**Programming Research Group**

# EXPERIMENTS IN LEARNING EVENT CALCULUS PROGRAMS

Stephen Moyle
Stephen Muggleton

PRG-TR-23-97

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

**Abstract**

The event calculus is a formalism for reasoning about actions and change in dynamic systems. It has been used in diverse areas including planning and communications protocol specification. Writing event calculus programs requires the construction of domain specific axioms (DSAs) - a programming task which is non-trivial, and one that hinders the broader use of the event calculus. This work demostrates that such axioms can be learned from temporal observations using Inductive Logic programming (ILP) techniques, in particular theory revision. The theory of logical back-propagation as a mechanism for theory revision is described and its implementation in the ILP system Progol is used here. A version of the event calculus is developed, as is a method for translating temporal observations to a form suitable for Progol. These methods were used to investigate learning DSAs for the traditional AI blocks world. In the experiments Progol, utilising logical back-propagation, learned correct DSAs. These results provide encouragement and highlight the possibility of discovering causal relationships from data in temporal databases, and also learning the domain specific knowledge necessary in the development of plans.

# 1 Introduction

The amount of data recorded electronically continues to grow. It is common for the time that the data was collected to be recorded along with the data, therefore classifying it as temporal. For example hospital databases record patient treatment histories, and banks keep customer account transaction records. In both science and commerce the requirements are to analyse and understand patterns in temporal data. However, increasingly human analysts are overwhelmed by the quantities of data involved. An automated method of generating knowledge from temporal data is required.

The situation calculus [McCarthy and Hayes, 1969] and event calculus [Kowalski and Sergot, 1986] are two declarative formalisms that represent time implicitly by describing the relationships between actions and change. Both formalisms provide a notion of default persistence which enable the derivation of properties that are unaffected by actions. Such reasoning about actions and change is considered to be a form of common sense reasoning [McCarthy, 1959].

Event calculus programs are typically normal logic programs [Lloyd, 1993] with a fixed clausal structure. An event calculus program $T_{EC}$, can be considered as being composed of two non-intersecting sets of clauses; (1) the core axioms $T_{Core}$, a fixed, immutable set of clauses that provide the logical basis for inferring the state of the event calculus program at any given time[1], and (2) the domain specific axioms $T_{DSA}$ which determine the derivable statements or *behaviour* of program. For any moderately complex dynamic system, providing the domain specific axioms (DSAs) to represent the *correct* system behaviour is difficult. This is hardly surprising as the generation of DSAs is a programming task.

The structural components of an event calculus program are common in the field of *theory revision* [De Raedt, 1992]. De Raedt (p.10 ) outlines the problem of theory revision as a process of updating a theory $T$ which comprises two sets of clauses $T_{fix}$ and $T_{mod}$. $T_{fix}$ represents the fixed components of the theory which may not be changed by the revising system, whilst attempting to revise $T_{mod}$ to form $T'_{mod}$ such that the resulting theory $T' = T_{fix} \cup T'_{mod}$ does not violate any integrity constraints $I$, and the evidence $E$ that was previously not derivable from $T$ is derivable from $T'$. The learning of event calculus programs from examples can be stated as:

- Given:
  - a set of integrity constraints $I$
  - a theory $T_{EC} = T_{Core} \cup T_{DSA}$ (where $T_{DSA}$ may be empty).
  - evidence $E$ (observations of dynamic behaviour)
- Find: a theory $T'_{EC} = T_{Core} \cup T'_{DSA}$ obtained by applying transformations on $T_{DSA}$ such that $T'_{EC}$ explains $E$ and is consistent with $I$.

Any ILP technique applied to learning event calculus programs must contend with the nature of the evidence or observations provided relative to the clauses to be constructed for the target theory $T$. It is common within ILP for single predicate learning to produce definitions for predicates that have the same heads as those of the examples[2]. In a single predicate learning setting, for each clause $C_i \in T$, there is a set (possibly empty) of examples $E_i$ such that the predicate symbol and arity ($p$) of the head of the clause is the same as that of the examples $p(C_i) = p(E_i)$. This is not

[1] The event calculus core axioms also provide the computational framework for deriving statements about the event calculus program when it is executed, for example, by a Prolog system.

[2] For instance, learning a definintion for `member/2` an example might be `member(2,[1,2,3])`.

the case for the event calculus where the observations are recorded by different predicates from those used to define legitimate DSAs ($p(C_i) \neq p(E_i)$).

The theory revision mechanism that is applied to learning event calculus programs in this paper is provided by logical back-propagation. Logical back-propagation makes use of a general form of inverting entailment [Muggleton, 1995]. Examples of an observational predicate $P$ are used to augment the definition of a related theoretical predicate $Q$. The implementation makes use of mechanisms from Stickel's Prolog Technology Theorem Prover [Stickel, 1992]. Derivation of the bottom clause is based on a relevance map which involves constructing a calling diagram among predicates. The theoretical and observational predicates involved in the generalisations are communicated by the user to CProgol5.0 by way of mode declarations.

The contribution of this paper is the extension and application of ILP techniques to automatically generate event calculus programs for dynamic systems from temporal observations of system properties. It is organised as follows - the event calculus is introduced and a version suitable for ILP developed. The theory of logical back-propagation as a method for theory revision is presented in section 3. This is followed by experimental results of applying logical back-propagation to the induction of DSAs. Section 5 presents a discussion of the findings and section 6 suggests areas for future research. Readers unfamiliar to the event calculus can find a familiar AI example, the blocks world, represented as an event calculus program in appendix.A

## 2 The Event Calculus

The event calculus was developed by Kowalski and Sergot [Kowalski and Sergot, 1986] as a general approach to representing and reasoning about events and their effects in a logic programming framework. An event calculus program allows a dynamic system to be modelled so that future states of the system can be determined, thus enabling the derivation of system values that *hold at* future times based upon events that have *happen*ed at previous times. Readers unfamiliar with the event calculus should refer to the blocks world example in the appendix.

