The AIIDE-17 Workshop on
Experimental AI in Games
WS-17-19

A Proposal for a Unified Agent Behaviour Framework

Javier M. Torres
Brainific SL
javier.m.torres@brainific.com

Abstract

Games have used different mechanisms along its history to provide agent behavior: FSMs, utility systems, behavior trees and planning methods. In this paper, we present an architecture that aims at incorporating all these approaches into trees of event handling nodes with behaviours as leaves, using rules for combining actions akin to utility systems. This formalism aims to develop hybrid systems in an easier way.

Introduction

Reactive NPCs in current games use techniques like Behaviour Trees (BTs) and Finite State Machines (FSMs), which keep a static structure however complex they may get, and are so specialised that interaction among them is difficult. We propose a formalism based on event-handling processes that can specialise into both techniques, and may naturally include other techniques like steering behaviours and planning algorithms.

Related work

Finite State Machines and Behaviour Trees are two of the most common formalisms used to describe the behaviour of agents in games. Whereas hierarchical state machines have a long history in video games, behaviour trees were introduced more recently by Damián Isla (Isla, 2005). The same structures are used in AND/OR search tree to solve nondeterministic planning tasks (Dechter and Mateescu, 2004a); in fact, as explored previously (Colledanchise, 2017), BTs are a generalization of several formalisms like the subsumption architecture (Brooks, 1986). In the last 12 years, they have been the subject of many improvements and refinements over this basic idea.

For example, Champandard and Dunstan (Champandard and Dunstan, 2013) present event driven behaviour trees, where only active behaviours are checked for changes in the environment. Decorators are also a staple of BT implementations, which modify the behaviour of a node; see (Epic Games 2017) as an example.

The CERA Cranium architecture (Arrabales, Ledezma et al., 2009) from the Conscious Robots team at the 2K Bot Prize at CIG '10, on the other hand, has uses a layered approach, separating perceptions, atomic actions and more complex plans.

Why yet another mechanism?

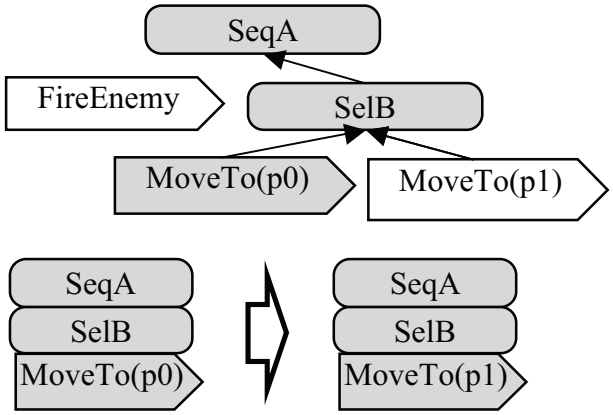
After almost 12 years of behavior trees, and more of HFSMs, why another formalism? On the one hand, both BTs and HFSMs are bound in their configurations (the number of nodes is known and bound at runtime). On the other hand, computer games have tight resources that are seldom dedicated to AI, and tight design processes that leave little room for experimentation. If we make an analogy with function calls, we would be programming AI with preallocated frames, even sometimes with several instantiations of the same function call.

A first change is to dynamically instantiate nodes in a BT instead of fixing the memory layout at design time. If we consider that a selector or sequence is far more common than a parallel node, and then we gray out the nodes, we can see that we can use a stack to store active nodes (see Figure 1). Every parallel node spawns a separate stack, but we can still restrict the depth of each stack. We can just make a "function call" to instantiate the nodes and even use more traditional constructs to obtain the next node, like creating a sequence of MoveTo nodes from a list of positions obtained at runtime, instead of resorting to decorators. We acknowledge that the current tools provide hooks for subtrees, for example, but we feel that an even much more dynamic approach should be taken.

However, this flow only concerns node creation. In a BT, Success and Failure (at least) events are handled from the leaves upwards and modify which nodes become active. It may be argued that, since the state is only reported

once, it is a true function call, although it would run across frames.

