents some benchmark results. A shorter version of this paper was presented in [Shanahan, 1997b].

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]. This section presents the bare outlines of the formalism.[1] 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.

We have the following axioms, whose conjunction is denoted EC.[2]

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

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

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

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

[1] [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.

[2] 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.

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

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

Happens(a,t1,t2) $\rightarrow$ t1 $\leq$ t2 (EC7)

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

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

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
Initially_P(β)	Fluent β holds from time 0
Initially_N(β)	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
Clipped(τ1,β,τ2)	Fluent β is terminated between times τ1 and τ2
Declipped(τ1,β,τ2)	Fluent β is initiated between times τ1 and τ2

Table 1: The Language of the Event Calculus

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_P, Initially_N, 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,

CIRC[Σ ; Initiates, Terminates, Releases] $\wedge$ CIRC[Δ ; Happens] $\wedge$ EC $\wedge$ Ω.

By minimising Initiates, Terminates and Releases we assume that actions have no unexpected effects, and by minimising Happens we assume that there are no unexpected event occurrences. In most of the cases we're interested in, Σ and Δ will be conjunctions of Horn clauses, and the circumscriptions will reduce to predicate completions [Lifschitz, 1994]. 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,

CIRC[Σ ; Initiates, Terminates, Releases] $\wedge$ CIRC[Δ ; Happens] $\wedge$ EC $\wedge$ $\Omega \vDash \Gamma$.

Conversely, as first pointed out by Eshghi [1988], planning in the event calculus can be considered as an abductive task. In terms of the circumscriptive event calculus, this task can be precisely characterised as follows.

Definition 2.1. A *domain description* is a finite conjunction of formulae of the form,

$\text{Initiates}(\alpha,\beta,t) \leftarrow \Pi$

or,

$\text{Terminates}(\alpha,\beta,t) \leftarrow \Pi$

or,

$\text{Releases}(\alpha,\beta,t) \leftarrow \Pi$

where Π is of the form,

$(\neg)\ \text{HoldsAt}(\beta_1,t) \wedge \ldots \wedge (\neg)\ \text{HoldsAt}(\beta_n,t)$,

α is a ground action term, and β and β_1 to β_n are ground fluent terms. □

Definition 2.2. An *initial situation* is a finite conjunction of formulae of the form,

$\text{Initially}_N(\beta)$

or,

$\text{Initially}_P(\beta)$

where β is a ground fluent term, in which each fluent name occurs at most once. □

Definition 2.3. A *goal* is a finite conjunction of formulae of the form,

$(\neg)\ \text{HoldsAt}(\beta,\tau)$

where β is a ground fluent term, and τ is a ground time point term. □

Definition 2.4. A *narrative* is a finite conjunction of formulae of the form,

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

or,

$\tau_1 < \tau_2$

where α is a ground action term, and τ_1 and τ_2 are time points. □

Definition 2.5. Let Γ be a goal, let Σ be a domain description, let Δ_0 be an initial situation, and let Ω be the conjunction of a pair of uniqueness-of-names axioms for the actions and fluents mentioned in Σ. A *plan* for Γ is a narrative Δ such that,

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

where,

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

is consistent. □

The consistency condition is guaranteed if the domain description does not permit a fluent to be both initiated and terminated at the same time.

Definition 2.6. A domain description Σ is *conflict free* if, for every pair of formulae in Σ of the form,

$\text{Initiates}(\alpha,\beta,t) \leftarrow \Pi_1$

and,

$\text{Terminates}(\alpha,\beta,t) \leftarrow \Pi_2$,

we have,

$\models \neg [\Pi_2 \wedge \Pi_1]$ □

Lemma 2.7. Let Σ be a conflict free domain description, let Δ_0 be an initial situation, let Δ be a totally ordered narrative, and let Ω be the conjunction of a pair of uniqueness-of-names axioms for the actions and fluents mentioned in Σ. The formula,

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

is consistent.

Proof. (Sketch) To construct a model of the formula, we have to assign a truth value to every fluent for every time point. This can be done in a STRIPS like fashion, starting at time zero and maintaining the set of fluents that hold after each action in Δ is carried out. We begin by selecting an arbitrary set of fluents compatible with Δ_0. These hold in the interval up to the first action in Δ. We then consider this first action. The formulae in Σ effectively permit us to form an addlist (fluents that are initiated) and a deletelist (fluents that are terminated or released) for the action. The conflict free condition ensures that these lists are mutually exclusive. Using these lists, we can construct the set of the fluents that hold in the interval between the first and second actions. This process continues until the final action in Δ. □

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. 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 three 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, the fluent Have(x) holds if the agent possesses item x, and the fluent Sells(x,y) holds if shop x sells item y. Let Σ be the conjunction of the following Initiates and Terminates formulae.

$$\text{Initiates(Go(x),At(x),t)}$$

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

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

Note that there is no distinction, in this formalism, between preconditions and contextual conditions. Both are expressed using HoldsAt. Let Δ_0 be the conjunction of the following formulae describing the initial situation.

$$\text{Initially}_P\text{(Sells(DIYShop,Drill))}$$

$$\text{Initially}_P\text{(Sells(Supermarket,Banana))}$$

$$\text{Initially}_P\text{(Sells(Supermarket,Milk))}$$

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

$$\text{UNA[Go, Buy]} \qquad \text{UNA[At, Have, Sells]}$$

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

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

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,

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

The above definition of a plan can easily be extended to encompass domains that include *domain constraints* or *state constraints*, which give rise to actions with indirect effects.

Definition 2.8. A *state constraint* is a formula of the form,

$$\text{HoldsAt}(\beta,t) \leftarrow (\neg)\ \text{HoldsAt}(\beta_1,t) \wedge \ldots \wedge (\neg)\ \text{HoldsAt}(\beta_n,t)$$

where β and β_1 to β_n are ground fluent terms. □

The circumscriptive solution to the frame problem adopted here accommodates the inclusion of state constraints (see [Shanahan, 1997a]), so long as they are conjoined to the theory outside the scope of any circumscription. This gives rise to the following definition.

Definition 2.9. Let Γ be a goal, let Σ be a domain description, let Δ_0 be an initial situation, let Ω be the conjunction of a pair of uniqueness-of-names axioms for the actions and fluents mentioned in Σ, and let Ψ be a finite conjunction of state constraints. A *plan* for Γ is a narrative Δ such that,

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

Here's an example involving state constraints. The goal will be to make a chemical plant safe, where plant safety is defined through the following state constraint.

$$\text{HoldsAt(PlantSafe,t)} \leftarrow$$
$$\text{HoldsAt(TankEmpty,t)} \wedge \text{HoldsAt(TemperatureLow,t)}$$

The tank can be emptied by draining it, as long as the pressure is normal. The plant's temperature can be brought down by cooling the tank.

$$\text{Initiates(DrainTank,TankEmpty,t)} \leftarrow \text{HoldsAt(PressureNormal,t)}$$
$$\text{Initiates(CoolTank,TemperatureLow,t)}$$

The pressure is normal if the valve is open or if the boiler is off. So we have two more state constraints.

$$\text{HoldsAt(PressureNormal,t)} \leftarrow \text{HoldsAt(ValveOpen,t)}$$
$$\text{HoldsAt(PressureNormal,t)} \leftarrow \text{HoldsAt(BoilerOff,t)}$$

We have actions for opening the valve and turning off the boiler.

$$\text{Initiates(OpenValve,ValveOpen,t)}$$
$$\text{Initiates(TurnOffBoiler,BoilerOff,t)}$$

Finally we have the requisite uniqueness-of-names axioms.