The event calculus utilises default persistence, formalising the notion that properties remain constant unless effected by the action of an event. Change is mediated solely through the action of events, where events are related to system properties if they initiate (or terminate) periods in which the properties hold (or do not hold).

A burden of all event calculi is that they require humans to provide domain specific axioms (DSAs) for definitions of predicates that encode the dynamics of the domain - a process that can be difficult[3]. This difficulty is exacerbated by the non-monotonicity of the formalism. One way to reduce this burden is to induce predicate definitions for the DSAs from observed time traces of property values from an existing dynamic system.

The event calculus is being used to describe an increasing number of domains including an Air Traffic Control system [Sripada, 1995], and a decision support system for vascular surgery [Abeysinghe, 1993]. Researchers in Leuven have successfully implemented a sliding window protocol with go-back-n in the event calculus [Denecker *et al.*, 1996], from which they were able to prove that the communications system as modelled was deadlock free. Importantly the event calculus has been used for the basis of solving planning problems (e.g. [Eshghi, 1988], [Kowalski, 1979], [Kakas and Michael, 1995], [Van Belleghem *et al.*, 1994]).

[3]This has been cofirmed in separate discussions with Marc Denecker and Murray Shanahan.

## 2.1 Components of Event Calculus programs

The ontological primitives used by the event calculus are *events* and *fluents*. Fluents[4] are time-varying properties of the world, which may have different values at different times. For example the fluent *on(light)* may be true (hold) at time 5 and false (not hold) at time 7. Events are instantaneous occurrences which may cause change in the values of fluents. A set of events is called a *narrative*. A *dependent fluent* is one that is assumed to rely on the properties of the world around it, while *independent fluents* are assumed fiat.

A single fluent *observation* is the value of a fluent at an instant in time. A *fluent trace* is the collection of fluent values for the same fluent at different times. An *observation trace* is a collection of fluent traces (possibly also narratives). *Trace constraints* represent the periods in which the values of the fluent trace must remain constant. *Up* and *down* transitions are the time points where fluents change value, and *up* and *down* data is data from fluent traces that are in the true and false state respectively.

The axiom structure of the event calculus itself can be thought of as consisting of two components. The first component is the set of *core axioms*, that are immutable and provide an encoding of default persistence, and the second component is the set of *domain specific axioms* which encode the causal relations between events and the properties of the system being modelled. (Refer to the example in the appendix for illustration.)

## 2.2 The Flip-Clip event calculus

The original event calculus [Kowalski and Sergot, 1986] is used less frequently, and is replaced by the Simplified Event Calculus (SEC) [Sadri and Kowalski, 1995]. However, the SEC presents difficulties learning DSAs using ILP. This section outlines the difficulties and develops a version of the event calculus called the Flip-Clip event calculus (FC EC).

One difficulty concerns the derivation of fluent properties that do not hold at particular times from an event calculus program. In the SEC a fluent `f` is considered true at time `t` if `holds_at(f,t)` is derivable. Should it not be derivable then it is considered as false. Deriving false fluent facts can be problematic in Prolog. This can be overcome by reifying the truth value of the fluent is within the `holds_at` predicate. Thus `holds_at/2` becomes `holds_at(Fluent,State,Time)` where the extra argument `State` can be `+` or `-` (representing the truth and falsity of the fluent value respectively). For example the fluent trace of a light might be represented in the event calculus as {`holds_at(on(light),-,0)`, `holds_at(on(light),+,1)`, `holds_at(on(light),-,2)`}. A domain constraint `:-clipped(0,on(light),+,5)` represents that the light must remain on from time point 0 to time point 5.

It is common in ILP to concentrate on learning a single predicate definition. However, in the SEC fluent properties can be triggered by both initiating events and terminating events. The second alteration to the SEC is the replacement of definitions for `initiates/3` and `terminates/3` predicates (i.e. the domain specific axioms) with a single predicate `flips/4`, with an intended meaning that an event flips the fluent into the specified truth state. For example the event of opening a switch, *open(switch)*, at some time cause the truth state of the fluent *on(light)* to become + in the successor time point. That is to say that the definitions for `initiates(Event,Fluent,Time)` are translated to `flips(Event,Fluent,+,Time)`, and the definitions for `terminates(Event,Fluent,Time)` to `flips(Event,Fluent,−,Time)`.

[4]McCarthy [McCarthy, 1959] states that the term *fluent* was used by Newton to describe a physical quantity that depends on time.

The complete definition of the core Flip-Clip event calculus (FC EC) axioms are as follows (c.f. EC1, EC2, and EC3 of the SEC as presented in appendix A).

(FC1)
```
holds_at(Fluent,State,Time2) :-
    happens(Event,Time1), Time1<Time2,flips(Event,Fluent,State,Time1),
    not clipped(Time1,Fluent,State,Time2).
```

(FC2)
```
clipped(Time1,Fluent,State1,Time2) :-
    happens(Event,Time), Time1=<Time, Time<Time2,
    switch(State1,State2), flips(Event,Fluent,State2,Time).
```

(FC3)
```
holds_at(Fluent,State,Time) :-
    given(Fluent,State,Time0), not clipped(Time0,Fluent,State,Time).
```

The body literal `switch/2` in FC2 defines what it is for a fluent to change truth state (i.e. `switch(+,-)` and `switch(-,+)`). The initial values of fluents are specified using `given/3` facts. Narrative information about events are specified using the `happens/2` predicate.

