Computational Intelligence, Volume 12, Number 3, 1996

HYPOTHETICAL REASONING ABOUT ACTIONS: FROM SITUATION CALCULUS TO EVENT CALCULUS

ALESSANDRO PROVETTI
C.I.R.F.I.D. - Università di Bologna. Via Galliera 3, Bologna, I-40121, Italy.
provetti@cirfid.unibo.it

Hypothetical reasoning about actions is the activity of pre-evaluating the effect of performing actions in a changing domain; this reasoning underlies applications of Knowledge Representation such as planning and explanation generation. Action effects are often specified in the language of Situation Calculus, introduced by McCarthy and Hayes in 1969. More recently, the Event Calculus has been defined to describe actual actions, i.e., those that have occurred in the past, and their effects on the domain. Altough the two formalisms share the basic ontology of atomic actions and fluents, Situation Calculus cannot represent actual actions while Event Calculus cannot represent hypotethical actions. In this paper, the language and the axioms of Event Calculus are extended to allow representing and reasoning about hypothetical actions, performed either at the present time or in the past, altough counterfactuals are not supported. Both Event Calculus and its extension are defined as logic programs so that theories are readily adaptable for Prolog query interpretation. For a reasonably large class of theories and queries, Prolog interpretation is shown to be sound and complete w.r.t. the main semantics for logic programs.

Key words: actions and change, hypothetical reasoning, narrative assimilation, knowledge representation, logic programming

1. INTRODUCTION

Hypothetical reasoning about actions is the activity of evaluating the effect of actions that affect a given domain; it is by now an established subfield of knowledge representation and it is mainly associated with the Situation Calculus (hereinafter SC), introduced by McCarthy and Hayes (1969) and later works in this area, such as the well-known Yale shooting problem of Hanks and McDermott (1987). The representation of temporally-scoped relations is another of such subfields in knowledge representation, where we can say Allen's (1984) Interval logic plays the rôle Situation Calculus plays in reasoning about actions.

The Event Calculus(EC) of Kowalski and Sergot (1986) is an example of temporal knowledge representation by means of a *Situation Calculus style* ontology of actions and fluents. Event Calculus' forte is the ability to *assimilate a narrative*, i.e. the description of a course of events, adjusting the effects of action and the time-line of the narrative as it becomes more and more precise, in an additive only fashion.

These two formalisms have received attention from several authors, with various perspectives; Table 1 is a summary of contributions related to this paper.

In their recent production, Pinto and Reiter (1993a and 1993b) have introduced an extended version of Situation Calculus (ESC) which makes it possible

- to represent dates and time-stamp actions and situations;
- to represent a narrative which actually occurred in the world as a branch of the tree of *possible developments of the world* that SC handles.

These features are obtained by introducing new predicate definitions, and new ordered types of constants for representing dates and functions such as $Start(action)$ or $End(action)$, linking actions to their dates. Pinto and Reiter argue that ESC matches the so called linear time formalisms, viz. Interval Logic and EC, on their

own ground: representing actions and change over time, while still being able to deal with hypothetical actions.

TABLE 1. A short summary of literature on Situation Calculus and Event Calculus.

Situation Calculus		Event Calculus	
Authors	Contribution	Author(s)	Contribution
McCarthy and Hayes 1969	defined SC		
		Kowalski and Sergot 1986	defined EC
Hanks and McDermott 1987	showed problems with semantics		
Gelfond et al. 1991	attracted new interest		
		Kowalski 1992	compared with SC
Pinto and Reiter 1993	ext. to actual actions and dates		
Miller and Shanhan 1994	defined narratives in SC	Kowalski and Sadri 1994	
		this work	ext. to hypothetical actions

Another point of comparison is the declarative semantics. Having stressed that in Event Calculus the use of negation as failure can produce unexpected results, viz. overcommitment, Pinto and Reiter discuss the semantics of ESC, which is given in two ways:

1. the extended axiomatization has a first-order semantics plus predicate circumscription for formalizing particular assumptions about the domain;
2. a logic programming implementation for a fragment of the formalisms is given that can be proven to be sound w.r.t. Clark's completion semantics.

Their claim seems to be that this kind of logical precision and clarity has not yet achieved for EC implementations.

This paper pursues a parallel objective, showing how SC's specific and, indeed, desirable features are easily implementable in a linear-time formalism like Event Calculus. Moreover, it is possible to define a precise declarative semantics for a reasonably broad class of domain formalizations in the extended EC.

In Section 2, we present a simple version of EC and discuss its relation with the assumptions we make on domains. In Section 3, new predicates are introduced for allowing reasoning about a fictional sequence of actions and projecting the value of fluents. This form of hypotethical reasoning can either be performed *in the future*, for exploring the result of alternative plans, or starting from a date in the past[1].

Since EC is defined as a logic program, it has a dual interpretation as a formalism for knowledge representation and a computational mechanism, thanks to SLDNF resolution and Prolog. Therefore, in Section 4 the declarative semantics aspect is discussed; if an EC axiomatization is seen as a logic program, then the most common declarative semantics agree, i.e., yield the same intended minimal model.

In Section 5, we argue that the extended EC behaves well w.r.t. termination and floundering, thus assessing a plain computational value.

[1] Altough a flavour of counterfactual reasoning is present, neither our formalization nor Pinto and Reiter's support it in full.

In the end, the author argues for a substantial equivalence between the extended EC and Pinto and Reiter's linear-time extension of SC.

In this paper some acquaintance with Situation Calculus and Logic Programming is assumed.

2. A TIME-POINTS ORIENTED EVENT CALCULUS

The Event Calculus has been proposed by Kowalski and Sergot (1986) as a system for reasoning about time and actions in the framework of Logic Programming.

Event Calculus is based on 3 ontologies. The first ontology is that of *events*, or action-tokens; these events are represented by means of constants that uniquely identify them. Each event is an instance of an $action-type$ (the second ontology) from which it inherits the description of its effects. The third ontology is that of *fluents*[2], which like in SC represent partial descriptions of the reality being modeled. A fluent *holds* over time from the moment that an event *initiates* it, i.e., the event makes it true in the world. Events may also *terminate* fluents, i.e., make tham false in the world. Event Calculus is based on forward default persistence —a fluent holds over time until a terminating event is recorded.

Since the first proposal, a number of improved formalizations have been proposed in literature that adapt the calculus to different tasks. Hence, the simple version of Shanahan (1989) is presented, because it can be taken as a common core definition embedded in late applications[3], such as abductive planning, diagnosis, temporal database and models of Law. Let us start by defining the logic programming framework and the basic language definitions used in the rest of the paper.

2.1. Logic programming framework

Assume a language of constants, function constants and predicate constants. Assume also that terms and atoms are built as in the corresponding first-order language. A rule ρ is an expression of the form:

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{1}$$

where $A_0, \ldots A_n$ are atoms and *not* is a logical connective called *negation as failure*. Also, for every rule let us define $head(\rho) = A_0$, $pos(\rho) = A_1, \ldots, A_m$, $neg(\rho) = A_{m+1}, \ldots, A_n$ and $body(\rho) = pos(\rho) \cup neg(\rho)$. The head of rules is never empty and, when $body(\rho) = \emptyset$, we refer to ρ as *a fact*.