UNA[PlantSafe, TankEmpty, TemperatureLow,
PressureNormal, ValveOpen, BoilerOff]

UNA[DrainTank, CoolTank, OpenValve, TurnBoilerOff]

Now suppose we have the following initial situation.

$\text{Initially}_N(\text{PlantSafe})$ $\text{Initially}_N(\text{TankEmpty})$

$\text{Initially}_N(\text{TemperatureLow})$ $\text{Initially}_N(\text{PressureNormal})$

$\text{Initially}_N(\text{ValveOpen})$ $\text{Initially}_N(\text{BoilerOff})$

Let Σ be the conjunction of the above Initiates formulae, let Δ_0 be the above initial situation, let Ω be the conjunction of the two uniqueness-of-names axioms, and let Ψ be the conjunction of the three state constraints. Now let Γ be,

HoldsAt(PlantSafe,T).

Finally let Δ be the conjunction of the following formulae.

Happens(OpenValve,T0)

Happens(DrainTank,T1)

Happens(CoolTank,T2)

$T0 < T1$

$T1 < T2$

$T2 < T$

Now we have,

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

In other words, Δ is a plan for Γ.

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 next section.

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 outline the use of logic programming techniques which can be used to render the previous section's logical specification of partial-order planning into a practical implementation. 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.

Eshghi [1988] was the first author to show how abduction could be used to solve event calculus planning problems. (The original event calculus of Kowalski and Sergot [1986], which was the inspiration for the present formalism, was also expressed a logic program. However, it cannot be executed directly as a Prolog program.) Other early approaches along these lines include [Shanahan, 1989], [Missiaen, 1991], and [Missiaen, *et al.*, 1995].

As reported in [Missiaen et al. 95], these systems sometimes generated erroneous partial plans, a problem that was solved in the system of [Denecker, *et al.*, 1992] by integrating abduction with constraint solving techniques. Denecker *et al.* also showed how such a system could be used for reasoning in the presence of uncertainty about actions. Later approaches that inorporated related ideas were presented in [Chleq, 1996] and [Jung, 1996]. The latter extends the planning methods of [Denecker, *et al.*, 1992] in the context of an object oriented formalism. More recently, Kakas, *et al.* [1998] have integrated an abductive solver and a finite domain CLP solver and applied them to planning.

As pointed out by [Missiaen, *et al.*, 1995], the event calculus axioms 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?" inadvertently 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 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. For example, the following "vanilla" meta-interpreter, when executed by Prolog, will mimic Prolog's own execution strategy.[3]

```
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).
```

[3] 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.

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).
```

The Prolog execution of this meta-level clause 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. Moreover, even if there were no looping problem, tackling `initiates` before `happens` results in a smaller search space than would be obtained if these sub-goals were tackled in the opposite order, since event occurrences irrelevant to the fluent `F` are not considered. (A discussion of this and related efficiency issues with respect to the event calculus can be found in [Chittaro & Montanari, 1996].)

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 $\neg$ HoldsAt(f,t) with `holds_at(neg(F),T)`. (Using the `neg` function, we retain some of the predicate calculus formalism's ability to handle incomplete information about which fluents hold. This would be lost if we used negation-as-failure.) 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 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 = $[\gamma_{1,1} \ldots \gamma_{1,n_1}] \ldots [\gamma_{m,1} \ldots \gamma_{m,n_m}]$] 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, allowing partially ordered narratives of events to be represented. Accordingly, when the meta-interpreter encounters a negated goal of the form 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.

This approach to the `before` predicate correctly handles examples that have troubled earlier abductive event calculus planners. Consider the following example, taken from [Missiaen, *et al.*, 1995].

```
initially(r).
initiates(e1,p,T).
initiates(e2,q,T).
terminates(e1,r,T) :- holds_at(q,T).
terminates(e2,r,T) :- holds_at(p,T).
```

Given the goal,

```
[holds_at(p,t),holds_at(q,t),holds_at(r,t)]
```

both the planner of [Shanahan, 1989] and the planner of [Missiaen, *et al.*, 1995] generate the following incorrect plan.

```
[happens(e1,t1), before(t1,t), happens(e2,t2), before(t2,t)]
```

The abductive event calculus planner presented in [Denecker, *et al.*, 1992] does not suffer from this problem, since it uses a dedicated constraint solver for checking the satisfiability of temporal ordering formulae with respect to a theory of totally ordered time points. The solution adopted in the present paper is very similar.

The `before` predicate is not the only source of incomplete information. 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 are 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 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) :-                              (A5)
  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]. Moreover, since `holds_at` goals will resolve with clause (A3) as well as clause (A5), no further code is required to handle examples with state constraints, such as that in Section 2.

A full listing of an implementation based on the techniques of this section is presented in the appendix. 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 conventional partial-order planning algorithm.

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.

```
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
```

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.

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 and Completeness

This section presents soundness and completeness results for a planner implemented using the techniques described in the previous section. A full listing of the planner is given in the appendix. This listing is also available electronically, and the URL is given in the appendix. This very basic version of the planner is highly inefficient and produces numerous duplicate solutions. An efficient, practical implementation based on this planner is described in the Section 7.

Let AEC be the logic program in the appendix. Note that negation-as-failure is written using the "\+" operator, and that equality and inequality are written in infix form. We'll assume a standard definition of SLDNF-refutation, such as that reproduced in [Shanahan, 1997a, Chapter 11]. The task is to prove that AEC is a sound and complete planner with respect to a certain class of event calculus theories. We begin by pinning down the class of theories we're interested in.

First, soundness and completeness are not guaranteed if there is incomplete knowledge about the initial situation. Hence the following definition.

Definition 4.1. A *complete initial situation* Δ is an initial situation such that for any fluent β either $\Delta \vDash \text{Initially}_N(\beta)$ or $\Delta \vDash \text{Initially}_P(\beta)$. □

Furthermore, unsoundness and incompleteness can result if, at some point in its computation, the planner calls clause 3 of `abdemo_naf` with a `holds_at` goal with an unbound fluent argument. One way to ensure this never happens is to stick to domain descriptions that are fluent range restricted, defined as follows.

Definition 4.2. A domain description Σ is *fluent range restricted* if, for every Initiates, Terminates and Releases clause ϕ in Σ, every non-temporal variable that occurs in ϕ also occurs in the fluent argument in the head of ϕ. □

Next we define a mapping from event calculus theories to logic programs. The mapping is very straightforward, although the definitions are a little unwieldy.

Definition 4.3. The function TRANS_T trivially maps predicate calculus atoms and terms onto corresponding logic program terms (inverting the case of the first letter of each variable, constant, function and predicate symbol, and changing the font to `courier`). □

In what follows, the trivial function TRANS_T will sometimes be omitted for clarity.

Definition 4.4. Let $\Pi = \psi_1 \wedge \ldots \wedge \psi_n$ be a conjunction where each ψ_i is of the form,

$(\neg)$ HoldsAt(β_i,t)

TRANS$_C$(Π) is defined as,

`[holds_at(`σ_1`,T),...,holds_at(`σ_n`,T)]`

where σ_i = TRANS$_T$(β_i) if ψ_i is positive and σ_i = `neg(`TRANS$_T$(β_i)`)` if ψ_i is negative. □

Definition 4.5. Let $\Sigma = \phi_1 \wedge \ldots \wedge \phi_n$ be a domain description. TRANS$_D$(Σ) is defined as follows.

TRANS$_D$(Σ) = {ψ_1, ..., ψ_n}

where,

$$\psi_i = \begin{cases} \texttt{axiom}(\text{TRANS}_T(\sigma), \text{TRANS}_C(\Pi)) \text{ if } \phi_i = \sigma \leftarrow \Pi \\ \texttt{axiom}(\text{TRANS}_T(\phi_i), []) \text{ if } \phi_i \text{ is an atomic formula} \end{cases}$$

□

Definition 4.6. Let $\Delta = \phi_1 \wedge \ldots \wedge \phi_n$ be an initial situation. TRANS$_I$(Δ) is defined as follows.

TRANS$_I$(Δ) = {ψ_1, ..., ψ_n}

where ψ_i is,

`axiom(initially(neg(`TRANS$_T$(β)`))),[])`

if ϕ_i = Initially$_N$(β), and,

`axiom(initially(`TRANS$_T$(β)`)),[])`

if ϕ_i = Initially$_P$(β). □

A number of further definitons are required that remove impure elements from the logic program AEC, namely `findall` and `gensym`. It should be clear that the resulting transformations are meaning-preserving in an intuitive sense. First, we have a function FDALL, which gives a pure version of `findall` tailored to the calls to `findall` in AEC.

Definition 4.7. Let Σ be a domain description. FDALL(Σ) is the set of all clauses of the form `findall(Gs3,`Ψ`,`Φ`)` such that Ψ is either,

