

The Event Calculus as a Programming Model for Game AI

Matthew Fuchs, PhD
mattfuchs@paideiacomputing.com

Introduction

Game AI has progressed over the years to cover a number of techniques and is an essential component of several best selling games. However, faced with pressure for every increasing complexity - the ability to converse with players, engage in increasingly sophisticated behavior, and increasingly complex storylines, there is a need for a programming model able to support a dizzying array of requirements:

- Monitor any number of player and game properties in real time
- Support an efficient implementation of (tens of) thousands of rules implementing player and game behavior
- Reevaluate rules on any time scale, i.e., every frame or less
- Susceptible to formal analysis, model checking, etc.
- Extendable to deal with probabilistic behavior

We propose the *Discrete Event Calculus* (DEC)[Mueller06] as a programming model for game AI able to fulfill, or be reasonably extended to fulfill, these requirements. To test this, we have implemented a reaction engine for controlling real-time systems using DEC as a programming model. DEC has evolved from a sequence of logics proposed in the AI community to deal with situations that change over time.

What is the Discrete Event Calculus

The Event Calculus is a many-sorted logic carefully designed to mimic time varying systems in a way that doesn't fall prey to the *frame problem* from logic. It adds three sorts to the predicate calculus, *fluents*, *events*, and *time*. Time may be either continuous (represented by the non-negative reals) or discrete (represented by the non-negative integers). We will be dealing exclusively with the discrete version.

DEC provides a means of describing the current state of affairs at time t , where $t \geq 0$; the events that may occur at time t ; and the rules that describe how the two of them give rise to the state of affairs at time $t + 1$.

Current systems using DEC either use them deductively – given the current state of affairs and a possible set of actions at particular times, what can we know about the future, or for abductive planning – to determine if there is a possible sequence of events (plan) leading from the current state of affairs to some particular future one. In both cases, DEC is used to simulate the future by examining the effects of events on the current state, one state at a time. We, however, will be using DEC to enact the present.

In describing DEC, we will start with a very simple example and add some wrinkles later on:

1. John is walking along in the high grass
2. John sees a tiger.
3. Anyone who sees a tiger runs.
4. John starts running.

We can organize all of this with the following set of axioms (we adopt the syntax from [Mueller06], where variables are lower case and all other names are upper case):

```
sort mob.
mob John, Tiger.
fluent Walking(mob), Running(mob).
event Sees(mob,mob).
HoldsAt(Walking(John), 0).
Happens(Sees(John,Tiger), 4).
[mob,time]Initiates(Sees(mob,Tiger),Running(mob),time).
[mob,time]Terminates(Sees(mob,Tiger),Walking(mob),time).
```

A *sort* is an enumerated type. In this case we have the *sort* *mob*, (for *mobile object*), with two members, John and Tiger. A *fluent* is a term which is true at certain instants and false at others, expressed as *HoldsAt*(*term*, *t*), meaning *term* is true at time *t*. A fluent is either *true* (or *holds*) at the start ($t = 0$) or it becomes true when it is *initiated*. A fluent remains true until it is *terminated*. For example, *Running*(John) may be a fluent representing whether John is running, and *Walking*(John) may be a fluent representing whether John is walking. If John is walking at time 0 then we can say *HoldsAt*(*Walking*(John), 0). In our example, this is stated as an *axiom*. It is a given, from the text, that John is walking at time 0. If John starts running at time 5, then we can say *HoldsAt*(*Running*(John), 5). Of course something must cause John to start running. Fluent values are changed by *events*.

