

Implementing the Event Calculus in the DemoII System

Davide Martinenghi

Department of Computer Science, Roskilde University, P.O. Box 260, DK-4000 Roskilde, Denmark

Abstract

This report is about the practical uses of the Event Calculus and the problems that might arise when implementing the axioms that define it. It is assumed that the reader is familiar with a meta-logical system such as the DemoII System, which represents the context of this paper.

1 Introduction

Event Calculus is a formalism for reasoning about the effects of actions. The particular variant of event calculus we stick to is basically the one proposed in [SHA99], with the only difference that all actions are considered "instantaneous" (i.e. we use the two-arguments version of the predicate *Happens* only). An overview of the Event Calculus "dialects" is given in [SAD95]. Event Calculus is regarded here as a facility for abducing plans that make given goals provable (see [MIL96]).

The axiomatization used, which we will refer to as EC, is reported below (all variables in this report are universally quantified with maximum scope, unless differently stated):

- $$\begin{aligned} \text{(EC1)} \quad & \text{HoldsAt}(f,t) \leftarrow \text{InitiallyTrue}(f) \wedge \neg \text{Clipped}(T_0,f,t) \\ \text{(EC2)} \quad & \text{HoldsAt}(f,t_2) \leftarrow \text{Happens}(a,t_1) \wedge \text{Initiates}(a,f,t_1) \wedge t_1 < t_2 \wedge \\ & \neg \text{Clipped}(t_1,f,t_2) \end{aligned}$$

- (EC3) $Clipped(t_1, f, t_2) \leftrightarrow Happens(a, t) \wedge t_1 \leq t < t_2 \wedge Terminates(a, f, t)$
 (EC4) $\neg HoldsAt(f, t) \leftarrow InitiallyFalse(f) \wedge \neg Declipped(T_0, f, t)$
 (EC5) $\neg HoldsAt(f, t_2) \leftarrow Happens(a, t_1) \wedge Terminates(a, f, t_1) \wedge t_1 < t_2 \wedge \neg Declipped(t_1, f, t_2)$
 (EC6) $Declipped(t_1, f, t_2) \leftrightarrow Happens(a, t) \wedge t_1 \leq t < t_2 \wedge Initiates(a, f, t)$

EC defines a three-sorted language where the sorts are *actions* (a, a_1, a_2, \dots), *fluents* (f, f_1, f_2, \dots) and *time points* (t, t_1, t_2, \dots). Actions happen at precise time points and are (in this paper) instantaneous. Fluents are to be understood as properties of the world, which are subject to the common sense law of inertia and that may be affected by actions. Time points are ordered by the relation “ \leq ” and there exists a least time point T_0 ; $t_1 \leq t_2$ is a shorthand for $(t_1 < t_2 \vee t_1 = t_2)$. Next follows a brief description of the predicates used in EC.

- $HoldsAt(F, T)$ indicates that fluent F holds at time T ;
- $Happens(A, T)$ means that action A happens at time T ;
- $InitiallyTrue(F)$ (respectively $InitiallyFalse(F)$) indicates that fluent F holds (respectively does not hold) at time T_0 ;
- $Initiates(A, F, T)$ (respectively $Terminates(A, F, T)$) states that F starts to hold (respectively not to hold) after action A at time T ;
- $Clipped(T_1, F, T_2)$ (respectively $Declipped(T_1, F, T_2)$) means that fluent F is terminated (respectively initiated) between times T_1 and T_2 .
- “ $<$ ” is a partial order over time points.

In addition to EC, domain-dependent axioms are needed in order to define the behavior of $Happens$, $InitiallyTrue$, $InitiallyFalse$, $Initiates$ and $Terminates$; additional axioms are also needed for the definition of the “ $<$ ” relationship.

In the remainder of the report familiarity with the DemoII system is assumed (see [CHR98] for a description). DemoII is available on the World Wide Web, with full source code and examples¹, at the following address:

<http://www.dat.ruc.dk/software/index.html>.

¹ All the examples commented in this report are available at the DemoII web site.

2 Making abductions with the Event Calculus in the DemoII environment

EC axioms can be defined in the DemoII object language as an object module in which negation is represented by an additional argument (*yes* or *no*) that indicates whether the literal in question is a positive or a negative atom.