```
(abresolve(terms_or_rels(A,F,T2),R1,Gs2,R1),
abresolve(happens(A,T2,T3),R1,[],R1),
append([before(T1,T3),before(T2,T4)|Gs2],Gs1,Gs3))
```

or,

```
(abresolve(inits_or_rels(A,F,T2),R1,Gs2,R1),
abresolve(happens(A,T2,T3),R1,[],R1),
append([before(T1,T3),before(T2,T4)|Gs2],Gs1,Gs3))
```

or,

```
(abresolve(G1,R1,Gs2,R1),append(Gs2,Gs1,Gs3))
```

and Φ is a list comprising all λ such that there is an SLDNF-refutation for the goal clause `:-` Ψ given the logic program TRANS$_D$(Σ) which generates the binding `Gs3=`λ. □

Note that FDALL(Σ) is always finite.

Next we need to purify $\text{TRANS}_D(\Sigma)$ with respect to `gensym`. AEC uses `gensym` to generate new skolem constants for the occurrence times of newly abduced events. The following technique can be used to transform any logic program using `gensym` to an equivalent pure logic program. Every predicate in the program is given two extra arguments, to carry the `gensym` number. Each clause is modified accordingly. For example, the clause,

```
p(X,Y) :- q(X,Y), r(X,Y).
```

becomes,

```
p(X,Y,M1,M3) :- q(X,Y,M1,M2), r(X,Y,M2,M3).
```

Then the following clause defining `gensym` is added.

```
gensym(t(M1),M1,s(M1)).
```

Goal clauses initialise the `gensym` number to 0. For example, the goal clause,