Suppose John sees a Tiger at $t = 4$ and starts running (so he's running at the next time step). We model this as the event *Sees*(John, Tiger). Since this occurs at $t = 4$, we can say *Happens*(*Sees*(John, Tiger), 4). Since anyone who sees a Tiger starts running (at our current state of complexity), we assert the axiom

```
[mob,time]Initiates(Sees(mob,Tiger),Running(mob),time).
```

Then, if John sees the Tiger at $t = 4$, *HoldsAt*(*Running*(John), 5) is *true*. Because we don't want anyone to run and walk at the same time, we also need to add

```
[mob,time]Terminates(Sees(mob,Tiger),Walking(mob),time).
```

(We use square brackets for universal quantification and curly ones for existential quantification. Therefore $[x, t]$ means "for all x and t " while $\{x, t\}$ means "there exist x and t ".) If you're already

running when you see the `Tiger`, then these rules have no effect – `Terminates(event, fluent)` only has effect if `fluent` holds when `event` happens, and `Initiates(event, fluent)` only has effect if `fluent` doesn't hold when `event` happens.

The Frame Problem

For this approach to work it needs to be the case that `~HoldsAt(Running(John), 0)`. However, as we don't explicitly state that, it cannot be proven from the given axioms. This is the *frame problem* – traditional first order logic requires an explicit statement of everything that is true or false for inference mechanisms to work as we would expect. We could state `~HoldsAt(Running(John), 0)` or add axioms stating that `Running` and `Walking` are mutually exclusive, but this becomes untenable as the size of the problem increases. The event calculus borrows a process called *circumscription* [Shanahan97] to handle this.

In essence, circumscription states the following about a DEC program:

- 1) No fluent holds at time 0 unless stated by an axiom.
- 2) Fluents only change value through `Initiates` or `Terminates`.
- 3) Events only happen if explicitly stated in the program or can be derived (i.e., on the right hand side of an implication).

Now we can infer that `~HoldsAt(Running(John), 0)`, as well as `~HoldsAt(Running(John), 4)` because circumscription ensures that John doesn't see the `Tiger` until $t=4$.

A DEC Engine for Games

DEC becomes interesting when it can interact with the outside world on a regular basis and guide the behavior of a process such as a game. To do that we map DEC time to game time - for example by having each instant in the DEC program correspond to a frame of the game - and use certain events as messages. Then at each instant, some events are added to the simulation from the outside world, the DEC axioms are (efficiently) evaluated, some fluents are updated, and some events are generated to the outside world. We will update our simple simulation to describe this behavior. In addition, we designate `John` as a non-player character (NPC) that's programmatically controlled and the `Tiger` as the Player. `Tiger` behavior becomes an extrinsic variable.

To accomplish this we rewrite our simple DEC scenario. We designate certain events as *external*, indicating they are transmitted outside the simulation. Other events are generated externally and simply happen within the simulation at the corresponding time step. We'll write the scenario in two ways. In the first, the external world moves the NPC with new positions added to the program as it moves. In the second, the DEC program will generate NPC movements. This will require some new machinery.

In the first version, we define a `Location` structure to keep track of where `John` and the `Tiger` are. The external events are `SetGait`, which establishes the speed `John` is moving at (and perhaps the animation as well) and `MovesTo` which gives `John`'s location at the next time step. The only other event we need is `Sees` which is generated when `John` first sees another moving object. Only two fluents are needed for this scenario - `Frightens` is true if the first mob frightens the second and `Position` tracks where everything is at each instant in time. When something is moving, this fluent will change at each frame,

but when objects stand still it will be more stable. There is also a predicate, `IsVisible`, which we don't define here, to determine if a `Location` is visible from another at a particular orientation.

