491 Knowledge Representation

# Event Calculus

Marek Sergot
Department of Computing
Imperial College, London

March 2005; Feb 2006

The Event Calculus:

- a general approach
- to representing and reasoning about events
- and their effects
- in a logic programming framework.

R.A. Kowalski & M.J. Sergot. 'A Logic-based Calculus of Events'.
*New Generation Computing* 4(1), Feb. 1986.

There are lots of different versions, more or less general, and various extensions.
Here is a typical version, with a brief outline of how it relates to the original treatment, and a list of some further extensions that are usually incorporated.

There are various implementation techniques to make this work efficiently for large (very large) examples. I omit those.

Event calculus has also been used for 'planning'. I only use it for 'narratives'. I will explain in outline how it can be used for planning *if* there is time at the end.

In AI terminology, a 'fluent' is a time-varying fact – a proposition whose value changes from state to state. Fluents correspond to what are called 'state variables' in other areas of Computer Science.



## General idea

Instead of starting with *situations* as in Situation Calculus focus instead on

- **events**.

Instead of *global states* ('situations') think in terms of

- **local states**,

one for each period of time for which a fluent $F$ holds continuously.

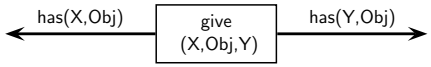
Events

- initiate and terminate

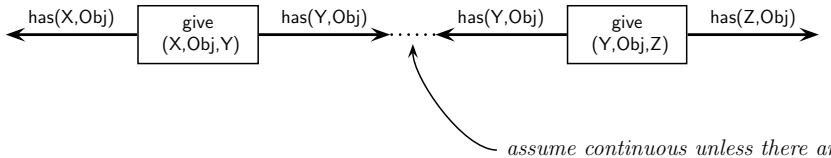
periods of time for which fluents hold continuously.

**Example**

```
give(X,Obj,Y) initiates  has(Y,Obj)
give(X,Obj,Y) terminates has(X,Obj)
```




And if we have several such (a 'narrative'):

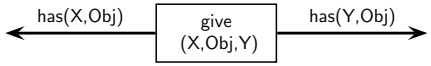



*assume continuous unless there are reasons to think otherwise (such as another* give *event or an observation that* Y *does not have* Obj *somewhere here)*

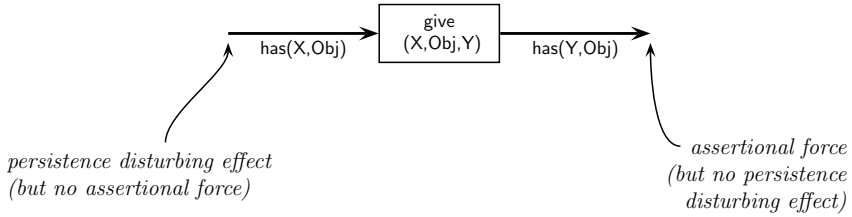
- In the **original** Event Calculus events could say something about the **past**, as in the example above.
  `E terminates U` means that `U` holds for some time period ending at event `E`.

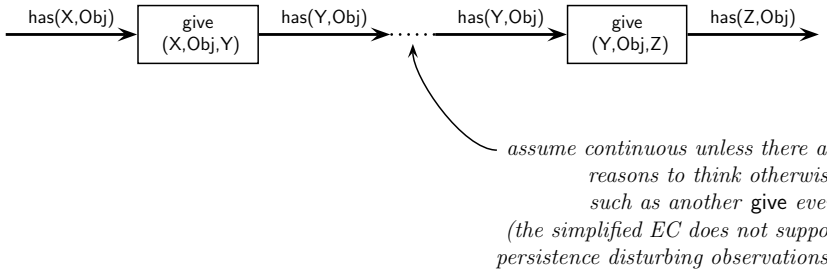
  Then there is some more or less complicated reasoning to infer which fluents persist past `E` and which do not.
- In the **simplified** Event Calculus **(SEC)**, which we will look at here, `E terminates U` does not assert anything about the past. It simply **blocks** persistence of `U` past the event `E`.