```
:- p(a,X), p(X,b).
```

becomes,

```
:- p(a,X,0,M1), p(X,b,M1,M2).
```

A sequence of ground terms of the form,

```
t(0), t(s(0)), t(s(s(0))), t(s(s(s(0)))), . . .
```

will then be generated by consecutive calls to `gensym`.

Definition 4.8. Let Σ be a domain description. $\text{TRANS}^{+}_{D}(\Sigma)$ is obtained from $\text{TRANS}_D(\Sigma)$ by defining `gensym` as outlined above. □

Now we can define the final mapping.

Definition 4.9. Let Σ be a domain description and Δ be an initial situation. $\text{TRANS}_P(\Sigma,\Delta)$ is defined as follows.

$$\text{TRANS}_P(\Sigma,\Delta) = \text{AEC} \cup \text{TRANS}^{+}_{D}(\Sigma) \cup \text{TRANS}_I(\Delta) \cup \text{FDALL}(\Sigma) \cup \vartheta$$

where ϑ is a set of standard definitions for `member`, `append`, equality and inequality. □

Now we can prove the soundness theorem. First we define a mapping from meta-level logic program goal clauses to object-level predicate calculus formulae.

Definition 4.10. For any `abdemo`, `abdemo_naf` or `abdemo_nafs` literal ϕ, $M_L(\phi)$ is defined as follows.

$M_L($`abdemo(`$[\gamma_1, ..., \gamma_n]$`,`$[\rho_1, ..., \rho_m]$`,R,`$[\eta_1, ..., \eta_k]$`,N`$) =$
$\gamma_1 \wedge ... \wedge \gamma_n \wedge \rho_1 \wedge ... \wedge \rho_m \wedge \neg\, \eta_1 \wedge ... \wedge \neg\, \eta_k$

$M_L($`abdemo_naf(`$[\gamma_1, ..., \gamma_n]$`,`$[\rho_1, ..., \rho_m]$`,R,`$[\eta_1, ..., \eta_k]$`,N`$) =$
$\neg\, (\gamma_1 \wedge ... \wedge \gamma_n) \wedge \rho_1 \wedge ... \wedge \rho_m \wedge \neg\, \eta_1 \wedge ... \wedge \neg\, \eta_k$

$M_L($`abdemo_nafs([`$[\gamma_1, ..., \gamma_n]$`,....,`$[\lambda_1, ..., \lambda_j]$`],`
`[`$\rho_1, ..., \rho_m$`],R,`$[\eta_1, ..., \eta_k]$`,N`$) =$
$\neg\, (\gamma_1 \wedge ... \wedge \gamma_n) \wedge ... \wedge \neg\, (\lambda_1 \wedge ... \wedge \lambda_j) \wedge$
$\rho_1 \wedge ... \wedge \rho_m \wedge \neg\, \eta_1 \wedge ... \wedge \neg\, \eta_k$. □

Definition 4.11. Given a goal clause Ψ of the form,

`:-` ϕ_1, ϕ_2, ..., ϕ_n

let $M(\Psi)$ be,

$$\bigwedge_{\phi \in \Pi} M_L(\phi)$$

where Π is the set comprising every ϕ_i that is an abdemo, abdemo_naf or abdemo_nafs literal. □

Theorem 4.12. (Soundness) For any complete initial situation Δ_0, any conflict free, fluent range restricted domain description Σ and any goal Γ, if there is an SLDNF-refutation for the goal clause,

`:- abdemo(`$\mathrm{TRANS_C}(\Gamma)$`,[[],[]],R,[],N).`

given the logic program $\mathrm{TRANS_P}(\Sigma,\Delta_0)$ which generates a binding `R=`$\mathrm{TRANS_C}(\Delta)$ for some Δ then Δ is a plan for Γ.

Proof. It suffices to show that there is an object-level proof that the goal follows from the plan within the meta-level SLDNF-refutation. Let the sequence of goal clauses $\Psi_1 \ldots \Psi_n$ be an SLDNF-refutation of the above description. By inspection of AEC, we can see that, for $1 \leq i \leq n-1$,

$$\Sigma \wedge M(\Psi_{i+1}) \vDash M(\Psi_i)$$

and therefore,

$$\Sigma \wedge M(\Psi_n) \vDash M(\Psi_0)$$ □

The completeness theorem relies on the following lemma.

Lemma 4.13. Let Σ be a domain description, let Δ_0 be an initial situation, let Δ be a narrative, and let Ω be the conjunction of a pair of uniqueness-of-names axioms for the actions and fluents mentioned in Σ. Let Φ be,

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

For any fluent β and time point τ, if,

$$\Phi \vDash \mathrm{HoldsAt}(\beta,\tau)$$

then either,

$$\Phi \vDash \mathrm{Initially}(\beta) \wedge \neg\, \mathrm{Clipped}(0,\beta,\tau)$$

or,

$$\Phi \vDash \exists\, e,t\, [\mathrm{Happens}(e,t) \wedge t < \tau \wedge \mathrm{Initiates}(e,\beta,t) \wedge \neg\, \mathrm{Clipped}(t,\beta,\tau)].$$

Proof. The proof of this lemma appeals to the fact that event calculus theories conform to the principle of directionality (see [Shanahan, 1997a]), which guarantees that whether or not a fluent holds at any given time point depends only on what is true before that time point. The details of the proof are omitted. □

This lemma is reminiscent of Chapman's modal truth criterion [Chapman, 1987] or Pednault's causality theorem cited in [Penberthy & Weld, 1992].

The function Prog, defined next, progresses an initial situation Δ by one action α. In other words, it yields a theory whose initial situation is the result of performing action α in initial situation Δ. This function is the basis of the inductive proof of completeness to follow.

Definition 4.14. Let Σ be a domain description, and let Ω be the conjunction of a pair of uniqueness-of-names axioms for the actions and fluents mentioned in Σ. Let $\Delta = \phi_1 \wedge \ldots \wedge \phi_n$ be a complete initial situation, where each ϕ_i is either $\mathrm{Initially_P}(\beta_i)$ or $\mathrm{Initially_N}(\beta_i)$. Let Φ be,

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

Let α be any action. Prog(Δ,Σ,α) is $\psi_1 \wedge ... \wedge \psi_n$, where,

$$\psi_i = \begin{cases} \text{Initially}_P(\beta_i) \text{ if} \\ \quad \phi \vDash \text{Initiates}(\alpha,\beta_i,o) \vee [\text{Initially}_P(\beta_i) \wedge \neg \text{Ter min ates}(\alpha,\beta_i,0)] \\ \text{Initially}_N(\beta_i) \text{ otherwise} \end{cases}$$

□

Theorem 4.15. (Completeness) For any complete initial situation Δ_0, any conflict free, fluent range restricted domain description Σ and any goal Γ, if there exists a totally ordered plan P for Γ, then there is an SLDNF-refutation for the goal clause,

:- `abdemo`(TRANS$_C$(Γ), `[[],[]],R,[],N)`.

given the logic program TRANS$_P$(Σ,Δ_0) which generates a binding `R`=TRANS$_C$(Δ) for some Δ such that P is a linearisation of a subset of Δ.

Proof. The proof uses induction over the length of plans.

Base case: In the case of a zero-length plan, there is a refutation that proceeds through clause 3 of `abdemo`, then, after various trivial calls, to clause 2 of `abdemo_naf`, which succeeds quickly since the residue is empty and there are no `happens` clauses to resolve against.

Inductive case: Assume the theorem holds for plans of length $\leq$ n. We need to show the theorem holds for plans of length $\leq$ n+1. If the length of P $\leq$ n, then the theorem holds for P. Otherwise P has n+1 steps. Let α be the first action in plan P, and let P– be a plan comprising the last n actions of P. Consider any SLDNF refutation for the goal clause,

:- `abdemo`(TRANS$_C$(Γ), `[[],[]],R,[],N)`

given the logic program TRANS$_P$(Σ,Prog(Δ_0,Σ,α)) that yields a binding `R`=TRANS$_C$(Δ–) for some Δ– such that P– is a linearisation of a subset of Δ–. It suffices to show how such a refutation can be modified to give a refutation for the same goal clause given the program TRANS$_P$(Σ,Δ_0) which yields a binding R=TRANS$_C$(Δ) for some Δ such that P is a linearisation of a subset of Δ.

To obtain a suitably modified refutation, certain sub-refutations need to be updated, namely those of the form,

:- `abdemo([holds_at(`β,τ`)`, γ_1, . . ., γ_m`]`, R1,R2,N1,N2`)`, Γ_1, . . ., Γ_k

. . .

:- `abdemo([`γ_1, . . ., γ_m`]`, R3,R4,N3,N4`)`, Γ_1, . . ., Γ_k

whose the second clause is obtained by resolving with clause 3 or 5 of the program. The rest of the refutation stays the same. Suppose β is positive. (The case for negative fluents is symmetrical.) Suppose R1=TRANS$_C$(Δ†). There are three cases.

Case 1: $\Sigma \wedge \Delta† \vDash$ Initially$_P$(β) $\wedge \neg$ Clipped(0,β,τ). (Modified initial situation and residue makes no difference.)

Case 2: $\Sigma \wedge \Delta† \vDash$ Initially$_P$(β), but $\Sigma \wedge \Delta† \nvDash \neg$ Clipped(0,β,τ). (We need α to get the $\neg$ Clipped.)

Case 3: $\Sigma \wedge \Delta† \nvDash$ Initially$_P$(β). (We need α to initiate β.)

For each case, we need to show how the old sub-refutation for `holds_at(`β,τ`)` from TRANS$_P$(Σ,Prog(Δ_0,Σ,α)) can be turned into a refutation for `holds_at(`β,τ`)` from TRANS$_P$(Σ,Δ_0).

Case 1. No modification is required.

Case 2. Since there exists a plan P comprising action α followed by plan P–, Lemma 4.13 tells us that the addition to $\Delta\dagger$ of some combination of temporal ordering constraints plus possibly a formula of the form Happens(α,τ') will yield a residue Δ such that $\Sigma \wedge \Delta \models \neg$ Clipped(0,β,τ). The task now is to show that there is a refutation for the goal clause,

`:- abdemo([holds_at(`β,τ`)],R1,R2,N1,N2)`

given the logic program TRANS$_P$(Σ,Δ_0) that yields a binding R2=TRANS$_C$(Δ) for such a Δ. By inspecting AEC, we can see that such a refutation exists. This will proceed through clause 3 of `abdemo`. After the calls to `add_neg` and `abdemo_nafs`, the refutation proceeds through clause 1 of `abdemo_naf`. The `findall` generates a `terms_or_rels` goal and two `before` goals for every potential clipping event for the fluent β. When `abdemo_nafs` is called again with these goals, the attempt to fail the `terms_or_rels` goal will lead, via clauses 3 and 4 of `abdemo_naf`, to the addition of `happens(`α`,`τ`')` to the residue, if necessary, to block the precondition of an event threatening to clip β. The attempt to fail the `before` goals will lead, via clauses 5 to 8 of `abdemo_naf`, to the addition of `before` facts to the residue if necessary to promote or demote an event threatening to clip β.

Case 3. The case is analogous to Case 2. The refutation proceeds via clause 4 of `abdemo`. The second call to `abresolve` adds `happens(`α`,`τ`')` to the residue, and the third call adds `before(`τ`',`τ`)`. Then there are calls to `add_neg` and `abdemo_nafs`, and the rest is the same as Case 2. □

Note that the program is guaranteed not to flounder, since the only uses of negation-as-failure are in calls to `demo_before` or `demo_beq` which follow a call to `abresolve(happens(...)...)` for the same temporal arguments. The call to `abresolve(happens(...)...)` will always ground those arguments.

