Lecture 10

Event Processing

The Event Calculus

Jeremy Pitt
Department of Electrical & Electronic Engineering
Imperial College London, UK
Email: j.pitt _at_ imperial _dot_ ac _dot_ uk
Phone: (int) 46318
Office: 1010 EEE

Aims and Objectives

- Aims
  - Introduce the Event Calculus, a(nother) logical formalism for reasoning about actions, events and change over time

- Objectives
  - Understand the EC engine for reasoning about actions and events
  - Able to formulate simple problems in EC
  - See the relation between state spaces, possible worlds and local states

The Yale Shooting Problem (YSP)

- A classic example, due to Steve Hanks and Drew McDermott in 1987
- One actor, Fred, who turns out to be a turkey, and a gun
- Two fluents (propositions whose values change over time)
  - One for the state of the gun, which can either be loaded or unloaded
  - One for the state of Fred, which can either be dead or alive
- Two actions
  - Load the gun, after which the gun is loaded
  - Shoot the gun, after which Fred is dead, and the gun unloaded

Logical Formulation of the YSP

- A *naive* formulation
  - $\neg loaded(N) \wedge load(N) \rightarrow loaded(N+1)$
  - $alive(N) \wedge loaded(N) \wedge shoot(N) \rightarrow dead(N+1)$
  - $loaded(N) \wedge shoot(N) \rightarrow \neg loaded(N+1)$

Reasoning about YSP (1)

- Given:
  - $\{alive(1), \neg loaded(1), load(1), shoot(2)\}$
- We can build a model for (satisfy):
  - $alive(2) \wedge loaded(2) \wedge dead(3) \wedge \neg loaded(3)$
- But we can also build a model for (Fred 'as 'eart attack):
  - $dead(2) \wedge loaded(2) \wedge dead(3) \wedge \neg loaded(3)$
- Because we did not say:
  - $alive(N) \wedge load(N) \rightarrow alive(N+1)$

Reasoning about YSP (2)

- Given:
  - $\{alive(1), \neg loaded(1), load(1), shoot(2), load(3), shoot(4)\}$
- We can build a model for (satisfy):
  - $alive(2) \wedge dead(3) \wedge dead(4) \wedge dead(5) \wedge loaded(2) \wedge \neg loaded(3) \wedge loaded(4) \wedge \neg loaded(5)$
- But we can also build a model for (Fatal Attraction Fred):
  - $alive(2) \wedge dead(3) \wedge alive(4) \wedge dead(5) \wedge loaded(2) \wedge \neg loaded(3) \wedge loaded(4) \wedge \neg loaded(5)$
- Because we did not say:
  - $\neg alive(N) \wedge load(N) \rightarrow \neg alive(N+1)$

The Frame Problem

- Do we have to do this for everything?!
  - Everything not explicitly changed stays the same?
  - That which is true does not 'spontaneously' become false
  - That which is false does not 'spontaneously' become true

- This is the **frame problem**

- (Note: adding $alive(N) \leftrightarrow \neg dead(N)$ isn't going to help)

The Event Calculus

- General purpose language for representing events, and for reasoning about effects of events
- ...to overcome the frame problem
- An(other) action language with a logical semantics. Therefore, there are links to:
  - Implementation directly in Prolog.
  - Implementation in other programming languages.
- Prolog:
  - Specification is its own implementation;
  - Hence: executable specification.