In the **original** Event Calculus:




In the **simplified Event Calculus**:

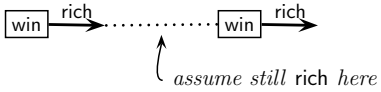


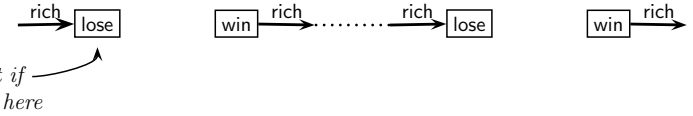



## Example

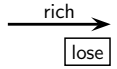
- winning the lottery initiates rich
  (but you might be rich already)
- losing your wallet terminates rich
  (but you might not be rich when you lose it)




And similarly:




Actually it is much more convenient to take end points like this:

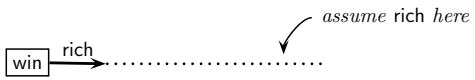



but then the pictures don't look so nice.

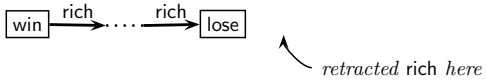
The actual formulation of the SEC (coming up) will do the end points correctly.

Remember we are assuming events are *instantaneous*.
How would you modify it if events have durations? (Optional tutorial exercise)

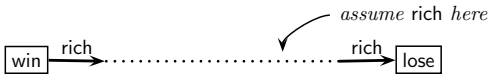
## Inferences are non-monotonic



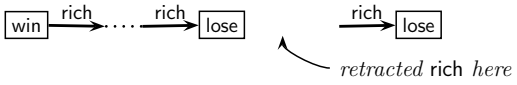

We add a new event occurrence:




Moreover, events can be assimilated **in a different order** from that in which they actually occurred.




But then we add another event occurrence:






## General formulation (simplified EC)

In the simplified Event Calculus (SEC)

- Events are given times (when they 'happen').
- We will assume that all events are instantaneous.
  (One can easily do a refinement where events have *durations*.
  Details are left for an (optional) tutorial exercise.)
- All examples here will have (non-negative) integer times.
  But this is for convenience — SEC makes no assumption that time
  is discrete and/or that time points are integers.
- Many events can happen at the same time.

In the original Event Calculus it is not necessary to give times to all events.
All we need is a *relative ordering* of events. This ordering could be *partial*.

There are two separate problems:

- Given a narrative, check that it is **consistent**
  (the event narrative is possible, given action pre-conditions and other constraints)
- Given a (consistent) narrative, determine what holds when — now, in the past, in the future.

Here is the general formulation:

| | |
|---|---|
| `holds_at(U, T)` | `U` is a fact or 'fluent'; `T` is a time point |
| `holds_for(U, P)` | `P` is a time period<br>(maximal interval of time points for which `U` holds continuously)<br>`P` is of the form `(`$T_1$ `,`$T_2$`]` or `since(T)`. |

The narrative (what has happened and when) is represented using

| | |
|---|---|
| `happens(E, T)` | An event/action of type `E` occurred/happened at time `T` |
| `initially(U)` | fluent `U` holds at the initial time point (which is usually `0`) |

The effects of actions are represented using the predicates `initiates` and `terminates` (usually written in infix form to aid readibility):

| | |
|---|---|
| `E initiates U` | The occurrence of an event of type `E` initiates a period of time for which fluent `U` holds; |
| `E terminates U` | The occurrence of an event of type `E` terminates a period of time for which fluent `U` holds. |

Examples:

```
birth(X) initiates alive(X)
give(X, Obj, Y) initiates has(Y, Obj)

death(X) terminates alive(X)
give(X, Obj, Y) terminates has(X, Obj)
```

Often `initiates` and `terminates` are context dependent. So there are also 3-ary versions of both.

Examples:

```
initiates( promote(X), rank(X, New), T ) ←
    holds_at(rank(X, Old), T),
    next_rank(Old, New)   % static ('rigid') fact

terminates( shoot, alive, T ) ←
    holds_at(loaded, T)
```

The conditions in these rules express **fluent pre-conditions**.