A logic program is defined as a collection of rules. Rules with variables are taken as shorthand for the sets of all their ground instantiations and the set of all ground atoms in the language of a program Π will be denoted by H_Π.
Queries are expressions with the same structure of rules but with an empty head.

In this paper, the *stable models* semantics (Gelfond and Lifschitz 1988) for logic programs is adopted, but we will see that the main results remain independent from this choice. Intuitively, a stable model is a possible view of the world that is *compatible* with the rules of the program. Rules are therefore seen as constraints on these views

[2]Elsewhere called *properties* or *relationships*.

[3]This version is even more simplified, because it assumes events are recorded in the database in the same order as they happened in reality. For the presentation of a full formalization, refer to works of Sergot (1990) and Sripada (1991).

of the world. Let us start defining stable models of the subclass of positive programs, i.e., those where, for every rule ρ, $neg(\rho) = \emptyset$.

Definition 1. (Stable model of positive programs)

The *stable model* $a(\Pi)$ of a positive program Π is the smallest subset of H_Π such that for any rule (1) in Π:

$$A_1, \ldots, A_m \in a(\Pi) \;\Rightarrow\; A_0 \in a(\Pi) \tag{2}$$

Clearly, positive programs have a unique stable model, which coincides with that obtained applying other semantics; in other words, positive programs are unambigous.

Definition 2. (Stable models of programs)

Let Π be a logic program. For any set S of atoms, let Π^S be a program obtained from Π by deleting

(i) each rule that has a formula *not* A in its body with $A \in S$;

(ii) all formulae of the form *not* A in the bodies of the remaining rules.

Clearly, Π^S does not contain *not* —so its stable model is already defined. If this stable model coincides with S, then we say that S is a *stable model* of Π. In other words, a stable model of Π is characterized by the equation:

$$S = a(\Pi^S). \tag{3}$$

Programs which have a unique stable model are called *categorical*.

Let us define entailment in the stable models semantics. A ground atom α is *true in* S if $\alpha \in S$, otherwise α is *false*, i.e., $\neg\alpha$ is *true* is S. This definition can be extended to arbitrary first-order formulae in the standard way.

We will say that Π *entails a formula* ϕ (written $\Pi \models \phi$) if ϕ is true in *all* the stable models of Π. We will say that the answer to a ground query γ is

yes if γ is true in all stable models of Π, i.e., $\Pi \models \gamma$;

no if $\neg\gamma$ is true in all stable models of Π, i.e., $\Pi \models \neg\gamma$;

unknown otherwise.

It is easy to see that logic programs are *nonmonotonic*, i.e., adding new information to the program may force a reasoner associated with it to withdraw its previous conclusions.

2.2. Basics of the language

The language of Event Calculus is typed, i.e. terms will belong to just one of these five disjoint sorts:

$\mathcal{T}$ Dates, which are represented by the first N integers and variables $t, t_1, \ldots$;

$\mathcal{F}$ Fluents, which are represented by constant and function symbols;
In the rest of the paper, the examples refer to the Blocks World domain; therefore fluent constants will be ground terms such as $On(A, B)$ and $Clear(C)$. Fluent variables are denoted $f, g \ldots$, etc.

$\mathcal{A}_{types}$ Action types;
In the Blocks World domain ground terms such as $Move(A, B)$ and $Unstack(B)$ will be used. Action-type variables are named $a\_type_1 \ldots$.

$\mathcal{A}_{tokens}$ Action tokens, which are represented by constants $E_1, E_2, \ldots$ (after "event") and variables a_token_1 etc..

The first predicates we introduce are $Happens : \langle \mathcal{A}_{tokens} \rangle$, $Date : \langle \mathcal{A}_{tokens}, \mathcal{T} \rangle$ and $Type : \langle \mathcal{A}_{tokens}, \mathcal{A}_{types} \rangle$. These are used for describing events, by means of sets of facts like the following:

$Happens(E_1)$
$Date(E_1, 17)$
$Type(E_1, Unstack(B))$

where action type $Unstack(B)$ is associated with token E_1.

The next predicates are $Initiates : \langle \mathcal{A}_{tokens}, \mathcal{F} \rangle$ and $Terminates : \langle \mathcal{A}_{tokens}, \mathcal{F} \rangle$. It is important to notice that action types are understood by looking at the $Initiates/Terminates$ definitions where it appears. In fact, these definitions set the description of the domain by describing the interplay between action types and fluents. For instance, in the Blocks World domain we have:

$$Initiates(e, On(x, y)) \leftarrow Type(e, Move(x, y))$$

$$Initiates(e, Clear(z)) \leftarrow Type(e, Move(x, y)), Date(e, t), HoldsAt(On(x, z), t), not\ z \equiv y$$

$$Terminates(e, Clear(y)) \leftarrow Type(e, Move(x, y))$$

$$Terminates(e, On(x, y)) \leftarrow Type(e, Move(x, z)), not\ z \equiv y$$

The $HoldsAt$ condition in the body of $Initiates$ expresses the dependence of the effect $Clear(z)$ on $On(x, z)$, which is time-dependent.

Now, let us introduce the topmost predicate: $HoldsAt(f, t) : \langle \mathcal{F}, \mathcal{T} \rangle$, which is understood as *"fluent f is true at time t."* Axiom ECI listed below expresses that a fluent holds[4] at a certain time if an earlier event initiated the fluent itself and there is no evidence in the database of the fluent ceasing to hold in the meantime.

$$(ECI)\quad HoldsAt(f, t) \leftarrow Happens(e), Initiates(e, f), Date(e, t_s), t_s < t, not\ Clipped(t_s, f, t)$$

[4]If observations on the value of fluents can be introduced in the formalization, i.e., $HoldsAt$ updates are allowed, a transformation of the axioms is necessary for giving consistent answers, at cost of a loss of elegance; Sripada (1991) presents a version of the calculus for accommodating such updates.

In other words, in the interval between the initiation of the fluent and the time the query is about, no terminating event has happened. This is made sure by axiom $ECII$ below. The *forward default persistence rule* is implemented by using negation as failure on $Clipped$ in ECI.

$$(ECII)\quad Clipped(t_s, f, t) \leftarrow \begin{array}{l} Happens(e^*), \\ Terminates(e^*, f), \\ Date(e^*, t^*), \\ t_s \leq t^*, \\ t^* < t \end{array}$$

The predicates $<$ and $\leq$ establish an ordering of events. We stipulate that temporal constants $T_1, T_2, T_3 \ldots$ are mapped on natural numbers, and that the ordering relations are also mapped on the same relations on naturals, thus inheriting their properties.

2.3. The Assumptions Underlying Event Calculus

The axiomatization of EC discussed so far embodies several assumptions on the structure of the domain, e.g., default persistence, as well as the ontological assumptions discussed earlier.

It is important to understand these assumptions, because they a) determine applicability of the formalism and b) provide a direction of research for making EC as general as possible. The main assumptions follow.