The following code shows how this can be done.

```
object_module( ec,
  \[
/*EC1*/ (holdsAt( yes, 'F', 'T') :- initiallyTrue('F'), clipped(no, t0, 'F', 'T')),
/*EC2*/ (holdsAt( yes, 'F', 'T2') :- happens('A', 'T1'), <('T1', 'T2'),
        initiates('A', 'F', 'T1'), clipped(no, 'T1', 'F', 'T2')),
/*EC3*/ (clipped( yes, 'T1', 'F', 'T2') :- happens('A', 'T'), <=('T1', 'T'),
        <('T', 'T2'), terminates('A', 'F', 'T')),
/*EC4*/ (holdsAt( no, 'F', 'T') :- initiallyFalse('F'), declipped(no, t0, 'F', 'T')),
/*EC5*/ (holdsAt( no, 'F', 'T2') :- happens('A', 'T1'), <('T1', 'T2'),
        terminates('A', 'F', 'T1'), declipped(no, 'T1', 'F', 'T2')),
/*EC6*/ (declipped( yes, 'T1', 'F', 'T2') :- happens('A', 'T'), <=('T1', 'T'),
        <('T', 'T2'), initiates('A', 'F', 'T'))
  ]).
```

In the same way, modules can be defined for expressing other needed properties and integrity constraints. For instance, the shorthand notation \leq can be defined in the following way:

```
object_module(less_or_equal,
  \[ (<=('A', 'A')),
    (<=('A', 'B') :- <('A', 'B'))
  ]).
```

We refrain from introducing transitivity of $<$ at this point, as such property would clearly be prone to formation of loops.

Being the deniable predicates adorned with an additional argument, the principle of non-contradiction easily translates to the idea that inconsistency is found whenever two such predicates, identical except for the first argument, can be proved².

² Such principle could be more compactly expressed with just one object clause:

```
bottom :- 'P' ^ [no|'AL'], 'P' ^ [yes|'AL']
```

Unfortunately the current implementation of DemoII (28/05/1999) does not allow object variables in the position of an argument list.

```

object_module(nonContradiction,
  \[
    (bottom :- clipped(no,'A','B','C'),clipped(yes,'A','B','C')),
    (bottom :- declipped(no,'A','B','C'),declipped(yes,'A','B','C'))
    %(bottom :- holdsAt(yes,'A','B'),holdsAt(no,'A','B'))
  ]).

```

Let us assume (at least for the examples shown in this report) that all the inconsistencies that one can find by looking at *HoldsAt* predicates can be found by looking at *Clipped* and *Declipped* predicates as well. In this way we can comment the third clause in the `nonContradiction` module and obtain a significant improvement of the performances.

Other inconsistencies that are likely to be entailed if no precautions are taken concern the $<$ relationships over time points: no time point can be less than itself and no time point is less than T_0 .

```

object_module(timeConsistency,
  \[
    (bottom :- <('T','T')),
    (bottom :- <('_',t0))
  ]).

```

We are now ready to make abductions about *Happens* facts (which we will call a *plan*), negated *Clipped* formulae and $<$ relations between time points, provided the system is fed with auxiliary information about the pertinent domain.

The patterns for these abducible parts of program can be expressed in terms of macro definitions:

```

easy_mk_bias_predicate(plan,
  \ happens(?A, ?T),
  (action(A), timepoint(T))).

easy_mk_bias_predicate(negative_conditions,
  \ (?P ^ [no, ?T1, ?F, ?T2]),
  ((P=clipped ; P=declipped), timepoint(T1), fluent(F), timepoint(T2))).

easy_mk_bias_predicate(timepoints,
  \ <(T1, ?T2),
  (timepoint(T1), timepoint(T2))).

```

timepoint, action and fluent are predicates defining the sorts of the language. timepoint simply requires its argument to be a constant:

```
timepoint(T) :- constant_(T).
```

action and fluent depend on the particular domain they refer to. For instance:

```
fluent(\\ onA).
fluent(\\ onB).
fluent(\\ onC).
action(\\ moveAB).
action(\\ moveBC).
```

A query to the system will typically consist of a known part of the world under consideration (the domain and the goal), and a part that needs to be abduced (the plan, the negative conditions and the time point ordering). The following predicate performs the required operations:

```
ec(Domain,Goal,\ (?P & ?N & ?T)) :-
  preds_(P,[happens]), plan(P),
  preds_(N,[clipped,declipped]), negative_conditions(N),
  preds_(T,['<']), timepoints(T),
  fails(\ (?Domain & ?T & timeConsistency), \\ bottom),
  fails(\ (?Domain & ec & less_or_equal & nonContradiction
    & ?P & ?N & ?T),\\bottom),
  demo(\ (?Domain & ec & less_or_equal & ?P & ?N & ?T), Goal),
  close_constraints(\ (?P & ?N & ?T)).
```

The search has been given more direction with some preds_ specifications, which are designed to interact with fails.