The 2-ary and 3-ary versions of `initiates` and `terminates` are easily related by adding the following general rules:

```
initiates(E, U, T) ← E initiates U
terminates(E, U, T) ← E terminates U
```

### General rules

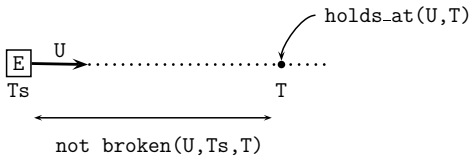
```
holds_at(U, T) ←
    happens(E, Ts),
    Ts < T,
    initiates(E, U, Ts),
    not broken(U, Ts, T)

broken(U, Ts, T) ←
    happens(Estar, Tstar),
    Ts < Tstar, Tstar < T,
    terminates(Estar, U, Tstar)
```

In order to deal with `initially` statements in the narrative we also need:

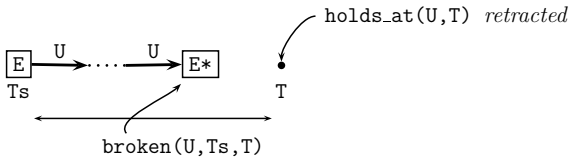
```
holds_at(U, T) ←
    0 ≤ T,
    initially(U),
    not broken(U, 0, T)
```

Note the way the time comparisons are written. A fluent does not hold at the time of the event that initiated it. This way round turns out to be more convenient than writing `Ts` $\leq$ `T` in the first `holds_at` clause.






Negation-by-failure `not` means the inference is **nonmonotonic**:




This is not a particularly **efficient** implementation but if you type it into Prolog (writing `:-` for ← and `\+` for `not`) it will run.

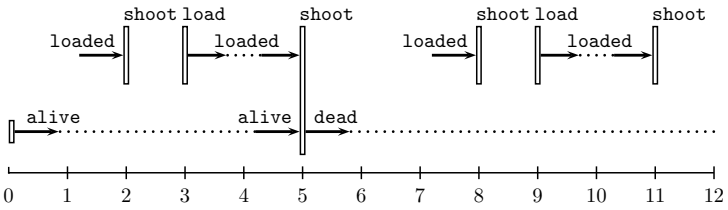
### General rules for computing periods

General rules for `holds_for(U, P)` are written in similar style.
Details are omitted. They are not difficult but I don't have time. Maybe if there is time at the end I will show them.
If not, I leave it as an (optional) tutorial exercise. You should be able to figure it out easily from the diagrams in the notes.

## Example ('Yale Shooting Problem')

(In Prolog notation)

```
initiates(load,loaded).
initiates(shoot,dead,T) :- holds_at(loaded,T).

terminates(shoot,loaded).
terminates(shoot,alive,T) :- holds_at(loaded,T).

initially(alive).

happens(shoot,2).
happens(load,3).
happens(shoot,5).
happens(shoot,8).
happens(load,9).
happens(shoot,11).
```




```
?- holds_at(alive, 4).
   yes

?- holds_at(alive, 5).
   yes                    % shows how end points are treated

?- holds_at(dead,6.22256).
   yes

?- holds_for(loaded,P).
   P = (3,5];
   P = (9, 11]
```

Don't try

```
?- holds_at(loaded, T).
```

Too many answers!! (Time is not discrete/integer)



## Action pre-conditions

You can't load and shoot a gun at the same time.
How do we express such *action pre-conditions*?

**Action pre-conditions = integrity constraints**

```
incons :-
     happens(load, T), happens(shoot, T).
```

Here `incons` stands for 'inconsistency'. It doesn't matter what you choose (as long as it isn't a Prolog built-in predicate.)
Now to check consistency of the narrative

```
?- incons.
```

Every time the narrative changes, run the query to check consistency.
(There are techniques for efficient *incremental* integrity constraint checking in deductive databases. Details omitted.)

It is often helpful to include an extra argument in `incons` to give an indication of the type of inconsistency that has been detected.

You can put any kind of message you like in this argument. Personally, I like to put a Prolog term representing the nature of the inconsistency, like this:

```
incons((load,T)-happens(shoot,T)) :-
     happens(load, T), happens(shoot, T).
```

Here `(load,T)-happens(shoot,T)` is just a Prolog term. (`-` is just a Prolog function symbol. It has no special meaning.)