1. *It is assumed that all the events are time-stamped;*
2. *it is assumed that no events occur other than those that are known to occur;*
3. *it is assumed that fluents persist until an event happens that influences them;*
4. *it is assumed that no action-type can affect a given fluent other than those that are known to do so;*
5. *conversely, it is assumed that every fluent has an explanation in terms of events.*

In logic programming, default assumptions are represented by means of negation-as-failure[5], albeit this may cause the phenomenon of *overcommitment* and in general makes assumptions not easy to lift.

In the rest of this section, we will briefly illustrate how the formalism of *extended logic programs* can help in making the assumptions explicit in the axiomatization and limit the side effects of negation-as-failure. Even though reformulating EC in extended logic programming is an intuitive *next step* in research, no formal result can be presented at this stage.

Extended logic programs have the same syntax of normal programs except for the introduction of a new form of negation *(¬)* know as *explicit* or *classical*. In the definitions of rules, atoms are replaced by *literals*, i.e., formulae of the form α or $\neg\alpha$ where α is an atom.

For this class of programs, Gelfond and Lifschitz (1991) have defined the *Answer Sets* semantics, a natural extension of stable models.

Let us reconsider the assumptions and sketch how extended programs can represent them explicitly.

[5] See (Baral and Gelfond 1994) for a discussion of these aspects.

1. *It is assumed that all the events are time-stamped;*

   This is not strictly a closure assumption and it has been forced upon domain descriptions with the pourpose of presenting a simple formalization.
   Lifting this assumption by allowing a partial order on events adds a further degree on nonmonotonicity and makes reasoning more complex, especially in terms of nonmonotonic effects and complexity[6].
   Notice also that Kowalski and Sergot's (1986) EC was not concerned with time and dates, but rather with relative ordering among events and naming of intervals.

2. *it is assumed that no events occur other than those that are known to occur;*

3. *it is assumed that fluents persist until an event happens that influences them;*

   These two assumptions are made explicit by taking the definition of *Clipped* as complete, i.e.,

$$\neg Clipped(t_s, f, t) \leftarrow not\ Clipped(t_s, f, t)$$

4. *it is assumed that no action type can affect a given fluent other than those which are known to do so;*

$$\neg Initiates(e\_token, f) \leftarrow not\ Initiates(e, f)$$

$$\neg Terminates(e\_token, f) \leftarrow not\ Terminates(e, f)$$

5. *it is assumed that every fluent has an explanation in terms of events.*

   As a result:

   - no fluent is true at time 0;
   - at least one initiating event is necessary for making a fluent true[7].

   This is the sharpest departure from SC and can be seen as a side effect of negation as failure. It is possible to get round this problem, even in normal programs, by introducing an extra predicate for describing fluents that are true in the initial state of the narrative:

$$Initially(Clear(A))$$
$$\ldots$$
$$Initially(Table(B))$$

$$(ECIII)\quad HoldsAt(f,t) \leftarrow Initially(f), not\ Clipped(0,f,t)$$

   Notice that it is sufficent to leave a fluent undefined to state that is not initially true.

[6] Refer to (Cervesato *et al.* 1993) for a discussion of the problem; also (Chittaro *et al.* 1994) addresses reasoning with partially ordered narratives in a more complex framework of modal interpretations.

[7] This is particularly interesting for generating explanations of fluents by abducing events (Shanahan 1989).

3. HYPOTHETICAL REASONING IN EC

In this section we define an extension of EC for reasoning about the effect of hypothetical sequences of actions. Pinto and Reiter (1993a) point out the utility of having temporal and hypothetical reasoning together:

> *By preserving the branching state property of the Situation Calculus, we can express and answer a variety of hypothetical queries, although counterfactuals cannot be expressed. For example "At time T_p in the past, when you put A on B, could A have been put on C instead?" can be simply expressed as:*
>
> $$during(T_p, s) \wedge actual(s) \supset possible(put(A, C), s)$$
>
> *"If I had performed $put(A, C)$, would F have been true?"*
>
> $$holds(F, do(put(A, C), S_p)) \ \wedge \ possible(put(A, C), S_p)$$
>
> *None of these features is possible in linear temporal logics. We need the branching structure of the situation calculus, coupled with a linear time line in that branching structure.*

Let us discuss which features have to be added to EC to make it able to handle this kind of reasoning.

3.1. The new predicates

The ideas motivating the new predicate definitions are the following:

- to rewrite Situation Calculus axioms within EC, in order to carry out projection;
- to provide a link between the point in time t where the simulation begins and the value of fluents in the simulation. That is, fluents that are true at t are still true during the simulation as long as an event does not terminate them. To this extent, the effect of the simulation depends on the time it starts;
- to make it possible to reason about hypothetical actions perfomed either *in the past* or *in the future* in the same fashion.

A new sort for situations ($\mathcal{S}$) is introduced; situations are defined using the function symbols Sit and Res:

$Sit : \mathcal{T} \ \rightarrow \ \mathcal{S}$

$Res : \mathcal{A}_{types} \times \mathcal{S} \ \rightarrow \ \mathcal{S}$

Variables $s_1, s_2, \ldots$ range over situations. Let us define the new predicates.

$$HypHolds(F, S)$$

states that fluent F is true in situation S.

$$Possible(A\_type, S)$$

captures the notion that it is possible to perform the action A_type starting from situation S. By

$$MayInitiate(A\_type, F, S)$$

it is intended that the action of type A_type performed in situation S brings about(makes true) the fluent F.

$$MayTerminate(A\_type, F, S)$$

it is intended that the action of type A_type performed in situation S makes fluent F false.

The axioms of the extended version, which are defined below, will be used in conjunction with a collection of facts such as $\{Happens(E_1),\ Date(E_1, 11),\ Type(E_1, Unstack),\ Initially(Clear(B))\}$ etc.

$$(EC1)\quad HoldsAt(f,t) \leftarrow Happens(e\_token),\ Date(e\_token, t_s),\ Type(e\_token, a\_type),\ t_s < t,\ MayInitiate(a\_type, f, Sit(t_s)),\ not\ Clipped(t_s, f, t)$$

$$(EC2)\quad Clipped(t_s, f, t) \leftarrow Happens(e\_token^*),\ Date(e\_token^*, t^*),\ Type(e\_token^*, a\_type^*),\ t_s \leq t^*,\ t^* < t,\ MayTerminate(a\_type^*, f, Sit(t^*))$$

$$(EC3)\quad HoldsAt(f,t) \leftarrow Initially(f),\ not\ Clipped(0, f, t)$$

The following axioms are for reasoning about hypothetical actions and closely resemble those of Situation Calculus.

$$(EC4)\quad HypHolds(f, Sit(t)) \leftarrow HoldsAt(f,t)$$

$$(EC5)\quad HypHolds(f, Res(a\_type, s)) \leftarrow Possible(a\_type, s),\ MayInitiate(a\_type, f, s)$$

$$(EC6)\quad HypHolds(f, Res(a\_type, s)) \leftarrow Possible(a\_type, s),\ HypHolds(f, s),\ not\ MayTerminate(a\_type, f, s)$$