SEC domain specific axioms can be translated simply to corresponding FC EC clauses. For example, the blocks world initiates axiom BDSA1 from the appendix becomes

```
flips(move(X,Y),on(X,Y),+,Time) :-
    holds_at(clear(X),+,Time), holds_at(clear(Y),+,Time),
    diff(X,Y),diff(X,table).
```

The FC EC is not a radical departure from other versions of the Event Calculus. A similar notion of truth reificaion can be found in Kakas and Miller's Language $\mathcal{E}$ [Kakas and Miller, 1997]. The format of the flips axiom is that the occurrence of a particular event will only cause a fluent to enter a truth state provided that at the time of the event all preconditions hold. The event calculus of the FC EC provides a framework whereby ILP techniques can be used to learn domain specific axioms in the predicate `flips/4` from examples of fluent temporal values (`holds_at/3`).

## 3 Theory Revision by Logical Back-propagation

The theory revision mechanism that is applied to learning event calculus programs in this paper is provided by logical back-propagation. Logical back-propagation is a form of generalised inverses entailment and is implemented in CProgol5.0

### 3.1 Generalised inverse entailment

The general problem of ILP can be specified as: *given* background knowledge $B$ and examples $E$ *find* the simplest consistent hypothesis $H$ such that

$$B \wedge H \models E.$$

In the case that both $E$ and $H$ are single Horn clauses then by inverse entailment [Muggleton, 1995] it is possible to generate the conjunction of all ground literals that are true in all models of $B \wedge \overline{E}$, denoted by $\overline{\bot}$.

$$B \wedge \overline{E} \models \overline{\bot} \models \overline{H}.$$

Logical back-propagation makes use of a general form of inverting entailment, whereby examples of an observational predicate $P$ are used to augment the definition of a related theoretical predicate $Q$. In the following example, examples of sentences (predicates for $s$) are used to augment a definition for noun phrase ($np$).

**Example 1 (Natural language processing)** *Given background knowledge*

$$B = \begin{cases} s(A,B) \leftarrow np(A,C), vp(C,D), np(D,B) \\ np(A,B) \leftarrow det(A,C), noun(C,B) \end{cases}$$

*example* $E = s([the, nasty, man, hit, the, dog], [])$*, and a prior hypothesis* $H = np(A,D) \leftarrow det(A,B), adj(B,C), noun(C,D)$ *then by inverse entailment*

$$\perp = s(A,B); np(A,C) \leftarrow det(A,D), adj(D,E), noun(E,C), vp(C,F), np(F,B)$$

The generation of $\overline{\perp}$ in the above example requires that there are no models for both the example ($s$) and the head of the hypothesis clause ($np$). However, the derivation of this type of negative information using a Horn clause theorem prover presents obvious difficulties.

### 3.2 Implementation

To overcome the need to derive negative information, the implementation of logical back-propagation makes use of mechanisms from Stickel's Prolog Technology Theorem Prover [Stickel, 1992]. Clauses are constructed to provide definitions for negated literals. For example, for the clause $p \leftarrow q$, the extra clause $\tilde{}q \leftarrow \tilde{}p$ is added, allowing the possibility of $\tilde{}q$ being derived using a Prolog interpreter. This construction or Stickelisation, is the basis for full clausal theorem proving.

Not all clauses of a theory are Stickelised when implementing generalised inverse entailment. A relevance map based on the calling diagram among predicates is used to determine the additional Stickel clauses required. Figure 1 shows the relevance map for the natural language processing example. The Stickel clause required to generate bottom for the example is $\tilde{}np(A,C) \leftarrow \tilde{}s(A,B), vp(C,D), np(D,B)$. This, combined with clauses defined to ensure input-output connectedness of predicates, enables the derivation of $\tilde{}np$ for the generation of $\overline{\perp}$.

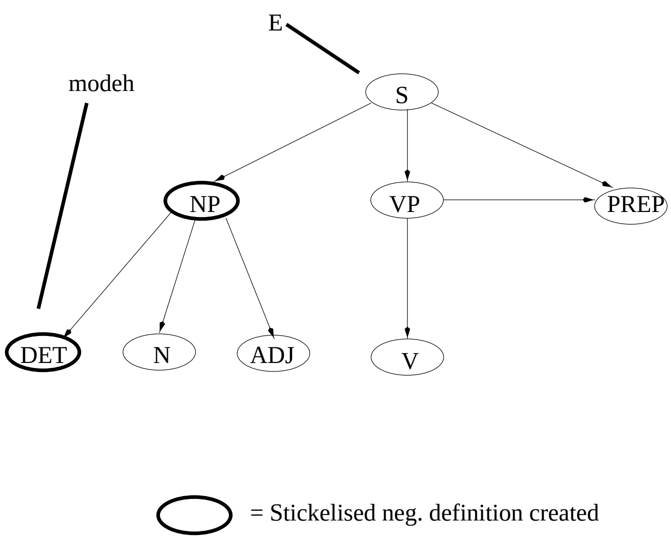


Figure 1: Relevance map for natural language processing example.

The theoretical and observational predicates involved in the generalisations are communicated by the user to CProgol5.0 by way of mode declarations. CProgol5.0 implements multi-predicate, multi-lattice learning using positive and negative examples.

# 4 Learning Event Calculus Programs

The aim of the experiments was to induce domain specific axioms for event calculus programs of dynamic systems from observed system properties using ILP techniques. The general procedure that was used is briefly described below (also summarised in figure 2).