Note also that the completeness theorem excludes plans with concurrent actions, even though they can be expressed in the event calculus. Moreover, the planner itself does not attempt to find such plans, unlike the event calculus planner reported in [Denecker, *et al.*, 1992].

The soundness and completeness results in this section aren't always useful as they stand. The fluent range restricted condition excludes many typical planning problems, including the shopping trip example as formulated in Section 2. However, the offending variables in such domain descriptions can easily be removed by some simple preprocessing which generates a separate clause for each of their ground instances.

Furthermore, the entire class of examples for which the planner is sound and complete includes many which fall outside the scope of these results. Indeed, the implemented planner works perfectly well on numerous domain descriptions that are not fluent range restricted, including the unpreprocessed shopping trip example. It also works for many examples that include state constraints, such as the plant safety example of Section 2. In particular, if the set of state constraints is stratified, we should expect the above soundness and completeness results to carry over without too much difficulty.

5 Hierarchical Planning

It's a surprisingly straightforward matter to extend the foregoing logical treatment of partial-order planning to planning via hierarchical decomposition.[4] This is achieved

[4] A hierarchical planner based on the event calculus is also presented in [Jung, 1998].

through the introduction of compound actions, which are quite naturally represented in the event calculus. Here's the formal definition.

Definition 5.1. A *compound action description* is a formula of the form,

$\text{Happens}(\alpha 1,\tau 1,\tau 2) \leftarrow \psi_1 \wedge \ldots \wedge \psi_n$

where $\alpha 1$ ia a ground action term, $\tau 1$ and $\tau 2$ are time point variables, and each ψ_i is of the form,

$(\neg)\ \text{HoldsAt}(\beta,\tau)$

where β is a ground fluent term and τ is a time point variable, or,

$\text{Happens}(\alpha 2,\tau 3,\tau 4)$

where $\alpha 2$ is a ground action term and $\tau 3$ and $\tau 4$ are time point variables, or,

$\neg\ \text{Clipped}(\tau 3,\beta,\tau 4)$

where β is a ground fluent term and $\tau 3$ and $\tau 4$ are time point variables, or,

$\tau 3 < \tau 4$

where $\tau 3$ and $\tau 4$ are time point variables, such that, for every time point variable τ occurring in ψ_i,

$\psi_1 \wedge \ldots \wedge \psi_n \vDash \tau 1 \leq \tau \leq \tau 2.$ □

Compound action descriptions are 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, the fluent In(r) holds if the robot is in room r, and the fluent In(p,r) holds if package p is in room r. The formula Connects(d,r1,r2) represents that door d connects rooms r1 and r2.

$\text{Initiates}(\text{Pickup}(p),\text{Got}(p),t) \leftarrow \text{HoldsAt}(\text{In}(r),t) \wedge \text{HoldsAt}(\text{In}(p,r),t)$

$\text{Releases}(\text{Pickup}(p),\text{In}(p,r),t) \leftarrow \text{HoldsAt}(\text{In}(r),t) \wedge \text{HoldsAt}(\text{In}(p,r),t)$

$\text{Initiates}(\text{PutDown}(p),\text{In}(p,r),t) \leftarrow \text{HoldsAt}(\text{Got}(p),t) \wedge \text{HoldsAt}(\text{In}(r),t)$

$\text{Initiates}(\text{GoThrough}(d),\text{In}(r1),t) \leftarrow \text{HoldsAt}(\text{In}(r2),t) \wedge \text{Connects}(d,r2,r1)$

$\text{Terminates}(\text{GoThrough}(d),\text{In}(r),t) \leftarrow \text{HoldsAt}(\text{In}(r),t)$

Now we have the first example of a compound action definition. Compound actions have duration, while primitive actions will usually be represented as instantaneous. The term ShiftPack(p,r1,r2,r3) denotes the action of retrieving package p from room r2 and delivering it to room r3, where the robot is initially in room r1. It comprises a number of sub-actions: two GoToRoom actions, a Pickup action and a PutDown action. The GoToRoom action is itself a compound action, to be defined shortly.

$$\begin{array}{l}
\text{Happens}(\text{ShiftPack}(p,r1,r2,r3),t1,t6) \leftarrow \\
\quad \text{Happens}(\text{GoToRoom}(r1,r2),t1,t2) \wedge t2 < t3 \wedge \\
\quad\quad \neg\ \text{Clipped}(t2,\text{In}(r2),t3) \wedge \neg\ \text{Clipped}(t1,\text{In}(p,r2),t3) \wedge \\
\quad\quad\quad \text{Happens}(\text{Pickup}(p),t3) \wedge t3 < t4 \wedge \text{Happens}(\text{GoToRoom}(r2,r3),t4,t5) \wedge \\
\quad\quad\quad\quad t5 < t6 \wedge \neg\ \text{Clipped}(t3,\text{Got}(p),t6) \wedge \neg\ \text{Clipped}(t5,\text{In}(r3),t6) \wedge \\
\quad\quad\quad\quad\quad \text{Happens}(\text{PutDown}(p),t6)
\end{array}$$

$\text{Initiates}(\text{ShiftPack}(p,r1,r2,r3),\text{In}(p,r3),t) \leftarrow \text{HoldsAt}(\text{In}(r1),t) \wedge \text{HoldsAt}(\text{In}(p,r2),t)$

The effects of compound actions should follow from the effects of their sub-actions, as can be verified in this case by inspection. Note that it is up to the author of the formulae to ensure that this is the case. Next we have the definition of a GoToRoom action, where GoToRoom(r1,r2) denotes the action of going from room r1 to room r2.

$$\begin{array}{l}
\text{Happens(GoToRoom(r,r),t,t)} \\
\text{Happens(GoToRoom(r1,r3),t1,t3)} \leftarrow \\
\quad \text{Connects(d,r1,r2)} \wedge \text{Happens(GoThrough(d),t1)} \wedge \\
\qquad \text{Happens(GoToRoom(r2,r3),t2,t3)} \wedge \text{t1} < \text{t2} \wedge \\
\qquad\quad \neg\, \text{Clipped(t1,In(r2),t2)} \\
\text{Initiates(GoToRoom(r1,r2),In(r2),t)} \leftarrow \text{HoldsAt(In(r1),t)}
\end{array}$$

This illustrates both conditional decomposition and recursive decomposition: 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.[5]

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.

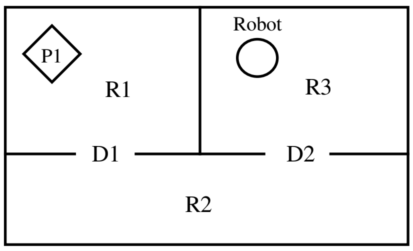

Figure 1: A Mail Delivery Domain

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.

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)

[5] In practise, the definition of GoToRoom would benefit from the use of a heuristic to guide the search.

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

$\text{Initially}_P(\text{In}(R3))$ $\quad$ $\text{Initially}_P(\text{In}(P1,R1))$

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)	
$T0 < T1$	$T1 < T2$
$T2 < T3$	$T3 < T4$
$T4 < T$	

Now we have, for example,

$$\begin{array}{l}\text{CIRC}[\Sigma_p \wedge \Sigma_c \text{ ; Initiates, Terminates, Releases}] \wedge \\ \quad \text{CIRC}[\Delta_0 \wedge \Delta_p \wedge \Delta_c \text{ ; Happens}] \wedge \text{EC} \wedge \Omega \wedge \Phi \vDash \\ \quad\quad \text{Happens(ShiftPack(P1,R3,R1,R2),T0,T4)}.\end{array}$$

We also have,

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

So Δ_p constitutes a plan.[6] Furthermore, we have,

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

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 would like the following to hold. For any fluent β and time point τ, HoldsAt(β,τ) follows from,

$$\begin{array}{l}\text{CIRC}[\Sigma_p \wedge \Sigma_c \text{ ; Initiates, Terminates, Releases}] \wedge \\ \quad \text{CIRC}[\Delta_p \wedge \Delta_c \text{ ; Happens}] \wedge \text{EC} \wedge \Omega \wedge \Phi\end{array}$$

if and only if it follows from,

$$\begin{array}{l}\text{CIRC}[\Sigma_p \text{ ; Initiates, Terminates, Releases}] \wedge \\ \quad \text{CIRC}[\Delta_p \text{ ; Happens}] \wedge \text{EC} \wedge \Omega \wedge \Phi.\end{array}$$

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.

[6] The inclusion of compound actions violates the conditions under which the circumscription of a narrative can be straightforwardly reduced to predicate completion, making proofs of such propositions more complicated. Recursion makes matters particularly tricky.

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 facilitates 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. This issue is discussed in more depth in [Shanahan, 1998].

This brings us to the issue of implementation. What modifications are required to the abductive meta-interpreter of Section 3 to enable it to perform hierarchical decomposition? In principle, the answer is almost none at all. 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.[7] However, in a practical planning system, it may be desirable to include special code for handling decomposition.

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.

Let me conclude this section with a few words about soundness and completeness. As far as soundness is concerned, Section 4's simple proof requires negligible modification to cover a suitably widened class of event calculus theories that permits the definition of compound actions. The completeness proof is more subtle, but it seems safe to conjecture that this too extends naturally to the same wider class of theories. However, no formal proof is offered in the present paper.

6 Shortcomings

This section discusses three of the planner's shortcomings.[8]

1. Unsoundness can result when a domain is not fluent range restricted.
2. Incompleteness can arise with incomplete information about the initial situation.
3. Unsoundness can result with some examples involving state constraints.

An example illustrating each shortcoming is given in turn. (It should be noted that each of these examples is excluded by the conditions attached to the soundness and completeness theorems of Section 4.)

[7] 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.

[8] Generally speaking, conventional planners suffer from exactly the same limitations.

The following example shows how a domain description that is not fluent range restricted can lead to unsoundness in the planner. When a person executes a `sell` action, they become rich, but they lose possession of everything that the shop they're in will buy.