We use nine axioms to create our scenario, with five defining the evolution of the system and the rest establishing the initial state. The first two establish that `MovesTo` events change the position of mobs. In the latter case we need to establish that the object is moving to a new position before terminating the old fluent – it is illegal to both initiate and terminate the same fluent in a single time step, as the results are indeterminate. The next rule establishes that if `mob1` can see `mob2` then a `Sees (mob1, mob2)` event happens. The next rule establishes that if a mob sees another mob that frightens it, it starts running. Next, assuming `moveAway` is a function defined elsewhere that takes two locations and returns the first with the angle changed to face away from the second, we turn the `John` away from the `Tiger`. The last three clauses place `John` and the `Tiger` at time 0, assert that the `Tiger` frightens `John`, and starts `John` walking.

```
sort mob, gait
mob John, Tiger
gait Walking, Running
structure Location(x,y,z,theta)
external event SetGait(person,gait)
external event MovesTo(mob,location)
event Sees(mob,mob)
predicate IsVisible(location,location)

fluent Frightens(mob,mob)
fluent Position(mob,location)

[mob,location,time]Initiates (MovesTo(mob,location), Position(mob,location), time) .

[mob,location1,location2,time]
HoldsAt (Position(mob,location1),time) & !(location2 = location1) ->
Terminates (MovesTo(mob,location2), Position(mob,location1),time) .

[mob1,mob2,location1,location2,time]
HoldsAt (Position(mob1,location1),time) & HoldsAt (Position(mob2,location2),time) &
IsVisible (location2,location1) -> Happens (Sees (mob1,mob2),time) .

[mob1,mob2,time]HoldsAt (Frightens (mob1,mob2),time) &
Happens (Sees (mob2,mob1),time) -> Happens (SetGait (mob2,Running), time) .

[mob1,mob2,location1,location2,time]
HoldsAt (Frightens (mob2,mob1),time) &
HoldsAt (Position (mob2,location2),time) & HoldsAt (Position (mob1,location1),time)->
Initiates (Sees (mob1,mob2), MovesTo (mob1, moveAway (location1,location2)),time) .
```

```

HoldsAt(Position(John, Location(0, 0, 0, 90)),0).
HoldsAt(Position(Tiger, Location(0, 0, 8, -90)),0).
Happens(SetGait(John, Walking),0).
HoldsAt(Frighens(Tiger,John),0).

```

In this initial version, we don't keep track of John's gait – we assume that the game engine underneath is doing that. However it is possible for the DEC program to determine John's position on a frame by frame basis. To do that we need to add a little more machinery.

There are two more predicates we can use to control fluents, $\text{Trajectory}(f, t, f', t')$ and $\text{AntiTrajectory}(f, t, f', t')$, where f, f' are fluents, t, t' are times). In the first case, if we have $\text{Initiates}(e, f, t)$, where e is an event, then we have $\text{HoldsAt}(f', t+t')$ until f is terminated. In the second, if we have $\text{Terminates}(e, f, t)$ then we have $\text{HoldsAt}(f', t+t')$ until f is initiated. These are interesting because the parameters of the fluent f' can vary with time. In our case we will use John's gait to drive his position. We then move John in the game by having f' imply an external MovesTo event.

We add a fluent, $\text{Gait}(\text{mob}, \text{gait})$, to our scenario which establishes John's gait - *Still*, *Walking*, or *Running*, for now. We also add five new axioms. The first two manipulate Gait and SetGait the way we did with MovesTo and Position previously. These pairs will eventually disappear into a macro or other syntactic sugar. The next axiom is a trajectory axiom and says that if a mob started walking at time t_1 and was at position (x, y, z, theta) , then as long as it is walking it will move one unit each time step along the angle theta . The next axiom has a similar significance for *Running*, but 3 times as fast. The final rule, which communicates back to the rest of the game, states that if a mob is moving (i.e., not *Still*), then its Position should be used to generate an external event to move the mob in the game.

```

[mob, gait, time]Initiates(SetGait(mob, gait), Gait(mob, gait), time).

[mob, gait1, gait2, time]HoldsAt(Gait(mob, gait1), time) & !(gait1 = gait2) ->
  Terminates(SetGait(mob, gait2), Gait(mob, gait1), time).

[mob, x, y, z, theta, t1, t2]HoldsAt(Position(mob, Location(x, y, z, theta)), t1) ->
  Trajectory(Gait(mob, Walking), t1,
    Position(mob, Location(x + (t2 - t1)*cos(theta), y,
      z + (t2 - t1)*sin(theta), theta), t2)).

[mob, x, y, z, theta, t1, t2]HoldsAt(Position(mob, Location(x, y, z, theta)), t1) ->
  Trajectory(Gait(mob, Running), t1,
    Position(mob, Location(x + 3 * (t2 - t1)*cos(theta), y,
      z + 3 * (t2 - t1)*sin(theta), theta), t2)).

[mob, gait, time, location]
!HoldsAt(Gait(mob, Still), time) & HoldsAt(Position(mob, location), time) ->

```

`Happens (MovesTo (mob, location) , time) .`

All in all, this basic scenario requires ten axioms.