1. Select a complete and working event calculus program
2. Generate a narrative of events for the event calculus program.
3. Execute the event calculus program from an initial state and record time traces for all observables.
4. Remove the domain specific axioms.
5. Use the following components as input to the ILP system Progol.
   - Core event calculus axioms
   - Narrative of events
   - Some (possibly all) of the fluent traces
   - Other background knowledge
6. Induce domain specific axioms using Progol5.0

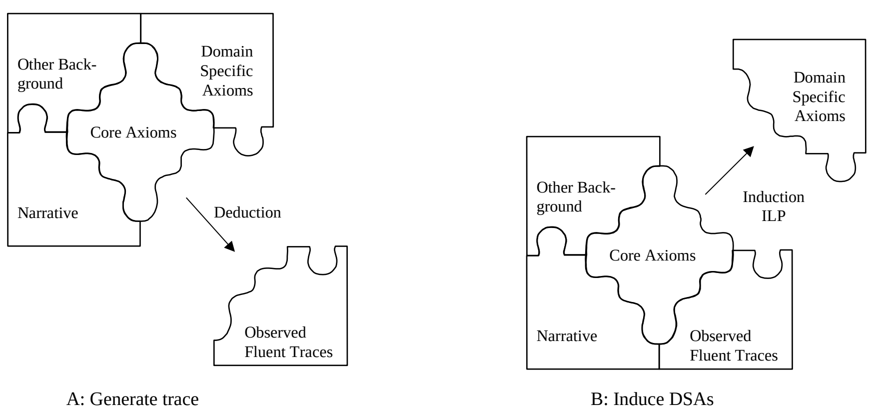


Figure 2: Experimental method: A. generate observed fluent traces; B. remove domain specific axioms and induce them from the observations.

## 4.1 Method

Previous experiments[5] highlighted difficulties in applying single predicate learning to DSAs. One feature of DSAs common to many event calculus programs is the coupling of DSAs. Typically there will at least one clause for initiating a fluent value, and one clause for terminating the

[5]Earlier experiments are not reported here.

fluent value (e.g. BDSA1 and BDSA2 in the appendix). However, should one clause be removed, then the behaviour generated by the event calculus program for the fluent will not be correct, leading to obvious difficulties when attempting single predicate learning. These difficulties were overcome by the following transformation procedure. The independent fluent traces were provided as background knowledge in the form of `basic/3` facts, from which the original observations (as `holds_at/3` facts) can be recovered via the additional background clause:

```
holds_at(Fluent,State,Time)  :-  basic(Fluent,State,Time).
```

The induction problem is uncoupled into two separate problems, one for learning transitions from the "-" to the "+" state for the dependent fluent (or *up* transitions), and one for transitions from "+" to "-" (*down* transitions). For the case of inducing DSAs for up transitions, the `holds_at/3` facts in the down state from the dependent fluent observational trace are translated to `basic/3` facts. Constraints are added to ensure that the *down* information remains available for learning *up* transitions - for every period of down time, a constraint of the form `:-clipped(Time1,Fluent,-,Time2).` is added. Such constraints state that the fluent is not clipped from the end of time point 1 until the end of time point 2. Analogous translations are made in preparation for inducing *down* DSAs.

## 4.2 Experiment - Shanahan's Blocks World

The blocks world of Shanahan is presented in appendix A (see figure 3). The DSAs for the blocks world can be easily translated to clauses in the FC EC. For example BDSA1 and BDSA2 become

```
flips(move(X,Y),on(X,Y),+,Time) :-
  holds_at(clear(X),+,Time), holds_at(clear(Y),+,Time),
  diff(X,Y), diff(X,table).
flips(move(X,Y),on(X,Z),-,Time) :-
  holds_at(clear(X),+,Time), holds_at(clear(Y),+,Time),
  holds_at(on(X,Z),+,Time), diff(X,Y), diff(Y,Z).
```

The complete FC EC program (i.e. core axioms and blocks world domain specific axioms) enables queries about the state of the blocks at different times. For example, the following Prolog query/responses were a sample of traces obtained.

```
setof(T,holds_at(on(block_c,table),+,T),Ts).
Ts = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
setof(T,holds_at(on(block_b,block_c),+,T),Ts).
Ts = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
setof(T,holds_at(clear(block_a),+,T),Ts).
Ts = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
setof(T,holds_at(on(block_a,block_d),+,T),Ts).
Ts = [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

The results of all queries similar to the above were the source of observed trace information used for induction.

### 4.2.1 Data for induction

The data provided to Progol for generating domain specific axioms includes the core event calculus axioms. A sample of the data for learning is presented in appendix B.

### 4.2.2 Results and discussion

The examples of `holds_at/3` are used in conjunction with logical back-propagation and the core axioms to generate a most specific clause. An example of a most specific clause generated by Progol from the *up* transition data set is:

```
flips(move(A,B),on(A,B),+,C) :-
    holds_at(clear(A),+,C), holds_at(clear(B),+,C), diff(A,block_b),
    diff(A,block_d), diff(A,table), diff(A,block_c),
    diff(A,B), diff(B,block_b), diff(B,table),
    diff(B,block_a), diff(B,block_c), diff(B,A).
```

Similar most specific clauses were generated from the *down* transition set e.g.

```
flips(move(A,B),on(A,C),-,D) :-
    holds_at(on(A,C),+,D), holds_at(clear(A),+,D), holds_at(clear(B),+,D),
    diff(A,B), diff(A,C), diff(A,E),
    diff(A,F), diff(B,A), diff(B,C),
    diff(B,E), diff(B,F), diff(C,A),
    diff(C,B), diff(C,E), diff(C,F),
    holds_at(on(B,F),+,D), holds_at(on(C,E),+,D), holds_at(on(E,F),+,D),
    holds_at(clear(F),+,D), diff(E,A), diff(E,B),
    diff(E,C), diff(E,F), diff(F,A),
    diff(F,B), diff(F,C), diff(F,E).
```

**Hypothesis generated for *up* transitions of the fluent `on/2`** The hypothesis generated by Progol for the up transitions of the fluent `on/2` was as follows.

(Induced Up)