Fluents and Events

- Focus on events rather than situations; local rather than global states
- Fluents
  - A fluent is a proposition whose value changes over time
  - A local state: period of time during which a fluent holds continuously
- Events
  - $initiate$ and $terminate$ ...
  - ... a period of time during which a fluent holds continuously
- Example
  - $give(X, obj, Y)$ initiates $has(Y, obj)$
  - $give(X, obj, Y)$ terminates $has(X, obj)$
- A sequence of such events forms a narrative

Simplified Event Calculus (SEC)

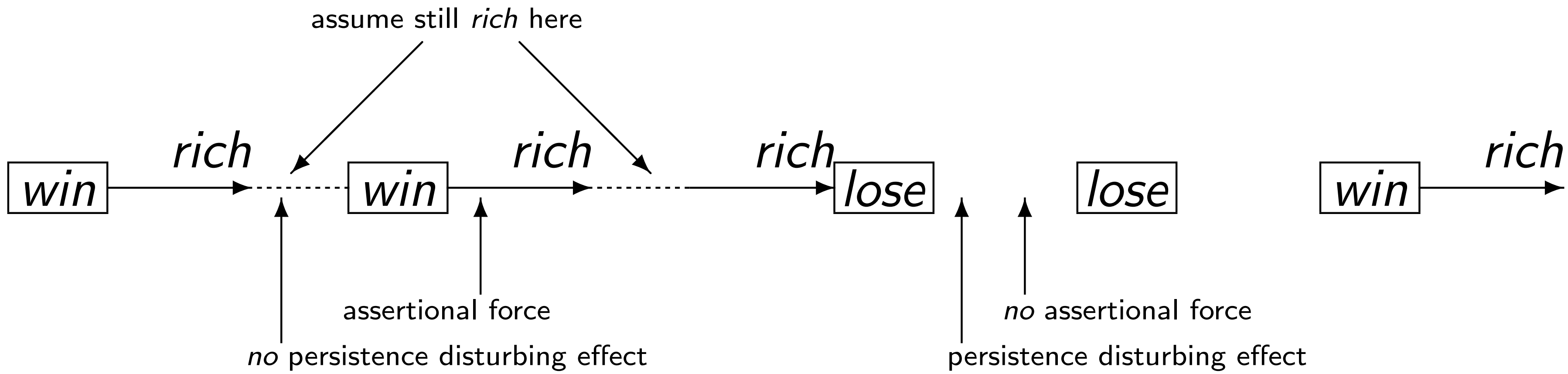
- Inertial fluents hold their values continuously
  - Values are assigned initially (at the start),
  - Values are given when asserted (initiated)
  - Values persist until disturbed (terminated)
  - Otherwise we have 'missing information'

- A formula of the form
  - $Event$ terminates $fluent$
  - Has persistence disturbing effect, but no assertional force

- A formula of the form
  - $Event$ initiates $fluent$
  - Has assertional force, but no persistence disturbing effect

Title

- Given
  - *win_lottery* initiates *rich*
    - Winning the lottery initiates rich (but you might be rich already)
  - *lose_wallet* terminates *rich*
    - Losing your wallet terminates rich (but you might not be rich when you lose it)



Events and Narratives in SEC

- Events occur at specific times (when they ‘happen’)
  - Assume that all events are instantaneous
  - Aside: there is a refinement of EC for events which have duration
- Here, we will use non-negative integer time-points
  - Does not mean we assume that time is discrete
  - Does not mean that time points have to be integers
  - We only need a relative/partial ordering for events
  - For non-negative integers, $<$ will do
  - Read $<$ as ‘earlier than’ or ‘before’
- A set of events, each with a given time, is called a *narrative*
  - Inference in the SEC is non-monotonic
  - Events in a narrative can be processed in a different order to that in which they occurred

General Formulation

- The narrative (what happens when) is represented by:
  - initially F
    - Fluent F holds at the initial time point (usually 0)
  - E happensat T
    - Event/action of type E occurred/happened at time T
- The effects of actions are represented by:
  - E initiates F at T
    - The occurrence of event of type E at time T starts a period of time for which fluent F holds
  - E terminates F at T
    - The occurrence of event of type E at time T ends a period of time for which fluent F holds

General Query

- The general query:
  - F holdsat T
    - Fluent F holds at time T
  - F holdsfor P
    - Fluent F holds for time period P
    - P is of the form $(T_1, T_2]$
- Time comparisons are strict: therefore a fluent does *not* hold at the time point in which it is initiated
- Recall
  - Closed intervals $[\_, \_]$ do include their end-points
  - Open intervals $(\_, \_)$ do not include their end-points
  - Therefore interval during which a fluent holds is (open,closed]