Now the consistency checking query

```
?- incons(X).
```

returns in `X` a record of what kind of inconsistency it is.

**Notice that**: action pre-conditions in event calculus will be *more complicated* than in situation calculus (typically). Many events can happen simultaneously in event calculus, and some combinations aren't possible. That can't happen in situation calculus — only one action at a time (unless you use some exotic version).

## Comments

**Frame problem(s):**

- the **qualification frame problem** is addressed by the built-in treatment of initiates/terminates and the use of negation-by-failure to make default inferences about how fluents persist
- the **computational frame problem** is addressed by having *local states* ('periods') and avoiding having to reason from one global situation to the previous one; moreover, it is possible to include some simple *indexing techniques* to make the search for potential initiating and terminating events very much more efficient. These can cope with applications of millions of event occurrences over thousands of event types and fluents. (Maybe more — I have never tried more.)
- the **ramification frame problem** is not addressed by event calculus. For a simple example: a person must be alive to be happy. So an *indirect* effect of shooting someone with a loaded gun is to terminate any period of time for which they are currently happy. In event calculus one must *anticipate* that, and write additional `terminates` clauses explicitly.

  The ramification problem is handled much better in another formalism $\mathcal{EC}+$ that I have under development with Rob Craven. Maybe more about that later if time permits.

There are many extensions to the basic SEC:

- events with duration
- partially ordered events
- structured events
- events with continuous change

Don't forget: the **full power** of logic programming is still available when writing `happens` and `initiates/terminates`. Two simple examples:

- `happens(flash, T) :- happens(shoot, T).`
- `holds_at(frightened, T) :- happens(shoot, T).`

  (Here `frightened` is a non-inertial fluent — it is only pointwise true)

(More examples later)



## Incompatible fluents

Consider `give(X, N, Y)` — "`X` gives `N` marbles to `Y`".

The effects of this action:

```
initiates( give(X, N, Y), amount(X, M_new), T )←
    holds_at(amount(X, M), T),
    M_new is M - N.
initiates( give(X, N, Y), amount(Y, M_new), T )←
    holds_at(amount(Y, M), T),
    M_new = M + N.

terminates( give(X, N, Y), amount(X, M), T )←
    holds_at(amount(X, M), T ).
terminates( give(X, N, Y), amount(Y, M), T )←
    holds_at(amount(Y, M), T ).
```

Actually because of the way `terminates` works (it is only 'persistence disturbing'), it is enough to write:

```
terminates( give(X, N, Y), amount(X, M), T ).
terminates( give(X, N, Y), amount(Y, M), T ).
```

But still. What a pain! And what if we forget the `terminates` ?

Here is one solution. There is a better and more general one (multi-valued fluents) in a moment.

**Incompatible fluents**

The original event calculus has `incompatible` fluents.

Examples:

- `incompatible(alive, dead)`
- `incompatible( amount(X, M), amount(X, N) )` ← `M` $\neq$ `N`

The general rules for `holds_at` and `holds_for` have to be modified.
But we only have to add

```
terminates(E, U, T) ← incompatible(U, V), initiates(E, V, T)
terminates(E, U, T) ← incompatible(V, U), initiates(E, V, T)
```

Now it is sufficient to write

```
initiates( give(X, N, Y), amount(X, M_new), T )←
    holds_at(amount(X, M), T),
    M_new is M - N.
initiates( give(X, N, Y), amount(Y, M_new), T )←
    holds_at(amount(Y, M), T),
    M_new = M + N.
```

The `terminates` are taken care of automatically by `incompatible`.

Similarly in the 'Yale Shooting Problem' we only have to write:

```
initiates(shoot,dead,T) :- holds_at(loaded,T).
```

With `incompatible(alive,dead)` the `terminates(shoot,alive)` can be omitted, and is handled automatically.

Note also we could write

```
incompatible(happy,dead)
```

as a way of getting a simple kind of indirect effect (ramification). But ramifications can be much more complicated than this.

`incompatible` works, but there is a much more powerful and more general solution that deals better with most of the common cases—multi-valued fluents.



## Multi-valued fluents

(Not provided in the published accounts of the event calculus, but I always include it in mine)

Instead of a Boolean fluent `amount(X,N)` use a **multi-valued fluent**, written `amount(X)=N`.