```
terminates(sell,have(X),T) :-
  holds_at(have(X),T), holds_at(at(Y),T),  buys(Y,X).
initiates(sell,rich,T).
buys(hws,drill).
```

The person is initially at the hardware shop, not at home, and has a drill.

```
initially(have(drill)).
initially(at(hws)).
initially(neg(at(home))).
```

Now consider the following query.

```
:- abdemo([holds_at(rich,t), holds_at(have(drill),t)],R).
```

Since the person cannot become rich without losing their drill, no plan exists for this goal, so the query should fail. Unfortunately, the query succeeds, returning the following trivial "plan".

```
R = [[happens(sell, t2, t2)], [before(t2, t)]]
```

To see how the planner can arrive at this faulty answer, we have to examine the way it processes the second goal. By the time it comes to solve the second goal, it has already built the incorrect plan. Given the `terminates` clause at the top of the domain description, the goal `holds_at(have(drill),t)` should fail. But the planner in fact proceeds as follows. In its misguided attempt to make `terminates(sell,have(drill),t2)` fail, the planner applies clause 3 of `abdemo_naf`, which tries to make `holds_at(at(Y),t2)` fail. To do this, it's happy to find *any* `Y` such that `holds_at(neg(at(Y)),t2)`, such as `Y = home`.

The problem is that the quantifiers have been treated incorrectly in a way that is directly analogous to floundering in negation-as-failure. Specifically, $\neg \exists$ x P(x) has been treated as $\exists$ x $\neg$ P(x). Enforcing the fluent range restricted condition ensures that this problem cannot arise. An alternative approach would be to explicitly enumerate the objects in the domain, and to prove $\neg$ P(x) for each such object x. Although this solution would only work for domains with a finite number of objects, this would be a useful future extension to the planner. It should also be noted that some of the other abductive planners, such as [Missiaen, *et al.*, 1995] and [Denecker, *et al.*, 1992]), solve this problem correctly.

Next we have an example, due to Rob Miller, that illustrates the need for more sophisticated processing in the presence of incomplete information about the initial situation. The domain comprises two actions `vaccinate1`, which immunises a patient if they have blood type 1, and `vaccinate2`, which immunises a patient if they have any other blood type.

```
initiates(vaccinate1,immune,T) :- holds_at(type1,T).
initiates(vaccinate2,immune,T) :- holds_at(neg(type1),T).
```

We know nothing about the initial situation, so we don't know the patient's blood type. This uncertainty about the initial situation is represented simply by the absence of an `initially` formula for the `type1` fluent. However, it's clear that the patient will be immune after a `vaccinate1` action followed by a `vaccinate2` action, whatever their blood type. But the query,

```
:- abdemo([holds_at(immune,t)],R)
```

produces no answer.

One method for tackling incompletely specified initial situations, which could be adapted for the present planner, is outlined in [Miller, 1995]. This involves generating each possible permutation of fluents consistent with what's known about the initial situation, and finding a plan that works for all of them.

Finally, the following example shows how the careless inclusion of state constraints can lead to unsoundness. First, we have the following state constraint.

```
holds_at(happy,T) :- holds_at(rich,T)
```

The domain's only action is `rob_bank`, which initiates `rich`.

```
initiates(rob_bank,rich,T)
```

In the initial situation, we have `neg(rich)` and `neg(happy)`.

```
initially(neg(rich))
initially(neg(happy))
```

Now, if we present the planner with the query,