3.2. Domain-dependent definitions

In extended EC, $MayInitiate$, $MayTerminate$ and $Possible$ are used to formalize domains, following this schema: perfoming action A_type in situation s initiates (respectively, terminates) F if:

i) the situation-independent conditions $\phi(A\_type, F)$ *(resp.* $\psi(A\_type, F)$ *)* are true;
ii) the fluent preconditions $C_1, \ldots C_m$ are the case in s;
iii) the fluent preconditions $C_{m+1} \ldots C_n$ are not the case in s.

$$MayInitiate(A\_type, F, s) \quad \leftarrow \phi(A\_type, F), \; HypHolds(C_1, s), \ldots HypHolds(C_m, s), \; not\ HypHolds(C_{m+1}, s), \ldots not\ HypHolds(C_n, s)$$

$$MayTerminate(A\_type, F, s) \quad \leftarrow \psi(A\_type, F), \; HypHolds(C_1, s), \ldots HypHolds(C_m, s), \; not\ HypHolds(C_{m+1}, s), \ldots not\ HypHolds(C_n, s)$$

In this schema the place-holders $\phi(A\_type, F)$ and $\psi(A\_type, F)$ stand for conjuncts of (positive or negative) atoms which do not have a situation as parameter. Clearly, a sensible domain description is one in which no action-type can both initiate and terminate the same fluent in any situation.
The same reading style is used for the definitions of $Possible(A\_type, s)$:

$$Possible(A\_type, s) \leftarrow \varphi(A\_type), \; HypHolds(C_1, s), \ldots HypHolds(C_m, s), \; not\ HypHolds(C_{m+1}, s), \ldots not\ HypHolds(C_n, s)$$

Let us illustrate the new domain-dependent predicates in the Block World example. To specify that *"a block becomes clear if we move the blocks above it elsewhere"* we write:

$$MayInitiate(Move(x, y), Clear(z), s) \leftarrow HypHolds(On(x, z), s), \; not\ z \equiv y$$

where

$$\phi(Move(x, y), Clear(z)) \equiv [not\ z \equiv y]$$

Contrast this definition with that of $Initiates(Move(x, y), Clear(z))$ on page 5.

3.3. The new predicates at work

The first question addressed by Pinto and Reiter:

> *At time T_p in the past, when you put A on B, could A have been put on C instead?*

translates into:

$$? - MayHappen(Put(A, C), T_p)$$

Conversely, the second example:

> *If I had performed $put(A, C)$, would F have been true?*

translates into:

$$? - HypHolds(F, Res(Put(A, C), T_p))$$

Plan validation. Another feature inherited from SC is the ability to check the validity of plans, albeit they cannot be directly generated. In other words, to check whether the plan *"do A, then do B and C"* is an effective way to achieve the goal G (expressable by a fluent) at the moment T, it is sufficent to interpret the query $? - HypHolds(G, Res(C, Res(B, Res(A, Sit(T)))))$.
Interestingly, the dependency of plan effects on the contest where the plan is perfomed is automatically obtained.

4. DECLARATIVE SEMANTICS

Pinto and Reiter (1993a) have compared the "first-order + circumscription" semantics of their formalism with that of EC:

> One advantage of this is the clean semantics provided by our axiomatization, in contrast to the event calculus reliance on the *Negation as failure* feature of logic programming, whose semantics is not well understood.

The argument is rather appropriate: EC has been defined within logic programming (Kowalski and Sergot 1986; Sergot 1990; Kowalski 1992) and the use of *negation as failure* for implementing default persistence is intrinsic to EC.

In this section, it will be shown that, if we restrict ourselves to *well-formed* domain descriptions, narratives and queries, it can be guaranteed that the resulting axiomatization falls in a class of programs for which the main declarative semantics for logic programming agree[8]. Morevoer, such programs are categorical; these results establish an unambigous declarative semantics.

After having defined well-formed domain descriptions precisely, we will briefly review some recent results in logic programming and apply them the semantics of extended EC.

4.1. Well-formed EC formalizations

Definition 3. (Well-formed narratives)

A narrative Θ is well-formed if

1. for every event token e_i the facts:

   $Happens(e_i)$
   $Date(e_i, t_i)$
   $Type(e_i, a\_type_i)$

   are in Θ and e_i does not appear elsewhere;
2. there are no references to other EC predicates;
3. possibly, there are facts $Initially(f)$.

Definition 4. (Well-formed domain descriptions)

[8]Refer to the contributions in (Minker 1988) and the survey paper of Apt and Bol (1994) for a discussion of semantics of logic programming and their equivalences.

A well-formed domain description Δ is a set of $MayInitiate$, $MayTerminate$ and $Possible$ rules conforming to the following schemata:

$$\begin{aligned} MayInitiate(A\_type, F, s) \leftarrow \quad & \phi(A\_type, F), \\ & HypHolds(C_1, s), \\ & not\ HypHolds(C_2, s) \end{aligned}$$

$$\begin{aligned} MayTerminate(A\_type, F, s) \leftarrow \quad & \psi(E\_type, F), \\ & HypHolds(C_3, s), \\ & not\ HypHolds(C_4, s) \end{aligned}$$

$$\begin{aligned} Possible(A\_type, s) \leftarrow \quad & \varphi(E\_type), \\ & HypHolds(C_1, s), \\ & not\ HypHolds(C_2, s) \end{aligned}$$

where formulae $\phi(A\_type, F)$, $\psi(A\_type, F)$ and $\varphi(A\_type)$ are conjunctions of atoms such that:

1. their predicate definitions are *stable* (see definition 20 on page 19);
2. no situation variable appears in their predicate definitions.

4.2. Logic Programming Background

Let us review the subclass of *acyclic programs*, introduced by Apt and Bezem (1991) and shown to be of particular interest for temporal reasoning.

Definition 5. (Level mapping)

A *level mapping* $|\ |$: $\mathrm{H}_\Pi \rightarrow \mathbb{N}$, is a function from program atoms to natural numbers.

Definition 6. (Acyclic programs)

A rule ρ (1) is acyclic w.r.t. $|\ |$ if

$$\forall A_i \in body(\rho).\ |A_0| > |A_i|$$

A program is *acyclic w.r.t.* $|\ |$ if all of its clauses are.
A program Π is *acyclic* if it is acyclic w.r.t. some level mapping $|\ |$.

Lemma 1. from Apt and Bol (1994)

Acyclic programs have the same unique model under all the major declarative semantics for logic programming.

Now, let us review some definitions and results from (Lifschitz and Turner 1994). These are originally given for the larger class of extended logic programs, but we limit ourselves to the normal programs case.