We've provided two ways to have the DEC program interact with the underlying game:

1. The game engine moves the characters around with the results reflected into the DEC program.
2. The DEC program decides what happens at each frame and reports that to the game engine.

There is a third alternative where the game engine moves the characters around, but the DEC program simulates where they should be using Trajectory axioms. In that case the game is not required to inform the DEC program at the same rate, but there needs to be consistency code developed.

Extending our DEC implementation

As an introduction, the example above stays simple, however it could be easily extended to far more complex situations:

- There might be multiple NPCs. These NPCs could use exactly the same rules as `John`. Rather than naming them individually, they can be represented by fluents themselves. There may also be rules about how they interact with each other.
- The NPC may have many more choices. For example, if it has weapons it may choose to shoot at the Tiger instead of running. In that case there would be events for the possible choices of action (`MayRun`, `MayShoot`) which would be evaluated by other rules until a course of action is chosen. That may include multiple steps defined by further fluents and events.
- There might be multiple animals around. The NPCs may be frightened of some of them and not others.

These situations are handled through the development of additional rules, predicates and functions to describe the further complexity. Predicates and functions may contain complex algorithms to determine values, such as path finding, etc.

These cases also convey the inherently multithreaded nature of a DEC program. The same program can monitor and respond to the behavior of multiple objects as if they were in completely separate threads, but still have rules that monitor their interactions. Two NPCs wandering around the same savannah would each generate different `MovesTo` events.

However the fundamental power of DEC sits underneath, providing a systematic way of evaluating a current situation and determining how to proceed.

In particular, there exist mechanisms for performing model checking on a DEC program. That's a traditional way of handling DEC. From model checking we can do constraint analysis, planning and regression testing.

For DEC, a model is an assignment of a value, true or false, to each possible event and fluent at each moment of time, i.e., $\text{HoldsAt}(\text{fluent}, \text{time}) = \text{true}$ or $\text{Happens}(\text{event}, \text{time}) = \text{false}$, such that all the axioms are true. Model checking has an interesting interaction with circumscription as described

previously. If we circumscribe our fluents and events, then there is only one possible assignment, and model checking devolves to checking if the given assignment works. Rather than the program being something that evolves through time, model checking sees the program and its evolution as a static object.

For DEC, a test case is an initial state (fluents that hold at $t=0$) and the external events that arrive. A set of constraints would be an additional set of axioms specifying conditions that either must or must not exist. In our little example, a constraint might be that no one should be both `Walking` and `Running` at the same time:

```
[mob, time, gait1, gait2] HoldsAt (Gait (mob, gait1), time) &  
HoldsAt (Gait (mob, gait2), time) -> gait1=gait2.
```

A test passes if none of the constraints is false given the run of the program. Many possible constraints wouldn't work as part of a program because they are not directly related to behavior. For example, that a shotgun is over the fireplace at time 0 implies that someone shoots the gun before the game is over is a condition whose truth value can be assessed but not easily one that can cause events to happen.

However, if we relax circumscription, we can allow a model checker to look for different initial states for fluents or different times that events occur to find a working model. For example, one can relax circumscription on events to determine if there is any way for a player to win in a game, given an initial state. Alternatively one can find an ideal initial state that will be most successful against any opponent strategy.

DEC can also be combined with planning strategies. Planning algorithms generally work on a set of atoms describing the current state (fluents) and deliver a list of actions to perform (events). This can be easily combined with a DEC program. Alternatively one can apply abductive planning [Shanahan89] to generate plans directly from the DEC program by finding a set of events leading from a start state to a desirable final state.