The SEC 'Engine'

$$
\begin{aligned}
&F \text{ holdsat } T \leftarrow \\
&\qquad 0 \leq T \;\wedge \\
&\qquad \text{initially } F \;\wedge \\
&\qquad \text{not } (F \text{ brokenbetween } 0 \text{ and } T)
\end{aligned}
$$

$$
\begin{aligned}
&F \text{ holdsat } T \leftarrow \\
&\qquad E \text{ happensat } Te \;\wedge \\
&\qquad Te < T \;\wedge \\
&\qquad E \text{ initiates } F \text{ at } Te \;\wedge \\
&\qquad \text{not } (F \text{ brokenbetween } Te \text{ and } T)
\end{aligned}
$$

$$
\begin{aligned}
&F \text{ brokenbetween } Te \text{ and } T \leftarrow \\
&\qquad E' \text{ happensat } Ti \;\wedge \\
&\qquad Te \leq Ti \;\wedge \\
&\qquad Ti < T \;\wedge \\
&\qquad E' \text{ terminates } F \text{ at } Ti
\end{aligned}
$$

Notes

- Negation-as-failure (not$(\ldots)$) ensures that inferences are non-monotonic

- Action pre-conditions can be expressed as integrity constraints
  - Some actions can't be performed at the same time
  - For example: $give(X, obj, Y) \wedge give(X, obj, Z) \wedge$ not$(Y = Z)$
  - Every time the narrative changes, query the integrity constraints to check consistency

- A simple extension allows many-valued (as well as boolean) fluents
  - Form is $F = V$: for boolean valued fluents, $V \in \{true, false\}$
  - We need to add the following rule to the 'engine'

$$\begin{aligned}&E \text{ terminates } F = V1 \text{ at } T \leftarrow\\ &\quad E \text{ initiates } F = V2 \text{ at } T \wedge\\ &\quad \text{not } (V1 = V2)\end{aligned}$$

Notes (2)

- There is a difference between:
  - $eat(X)$ initiates $have(X) = false$ at T
  - $eat(X)$ terminates $have(X) = true$ at T

- Suppose there was also the axiom:
  - $regurge(X)$ initiates $have(X) = true$ at T

- Then, with this axiom for $regurge$ and each alternative axiom for eat, think about the narratives
  - $\begin{array}{l} happens\ eat(cake)\ at\ t_1 \\ happens\ regurge(cake)\ at\ t_1 \end{array}$ vs. $\begin{array}{l} happens\ eat(cake)\ at\ t_1 \\ happens\ regurge(cake)\ at\ t_2 \end{array}$
  - Disproving $X = true$ doesn't (necessarily) mean proving $X = false$

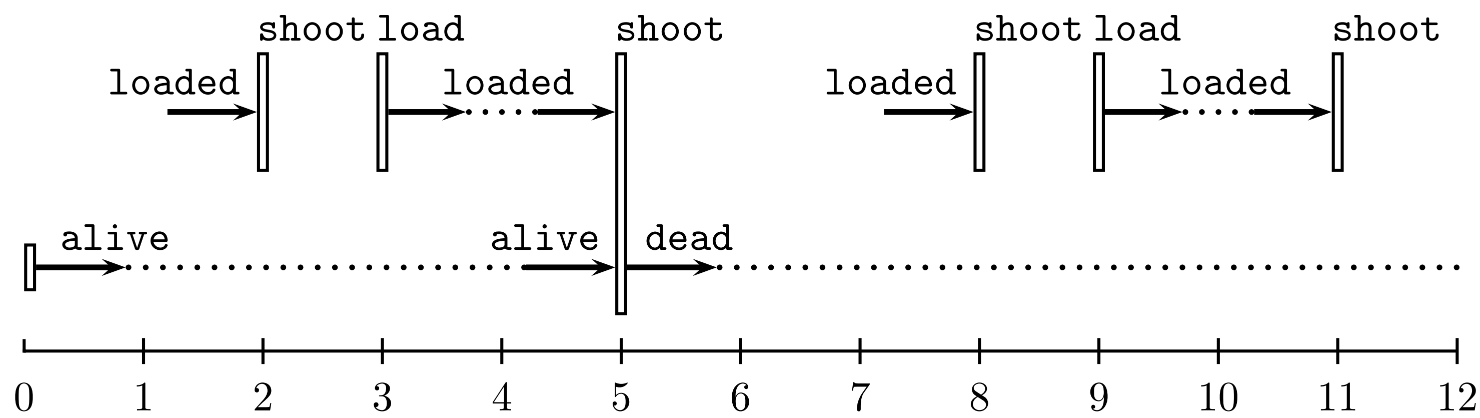
EC Formulation of YSP (due to Marek Sergot)