```
flips(move(A,B),on(A,B),+,C) :-
    holds_at(clear(A),+,C), holds_at(clear(B),+,C),
    diff(A,table), diff(A,B).
```

This clause is identical (modulo variable renaming) to the original domain specific axiom for the up transition. This clause includes literals that relate both temporal and static relationships between the objects in the domain.

**Hypotheses generated for *down* transitions of the fluent `on/2`** The hypotheses generated by Progol for the up transitions of the fluent `on/2` were as follows.

(Induced Down 1)

```
flips(move(A,B),on(A,C),-,D) :- holds_at(on(B,C),+,D).
```

(Induced Down 2)

```
flips(move(A,B),on(A,C),-,D) :-
    holds_at(on(A,C),+,D), holds_at(clear(A),+,D),
    holds_at(clear(B),+,D).
```

The first hypothesis Induced Down 1 was not part of the original set of domain specific axioms for the blocks world program. This clause trivially states that moving a block to another will not make the first be on a third if there is already the second on the third. The second hypothesis generated although not identical to the original domain specific axiom for the up transition does imply the original.

## 5 Discussion

The experimental evidence presented demonstrates that it is possible to induce domain specific axioms for event calculus programs using temporal observations of system properties. Using logical back-propagation, Progol was able to learn the domain specific axioms to describe the dynamics of the blocks world from observations of the world over time.

The development of the Flip Clip event calculus introduces two main advantages for the induction of event calculus programs using ILP. First, the reification of the truth of a fluent allows negative fluent information to be derived. Although this is not a new idea (see [Kakas and Miller, 1997]), it removes the need to implement some form of constructive negation (for example see [Chan, 1988]). Second, the reification allows the clauses for both initiating and terminating axioms to be translated into a single definition (`flips/4`), allowing single predicate learning techniques to be applied.

The domain specific axioms in event calculus programs that refer to specific fluents often contain clauses for the initiation and termination of fluent properties for correct dynamic modelling. The removal of one set of these clauses will result in the derivation of incorrect fluent values, caused by the non-monotonicty of the formalism[6]. The application of single predicate learning necessitates that only clauses for one transition be learned at a time. This produces difficulties in over generalisation and the incorrect generation of intermediate theories. By careful use of constraints derived from observations, the non-monotonicity can be avoided for the purpose of induction.

The immutable core event calculus axioms play an important role in the theory revision process. These axioms participate in the induction in two key areas, (1) for generating a most specific hypothesis, and (2) searching the subsumption lattice of possible clauses. First, logical back-propagation is used in the generation of the most specific hypothesis where examples presented for generalisation are the fluent traces or sets of ground `holds_at/3` facts. Attempt to generalise `holds_at/3` does not succeed directly as no `modeh` definition for `holds_at/3` exists. The logical back propagation algorithm searches for a predicate definition which has a `modeh` definition and it also has `holds_at/3` in its ancestor set. In this example the first predicate to conform to these requirements is `flips/4` which appears in the body of the core axiom FC1. This core axiom, and all those that have heads in the ancestor set, have a Stickel transformation applied, and the negated atom (`~flips/4`) is tested for derivability. If it is derivable then the bindings returned are used as part of the start set for inverse entailment (see [Muggleton, 1995]), leading to the generation of the most specific clause. In producing a most specific clause, logical back propagation translates a single item of observation data to a saturated domain specific axiom. Although a single observation is used to generate a most specific hypothesis all of the observations (and constraints) play a role in the search for a consistent, compressive clause.

Second, the core event calculus axioms aid in the constraining the search for the most compressive hypothesis. The search for a consistent clause $H$, where $\Box \preceq H \preceq \bot_{(flips, holds\_at)}$, makes use of the second core axiom (FC2) and the ground trace constraints of the form `:-clipped(Time1,Fluent,-,Time2)`. The second core axiom contains `flips/4` as a body literal. Every hypothesis clause that is considered in the subsumption lattice is checked for constraint satisfaction via the second core axiom. With no negative examples, trace constraints are included to prevent over generalisation.

[6]Default persistence is a key feature of the event calculus.

### 5.1 Relation to other work

Although the possibility of learning causal rules in the event calculus is mentioned by Kakas and Miller ([Kakas and Miller, 1997]), this work is the first to achieve it.

Previous researchers have studied ILP in temporal domains. Bratko and colleagues [Bratko *et al.*, 1992], [Bratko, 1994] describe the application of inducing qualitative models [Kuipers, 1986] of dynamic systems from example behaviours. A qualitative model was also used as background knowledge to learn temporal diagnostic rules for diagnosing satellite faults [Feng, 1992]. The accuracy of diagnosis for this problem was increased over a static, snap shot of the system, by considering temporal aspects of the domain. Sammut and Hume have studied the learning of action sequences in a simulated robot world [Hume and Sammut, 1992]. ILP systems have also been applied to the problem of learning the rules for turning an aircraft from simulated F16 flight data [Srinivasan and Camacho, 1995], and to learning traffic flow rules [Eastmond, 1996].

## 6 Concluding remarks

This paper has described the event calculus and developed a modified version suited to ILP. Logical back-propagation was presented as a mechanism for performing theory revision. The clausal structure of the event calculus combined with logical back-propagation was used successfully to learn simple event calculus programs from observations of desired program behaviour.