We can now develop each node in the tree completely decoupled from the rest. It is true that we cannot see the complete tree anymore, though, and we need to develop the nodes in a written language and not on a visual tool. As such, each node is subject to reuse, just like a function definition. Note that this also allows us to insert subtrees on the fly, even generated subtrees via planning, since nodes are data structures.




```
SeqA (String arg) {
  return FireEnemy(enm1) &&
    SelA; }
SelB (Int arg) {
  return MoveTo(p0) ||
    MoveTo(p1); }
```

Figure 1 Traditional Behaviour Tree and its view as a function stack

A second improvement is to decouple node creation and lifecycle from event handling, so once created a node can react to events from its children, process them and send events upwards, and not just a single "return" event. A suitable formalism to express these systems can be Communicating Sequential Processes (Hoare, 1985). Event driven behavior trees work in a similar way, but by relaxing the conditions on event types and event processes we can approach formalisms like HFSMs. Also, separating the frame-by-frame aspect in a behavior (e.g. a steering behavior) from the node in a behavior tree representing it should allow the designer to focus on immediate reactive behavior vs. more complex behavior.

A final aspect is to separate behaviors from the final set of actions taken. Take a parallel node that activates Cover and FireEnemy behaviours. It would be very simple and natural to decide which one to finally perform by combining them in a utility-based selector. This would be independent of where in our tree the actions have been activated: whether taking cautious approach or cautious retreat, or whether we use fire to take down or suppress an enemy, the balance between cover and exposure would feel similar in all situations.

Architecture

The proposed Action Framework architecture comprises the following elements, featured in Figure 2:

Behaviour Composer Table	
Cover	U = K1*HP*BulletsInFlight
FireEnemy	U = K2/BulletsInFlight

Figure 2 Elements in the Action Framework

Behaviours, that output actions reactively every time frame and send events upwards the action tree; they are percepts as much as actions. Some behaviours do not output actions but perform some algorithm on internal data over a series of time frames.

Action nodes, that help select behaviours. They are very similar to nodes in behaviour trees and states in FSMs, with the following properties:

They send events of a certain type to their parent and handle events of another type from their active children. In the case of behaviour trees, these events are often Success and Failure. FSMs can recognise other events.

They use a transition function that takes the current subtree whose root is itself, together with its children and its state, and a list of upcoming events, and replaces it with another potentially new subtree with a new state, whose only requirement is to send upwards the same type of events as the previous subtree.

A **behaviour composer**, that resolves conflicts between behaviours. In this layer, compatible actions like steering behaviours are combined and incompatible actions are selected according to their utility.

Action Tree

The action tree is composed of the hierarchical structure of currently action nodes and behaviours. Every time an event arrives at the action tree from a behavior node, it traverses the tree upwards, possibly causing changes in the action tree.

Note that there is no global and total description of all the configurations that the tree of active nodes and behaviours may take, as in a programming language there is no global and total description of all the possible function expansions. Instead, we can inspect the currently running tree of action nodes and behaviours. Debugging is made possible by inspecting traces of incoming and emitted events, and the current action tree.

Events that are not handled in lower layers are passed upwards to be handled by higher order nodes. The action tree changes potentially every tick. Figure 3 shows an example of an action tree. It should not be mistaken as a fully specified Behaviour Tree, but rather as the tree of currently active nodes in a Behaviour Tree.