File 'EC.pl' implements the predicates and object modules described in this paragraph.

3 Examples

3.1 A unidirectional example - DiskABC

Standard event calculus examples rely on simplifications of the world such as the blocks world (see e.g. [MOY97]). The first example we describe here is based on a simple domain in which there are three pegs (A , B , C) and a disk, which is put on peg A at the initial time point T_0 .

From peg A we can move the disk to peg B , and from peg B to peg C .

The goal is to have the disk on peg *C* at time $T_F > T_0$.

In the event calculus terminology, the domain of this example can be sketched in term of two actions (*MovesAB* and *MovesBC*) and three fluents (*OnA*, *OnB* and *OnC*), whose interaction is made precise by the following object module.

```
object_module( domain,
  \[ (initiates(movesAB,onB,'T') :- holdsAt(yes,onA,'T')),
    (initiates(movesBC,onC,'T') :- holdsAt(yes,onB,'T')),

    (terminates(movesAB,onA,'T') :- holdsAt(yes,onA,'T')),
    (terminates(movesBC,onB,'T') :- holdsAt(yes,onB,'T')),

    initiallyTrue(onA),
    initiallyFalse(onB),
    initiallyFalse(onC)
  ]).
```

The sorts of the language are the same as the ones defined in §2:

```
fluent(\\ \\ onA).
fluent(\\ \\ onB).
fluent(\\ \\ onC).
action(\\ \\ moveAB).
action(\\ \\ moveBC).
```

A plan leading to the required goal can be found with the following query:

```
ec(\domain, \\holdsAt(yes,onC,tF),A).
```

which generates the result

```
A = \ ( [ (happens(movesBC,newConst0):-true),
  (happens(movesAB,newConst1):-true)] &

  [ (clipped(no,t0,onA,newConst1):-true),
  (clipped(no,newConst1,onB,newConst0):-true),
  (clipped(no,newConst0,onC,tF):-true)] &

  [(newConst1<newConst0:-true),
  (newConst0<tF:-true)] ? ;

no
```

The first part of the answer is the plan, which is clearly a sequence of two moves happening at some new time points *newConst0* and *newConst1* automatically generated by the system. The order of such time points is specified in the third

part of the answer by means of two $<$ relationships. The abduced negated *Clipped* formulae stand as conditions for preventing a given fluent from being terminated between the relevant time points. Notice that this solution is *exactly* the one we were looking for, as it only says that actions *MovesAB* and *MovesBC* must be performed in sequence and nothing more is said about the time points at which such actions happen (`newConst0` and `newConst1` stand virtually for all time points in the interval $[T_0, T_F]$, provided that `newConst0 < newConst1`). However, as we shall see later on, we were here both particularly careful in specifying a "unidirectional" problem that cannot "flounder" and lucky in getting such a clear answer. In fact, the actions are such that there is no chance to come back to an already visited "state". The solution found is therefore the good one and when asked for other solutions, the system correctly answers "no".

File 'DiskABC.pl' contains the implementation of this example.

3.2 The Shakey robot - revisited

In [SHA99] a domain is described in which a robot is supposed to be able to move from one room to another, provided that a door connects the two rooms. For example *Connects*(D_1, R_1, R_2) means that door D_1 connects rooms R_1 and R_2 . Going through a given door takes the robot from the original room to the one the door connects it to. Clearly, *Connects* is meant to be a symmetric predicate; however, in the current implementation of this example we had to comment symmetry out in order to avoid infinite loops (which are due to the search strategy generated by the combination of event calculus and the DemoII system). The resulting domain is therefore a rough simplification of the one defined in [SHA99], which, obviously, will not work in general. The robot is initially found in room R_3 and the goal is to have it in room R_6 at a certain time point $T > T_0$. The existing rooms and doors are described in the object module below. A convenient plan can in this case be found even without symmetry of room connectivity, and therefore this particular example will work.

```
object_module( domain,
  \[
    connects(d1,r1,r2),
    connects(d2,r2,r3),
    connects(d3,r2,r4),
    connects(d4,r3,r4),
    connects(d5,r4,r5),
    connects(d6,r4,r6),
  %   (connects('D','R1','R2') :- connects('D','R2','R1')),
```

```

(initiates(gothrough('D'),inroom('R1'),'T') :-
    connects('D','R2','R1'), holdsAt(yes,inroom('R2'),'T')),

(terminates(gothrough('D'),inroom('R'),'T') :-
    holdsAt(yes,inroom('R'),'T')),

initiallyTrue(inroom(r3))
]).