Definition 7. (Splitting set)

A *splitting set* for a program Π is any set U of atoms such that for every rule in Π if its head belongs to the the splitting set then all the atoms appearing in the body of the rule belong to the set itself:

$$\forall \rho \in \Pi.\ head(\rho) \in U \quad \Rightarrow \quad body(\rho) \in U$$

The set of rules of Π such that $head(\rho) \cup body(\rho) \in U$ is called *the bottom* of Π w.r.t. U, denoted $b_U(\Pi)$.
The subprogram $\Pi \setminus b_U(\Pi)$ is called *the top of* Π *relative to* U. Clearly,

$$\rho \in \Pi \setminus b_U(\Pi) \;\Rightarrow\; head(\rho) \in \mathrm{H}_\Pi \setminus U$$

Definition 8. (Partial evaluation)

The partial evaluation of a program Π with splitting set U w.r.t. a set of atoms X is the program $e_U(\Pi, X)$ defined as follows. For each rule $\rho \in \Pi$ such that

$$(pos(\rho) \cap U) \in X \quad \wedge \quad (neg(\rho) \cap U) \cap X = \emptyset$$

put the rule ρ' in $e_U(\Pi, X)$, defined as follows,

$$head(\rho') = head(\rho),\ pos(\rho') = pos(\rho) \setminus U,\ neg(\rho') = neg(\rho) \setminus U$$

Definition 9. (Solution)

Let U be a splitting set for a program Π. A solution to Π w.r.t. U is a pair $\langle X, Y \rangle$ such that:

- X is a stable model for $b_U(\Pi)$;
- Y is a stable model for $e_U(\Pi \setminus b_U(\Pi), X)$.

Lemma 2. (Splitting Lemma)

Let U be a splitting set for a program Π. A set A of atoms is a stable model for Π if and only if $A = X \cup Y$ for some solution $\langle X, Y \rangle$ to Π w.r.t. U.

□

Corollary 1. For every program Π with splitting set U:

$\forall l \in U.$ $\quad \Pi \models l \;\Leftrightarrow\; l \in X_i$ $\quad$ for every solution $\langle X_i, Y_i \rangle$ to Π w.r.t. U.

$\forall l \in \mathrm{H}_\Pi \setminus U.$ $\quad \Pi \models l \;\Leftrightarrow\; l \in Y_i$ $\quad$ for every solution $\langle X_i, Y_i \rangle$ to Π w.r.t. U.

Proof. Immediate by applying the definitions above.

■

4.3. Declarative semantics of extended event calculus

Proposition 1. $\Pi = \Theta \cup \Delta \cup \{EC1, \ldots EC6\}$ is not stratified.

As an instance, consider the unfolding of the $HoldsAt$ definition:

$$HoldsAt(f,t) \leftarrow \ldots\ not\ Clipped(t_s, f, t)$$
$$Clipped(t_s, f, t) \leftarrow \ldots\ MayTerminate(e, f, t_1)$$
$$MayTerminate(e, f, t_1) \leftarrow \ldots\ HypHolds(c_3, Sit(t_1))$$
$$HypHolds(c, Sit(t_1)) \leftarrow HoldsAt(c, t_1)$$

$HoldsAt$ is defined in terms of itself via the negation on $Clipped$. □

Proposition 2. $\Pi = \Theta \cup \Delta \cup \{EC1, \ldots EC6\}$ is not locally stratified in general.

Local stratification depends on fluent-to-fluent dependencies, so let us start proving Proposition 2 by considering how the stratification should be defined over $HoldsAt$ atoms.
For each fluent f, look at the $HoldsAt$ literals appearing in the bodies of $MayInitiate(a\_type, f, t)$ and $MayTerminate(a\_type, f, t)$.
For each negative occurence *not* $HypHolds(c, s)$ in the body of $Initiates$ and for each positive or negative occurence in the body of $Terminates$, the condition on stratification is (read "$\| \|$" as *"level of"*):

$$\forall t_1, t_2 : \|HoldsAt(c, t_2)\| \prec \|HoldsAt(f, t_1)\|$$

The whole set of constraints on $HoldsAt$ atoms can be seen as a dependency graph having the fluent names as vertices. If the graph is acyclic, the program can be given a local stratification, but this might well not be the case. □

Notice that the "Situation Calculus" part, viz. axioms $EC4, EC5$, is easy to prove locally stratified, because of the use of Res functions: $\|HypHolds(f, s)\| \prec \|HypHolds(f, Res(a, s))\|$ and so on.

Theorem 1. (Extended EC is categorical)

For every well-formed narrative Θ and domain description Δ, the program $\Pi = \Theta \cup \Delta \cup \{EC1 \ldots EC6\}$ has a unique minimal model in the stable models semantics.

Proof. It is easy to show that $U = H_{\{Happens, Date, Type, Initially\}}$ is a split set for Π, and that the bottom $b_U(\Pi)$ is exactly Θ.
Moreover, Θ is a set of ground facts and, therefore, it has a unique minimal model:

$$M_{\text{bottom}}\,(A \in M_{\text{bottom}} \Leftrightarrow \{A \leftarrow\} \in \Theta)$$

Let us consider the partial evaluation $e_U(\Pi, M_{\text{bottom}})$; what we expect to find in the result of the partial evaluation are clauses like the following:

$$(EC1') \quad HoldsAt(f, t) \leftarrow MayInitiate(e\_type, f, Sit(t_s)), \; not\ Clipped(t_s, f, t) \qquad \{ \textit{ where } t_s < t\}$$

$$(EC2') \quad Clipped(t_s, f, t) \leftarrow MayTerminate(e\_type^*, f, Sit(t^*)) \qquad \{ \textit{ where } t_s \leq t^* < t\}$$

whereas the instances of $\{EC3 \ldots EC5\}$ and clauses in Δ remain unchanged.
Now, it is possible to show that $e_U(\Pi, M_{\text{bottom}})$ is acyclic. First, consider the following *recursive term mapping*, where t is a natural number:

$$|Sit(t)|_{term} = 4t + 5$$

$$|Res(a, s)|_{term} = |s|_{term} + 2$$

Now, take the following level mapping schemata:

$|HoldsAt(f,t)| \;=\; 4t+4$

$|MayInitiate(e\_type,f,s)| \;=\; |s|_{term}+1$

$|Clipped(t_s,f,t)| \;=\; 4t+3$

$|MayTerminate(e\_type,f,s)| \;=\; |s|_{term}+1$

$|HypHolds(f,Sit(t))| \;=\; |s|_{term}$

$|Possible(e\_type,s)| \;=\; |s|_{term}+1$

$|\phi(E\_type,F)| \;=\; |\psi(E\_type,F)| \;=\; |\varphi(E\_type)| \;=\; 0$

Finally, for every atom A: $|not\ A| \;=\; |A|$.

It remains easy to show that each clause in $e_U(\Pi, M_{\text{bottom}})$ satisfies the acyclicity condition; hence, there exists a unique model M_{top} for $e_U(\Pi, M_{\text{bottom}})$.

As a result, $M \;=\; \langle M_{bottom}, M_{top}\rangle$ is the only solution for Π w.r.t. U and, therefore, the unique stable model of Π.

■

Corollary 2. For every well-formed history Θ and domain description Δ, the program $\Pi \;=\; \Theta \cup \Delta \cup \{EC1 \ldots EC6\}$ has a unique minimal model in all the main logic programming semantics.

Proof. Immediate from Theorem 1 and Lemma 1.