Alternative models

Any number of approaches have been used to implement AI in games. (A*, MDP) The most ubiquitous are scripts and finite state machines. Behavior trees are a model with increasing frequency and there have been occasional efforts to implement planning and more traditional rule-based systems. In all of these there is a distinction between the code and AI model, and none of the AI models has an explicit notion of time. DEC provides a model that conveniently subsumes all of the above while providing an explicit notion of time.

... Additional material to go here ...

What systems can be represented by DEC?

EC developed out of the AI community and has been used to represent a wide variety of systems. M. Shanahan [Shanahan97] first used EC for robotic path planning. Mueller [Mueller06] describes using EC for a wide variety of applications, including electrical systems, business systems, story understanding, representing default knowledge, beliefs, emotional behavior, among others. [vanLambalgen05] applies EC to derive a sophisticated accounting for events in natural language semantics. This shows DEC is

applicable to a wide variety of AI problems.

More generally it can represent a wide variety of transition systems. In essence, the set of fluents that hold at any point describe the current state of the system, some events may occur representing events in the world (such as John seeing a Tiger), and the current state and these events combine with the rules to generate the next state of the world. Because the rules are defined using the predicate calculus they can be very powerful. In fact, the rules are clearly powerful enough to describe a Turing machine.

Until now, EC has been used for reasoning, rather than execution. Given a state of affairs and a set of rules, one can query the system by adding an axiom about the system at some future time and demonstrating that the system is not inconsistent, either by running the system until then, or potentially by reduction to SAT and demonstrating a model. Another interesting line of work has been on abductive planning [Shanahan89]. Here we take describe the current state and a future state and let the system abduce a sequence of events moving the current system to the desired one.

Implementation Issues

How to make DEC fast

DEC axioms are of the form *if X then Y*, where X is a formula over `HoldsAt` and `Happens` terms, occasionally mixed with other functions. With no backtracking, the left side of the implication can only be true at any time *t* if there are (or are not, for negated terms) `HoldsAt (fluent, t')` or `Happens (event, t)` atoms floating around at time *t*. Now if the rule is

$$\text{HoldsAt}(\text{fluent}, t) \ \& \ \text{HoldsAt}(\text{fluent}', t) \ \rightarrow \ \text{Happens}(\text{event}, t).$$

and it's true that `HoldsAt (fluent', t)` but perhaps not `HoldsAt (fluent'', t)`, then we can consider the rule partially activated and the degree of activation of any rule may vary over time as certain events happen, or certain fluents hold or don't (if *fluent* suddenly no longer holds, the rule is completely inactive, and if *fluent'* also holds then it fires). Given this changing structure over time, we can apply the Rete algorithm [Doorenbos95] to optimizing the activation and firing of DEC rules. The Rete algorithm, from the Greek word for net, compiles a network of rules, generally for a forward chaining expert system such as Jess or Drool.

If one considers a naïve implementation of DEC, then at each clock tick, the system looks at all the axioms, fluent, and events, and decides which axioms are applicable. This process scales badly with the number of axioms, events and fluents in the system. The Rete algorithm reorganizes the axioms (or rules in a forward-chaining system) into a network so that each fluent or event is matched only once with all the axioms it might activate. If a fluent is terminated, then it is removed once. When enough fluents and events are available that the left hand side of an axiom is true, that rule is immediately executed. The Rete algorithm (and its successors) therefore represents a huge improvement in runtime efficiency and is essential to a useable implementation. Given the efficiency of current Rete implementations, rule bases in the tens of thousands of rules are feasible.

Current Implementation

Rather than implementing DEC from scratch and needing to implement both the language and the Rete algorithm, we chose to implement DEC by translating to Jess[Jess], a popular forward chaining production system written in Java with a syntax derived from CLIPS. Using Jess gives us immediate access to the external world through its Java interface as well as a robust Rete implementation.

As a forward-chaining system, Jess represents the world by a set of atoms, or facts, represented by ordered or unordered S-expressions, such as (*functor*, *param1*, *param2*), where *functor* is the name of the fact and *param1* and *param2* are values. If the atom is unordered, then the parameters are a list of pairs, where the first value is the name of the parameter, or “slot name” and the second is its value.