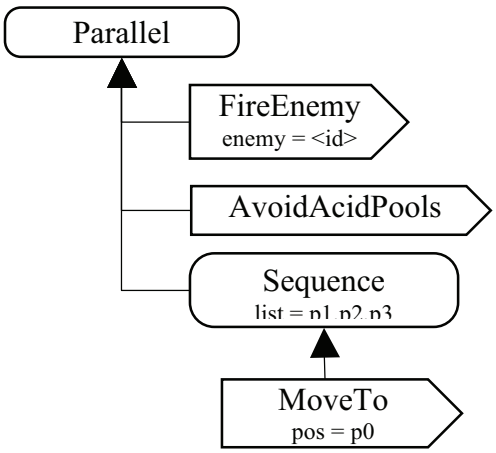


Figure 3 - Action Tree example

Action nodes

Action nodes in a tree are defined by some internal state, their children nodes, and a transition function. Note this specification is a generalization of both behaviour trees and HFSMs: whereas in _BTs the state and the children nodes are updated (e.g., a Sequence node updates the current children by taking a node from its list of actions, as well as this same list by removing its head), in FSMs the active "node" itself is turned into another "node" (note that state nodes with children can be used to model hierarchical FSMs). In the following textbox a sequence node and an FSM are described in a Java-like pseudocode.

Note that the onEvent functions may handle any type of event, notably an event containing a subtree that then gets returned and inserted into the action tree.

```
class Sequence extends Node
   def Sequence ([Factory<Node>] seqList):
      [head | tail] = seqList
      currentNode = head.new()
      self.next = tail
      return (self, [currentNode])
   def ( Optional<(Node, [Node])>, [SuccType] ) onEvent
(Node n, [Factory<Node>] next, SuccType evt):
      case evt of:
         Failure: return (Nothing, [Failure])
         Success: case self.next of:
            []: return (Nothing, [Success])
            [head | tail]: ( (self, [head.new()]), [] )
         end
      end
class StateA extends Node
   def StateA ():
      return (self, [])
   def ( Optional<(Node, [Node])>, [OutEvtType] )
onEvent (Node n, [Factory<Node>] next, InEvtType evt):
      case evt of:
         inEvtA: return ( (StateB.new(), []), [outEvtA] )
         inEvtB: return ( (self, []), [] )
      end
class StateB extends Node
   def StateB ():
      return (self, [])
   def ( Optional<(Node, [Node])>, [OutEvtType] )
onEvent (Node n, [Factory<Node>] next, InEvtType evt):
      case evt of:
         inEvtA: return ( (self, []), [] )
         inEvtB: return ( (StateA.new(), []), [outEvtB] )
      end
```

Implementation

There is an ongoing and preliminary open source implementation of this architecture in Haskell [1] being integrated in the Windows platforms using the Haskell FFI. It is used to explore and prototype new concepts. It currently implements common BT nodes and an FSM.

Conclusion

The Action Framework defines common concepts that can be specialized and combined to form some of the existing mechanisms for agent AI in games, like FSMs, behavior trees or planning algorithms; mainly, arbitrary event definitions and event handling trees. We aim at easily reformulating existing mechanisms, but also creating interesting new possibilities that can be obtained by assembling variations of these concepts.

[1] https://bitbucket.org/brainific/action-fw

References

Arrabales, Raúl, Agapito Ledezma, and Araceli Sanchis. "Towards conscious-like behavior in computer game characters." Computa-tional Intelligence and Games, 2009. CIG 2009. IEEE Symposium on. IEEE, 2009.

Brooks, R. (1986). "A robust layered control system for a mobile robot". Robotics and Automation, IEEE Journal of [legacy, pre-1988]. 2 (1): 14–23. doi:10.1109/JRA.1986.1087032. Retrieved 2008-04-14.

Champandard, Alex J., and Philip Dunstan. "The Behavior Tree Starter Kit." Game AI Pro: Collected Wisdom of Game AI Profes-sionals (2013): 73.

Colledanchise, Michele. Behavior Trees in Robotics. PhD diss. KTH Royal Institute of Technology, 2017.

Dechter, R., & Mateescu, R. (2004a). Mixtures of deterministic-probabilistic networks and their AND/OR search space.

Epic Games. 2017. “Unreal engine documentation”

https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/NodeReference/Decorators/.

Graham, David. "An Introduction to Utility Theory." Game AI Pro (2014): 113-126.

Hoare, C. A. R.: “Communicating Sequential Processes”, Prentice Hall, 1985, ISBN 0-13-153289-8.

Isla, D. “Handling complexity in the Halo 2 AI.” Game Developers Conference, 2005.