Instead of a Boolean fluent `loc(X,Loc)` use a multi-valued fluent `loc(X)=Loc`.

The idea is that a multi-valued fluent can have *at most one value at a time.*
Adjust the general rules again. We add:

```
terminates(E, U=V, T) ← initiates(E, U=V', T)
```

(Because of the way `terminates` works it is not necessary to add the condition `V` $\neq$ `V'`. We get the desired effect without it.)

For many examples, use of multi-valued fluents will be the best solution. (See examples to follow.)

### Example

For a slight variant of the YSP, suppose it is necessary to aim the gun (at a 'target') before shooting. Let `target` be a multi-valued fluent.

```
initiates( shoot, dead(X), T) ←
    holds_at(loaded, T),
    holds_at(target=X, T)
aim(X) initiates target=X
```

When the gun is aimed at someone else `target=X` is automatically terminated.

### Negative fluents

The `incompatible` device is clumsy.
It would be more natural to introduce negative fluents. For example:

```
birth(X) initiates alive(X)
death(X) initiates ¬alive(X)
```

In fact we already have this, by treating Boolean fluents as a special case of multi-valued fluents. So write e.g.

```
birth(X) initiates alive(X)=t
death(X) initiates alive(X)=f
```

## Some subtleties

Note: it might seem that we can dispense with `terminates` altogether.
But there is a difference between

```
death(X) initiates alive(X)=f
```

and

```
death(X) terminates alive(X)=t
```

The first makes an *assertion*; the second only has a *persistence disturbing* effect. Both are useful, but are not equivalent.

Note also:

- `holds_at(p=f,T)` is *not the same as* `not holds_at(p=t,T)`
- `holds_at(p=t,T)` is *not the same as* `not holds_at(p=f,T)`

Think about it: it has to do with *missing information*.

(The same point arises in 'extended logic programs' which mix negation-by-faiour `not` with classical truth-functional negation ¬.)



## Example (cars)

Static ('rigid') facts:

```
candrive(jim, carA).
candrive(jim, carB).
candrive(jim, carC).
candrive(mary, carB).
candrive(bob, carA).
```

Initial facts:

```
initially(loc(jim)=work).
initially(loc(mary)=home).
initially(loc(bob)=shops).
initially(loc(carA)=home).
initially(loc(carB)=home).
initially(loc(carC)=work).
```

Now the effects of actions:

```
initiates(walk(X,Loc), loc(X)=Loc).
initiates(drive(X,_,Loc), loc(X)=Loc).
initiates(drive(_,Car,Loc), loc(Car)=Loc).
```

We don't need to include `terminates` for these because these are multi-valued fluents.

Of course, walking and driving aren't really *instantaneous* but I am ignoring the version of the event calculus with events with duration.

A person is happy if (s)he is in the pub.
We could write

```
holds_at(happy(X), T) :- holds_at(loc(X)=pub, T).
```

But then (in simplified EC at least) `happy(X)` won't persist.

So we write instead that `walk(X,pub)` and `drive(X,Car,pub)` initiate `happy(X)`.
It doesn't matter if `X` is happy already — cf. winning the lottery and becoming/being rich.

```
% define a new kind of event, for convenience
   happens(go(X,Loc), T) :- happens(walk(X,Loc), T).
   happens(go(X,Loc), T) :- happens(drive(X,_,Loc), T).

% arriving at the pub makes X happy
   initiates(go(X,pub),happy(X)).

% and happy(X) persists until X gets to work
   terminates(go(X,work), happy(X)).
```

Notice that I have chosen above to make `happy(X)` a regular Boolean fluent rather than a multi-valued fluent with values `t` or `f`.
No particular reason, except to demonstrate that Boolean and multi-valued fluents can be mixed like this.

Next, `X` is tired when `X` leaves work.
(This does not imply that leaving work *causes* `X` to be tired.)
And `tired(X)` persists until `X` gets home or to the pub.

```
initiates(go(X,_),tired(X),T) :- holds_at(loc(X)=work,T).

terminates(go(X,home), tired(X)).
terminates(go(X,pub), tired(X)).
```

The defined action `go` saves us having to deal with `walk` and `drives` separately.