```
:- abdemo([holds_at(happy,t)],R)
```

we get the answer,

```
R = [[happens(rob_bank, t2, t2)], [before(t2, t)]].
```

At first glance, this seems right. But in fact, the conjunction of this narrative `R` with the domain description and event calculus axioms yields both `holds_at(happy,t)` and `holds_at(neg(happy),t)`, and the planner is therefore not generally sound in the presence of state constraints. The source of the problem is the formula `initially(neg(happy))`. Inspection of the event calculus axioms reveals that, if a fluent is explicitly declared as initially false, then it can only be initiated directly by an event, not indirectly by a state constraint. Omission of this initially formula yields the desired behaviour.

A possible way to extend the soundness and completeness theorems of Section 4 to encompass state constraints might be to enforce a partition of fluents into primitive and derived. Only derived fluents would be permitted on the left-hand-side of a state constraint, and only primitive fluents would be allowed as arguments to Initiates and Terminates formule.

7 Implementation and Efficiency Issues

The planner in the appendix is very inefficient, and is only intended as a theoretically elegant starting point for building a more practical system. A more efficient and usable planner can be obtained from the elegant planner via the following steps.

- Numerous cuts can be inserted to eliminate redundant backtracking and duplicate solutions.
- The transitive closure of temporal ordering constraints in the residue can be maintained and passed as a parameter along with the residue itself, instead of being computing by calls to `demo_before`. A number of experiments were carried out which demonstrated that this enhancement considerably improves the performance of the planner.
- When negated `clipped` and `declipped` lemmas are re-proved in response to additions of actions to the residue, redundant computation can be reduced by

focusing on only the most recently added action, and by looking only at the `clipped` and `declipped` lemmas that refer to intervals within which that event falls.

A fully commented listing of the planner is available electronically, and the URL is given in the appendix.

The performance of the efficient version of the planner was evaluated using two benchmark problems. The aim here is not to compete with other planners in terms of efficiency , but rather to demonstrate that the adoption of a theorem proving approach hasn't incurred any hidden exponential complexity. In Test A, the planner was run on the shopping trip example from Section 2, with an increasing number of objects to be bought. The results are given in Table 1. All the test were carried out using an Apple Macintosh PowerBook 1400cs, running LPA MacProlog 32, with unoptimised code. Timings are in seconds. The average of three runs is given to compensate for small variations from run to run.

Objects	6	8	10	12	14	16
Time	0.44	0.84	1.52	2.22	3.69	5.52

Table 1: Test A Results

Actions in plans in Test A can be carried out in any order: the agent can choose any order in which to buy the objects. In Test B, on the other hand, the planner was run on an example involving a chain of preconditions, and generates totally ordered plans. The performance of the planner was investigated as the number of preconditions increases. The results are given in Table 2.

Preconds	6	8	10	12	14	16
Time	0.06	0.11	0.13	0.20	0.26	0.35

Table 2: Test B Results

In both tests, the planner's performance is polynomial, and in Test B it is nearly linear. This suggests that there is indeed no hidden exponential cost to adopting a theorem proving approach to planning. However, the planner fares less well on examples, such as the flat tyre problem, on which recent planners such as Satplan [Kautz & Selman, 1996] and Graphplan [Blum & Furst, 1997] perform more impressively. This is unsurprising, though, as partial-order planners like UCPOP also compare unfavourably to these planners on such examples.

A further planner has been implemented, having the following features.

• If run directly in Prolog, the planner in the appendix inherits Prolog's depth-first search strategy. Completeness is then sacrificed, and many straightforward planning problems give rise to looping. Accordingly, the practical implementation uses an iterative deepening search strategy, where length of plan is used as the depth bound.

• With respect to hierarchical planning, rather than relying on the automatic decomposition carried out by SLDNF-resolution as described in the last section, the practical planner has special code for decomposing compound actions. This enables decomposition and partial-order planning to be carefully interleaved in such a way that useful partial results are obtained in the middle of the planning process. (See [Shanahan, 1998].)

This planner can also be downloaded, via the URL given in the appendix.

Although it would be hard to construct a formal proof, the intention is that all of the above described modifications to the basic planner in the appendix are meaning-preserving in a sense that preserves soundness and completeness.

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 by Missiaen, *et al.* [1995], although that planner deals incorrectly with certain problems, a drawback addressed by the planner of [Denecker, *et al.*, 1992]. Other abductive event calculus planners have been developed by Chleq [1996] and 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 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, like the planner of [Denecker, *et al.*, 1992], performs correctly on the anomalous examples described in [Missiaen, *et al.*, 1995].

A number of general-purpose procedures for abductive logic programming exist, such as SLDNFA [Denecker & De Schreye, 1998], ACLP [Kakas, *et al.*, 1998], and the IFF procedure [Fung & Kowalski, 1997], that can also been applied to planning or reasoning about action. These procedures are more complex than the present abductive planner, but soundness and completeness theorems have been developed for them which cover a wider class of examples than the fluent range restrictedness condition used here will allow. A detailed comparison of these procedures with the present planner is highly desirable, but is beyond the scope of the present paper.

Recently, a resource-bounded version of the planner, alongside a similar implementation of abductive sensor data assimilation, has been deployed on a miniature mobile robot in a simple mail delivery domain [Shanahan, 1998]. Work continues along these lines.

Acknowledgments

Thanks to Rob Miller, Marc Denecker, and two anonymous referees. This work was carried out as part of the EPSRC funded project GR/L20023 "Cognitive Robotics".

References

[Blum & Furst, 1997] A.L.Blum and M.L.Furst, Fast Planning Through Planning Graph Analysis, *Artificial Intelligence*, vol. 90 (1997), pp. 281–300.

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

[Chitaro & Montanari, 1996] L.Chittaro and A.Montanari, Efficient Temporal Reasoning in the Cached Event Calculus, *Computational Intelligence*, vol.12, no.3 (1996), pp. 359–382.

[Chleq, 1996] N.Chleq, Constrained Resolution and Abductive Temporal Reasoning, *Computational Intelligence*, vol.12, no.3 (1996), pp. 383–406.

[Denecker, *et al.*, 1992] M.Denecker, L.Missiaen and M.Bruynooghe, Temporal Reasoning with Abductive Event Calculus, *Proceedings ECAI 92*, pp. 384–388.

[Denecker & De Schreye, 1998] SLDNFA: An Abductive Procedure for Abductive Logic Programs, *Journal of Logic Programming*, vol. 34, no. 2, pp. 111–167.

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

[Fung & Kowalski, 1997] T.H.Fung and R.A.Kowalski, The IFF Proof Procedure for Abductive Logic Programming, *Journal of Logic Programming*, vol. 33, no. 2, pp. 151–165.

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

[Jung, 1998] C.G.Jung, Situated Abstraction Planning by Abductive Temporal Reasoning, *Proceedings ECAI 98*, pp. 383–387.

[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.

[Kakas, *et al.*, 1998] A.C.Kakas, A.Michael and C.Mourlas, ACLP: A Case for Non-Monotonic Reasoning, *Proceedings of the 7th International Workshop on Non-Monotonic Reasoning (NM 98).*

[Kautz & Selman, 1996] H.Kautz and B.Selman, Pushing the Envelope: Planning, Propositional Logic and Stochastic Search, *Proceedings AAAI 96*, pp. 1194–1201.

[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.

[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*, vol. 31 (1997), pp. 59–83.

[Lifschitz, 1994] V.Lifschitz, Circumscription, in *The Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, ed. D.M.Gabbay, C.J.Hogger and J.A.Robinson, Oxford University Press (1994), pp. 297–352.

[Miller, 1995] R.S.Miller, Situation Calculus Specifications for Event Calculus Logic Programs, *Proceedings of the Third International Conference on Logic*

Programming and Non-Monotonic Reasoning, Lexington, KY, USA, Springer Verlag, 1995, pp. 217–230.

[Missiaen, 1991] L.Missiaen, Localized abductive planning for robot assembly, *Proceedings 1991 IEEE Conference on Robotics and Automation*, pp. 605–610.

[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.

[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.

[Russell & Norvig, 1995] S.Russell and P.Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall International (1995).

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

[Shanahan, 1998] M.P.Shanahan, Reinventing Shakey, *Working Notes of the 1998 AAAI Fall Symposium on Cognitive Robotics*, pp. 125–135.

Appendix

A companion website to this paper can be accessed via http://www.ee.ic.ac.uk/~mpsha/planners.html. This website contains the following.

- Version 4.2 of the planner. This is the bare, uncommented, pure but inefficient version given below.
- Version 1.9a of the planner. This is the faster version that was used for the benchmarks described in Section 6. It's reasonably efficient at partial-order planning, but doesn't handle hierarchical planning well.
- Version 1.13 of the planner. This version of the planner uses an iterative deepening search strategy, and has special code for hierarchical decomposition. It's less efficient for straight partial-order planning than Version 1.9a.
- Code for each of the examples given in the paper.
- Code for the two benchmark examples described in Section 6.

All the code is written in LPA MacProlog 32, but should be easy to port to other Prolog systems.

There follows a complete listing, without comments, of the pure but inefficient version of the planner used for the proofs in Section 6.

```
/*

   ABDUCTIVE EVENT CALCULUS

   MURRAY SHANAHAN

   Version 4.2
   Stripped down, cut-free version, without comments

*/

abdemo(Gs,R) :- init_gensym(t), abdemo(Gs,[[],[]],R,[],N).