```

The sorts of the language are defined as follows³:

```

fluent(\\ inroom(?R)) :- constant_(R).
action(\\ gothrough(?D)) :- constant_(D).

```

The system can be queried as usual in the following way:

```

ec(\domain,\\holdsAt(yes,inroom(r6),t),A).

```

with the corresponding answer:

```

A = \ ( [(happens(gothrough(d6),newConst0):-true),
    (happens(gothrough(d4),newConst1):-true)] &

    [(clipped(no,t0,inroom(r3),newConst1):-true),
    (clipped(no,newConst1,inroom(r4),newConst0):-true),
    (clipped(no,newConst0,inroom(r6),t):-true)] &

    [(newConst1<newConst0:-true),
    (newConst0<t:-true)] ? ;

no

```

which contains the wanted solution (see §3.1 for comments about `newConst0` and `newConst1`).

File 'Shakey.pl' contains the implementation of this example.

3.3 Robots - another example

An example in [MIL98] describes a simple scenario of a robot that can go outside a room by moving through a door, which can be locked and unlocked with a key. Fluents and actions are as follows:

³ Event calculus becomes in this case a four-sorted language, as a new sort for "domain objects" (like rooms in `inroom` fluents and doors in `gothrough` actions) is introduced.

```

fluent(\\ haskey). % the robot has the key
fluent(\\ locked). % the door is locked
fluent(\\ inside). % the robot is inside
action(\\ pickup). % the robot picks up the key
action(\\ insert). % the robot inserts the key in the door
action(\\ gothrough). % the robot goes through the door

```

Picking up the key makes the robot holding the key.

Inserting the key in the door locks the door if the door was unlocked or unlocks it if it was locked, provided that the robot holds the key.

Going through the door takes the robot inside, provided that it is outside and the door is not locked. If the robot is inside and the door is not locked, going through the door terminates the condition of being inside. Initially the door is locked and the robot is inside. The robot picks up the key at time T_2 , inserts the key at time T_4 and goes through the door at time T_6 , with $T_2 < T_4 < T_6$.

```

object_module( domain,
  \[
    initiates(pickup,haskey,'T'),
    (initiates(insert,locked,'T') :-
      holdsAt(no,locked,'T'),holdsAt(yes,haskey,'T')),
    (initiates(gothrough,inside,'T') :-
      holdsAt(no,locked,'T'),holdsAt(no,inside,'T')),

    (terminates(gothrough,inside,'T') :-
      holdsAt(no,locked,'T'),holdsAt(yes,inside,'T')),
    (terminates(insert,locked,'T') :-
      holdsAt(yes,locked,'T'),holdsAt(yes,haskey,'T')),

    initiallyTrue(locked),
    initiallyTrue(inside),

    happens(pickup,t2),
    happens(insert,t4),
    happens(gothrough,t6),

    %%% TIMEPOINT ORDERING
    t2<t4, t2<t6, t4<t6, t2<t8, t4<t8, t6<t8
  ]).

```

The goal is to prove that the robot is not inside at time $T_8 > T_6$, which can be done with the following query:

```

ec(\domain,\\holdsAt(no,inside,t8),A).

```

The answer

```

A = \ ( [] & % plan

```

```

[(clipped(no,t0,locked,t4):-true),
 (clipped(no,t2,haskey,t4):-true),
 (declipped(no,t4,locked,t6):-true),
 (clipped(no,t0,inside,t6):-true),
 (declipped(no,t6,inside,t8):-true)] &

[]) % timepoints

```

provides an empty plan and no further time points, which means that the data the system is fed with are sufficient for proving the goal. The negative *Clipped* and *Declipped* formulae are, as usual, conditions for preventing a certain fluent from being initiated or terminated between the relevant time points.

Unfortunately, when asking for a second answer, the system goes into an infinite loop. (see also §5)

File 'key.pl' contains the implementation of this example.