It is possible that these methods could be extended for use in intelligent planning applications. The learning system could be used to monitor the actions of a goal directed agent, and the resulting transformations to the domain. From these observations the learner could induce domain specific temporal axioms to describe the execution of the observed plan. Plans for a new goal could then be generated by an abduction system whose task would be to use the DSAs and the initial state to produce a partially ordered set of events that achieve the goal (e.g. see [Eshghi, 1988] and [Kakas and Michael, 1995]).

Future work should consider domains where the effects of events are delayed. Another alternative area for future research would be the extension of the event calculus formulation to include continuous change (e.g. [Shanahan, 1990]) which may involve the use of constraint logic programming (for a survey see [Jaffar and Mahar, 1994]). Learning rules about action and change may also be possible using an analogous logic programming implementation of the situation calculus[7].

A more sophisticated method for avoiding the non-monotonicty is required. One option, albeit inefficient, would be to generate all possible bottom clauses for both *up* and *down* transitions, and perform some form of parallel communicating search for compressive hypotheses.

This work has focused on the case where events and fluents are observed and labeled as such. It is possible that for some data sets it may be difficult to pre-classify which of the observations are events and which are fluents. This could lead to an investigation of an event calculus formalism based upon describing ramifications similar to that used in [Abeysinghe, 1993].

[7] This was suggested by Murray Shanahan.

## References


[Abeysinghe, 1993] Geetha Kalyani Abeysinghe. *Event Calculus to Support Temporal Reasoning in a Clinical Domain.* Ph.D. thesis, University of Southampton, 1993.

[Bratko, 1994] I. Bratko. Machine Learning and Qualitative Reasoning. *Machine Learning*, 14(3):305-312, 1994.

[Bratko *et al.*, 1992] I. Bratko, S. Muggleton, and A. Varsek. Learning Qualitative Models of Dynamic Systems. In S. Muggleton, editor, *Inductive Logic Programming*, p. 437-452. Academic Press, London, 1992.

[Chan, 1988] David Chan. Constructive Negation based on the Completed Database. In *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, 1988. MIT Press.

[De Raedt, 1992] Luc De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach.* Academic Press, London, 1992.

[Denecker *et al.*, 1996] Marc Denecker, Kristof Van Belleghem, Guy Duchatelet, Frank Piessens, and Danny De Schreye. A Realistic Experiment in Knowledge Representation in Open Event Calculus: Protocol Specification. In M. Maher, editor, *Proceedings of JICSLP'96, the Joint International Conference and Symposium on Logic Programming*, p. 170-184. Bonn, Germany, 1996. MIT Press.

[Eastmond, 1996] Matthew Eastmond. *ILP applied to traffic flow problems on the M25.* M.Sc. thesis, Oxford University, 1996.

[Eshghi, 1988] K Eshghi. Abductive Planning with Event Calculus. In *Proceedings of the Fifth International Conference on Logic Programming*, p. 562 - 579. 1988.

[Feng, 1992] C. Feng. Inducing Temporal Fault Diagnostic Rules from a Qualitative Model. In S. Muggleton, editor, *Inductive Logic Programming*, p. 473-494. Academic Press, London, 1992.

[Hume and Sammut, 1992] D. Hume and C. Sammut. Applying Inductive Logic Programming in Reactive Environments. In S. Muggleton, editor, *Inductive Logic Programming*, p. 539-549. Academic Press, London, 1992.

[Jaffar and Mahar, 1994] J. Jaffar and M.J. Mahar. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19,20:503-518, 1994.

[Kakas and Michael, 1995] A. C. Kakas and A. Michael. Integrating Abductive and Constraint Logic Programming. In *Proceedings of the 12th International Conference on Logic Programming*, Japan, 1995.

[Kakas and Miller, 1997] Antonis C. Kakas and Rob S. Miller. A Simple Declarative Language for Describing Narratives with Actions. *The Journal of Logic Programming*, Vol 31(1-3) (Special Issue on Reasoning about Action and Change):157 - 200, 1997.

[Kowalski, 1979] Robert Kowalski. *Logic for Problem Solving.* Elsevier, North Holland, 1979.

[Kowalski and Sergot, 1986] Robert Kowalski and Marek Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4:67 - 95, 1986.

[Kuipers, 1986] B Kuipers. Qualitative simulation. *Artificial Intelligence*, 24:169-203, 1986.

[Lloyd, 1993] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1993.

[McCarthy, 1959] John McCarthy. Programs with Common Sense. In *Proceedings of the Symposium on Mechanisation of Thought Processes*, London, 1959. Her Majesty's Stationery Office.

[McCarthy and Hayes, 1969] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, p. 463 - 502. Edinburgh University Press, 1969.

[Michael, 1991] Antonia Michael. *Urban Traffic represented in Event Calculus.* M.Sc. thesis, Imperial College, 1991.

[Muggleton, 1995] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13(3 and 4):245–286, 1995.

[Sadri and Kowalski, 1995] Fariba Sadri and Robert Kowalski. Variants of the Event Calculus. In *Proceedings of the 12th International Conference on Logic Programming*, Japan, 1995.

[Shanahan, 1990] Murray Shanahan. Representing Continuous Change in the Event Calculus. In *Proceedings of ECAI 90*, p. 598-603. 1990.

[Shanahan, 1997] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[Srinivasan and Camacho, 1995] Ashwin Srinivasan and Rui Camacho. Numerical reasoning in ILP. In *Machine Intelligence 15*, Oxford, 1995. (to appear).

[Sripada, 1995] Suryanarayana M. Sripada. Efficient Implementation of the Event Calculus for Temporal Database Applications. In *Proceedings of the 12th International Conference on Logic Programming*, Japan, 1995.

[Stickel, 1992] Mark E. Stickel. A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science*, 104(1):109–128, 1992.

[Van Belleghem *et al.*, 1994] Kristof Van Belleghem, Marc Denecker, and Danny De Schreye. The Abductive Event Calculus as a General Framework for Temporal Databases. In *Proceedings of the International Conference on Temporal Logic*, 1994.