■

Remark. (Shanahan's Proof)

Introducing the version of EC used in Section 2, Shanahan (1989) has shown an intuitive, albeit nonstandard, transformation on the ground version of the program that allows to prove local statification. In this paper, the result follows from techniques not available then and addresses a more general problem, given the presence of new axioms. ◇

5. COMPUTATIONAL PROPERTIES

This section is concerned with showing that the EC axiomatization presented in this paper enjoys good computational properties.

We seek a proof that the usual Prolog interpretation of queries, such as $? - Holds(f,t)$ or $? - HypHolds(f,s)$, against a well-formed EC program always terminates giving sound answers w.r.t. to the declarative semantics of the program itself.

We will show that, under reasonable conditions on the structure of queries, this is the case. In fact, let us restrict ourselves to *EC-allowed* queries, defined as follows.

Definition 10. (EC-allowed queries)

A query γ is EC-allowed if it satisfies two requirements, one static and one dynamic:

1. there are no time or situation variables;
2. negative subqueries are always interpreted ground.

As an instance,

$$\leftarrow HoldsAt(f, 11),\ not\ HoldsAt(f, 12)$$

is EC-allowed because, should the first sub-query succeed with substitution, say, $f/Clear(A)$, then $not\ HoldsAt(Clear(A), 12)$, which is ground, would be interpreted.

5.1. Termination

Let us review the characterization of Prolog termination in (Apt and Pedreschi 1993). By the term *goal*, we intend a query interpreted by Prolog against a program.

Definition 11. ($LDNF$ derivation)

LDNF derivation differs from standard SLDNF derivation in the use of Prolog *first-left* selection rule.

We intend to show that for a certain class of programs and queries (resp. acceptable programs and bounded queries) all the LDNF derivations are finite. In order to define the key concept of *acceptability*, some auxiliary definitions are needed.

Definition 12. (Depends on)

Let π and ϖ be predicates defined in Π. We will say that

π refers to ϖ iff there is a rule ρ in Π such that $head(\rho) = \pi(\underline{x})$ and $\varpi(\underline{y}) \in body(\rho)$.

π depends on ϖ iff $\langle \pi, \varpi \rangle$ is in the reflexive transitive closure of "refers to."

Definition 13. (Π^-)

For every program Π, let Neg_Π be the set of predicates in Π that occur negatively in the body of a rule in Π, i.e.,

$$\mathrm{Neg}_\Pi = \{\pi : \exists \rho.\, \pi(\underline{x}) \in neg(\rho)\}$$

$$\mathrm{Neg}^*_\Pi = \{\pi : \exists \varpi.\, \varpi \in \mathrm{Neg}_\Pi \wedge \varpi \text{ depends on } \pi\}$$

$$\Pi^- = \{\rho : head(\rho) \in \mathrm{Neg}^*_\Pi\}$$

In order to keep the next definitions simple, let us introduce the following.

Definition 14. (Suitable model)

A model I of Π is called *suitable* if its restriction to the predicates in Neg^*_Π is a model of $comp(\Pi^-)$.

For EC programs, we have already seen (Corollary 2) that considering the completion of the program is equivalent to considering its stable models.

Definition 15. (Acceptability)

A program Π is *acceptable w.r.t.* $|\ |$ *and a suitable model* I if for every rule $A_0 \leftarrow L_1, \ldots L_m$ where $L_i = A_i \vee$ *or* A_i:

$$\forall \rho \in \Pi.\, I \models L_1, ..., L_{i-1} \ \Rightarrow\ |A_0| > |L_i|$$

A program is called *acceptable* if it is acceptable with respect to some level mapping $|\ |$ and a suitable model I of it.

Now, let us define bounded queries.

Definition 16. (Bounded queries)

Let Π be a program with a level mapping $|\ |$ and a suitable model I. For each goal $\gamma : \leftarrow G_1 \ldots G_n$, a finite multiset $|\gamma|_I$ is defined as follows:

$$|\gamma|_I = bag(|G_1|, \ldots |G_{\overline{n}}|)$$

where $\overline{n} = min(\{n\} \cup \{i \in [1 \ldots n] : I \not\models G_i\})$. Moreover, for each goal γ the following set of multisets $|[\gamma]|_I$ is defined:

$$|[\gamma]|_I = \{|\gamma'|_I : \gamma' \text{ is a ground instance of } \gamma\}$$

We will say that a goal γ *is bounded by* k *wrt* I if $\forall l.l \in \cup|[\gamma]|_I \ \Rightarrow\ k \geq l$, where $\cup|[\gamma]|_I$ is the set-theoretic union of elements of γ. Finally, a goal γ *is bounded* w.r.t. $|\ |$ and I if it is bounded for some $k \geq 0$ w.r.t. I.

Notice that, by our definition of level mapping in Theorem 1, every atom with no time or situation variable is bounded and $\forall A'.\, |A'|_I = |[A]|_I$.

Lemma 3. (Corollary 4.11 of Apt and Pedreschi 1993)

Let Π an acceptable program and γ a bounded goal, then all LDNF derivations of $\Pi \cup \gamma$ are finite.

Theorem 2. (Acceptability)

For every well-formed Θ and Δ, $\Pi = \Theta \cup \Delta \cup \{EC1 \ldots EC6\}$ is acceptable.

Proof. Consider the level mapping $|\ |$ used in Theorem 1 with the following extension:

$|Date(e\_token, t_s)| = 0$

$|Type(e\_token, e\_type)| = 0$

$|t_s < t| = |t_s \leq t| = 0$

Consider the ground instancies of $EC1$ and assume:

$$I \models Happens(e\_token), \ldots, t_s < t.$$

By the definition of $<$ in Π, we are certain that t_s is less[9] than t, in other words t_s can be at most $t-1$. Applying the level mapping described earlier and the condition for acceptability we obtain:

[9]Here t and t_s are placeholders for temporal constants of the language, e.g. naturals.

$$|HoldsAt(f,t)| > |MayInitiate(e\_type, f, Sit(t))| \Leftrightarrow$$

$$4t + 4 > |Sit(t_s)|_{term} + 1 \Leftrightarrow$$

$$4t + 4 > 4t_s + 6$$

Now, $Max(t_s) = t - 1$ and substituting: $4t + 4 > 4(t-1) + 6 \Rightarrow 4t + 4 > 4t + 2$ which is true for any t.

The same line of reasoning can be followed for proving, w.r.t. axiom $EC2$, that if $I \models Happens(e\_token^*), \ldots, t^* < t$ then:

$$|Clipped(t_s, f, t)| > |MayTerminate(A\_type, f, Sit(t^*))|$$

For the remaining axioms, the proof is straightforward, thanks to the definition of term mapping, by which heads containing $Res(a, s)$ are always *greater* than the atoms in their bodies, which contain the situation constant s.

■

Lemma 4. (EC-allowed queries are bounded)

Every query γ with no temporal variable appearing within is bounded.