4 Inaccuracy of the method

Let us consider the example from §3.1, but where we introduce in the domain an extra event happening at a time point $T_2 < T_F$.

```
happens(movesAB,t2), t2<tF
```

If we still ask the system for a plan that takes the disk to peg *C* at time T_F , we now get the following answers:

```

A = \ ( [(happens(movesBC,t2):-true),
 (happens(movesAB,newConst0):-true)] &
 [(clipped(no,t0,onA,newConst0):-true),
 (clipped(no,newConst0,onB,t2):-true),
 (clipped(no,t2,onC,tF):-true)] &
 [newConst0<t2:-true]) ? ;

A = \ ( [(happens(movesBC,newConst0):-true)] &
 [(clipped(no,t0,onA,t2):-true),
 (clipped(no,t2,onB,newConst0):-true),
 (clipped(no,newConst0,onC,tF):-true)] &
 [(t2<newConst0:-true),(newConst0<tF:-true)]) ? ;

A = \ ( [(happens(movesBC,newConst0):-true),
 (happens(movesAB,newConst1):-true)] &
 [(clipped(no,t0,onA,newConst1):-true),
 (clipped(no,newConst1,onB,newConst0):-true),
 (clipped(no,newConst0,onC,tF):-true)] &
 [(newConst1<newConst0:-true),(newConst0<tF:-true)]) ? ;

```

no

In the second answer, the system, as expected, invents a new time point $NewConst_0 > T_2$ at which the action of moving the disk from B to C happens.

In the first answer, something strange happens. The disk is moved from A to B at a new time point $NewConst_0 < T_2$ and then moved from B to C at time T_2 . So we have at time T_2 two actions happening simultaneously ($MovesAB$ and $MovesBC$), which is not prohibited by event calculus, the first of which has no effect, because its prerequisite (OnA) does not hold.

In the third case time point T_2 is completely disregarded. Actions $MovesAB$ and $MovesBC$ happen at two new time points ($NewConst_1$ and $NewConst_0$). This answer is somehow still acceptable, as, no matter where T_2 is placed wrt $NewConst_1$ and $NewConst_0$, at time T_F the disk will still be on peg C . The reason is that either the action happening at T_2 or the one happening at $NewConst_1$ has no effect, because its prerequisite (OnA) does not hold. Let us examine all the possible cases:

- 1) $T_2 < NewConst_1 < NewConst_0$. Action $MovesAB$ happening at $NewConst_1$ has no effect, as its prerequisite (OnA) does not hold (the disk is already on B).
- 2) $T_2 = NewConst_1 < NewConst_0$. Action $MovesAB$ happens simultaneously twice, but the effect is still that of moving the disk on peg B (and finally on peg C , from $NewConst_0$ on).
- 3) $NewConst_1 < T_2 \leq NewConst_0$. Symmetric of case 1. Now the action at T_2 has no effect.
- 4) $NewConst_1 < NewConst_0 < T_2$. The action at T_2 has no effect here because the disk is already on C .

Let us slightly modify the program by adding the fact

```
clipped(no, t0, onA, t2)
```

saying that the extra action we added at time T_2 cannot be clipped in $[T_0, T_2]$. Surprisingly enough, we still get the same three answers discussed in this paragraph, although the third one seems to give a clear contradiction: either the action at T_2 clips the one at $NewConst_1$ (subcase 1) or vice versa (subcases 3 and 4). The only possible case is $T_2 = NewConst_1$ (subcase 2), which is not very interesting, as the same action happens twice at the same time. Furthermore:

- we are only interested in introducing new time points if those time points are different from the ones already mentioned in the program (therefore subcase 2 should be dropped);
- contradiction in subcase 1 cannot be proved because there is missing information about the time point ordering. In particular, when $NewConst_1$ is introduced by the system, the ordering $t_2 <_{newConst_1}$ should be abduced (and the fact $t_0 < t_2$ should be present in the system as well). In this way, $clipped(yes, t_0, onA, newConst_1)$ can be derived; hence, the contradiction.
- analogously, contradiction in subcases 3 and 4 cannot be proved. In particular, the ordering facts $newConst_1 < t_2$ and $t_0 <_{newConst_1}$ should be abduced. In this way, $clipped(yes, t_0, onA, t_2)$ can be derived; hence, the contradiction.