# A An Introduction to the Event Calculus: a Blocks World Example

As an introduction to the event calculus[8] consider the blocks world shown in figure 3[9]. This appendix develops an event calculus program $T_{Blocks}$, that permits reasoning about what fluents hold at different times in the blocks world. Reasoning about events in the event calculus is based on a narrative. For example, the process of moving the two blocks shown in figure 3 individually requires that block A is first moved (first event), and then block B is moved (second event).

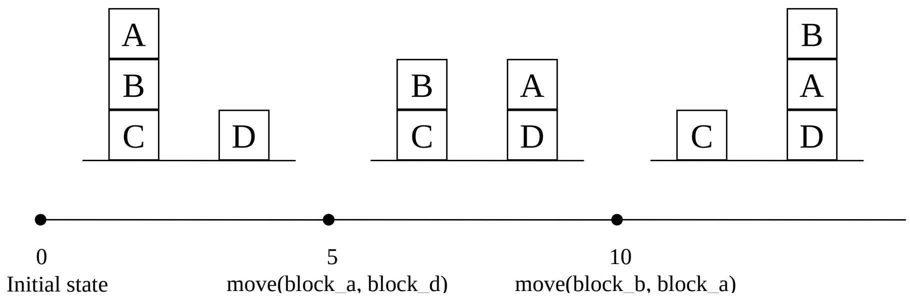


Figure 3: The Blocks World Narrative

The event calculus program $T_{Blocks}$ , contains clauses that describe the initial state of the blocks world, $T_0$ , which can be represented by the following Horn clauses.

(BI)
$$\begin{array}{lll} initially(on(c, table)) & initially(on(b, c)) & initially(on(a, b)) \\ initially(on(d, table)) & intially(clear(a)) & intially(clear(d)) \\ intially(clear(table)) & & \end{array}$$

The narrative $T_N$, detailing the occurrence of events for the rearrangement of the blocks world is given by the clauses

(BH1) $happens(move(a, d), 5)$

(BH2) $happens(move(b, a), 10)$

The intended meaning of clause BH1 is that the event moving block A onto block D ($move(a, d)$) occurs at time instant 5.

At this point the list of clauses presented is not sufficient to reason about the temporal nature of the state of the blocks. That is to say that the conjunction of $T_0$ and $T_N$ does not entail statements about the position of the blocks at later times (time 6 for example). For example

$$T_0 \wedge T_N \nvDash holds\_at(on(a, d), 6)$$

To derive information about the state of the world at later times two more components of the event calculus must be included; the *core axioms* and the *domain specific axioms*. The *core*

[8]The event calculus developed here is known as the Simplified Event Calculus (SEC) [Sadri and Kowalski, 1995].
[9]This example is from [Shanahan, 1997].

*axioms* of the event calculus, $T_{Core}$, consist of the following normal program clauses[10].

$$\text{(EC1)} \quad \begin{array}{l} holds\_at(Fluent, Time_2) \leftarrow \\ \quad happens(Event, Time_1) \wedge\ initiates(Event, Fluent, Time_1) \\ \quad \wedge\ Time_1 < Time_2 \wedge\ not\ clipped(Time_1, Fluent, Time_2) \end{array}$$

$$\text{(EC2)} \quad \begin{array}{l} clipped(Time_1, Fluent, Time_2) \leftarrow \\ \quad happens(Event, Time) \wedge\ terminates(Event, Fluent, Time_1) \\ \quad \wedge\ Time_1 =< Time \wedge\ Time < Time_2 \end{array}$$

$$\text{(EC3)} \quad \begin{array}{l} holds\_at(Fluent, Time) \leftarrow \\ \quad initally(Fluent) \wedge\ not\ clipped(0, Fluent, Time) \end{array}$$

The intended meaning of clause EC1 is that a fluent value holds at a later time, $Time_2$, if an event happens at an earlier time, $Time_1$, and that event initiates the fluent value, provided that the fluent value is not clipped between those two times. Clause EC2 defines what it is for a fluent value to be *clipped*. A fluent value is clipped between times $Time_1$ and $Time_2$ if there is an event that occurs at some time in between the two that terminates the fluent value. Finally, the third core axiom (EC3) caters for the special case of the initial state from which a fluent continues unchanged provided that it has not been clipped. The inclusion of the core axioms with those of $T_0$ and $T_N$ still does not permit the correct derivation of properties other than those from the initial conditions.

Finally, a description of which events trigger the state of particular fluents is required. These are the domain specific axioms. The clauses that describe the relationship between events in the blocks world and the state of the fluents in the blocks world, $T_{DSA}$, are

$$\text{(BDSA1)} \quad \begin{array}{l} initiates(move(X,Y), on(X,Y), Time) \leftarrow \\ \quad holds\_at(clear(X), Time) \wedge\ holds\_at(clear(Y), Time) \\ \quad \wedge\ diff(X,Y) \wedge diff(X, table) \end{array}$$

$$\text{(BDSA2)} \quad \begin{array}{l} terminates(move(X,Y), on(X,Z), Time) \leftarrow \\ \quad holds\_at(clear(X), Time) \wedge\ holds\_at(clear(Y), Time) \\ \quad \wedge\ holds\_at(on(X,Z), Time) \wedge\ diff(X,Y) \\ \quad \wedge\ diff(X, table) \end{array}$$

$$\text{(BDSA3)} \quad \begin{array}{l} initiates(move(X,Y), clear(Z), Time) \leftarrow \\ \quad holds\_at(clear(X), Time) \wedge\ holds\_at(clear(Y), Time) \\ \quad \wedge\ holds\_at(on(X,Z), Time) \wedge\ diff(X,Y) \\ \quad \wedge diff(Y,Z) \end{array}$$

$$\text{(BDSA4)} \quad \begin{array}{l} terminates(move(X,Y), clear(Y), Time) \leftarrow \\ \quad holds\_at(clear(X), Time) \wedge\ holds\_at(clear(Y), Time) \\ \quad \wedge\ diff(X,Y) \wedge diff(X, table) \end{array}$$

The intended interpretation of the predicate $diff(X,Y)$ is that objects $X$ and $Y$ are unique.

Let $T_{Blocks}$ be the conjunction of $T_0$, $T_N$ and $T_{DSA}$. The addition of the core axioms completes the event calculus program. It is now possible to derive statements about the state of the blocks world after the initial configuration. For example

$$T_{Blocks} \wedge T_{Core} \models holds\_at(on(a,d), 6)$$

[10] A *normal program clause* is a clause that may contain one or more negated literals in its body. e.g. $p \leftarrow not\ q$.

More elaborate domains have been described using the event calculus including traffic flow [Michael, 1991], air traffic control [Sripada, 1995], and a tele-communications protocol [Denecker *et al.*, 1996].

# B Data for induction of the Blocks World DSAs

The data for generating the domain specific axioms is presented below. There are four subsections. The first contains background knowledge common to both learning up and down transition; the second and third components is the data required for learning the up and down transitions respectively; and the last subsection details common mode declarations, settings and constraints.

## B.1 Common background knowledge

The the following components of background knowledge was presented for both the learning of the up and down transition DSAs.

- Independent observational fluent trace facts. For example the beginning of each trace starts with the facts below.