Proof. Follows from considering that $|\ |$ is defined in terms of temporal constants appearing in the atoms; therefore, for each time-variable-free atom G: $|G| : |[G]|$ and $|[G]|$ always has a limit $l = K(T)$. In particular, $\forall \tau \in \mathcal{F}. |Holds(\tau, T)| < 4T + 5$.

■

Corollary 3. (Finite interpretation of extended EC)

For every well-formed history Θ and domain description Δ, the LDNF interpretation of a bounded query γ against $\Pi = \Theta \cup \Delta \cup \{EC1 \ldots EC6\} \cup \{\gamma\}$ is finite.

Proof. Immediate from Theorems 2 and 4, and Lemma 3.

■

At the end of the next section, we will see how these result extend to actual Prolog interpretation.

5.2. Floundering

So far we have considered LDNF, which is a *safe* computation rule, i.e., it never interprets non-ground negative queries. In fact, it is well-known that this form of interpretation is often unsound. When this is the case, the interpretation is said to *flounder* and LDNF fails[10].

On the other hand, actual Prolog interpreters apply the left-first selection rule *unsafely*, thus becoming occasionally unsound.

In this section, we show that the Prolog interpretation of EC-allowed queries against well-formed programs never leads to floundering and, therefore, Prolog evaluation is guaranteed sound w.r.t. the declarative semantics of these programs.

In the following, we take advantage of the results of Stroetman (1993).

[10] It is also known (Apt and Pedreschi 1993) that floundering is a factor in temination analysis.

Definition 17. (Mode)

A *mode* divides the occurence of variables appearing in an atom between *input positions* "+" and *output positions* "−". $\sigma_\Pi : \{1 \ldots n\} \rightarrow \{+, -\}$

Intuitively, input variables are those that are substituted by a ground term when the atom is interpreted. Output variables are either ground at the start or returned instantiated at the end of the interpretation.

Definition 18. (Input variables and output variables)

For every atom $\pi(\tau_1, \ldots, \tau_n)$ we define the multiset $FV^+(\pi(\tau_1, \ldots, \tau_n))$ of input variables as follows:

$$FV^+(\pi(\tau_1, \ldots, \tau_n)) \;=\; \biguplus\{FV^+(\tau_i) \;:\; i \in \{i \ldots n\} \;\wedge \sigma_\pi(i) = \;+\}$$

where $\biguplus$ stays for union of multisets. Likewise, the multiset $FV^-(\pi(\tau_1, \ldots, \tau_n))$ of output variables is defined as follows:

$$FV^-(\pi(\tau_1, \ldots, \tau_n)) \;=\; \biguplus\{FV^-(\tau_i) \;:\; i \in \{i \ldots n\} \;\wedge \sigma_\pi(i) = \;-\}$$

Clearly, $FV \;=\; FV^+ \cup FV^-$. Moreover, for negative formulae *not* A we define:

$$FV^+(\mathit{not}\ A) \;=\; FV(A) \;\;\wedge\;\; FV^-(\mathit{not}\ A) \;=\; \emptyset$$

The rationale of this definition is to force negative formulae to be interpreted as ground.

Definition 19. (I/O specification)

An *I/O specification* σ is a function that assign a mode to every predicate symbol.

Definition 20. (Stable)

Given a program Π and a query γ, we will say that a rule ρ of Π is stable w.r.t. an I/O-specification σ iff

1. $\forall x \in FV^-(A_0).\left\{ \begin{array}{ll} & x \in FV^+(A_0) \\ or & \\ & \exists j \in \{1 \ldots n\}.\, x \in FV(A_j) \end{array} \right.$
2. $\forall x \in FV^+(A_i).\left\{ \begin{array}{ll} & x \in FV^+(A_0) \\ or & \\ & \exists j \in \{1 \ldots i-1\}.\, x \in FV(A_j) \end{array} \right.$

Also, we will say that a goal γ is stable w. r. t. an I/O-specification σ iff

$$\forall x \in FV^+(A_i).\, \exists j \in \{1 \ldots i-1\}.\, x \in FV(A_j)$$

We will say that $\Pi \cup \gamma$ is stable w.r.t. an I/O-specification σ iff γ and every rule in Π is.

Lemma 5. (Stable programs and queries do not flounder)

If Π and γ are stable w. r. t. an I/O-specification σ then:

1. no Prolog computation of $\Pi \cup \gamma$ flounders;
2. all the computed answers of $\Pi \cup \gamma$ are ground.

Now we can prove that our programs are stable.

Theorem 3. (EC with EC-allowed queries is stable)

For every well-formed program $\Pi \equiv \Theta \cup \Delta \cup \{EC1 \ldots EC6\}$, if γ is EC-allowed then $\Pi \cup \gamma$ is stable.

Proof. Let us consider the following I/O-specification:

$\sigma(HoldsAt(f,t)) = \langle -,+ \rangle$
$\sigma(HypHolds(f,s)) = \langle -,+ \rangle$
$\sigma(Clipped(t_s,f,t)) = \langle +,+,+ \rangle$
$\sigma(MayInitiate(a,f,s)) = \langle -,-,+ \rangle$
$\sigma(MayTerminate(a,f,s)) = \langle -,-,+ \rangle$
$\sigma(Possible(a,s)) = \langle -,+ \rangle$
$\sigma(t_1 < t_2) = \langle +,+ \rangle$
$\sigma(t_1 \leq t_2) = \langle +,+ \rangle$
$\sigma(Happens(a)) = \langle - \rangle$
$\sigma(Date(a,t)) = \langle -,- \rangle$
$\sigma(Type(a,ty)) = \langle -,- \rangle$

Now, since γ is EC-allowed, it is also stable w.r.t. σ. In particular, there are no situation and time variables, e.g., for $\gamma : HoldsAt(f,T)$ we have $FV^+(\gamma) = \emptyset$ and $FV^-(\gamma) = \{f\}$.
Let us show that Π is stable. Consider the rule EC1:

$$FV^-(HoldsAt(f,T)) = \{f\} \;\&\; f \in FV(MayInitate(e\_type, f, Sit(t_s))).$$

$$FV^+(Happens(e\_token)) = \emptyset.$$

$$FV^+(Date(e\_token, t_s)) = \emptyset.$$

$$FV^+(Type(e\_token, a\_type)) = \emptyset.$$

$$FV^+(t_s < t) = \{t_s, t\} \;\&\; \begin{cases} t_s \in FV(Date(e\_token, t_s)); \\ \\ t \in FV^+(Holds(f,T)). \end{cases}$$

$$FV^+(MayInitiate(a\_type, f, Sit(t_s))) = \{t_s\} \;\&\; t_s \in FV(Date(e\_token, t_s)).$$

$$FV^+(Clipped(t_s,f,t)) = \{t_s,f,t\} \;\&\; \begin{cases} t_s \in FV(Date(e\_token, t_s)); \\ \\ f \in FV(MayInitiate(a\_typ, f, Sit(t_s))); \\ \\ t \in FV^+(Holds(f,T)). \end{cases}$$

Hence, EC1 is proved stable. Following this strategy it remains easy to show that the remaining axioms are also stable. Let us focus on the rules in Δ and take the definition of $MayInitiate(a\_type, f, Sit(t_s))$; for each positive or negative $HypHolds$ condition, it is easy to show:

$$FV^+(HypHolds(C,s)) = \{s\} \ \& \ s \in FV^+(MayInitate(a\_type, f, Sit(t_s)))$$

The residual case is that of formulae in $\phi(A\_type, f)$; but for them, we have defined earlier that they were stable, i.e., (by abuse of notation) $FV^+(\phi(A\_type, F)) = \emptyset$.