These considerations, together with the fact that in our definition of time point ordering we dropped both transitivity and non-convergency⁴, take us to the conclusion that a specialized version of `close_constraints` should be used for these event calculus examples such that for each automatically generated time point:

- it should be different from all the other time points;
- the transitive closure of all possible orderings of the new time points wrt the others is abduced (in all possible ways on backtracking).

File 'Order.pl' contains the implementation of this example.

5 Conclusions

In this report we showed how easily event calculus problems are formulated with a meta-logical system such as DemoII. The search strategy set by this combination of formalisms (§2) proves good when dealing with simple cases (§3.1), but tends to be problematic in more critical circumstances (§3.3 and §4), so that simplifications are needed (§3.2).

Other strategies are possible (see e.g. the "hierarchical planning" described in [SHA99]; see also the abductive system described in [KAK98]), like, for example, a *first iterative deepening* based on the number of new actions or new time points introduced (typically to be minimal). A more appropriate

⁴ $\forall(t_1, t_2) ((t_1 \leq t_3 \wedge t_2 \leq t_3) \rightarrow (t_1 \leq t_2 \vee t_2 \leq t_1))$, which can be simplified here to $\forall(t_1, t_2 \in T) (t_1 < t_2 \vee t_2 < t_1 \vee t_1 = t_2)$, where T is the sort of time points and the \vee in the last formula are exclusive ors.

instantiating device than the default `close_constraints` needs then to be programmed. Furthermore, practical applications cannot in general be comfortably approached without feeding the search with some heuristics.

A suggested way for avoiding infinite loops, which typically arise because a proof step somehow generates itself, would be to program a meta-predicate that keeps a notion of "state" for a proof and discards some of the abductions made by `demo`. If we assume that the state for a given event calculus domain corresponds to the set of fluents that hold at a given timepoint, one could (mathematically) define a meta-predicate `no_cycles` in the following way:

$$\neg\exists(t_1, t_2, t_3 \in T, a \in A)(t_1 \leq t_2 < t_3 \wedge \text{Happens}(a, t_2) \wedge (\forall f \in F (\text{HoldsAt}(f, t_1) \equiv \text{HoldsAt}(f, t_3))))$$

where `T` is the sort of time points, `A` the sort of actions and `F` is the sort of fluents.

Clearly, this interesting technique does not easily translate into a feasible implementation, as each *HoldsAt* predicate requires a (heavy) event calculus proof. A more tractable reformulation is therefore called for.

In the course of the study of the combination of event calculus and DemoII, we found out that non-convergency (note 4) cannot be expressed in terms of the DemoII object language and the `demo` and `fails` predicates. The consequences were that, in some cases, new time points were not unambiguously disposed in the time ordering and therefore some of the erroneous solutions could not be discarded.

A call to `fails` with a clause

$$\backslash (\text{bottom} :- 'T1' < 'T2' , 'T2' <='T1') \quad (\text{c1})$$

would simply check that all the existing `<` relationships do not violate the constraint, but would not do it for all the inferable ones. The constraint we should be able to express is of this kind:

$$\backslash (\text{bottom} :- \text{not}('T2' <='T1' ; 'T1' < 'T2')) \quad (\text{c2})$$

which corresponds to a sort of "double negation" of (c1), which we are not able to express with a single call to `fails`.

Bibliography

- [CHR98] Christiansen, H. *Automated reasoning with a constraint-based metainterpreter*, March 4, 1998, The Journal of Logic Programming 37 (1998) 213-254.
- [KAK98] Kakas, A. C., Michael, A., and Mourlas, C., *ACLP: Abductive Constraint Logic Programming*, submitted draft, J. Logic Programming 1998, 1-45
- [MIL96] Miller, R., *Notes on Deductive and Abductive Planning in the Event Calculus*, July 1996
- [MIL98] Miller, R., and Pinto, J. *Temporal Languages for Reasoning About Action*, November 7, 1998.
- [MOY97] Moyle, S., and Muggleton, S., *Learning Programs in the Event Calculus*, p. 205-212 in Nada Lavrac, Saso Dzeroski (Eds.): *Inductive Logic Programming, 7th International Workshop, ILP-97*, Prague, Czech Republic, September 17-20, 1997, Proceedings. Lecture Notes in Computer Science, Vol. 1297, Springer, 1997, ISBN 3-540-63514-9
- [SAD95] Sadri, F., and Kowalski, R., *Variants of the Event Calculus*, ICLP 95
- [SHA99] Shanahan, M. *Reinventing Shakey*, 1999, Tracking Number: A319, to appear.