Finally, to give an example of a derived, non-inertial fluent, let's suppose that Mary is grumpy whenever she is tired, or at the shops.

```
holds_at(grumpy(mary),T) :- holds_at(tired(mary),T).
holds_at(grumpy(mary),T) :- holds_at(loc(mary)=shops, T).
```

`grumpy(mary)` doesn't persist: it is true when she is tired or at the shops.

**Action-preconditions**

These are going to be quite complicated because we also have to think of what *combinations* of events can happen simultaneously.

First let's deal with individual events. (You can ignore the labels in `incons` for a first reading.) I usually deal with combinations first but it's perhaps easier to read this way round.

```
% check candrive facts: (drive(X,Car,L) can't happen if X can't drive car Car
   incons((drive(X,Car,L), T)-(\+ candrive(X,Car))) :-
       happens(drive(X,Car,L), T),
       \+ candrive(X,Car).   % \+ is 'not' in Prolog

% can't drive a car if it's somewhere else
   incons((drive(X,Car,L), T)-(loc(X)=L1,T)-(loc(Car,L2),T)) :-
       happens(drive(X,Car,L), T),
       holds_at(loc(X)=L1, T),   % assumes we always know loc(X)
       holds_at(loc(Car)=L2, T), % assumes we always know loc(Car)
       L1 \= L2.

% can't drive from the pub, except for mary
   incons((drive(X,Car,L), T)-(loc(X)=pub,T)) :-
       happens(drive(X,Car,L), T),
       X \= mary,
       holds_at(loc(X)=pub, T).
```



```
% too far to walk from work to the shops
   incons((walk(X,L1), T)-(loc(X)=L2, T)-too_far_to_walk(L1,L2)) :-
       happens(walk(X,L1), T),
       holds_at(loc(X)=L2, T),
       too_far_to_walk(L1,L2).

   too_far_to_walk(work,shops).  % static ('rigid') fact
   too_far_to_walk(shops,work).
```

Now for impossible combinations of events:

```
% X can't walk to two different places at the same time
   incons((walk(X,L1), T)-(walk(X,L2), T)) :-
       happens(walk(X,L1), T),
       happens(walk(X,L2), T),
       L1 \= L2.

% X can't drive two different cars at the same time,
% or the same car to two different places
   incons((drive(X,Car1,L1), T)-(drive(X,Car2,L2), T)) :-
       happens(drive(X,Car1,L1), T),
       happens(drive(X,Car2,L2), T),
       (X,Car1,L1) \= (X,Car2,L2).

% two different people can't drive the same car
% to the same location, or different locations
   incons((drive(X1,Car,L1), T)-(drive(X2,Car,L2), T)) :-
       happens(drive(X1,Car,L1), T),
       happens(drive(X2,Car,L2), T),
       (X1,Car,L1) \= (X2,Car,L2).

% X can't drive a car and walk at the same time
   incons((drive(X,Car1,L1), T)-(walk(X,L2), T)) :-
       happens(drive(X,Car1,L1), T),
       happens(walk(X,L2), T).
```

I think that's it.

**Example narrative (bad)**

```
happens(walk(bob,work),4).   % bad
happens(drive(jim,carA,pub),5).   % bad
happens(walk(jim,shops),8).
happens(drive(mary,carB,pub),5).
happens(walk(jim,home),10).
happens(drive(mary,carB,home),10).
happens(drive(jim,carB,pub),12).
happens(walk(jim,work),13).  % bad
happens(drive(jim,carC,work),13). % bad
```

Check for consistency with action pre-conditions:

```
| ?- incons(X).
X = (drive(jim,carC,work),13)-(walk(jim,work),13) ? ;
X = (drive(jim,carA,pub),5)-(loc(jim)=work,5)-(loc(carA,home),5) ? ;
X = (drive(jim,carC,work),13)-(loc(jim)=pub,13)-(loc(carC,work),13) ? ;
X = (drive(jim,carC,work),13)-(loc(jim)=pub,13) ? ;
X = (walk(bob,work),4)-(loc(bob)=shops,4)-too_far_to_walk(work,shops) ? ;
(no more solutions)
```

This narrative is no good!