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

terminates(shoot,loaded,T).
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).
```



FWGC in EC

- We can specify the FWGC (farmer-wolf-goat-cabbage) problem in EC
- State representation:
  - Two values (l and r) suggests boolean fluents
  - Let f (is true) mean "the farmer is on the left bank"
  - So $\neg f$ means "the farmer is on the left bank"
  - etc.
- So the initial state:

```
initially( f ).
initially( w ).
initially( g ).
initially( c ).
```

FWGC in EC: Actions

- There are four possible actions (events)
  - The farmer moves on his own
  - The farmer takes the wolf
  - etc.

FWGC in EC: Initiates/Terminates

- The farmer moves on his own, so f 'toggles'
  - If f is false, initiate a period of time when f is true, and
  - If f is true, terminate a period of time when f is true
  - But: if the wolf is on the same bank as the farmer, then the goat had better be on the other bank
  - And, if the goat is on the same bank as the farmer, then the cabbage had better be on the other bank

```
initiates( moveFarmer, f, T ) :-
        \+ holdsAt( f, T ),
        (\+ holdsAt( w, T ) -> holdsAt( g, T );true),
        (\+ holdsAt( g, T ) -> holdsAt( c, T );true).

terminates( moveFarmer, f, T ) :-
        holdsAt( f, T ),
        (holdsAt( w, T ) -> \+ holdsAt( g, T );true),
        (holdsAt( g, T ) -> \+ holdsAt( c, T );true).
```

FWGC in EC: Narratives

- A sequence of events

```
/*
** with errors
*/
happens( moveFarmer, 1 ).
happens( moveFarmerGoat, 2 ).
happens( moveFarmerCabbage, 3 ).
happens( moveFarmer, 4 ).
happens( moveFarmerWolf, 5 ).
happens( moveFarmerGoat, 6 ).
happens( moveFarmerCabbage, 7 ).
happens( moveFarmer, 8 ).
happens( moveFarmerGoat, 9 ).
```

FWGC in EC: Planning

- Assume: perform one action at each consecutive time point
- At each time point, select an action, add (assert) to database
- If it leads to a consistent narrative, create history and try another
- If if leads to a 'dead end', retract, backtrack and try another

```
next_action( T, History, Tmax ) :-      % current time and sequence of states
        T1 is T + 1,                     % get the next time point
        possible_action( A ),            % each possible action at this time
        retractable_assert( A, T1 ),     % assert event happens at that time
        T2 is T + 2,                     % what is state after event occurs
        farmer_change_state( T1, T2 ),   % problem invariant: farmer changes state
        new_state( T2, History, NewH ),  % loop checking
        next_action( T1, NewH, Tmax ).   % find the next action
```

Summary and Conclusions

- The Event Calculus provides reasoning with fluents
- Planning and reasoning are important capabilities for implementing **intelligent agents** in **multi-agent** systems
- Symbolic reasoning is a necessary complement to sub-symbolic reasoning