The same argument can be applied to each definition of $MayTerminate$ and $Possible$ in order to show that they are stable.

■

The immediate consequence of stability is termination.

Corollary 4. (Termination)

The Prolog interpretation of EC-allowed queries against a well-defined EC program is terminating without floundering.

Proof. Follows from Corollary 3, Lemma 5 and Theorem 3. ■

This result has important consequences. Since there are neither infinite nor floundered Prolog derivations (for EC-allowed queries), Stroetmann (1993) has shown, in a slightly different theoretical context, that the Prolog interpretation is sound and complete w.r.t. the completion of the program and, therefore, a stable model of Π.
In other words, for every query q and variable substitution σ such that $\Pi \models q\sigma$, there exists a finite derivation for $\Pi \cup q$ with computed answer substitution ξ such that $q\sigma \equiv q\xi$.

6. CONCLUSIONS

Similarities and differences between Event Calculus and Situation Calculus have been subject of much attention in the knowledge representation literature (Kowalski 1992; Pinto and Reiter 1993a, 1993b; Kowalski and Sadri 1994; Miller and Shanahan 1994).

On the one hand, Pinto and Reiter have successfully extended SC to treat time and actual actions; This work, on the other hand, has shown an improved version of EC which performs hypothetical reasoning on the effect of actions, one of the features that motivated the introduction of SC.

The extended EC has a clear semantics for a relevant class of theories, and the explicit representation of completeness assumptions is made possible by adopting Gelfond-Lifschitz's extended logic programs. On the computational side, the Prolog interpretation of EC-allowed queries was shown to be sound and complete.

These results suggest that logic programming is a framework of choice for action specification and formalization of reasoning about actions. Beside, the author finds the EC ontology of action types/action tokens more intutive than that of action types/situations of ESC.

Other aspects of EC remain open for further research, for instance the idea of of providing names for intervals of time bounded by partially-known events, which was present in the first formulation but lead to inconsistencies (Pinto and Reiter 1993b).

ACKNOWLEDGEMENTS

The author is indebted to Michael Gelfond for his careful advice and inspiration during the author's graduate studies and beyond.

Chitta Baral, Stefania Costantini and Gaetano Lanzarone have given valuable encouragement and advice.

Paulo Azevedo, Ann Gates and Angelo Montanari have kindly proof-read and discussed early versions of this paper.

This work was done mainly during the author's stay at the Computer Science Department of The University of Texas at El Paso, which is gratefully acknowledged.

REFERENCES

ALLEN J. F. 1984. Towards a general theory of action and time. Artificial Intelligence, **23**(5):123–154.

ALLEN J. F. and G. FERGUSON. 1994. Actions and Events in Interval temporal Logic. Journal of Logic and Computation, **4**(5):531–579.

APT K. R. and M. BEZEM. 1991. Acyclic Programs. New Generation Computing, **9**(3,4):335-363.

APT K. R. and D. PEDRESCHI. 1993. Reasoning about Termination of Pure Prolog Programs. Information and Computation, **16**(1):109–157.

APT K. R. and R. N. BOL. 1994. Logic programming and negation: a survey. Journal of Logic Programming, **19,20**:9–71.

BARAL C. and M. GELFOND. 1994. Logic programming for knowledge representation. Journal of Logic Programming, **19,20**:73–148.

CERVESATO I., A. MONTANARI A. and A. PROVETTI. 1993. On the Nonmonotonic Behaviour of Event Calculus for Computing Maximal Time Intervals. Interval Computations, **2**:83–119.

CHITTARO L., A. MONTANARI A. and A. PROVETTI. 1994. Skeptical and Credulos Event Calculi for Supporting Modal Queries. Proceedings of European Conference on Artificial Intelligence (ECAI'94), pages 361–365.

GELFOND M. and V. LIFSCHITZ. 1988. The Stable Model Semantics for Logic Programming. In *Kowalski, R.* and *K. Bowen* (editors) Logic Programming: proceedings of the Seventh international Conference, pages 1070–1080.

GELFOND M. and V. LIFSCHITZ. 1991. Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing, **9**(3,4):365–386.

GELFOND M., A. RABINOV and V. LIFSCHITZ. 1991. What are the Limitations of Situation Calculus? In *Boyer R. S. (editor),* Automated Reasoning. Essays in Honor of Woody Bledsoe, pages 167–181. Kluwer Academic.

GELFOND M. and V. LIFSCHITZ. 1993. Representing Actions and Change by Logic Programs. Journal of Logic Programming, **17**(2,3,4):301–355.

HANKS S. and D. MCDERMOTT. 1987. Nonmonotonic Logic and Temporal Projection. Artificial Intelligence, **3**(33):379–412.

KOWALSKI, R. and M. SERGOT. 1986. A Logic-based Calculus of Events. New Generation Computing, **4**:67–95.

KOWALSKI, R. 1992. Database Updates in the Event Calculus. Journal of Logic Programming, **12**:121–146.

KOWALSKI, R. and F. SADRI. 1994. The Situation Calculus and Event Calculus compared. Proceedings of the International Symposium on Logic Programming (ILPS '94), pages 539–553.

LIFSCHITZ V. and H. TURNER. 1994. Splitting a Logic Program. Proceedings of the International conference on Logic Programming (ICLP '94), pages 23–37.

MCCARTHY J. and P. HAYES. 1969. Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence, **4**:463–502.

Miller, R. and M. P. Shanahan. 1994. Narratives in the Situation Calculus. Journal of Logic and Computation, **4**(5):513–530.

Minker, J. (editor). 1988. Foundations of Deductive databases and Logic Programming. Morgan Kaufmann.

Pinto, J. and R. Reiter. 1993a. Adding a Time Line to the Situation Calculus. Working notes of AAAI symposium on logical formalizations of common sense reasoning. Austin(Tx).

Pinto, J. and R. Reiter. 1993b. Temporal Reasoning in Logic Programming: A Case for the Situation Calculus. Proceedings of the International Conference on Logic Programming (ICLP'93), pages 203–221.

Sergot, M. J. 1990. (Some topics in) Logic Programming in AI. Lecture notes of the GULP advanced school on Logic Programming. Alghero, Italy.

Shanahan, M. P. 1989. Prediction is Deduction but Explanation is Abduction. Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'89), pages 1055–1050.

Sripada, S. 1991. Temporal Reasoning in Deductive Databases. PhD Thesis in Computing. Imperial College, London. A presentation of this work can be found in the Proceedings of IJCAI'93, pages 860–865.

Stroetman, K. 1993. A Completeness Result for SLDNF-Resolution. Journal of Logic Programming, **15**:337-355.