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

abdemo([holds_at(F1,T)|Gs1],R1,R3,N1,N4) :-
     F1 \= neg(F2), abresolve(initially(F1),R1,Gs2,R1),
     append(Gs2,Gs1,Gs3), add_neg([clipped(0,F1,T)],N1,N2),
     abdemo_naf([clipped(0,F1,T)],R1,R2,N2,N3),
     abdemo(Gs3,R2,R3,N3,N4).

abdemo([holds_at(F1,T3)|Gs1],R1,R5,N1,N4) :-
     F1 \= neg(F2), abresolve(initiates(A,F1,T1),R1,Gs2,R1),
     abresolve(happens(A,T1,T2),R1,[],R2),
     abresolve(before(T2,T3),R2,[],R3),
     append(Gs2,Gs1,Gs3),
     add_neg([clipped(T1,F1,T3)],N1,N2),
     abdemo_nafs(N2,R3,R4,N2,N3),
     abdemo(Gs3,R4,R5,N3,N4).

abdemo([holds_at(neg(F),T)|Gs1],R1,R3,N1,N4) :-
     abresolve(initially(neg(F)),R1,Gs2,R1),
     append(Gs2,Gs1,Gs3), add_neg([declipped(0,F,T)],N1,N2),
     abdemo_naf([declipped(0,F,T)],R1,R2,N2,N3),
     abdemo(Gs3,R2,R3,N3,N4).

abdemo([holds_at(neg(F),T3)|Gs1],R1,R5,N1,N4) :-
     abresolve(terminates(A,F,T1),R1,Gs2,R1),
     abresolve(happens(A,T1,T2),R1,[],R2),
     abresolve(before(T2,T3),R2,[],R3),
     append(Gs2,Gs1,Gs3),
     add_neg([declipped(T1,F,T3)],N1,N2),
     abdemo_nafs(N2,R3,R4,N2,N3),
     abdemo(Gs3,R4,R5,N3,N4).

abdemo([G|Gs1],R1,R3,N1,N2) :-
     abresolve(G,R1,Gs2,R2), append(Gs2,Gs1,Gs3),
     abdemo(Gs3,R2,R3,N1,N2).

abresolve(terms_or_rels(A,F,T),R,Gs,R) :- axiom(releases(A,F,T),Gs).

abresolve(terms_or_rels(A,F,T),R,Gs,R) :-
     axiom(terminates(A,F,T),Gs).
```

```
abresolve(inits_or_rels(A,F,T),R,Gs,R) :- axiom(releases(A,F,T),Gs).

abresolve(inits_or_rels(A,F,T),R,Gs,R) :- axiom(initiates(A,F,T),Gs).

abresolve(happens(A,T),R1,Gs,R2) :-
     abresolve(happens(A,T,T),R1,Gs,R2).

abresolve(happens(A,T1,T2),[HA,BA],[],[HA,BA]) :-
     member(happens(A,T1,T2),HA).

abresolve(happens(A,T,T),[HA,BA],[],[[happens(A,T,T)|HA],BA]) :-
     executable(A), skolemise(T).

abresolve(before(X,Y),R,[],R) :- demo_before(X,Y,R).

abresolve(before(X,Y),R1,[],R2) :-
     \+ demo_before(X,Y,R1), \+ demo_beq(Y,X,R1),
     add_before(X,Y,R1,R2).

abresolve(diff(X,Y),R,[],R) :- X \= Y.

abresolve(G,R,Gs,R) :- axiom(G,Gs).

abdemo_nafs([],R,R,N,N).

abdemo_nafs([N|Ns],R1,R3,N1,N3) :-
     abdemo_naf(N,R1,R2,N1,N2), abdemo_nafs(Ns,R2,R3,N2,N3).

abdemo_naf([clipped(T1,F,T4)|Gs1],R1,R2,N1,N2) :-
     findall(Gs3,
          (abresolve(terms_or_rels(A,F,T2),R1,Gs2,R1),
          abresolve(happens(A,T2,T3),R1,[],R1),
          append([before(T1,T3),before(T2,T4)|Gs2],Gs1,Gs3)),Gss),
     abdemo_nafs(Gss,R1,R2,N1,N2).

abdemo_naf([declipped(T1,F,T4)|Gs1],R1,R2,N1,N2) :-
     findall(Gs3,
          (abresolve(inits_or_rels(A,F,T2),R1,Gs2,R1),
          abresolve(happens(A,T2,T3),R1,[],R1),
          append([before(T1,T3),before(T2,T4)|Gs2],Gs1,Gs3)),Gss),
     abdemo_nafs(Gss,R1,R2,N1,N2).

abdemo_naf([holds_at(F1,T)|Gs],R1,R2,N1,N2) :-
     opposite(F1,F2), abdemo([holds_at(F2,T)],R1,R2,N1,N2).
```

```
abdemo_naf([holds_at(F,T)|Gs],R1,R2,N1,N2) :-
     abdemo_naf(Gs,R1,R2,N1,N2).

abdemo_naf([before(X,Y)|Gs],R,R,N,N) :- X = Y.

abdemo_naf([before(X,Y)|Gs],R,R,N,N) :- X \= Y, demo_before(Y,X,R).

abdemo_naf([before(X,Y)|Gs],R1,R2,N1,N2) :-
     X \= Y, \+ demo_before(Y,X,R1),
     abdemo_naf(Gs,R1,R2,N1,N2).

abdemo_naf([before(X,Y)|Gs],R1,R2,N,N) :-
     X \= Y, \+ demo_before(Y,X,R1),
     \+ demo_beq(X,Y,R1), add_before(Y,X,R1,R2).

abdemo_naf([G|Gs1],R,R,N,N) :-
     G \= clipped(T1,F,T2), G \= declipped(T1,F,T2),
     G \= holds_at(F,T), G \= before(X,Y),
     \+ abresolve(G,R,Gs2,R).

abdemo_naf([G1|Gs1],R1,R2,N1,N2) :-
     G1 \= clipped(T1,F,T2), G1 \= declipped(T1,F,T2),
     G1 \= holds_at(F,T), G1 \= before(X,Y),
     findall(Gs3,(abresolve(G1,R1,Gs2,R1),append(Gs2,Gs1,Gs3)),Gss),
     Gss \= [], abdemo_nafs(Gss,R1,R2,N1,N2).

demo_before(X,Y,[HA,BA]) :- demo_before(X,Y,BA,[]).

demo_before(0,Y,R,L) :- Y \= 0.

demo_before(X,Y,R,L) :- X \= 0, member(before(X,Y),R).

demo_before(X,Y,R,L) :-
     X \= 0, \+ member(before(X,Y),R), member(X,L).

demo_before(X,Y,R,L) :-
     X \= 0, \+ member(before(X,Y),R), \+ member(X,L),
     member(before(X,Z),R), demo_before(Z,Y,R,[X|L]).

demo_beq(X,X,R).

demo_beq(X,Y,R) :- X \= Y, demo_before(X,Y,R).

add_before(X,Y,[HA,BA]) :- member(before(X,Y),BA).

add_before(X,Y,[HA,BA],[HA,[before(X,Y)|BA]]) :-
```

```
     \+ member(before(X,Y),BA), \+ demo_beq(Y,X,[HA,BA]).

add_neg(N,Ns,Ns) :- member(N,Ns).

add_neg(N,Ns,[N|Ns]) :- \+ member(N,Ns).

skolemise(T) :- gensym(t,T).

opposite(neg(F),F).

opposite(F1,neg(F1)) :- F1 \= neg(F2).
```