```
basic(clear(block_d ),+, 1).
basic(clear(block_a ),+, 1).
basic(clear(block_c ),+, 11).
basic(clear(block_a ),-, 1).
basic(clear(table ),+, 1).
basic(clear(block_d ),-, 6).
basic(clear(block_b ),+, 6).
```

- background facts including time points, and typing information. For example

```
time_sort(0). time_sort(1). ... time_sort(15).
fluent(on(X,Y)). fluent(clear(X)).
event(move(X,Y)).
```

- Facts about the uniqueness of objects as expressed by the predicate `diff/2` were also provided.

```
diff(block_a,block_b).
diff(block_a,block_c).
⋮
diff(block_d,table ).
```

- The narrative for this experiment consists of two events.

```
happens(move(block_a,block_d),5).
happens(move(block_b,block_a),10).
```

## B.2 Dependent observational fluent trace information for learning *up* transition DSAs.

- Dependent "+" `holds_at/3` facts. For example

```
holds_at(on(block_c, table ),+, 1).
holds_at(on(block_c, table ),+, 2).
⋮
holds_at(on(block_c, table ),+,15).
holds_at(on(block_b, block_c),+, 1).
holds_at(on(block_b, block_c),+, 2).
⋮
holds_at(on(block_b, block_c),+, 10).
holds_at(on(block_a, block_b),+,1).
holds_at(on(block_a, block_b),+,2).
⋮
holds_at(on(block_a, block_b),+,5).
holds_at(on(block_d, table),+,1).
holds_at(on(block_d, table),+,2).
⋮
holds_at(on(block_d, table),+,15).
holds_at(on(block_a, block_d),+,6).
holds_at(on(block_a, block_d),+,7).
⋮
holds_at(on(block_a, block_d),+,15).
holds_at(on(block_b, block_a),+,11).
holds_at(on(block_b, block_a),+,12).
⋮
holds_at(on(block_b, block_a),+,15).
```

- Dependent "-" facts encoded as `basic/3`. For example

```
basic(on(block_a, block_a),-, 6).
basic(on(block_a, block_b),-, 6).
basic(on(block_a, block_c),-, 6).
basic(on(block_a, table),-, 6).
basic(on(block_a, block_b),-, 11).
basic(on(block_b, block_c),-, 11).
basic(on(block_b, block_d),-, 11).
basic(on(block_b, table),-, 11).
```

- Constraints for periods of "-". For example

```
:- clipped(5,on(block_a, block_b),-, 15).
:- clipped(5,on(block_a, table ),-, 15).
⋮
:- clipped(10,on(block_b, block_d),-, 15).
```

### B.3 Dependent fluent trace information for learning *down* transition DSAs

- Dependent "-" `holds_at/3` facts. For example

```
holds_at(on(block_a, block_a),-, 6).
holds_at(on(block_a, block_a),-, 7).
⋮
holds_at(on(block_a, block_a),-,15).
holds_at(on(block_a, block_b),-, 6).
holds_at(on(block_a, block_b),-, 7).
⋮
holds_at(on(block_a, block_b),-, 15).
holds_at(on(block_b, block_b),-,11).
holds_at(on(block_b, block_b),-,12).
⋮
holds_at(on(block_b, block_b),-,15).
holds_at(on(block_b, block_c),-,11).
holds_at(on(block_b, block_c),-,12).
⋮
holds_at(on(block_b, block_c),-,15).
holds_at(on(block_b, block_d),-,11).
holds_at(on(block_b, block_d),-,12).
⋮
holds_at(on(block_b, block_d),-,15).
holds_at(on(block_b, table),-,11).
holds_at(on(block_b, table),-,12).

⋮
holds_at(on(block_b, table),-,15).
```

- Dependant "+" facts encoded as `basic/3`. For example

```
basic(on(block_c, table ),+, 1).
basic(on(block_b, block_c),+, 1).
basic(on(block_a, block_b),+, 1).
basic(on(block_d, table ),+, 1).
basic(on(block_a, block_d),+, 6).
basic(on(block_b, block_a),+, 11).
```

- constraint for periods of "-". For example

```
:- clipped(1,on(block_c, table ),+, 15).
:- clipped(1,on(block_b, block_c),+, 10).
⋮
:- clipped(10,on(block_b, block_a),+, 15).
```

### B.4 Mode declarations for input to Progol 5.0.

```
:-modeh(1,flips(move(+object,+object),on(+object,+object),#state,+time_sort))?
:-modeb(*,holds_at(clear(+object),#state,+time_sort))?
:-modeb(*,holds_at(on(+object,+object),#state,+time_sort))?
:-modeb(*,diff(+object,#object))?
:-modeb(*,diff(+object,+object))?
```