**Example narrative (good)**

```
happens(walk(bob,home),4).       % fixed
happens(drive(bob,carA, work),5).
happens(drive(jim,carC,pub),5).  % fixed
happens(walk(jim,shops),8).
happens(drive(mary,carB,pub),5).
happens(walk(jim,home),10).
happens(drive(mary,carB,home),10).
happens(drive(jim,carB,pub),12).
happens(walk(jim,work),13).
```

Check for consistency with action pre-conditions:

```
| ?- incons(X).
no
```

Fine! Now where is `carC`?

```
| ?- holds_at(loc(carC)=Loc,14).
Loc = pub ? ;
(no more solutions)
```

When was `jim` at work?

```
| ?- holds_for(loc(jim)=work, P).
P = [0,5] ? ;
P = since(13) ? ;
(no more solutions)
```

Where has `mary` been?

```
| ?- holds_for(loc(mary)=Loc, P).
Loc = home, P = [0,5] ? ;
Loc = pub,  P = (5,10] ? ;
Loc = home, P = since(10) ? ;
(no more solutions)
```

## Event calculus: Summary

| | |
|---|---|
| `holds_at(U, T)` | `U` is a fact or (multi-valued) 'fluent'; `T` is a time point |
| `holds_for(U, P)` | `P` is a time period<br>(maximal interval of time points for which `u` holds continuously)<br>`P` is of the form `[T1 ,T2)` or `since(T)`. |

The narrative (what has happened and when) is represented using

| | |
|---|---|
| `happens(E, T)` | An event/action of type `E` occurred/happened at time `T` |
| `initially(U)` | Fluent `U` holds at initial time (time 0) |

The effects of actions are represented using the predicates `initiates` and `terminates`.

| | |
|---|---|
| `initiates(E,U,T)` | The occurrence of an event of type `E` at time `T`<br>initiates a period of time for which fluent `U` holds; |
| `terminates(E,U,T)` | The occurrence of an event of type `E` at time `T`<br>(weakly) terminates a period of time for which fluent `U` holds. |

There are also binary versions without the time argument for convenience when the effects of an event are not context-dependent.

**General rules**

```
holds_at(U, T) ←
    0 ≤ T,
    initially(U),
    not broken(U, 0, T)
holds_at(U, T) ←
    happens(E, Ts),
    Ts < T,
    initiates(E, U, Ts),
    not broken(U, Ts, T)
broken(U, Ts, T) ←
    happens(Estar, Tstar),
    Ts < Tstar, Tstar < T,
    terminates(Estar, U, Tstar)
initiates(E,U,T) ← initiates(E,U)     % for convenience
terminates(E,U,T) ← terminates(E,U)
terminates(E, F=V, T) ← initiates(E,F=V′, T)     % multi-valued fluents
```

**General rules for computing periods**

Similar. Details omitted. Not difficult but not enough time.

I leave it as another optional tutorial exercise. You should be able to figure it out easily from the diagrams in the notes.



**Action pre-conditions**

Action pre-conditions are expressed as **integrity constraints**:

`incons :-` ... *conditions*

Or better:

`incons(X) :-` ... *conditions*

Every time the narrative is updated, check `incons(X)`.

Be careful when writing integrity constraints: consider whether you want

- `holds_at(fluent=f, T)` (you know you will know this); or
- `not holds_at(fluent=t, T)` (you might not know)

**Last point**

By analogy with multivalued fluents, write events in the same style wherever possible.

Examples:

- `aim=X` — the gun is aimed at target `X`; there can only be one target at a time;
- `walk(X)=Loc` — `X` walks to location `Loc`; `X` can only walk to one location at a time;
- `drive(X)=(Car,Loc)` — `X` drives car `Car` to location `Loc`; `X` can only drive one combination of `(Car,Loc)` at any one time

Include one general integrity constraint of the form:

```
incons((A=V1,T)-(A=V2,T)) :-
    happens(A=V1, T),
    happens(A=V2, T),
    V1 \= V2.
```

This doesn't remove the need to write other integrity constraints but it covers one important class in one go. (Just a tip.)