Beyond translating DEC to Jess, we also added a couple of essential constructs – compound structures and lists (which, of course, are compound structures) – to be able to represent the kind of information used by actual applications. By remaining functional and not permitting the mutation of structures or explicit use of pointers, we avoid circular structures and can use reference counting for garbage collection of structures and lists. Events and fluents do not require reference counting to be collected – events expire after they occur, so we can easily collect and retract (in the language of Jess) all events less than the current moment. Fluents are collected when they no longer hold.

We represent fluents and events as ordered facts, with the first value always being the time that the fact was created. As axioms may assert the existence of events or the truth of fluents at arbitrary time points, but processing proceeds linearly, it is essential to know which events occurred at the current time and not in the future, and to ensure that a fluent holds now. Rules matching these generally check to ensure that the time values are correct. Structures are organized as unordered structures, as we use the slot names in garbage collection.

The DEC process is run from a Java driver that communicates between the DEC and the rest of the world, reading in events from the world for DEC and sending DEC events to other processes. At each time step the process goes through four stages:

1. Fluent contradictions, where there is an attempt to both initiate and terminate a fluent, are trapped. If there are no contradictions, the string representations of externalized events (the means of communicating out to the rest of the system) are removed.
2. Terminated fluents are removed by matching existing fluents against `retractFluent` facts. The origin time point of the retracted fluent must be less than or equal to the time that the `retractFluent` fact was generated. Likewise, new fluents are created from `createFluent` facts, with the `createFluent` fact containing all the information for the new fluent. The `createFluent` and `retractFluent` facts were created during the last round. We also need to clean up events remaining around. For each event and fluent we need to check which fields refer to structures to decrement reference counters. For each of these we generate a `modifyRefs` fact. To manage this, rules are generated to handle all of these correctly.
3. We garbage collect all the structures by matching all the `modifyRefs` facts generated in the previous

phase. Structures whose reference counters are 0 are retracted and their children's reference counters are also decremented. These structures may be more easily dealt with by just creating Java beans, in which case the garbage collection will be handled by the Java runtime. However an important part of rule matching is structure matching within the rules. This is currently handled inside the generated Jess code, although it is not a complete resolution algorithm. If we move this to Java, we would need an algorithm for performing resolution over Java beans. We wait to this point to do the garbage collection because there may be a mix of increments and decrements for a single structure if it is passed from an event to a fluent (the first generates a decrement when the event is collected, the second generating an increment when the fluent is created).

4. Having retracted/asserted facts for all the fluents generated from the events of the last round, we can now execute all the axioms that generate new events. Before that, though, we add in events coming from outside. Some generated events will also need to be communicated outside the simulation. For these, additional rules are generated that create a string representation of the event, including embedded structures.

The Java driver executes phases 0 – 2, adds events into the DEC program, executes phase 3 and finally collects events for the outside world. We use a Jess command to run through the list of events and gather representations of all the external events. At startup we run through one complete cycle to let the DEC program generate the initial events for the outside world.

Bibliography

[Doorenbos95] Robert Doorenbos, *Production Matching for Large Learning Systems*, PhD Thesis, Carnegie Mellon, Pittsburgh, 1995.

[Jess] *Jess, The Rule Engine for the Java Platform*, <http://www.jessrules.com/>

[Mueller06] Erik Mueller, *Commonsense Reasoning*, Morgan Kaufmann, 2006

[Mueller04] Erik Mueller, "Event Calculus Reasoning Through Satisfiability", *Journal of Logic and Computation*, 14(5), 703-730, 2004.

[Shanahan97] Murray Shanahan, *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press, 1997

[Shanahan89] Murray Shanahan, "Prediction Is Deduction but Explanation Is Abduction", *Proceedings IJCAI 89*, 1055-1060, Detroit.

[vanLambalgen05] M. van Lambalgen and F. Hamm, *The proper treatment of events*, Blackwell Publishing, Oxford, 2005.