Implementation Walkthrough of a Homegrown "Abstract State Machine" Style System in a Commercial Sports Game

Brian Schwab

2335 Seaside Street
San Diego, CA, 92107
Brian_schwab@yahoo.com

Abstract

When I began working at Sony Computer Entertainment of America in 2002, the AI system they were using was very dated. Over the next few years, I designed and developed an almost completely data driven system that has proven to be very powerful, extremely extensible, and designer friendly. This system uses a homegrown data structure, the use of which in many ways resembles the software method of using Abstract State Machines for decomposing complex logical constructs iteratively. This paper will provide an overview of the construction and usage of the system, as well as the pros and cons of this type of game AI engine.

Basic System Architecture and Terms[·]

The overall AI engine I use is divided into two main parts: the **Situation** system, and the **Behavior** system. The Behavior system is the lower level, animation selection code. It handles animation priority issues, animation branching, blending, and alignment/positioning issues. It was implemented as a more straightforward HFSM architecture with a graphical editor. A vast amount of very complex behaviors can be implemented within this editor, and in some ways its functionality overlaps the Situation system. I will not delve any more into this part of the system, as the main thrust of this article concerns the Situation engine.

The Situation system sits a layer above the Behavior system, and in a nutshell defines the *when* to the Behavior system's *how*. You could think of the Behavior side as being the tactical decision maker, with the Situation system producing the higher level strategic decisions. The majority of the communication between these two systems is either message based, or uses the overall game blackboard, which is a shared data area that almost any part of the game can access.

Situation Basics

Each Situation defines a few, distinct elements:

- Overall game **conditions**. This defines the current state of game and/or team variables and conditions that must be true in order for the overall situation to fire.
- Any number of involved **roles**. Each role is an actor specification that defines a description of the type of player that will fill the role successfully. Role definitions can be as general (e.g., any offensive player on the left half of the court) or as specific (e.g., any defensive player that is 6'10" or taller that has a shooting skill greater than 75 that is less than 3 feet from both the ball handler and the right sideline). Since roles are really the main selection method for state, many different types of roles can be defined. You have not only the various player level variables and conditions listed above, but also considerations like: allowing multiple agents to bind to a single role (and thus receive identical behavior assignments), setting up exclusion bindings (which invalidate the situation from firing if the role is bound, useful for exceptions to rules), specifying if the particular role is essential to the firing of the overall situation (or optional), and if the role binding also requires the agent to actually be able to start the behavior assignment immediately or if queuing is allowed.
- Optional **behavior assignments**. Any role defined in the situation may be given behaviors to engage in if the overall situation is chosen for execution. Behaviors can be singular (run to here), or given in a behavior **chain** (run to here, then pass the ball to that guy, then run over there, then wave your hand).
- **Monitor definitions**. Monitors are small variable and/or expression checkers that can be used at almost any level of a situation (the high situation level, on any given role, or embedded within behavior chains). When true, monitors can set off events and/or write out values to the blackboard.

Situational Connectivity

All the individual situations are then connected in a tree-like structure called a Situation Network. In this system, the network is "tree-like" in that it connects like minded situations in a tree style way, but loops and other

specialized traversal methods can be set up, making the network more like a graph.

Because of the massive dynamic interconnectedness of the states within a real-time sports game, I didn't want to go with a straight state graph. The lines of state connectivity would quickly become unwieldy. Other HSM diagramming methods (like embedded states: where a large box is a parent state, and any smaller boxes inside of it are thus child states) have proven very un-intuitive to designers, and also makes loops hard to see. The decision tree structure makes a lot of sense to most people, and by extending the way that we traverse a decision tree, we can still get graph like behavior out of it.

However, by using a tree like structure, we have also made it slightly more difficult to see specific loops or jumps in logic. What I use instead is specific embedded formatting visualizations within the editor (color coding, use of icons, and a scoping window), so that authors using the tool can see jumps in state flow without mucking through tons of connectivity lines.

Polling Group Basics

Each situation can live on its own, or have any number of Situation "Polling Groups" (PGs) beneath it. These PGs are actually just folders of additional situations that provide the author with the method for grouping child situations under specific settings for *situational type*, *scoping*, *role inheritance*, *priority*, *polling policies*, and *exit policies*.

- **Situational types** include persistency, and abstractness. Persistency is like the program counter of the overall Situation network; it marks the currently running master situation that controls the current scope of the tree. Only one persistent situation is running at any given time. Entering a new persistent situation will cause all roles involved to stop what they were doing (if possible) and start something else when they fire. Non-persistent situations don't upset the current state of the tree; they only override specific individuals with behaviors. So, if the overall game is running a persistent situation, several players can be given other tasks without interrupting the overall state of the game. An abstract situation is one that includes no behavioral assignments to any of the included roles. These situations are pure decision tree nodes, serving only as placeholders for logic (or monitors). They fire if true, but then immediately poll their children, so that a non-abstract situation can be reached that will give the players some behavior.
- **Scoping** involves a few things. At any level of the tree, situational authors can define interruptability (meaning, which situations can interrupt any given executing situation) with a few different methods. The structure of the tree itself obviously gives one level of scoping. Interrupt Flags can also filter out situations that you don't wish to pre-empt currently running situations. These are just group bitfield flags, which function as a rudimentary ID, that you can tag onto any situation. Each situation can then list which flagged situations it will allow to interrupt. Lastly, overall PG priority level (discussed below) will also limit other groups with lower priority from coming along and taking control.
- **Role Inheritance** allows the authors to actually bind players to roles in an inherited fashion. You can either define roles as further specifications on older roles, or you can inherit the specific player that was bound to a role in any parent situation. This is very useful as a kind of role history, where you are able to refer back to roles that happened in the past and gain access to the specific player involved.
- **Polling priority** is simply a number that the system uses for ranking situations in terms of "importance." High priority situations are not only checked first, but cannot be interrupted by lower priority situations.
- **Polling policies** determine how a given folder of situations is "tested for truth". The policy can be set to: polling on a tick basis (either every tick, or some schedule), purely event based (meaning not polled at all, but becoming active only when a specific event comes in), and a number of miscellaneous policies like OnReset (which means "poll this folder every tick until a situation fires, and then don't poll at all until that situation exits or is aborted"). Other miscellaneous policies include OnEntry/OnExit (which are folders that are only checked upon entry or exit of the parent situation), and OnTickCheckPriors (which polls an entire folder until something fires, and then will continue to poll situations higher in the folder then the one currently executing; this is useful for heavily prioritized setups).
- PG **exit policies** are just as varied. Some examples include: exit when all behaviors are done, after some set time period, or when any leaf node underneath you has exited. You can also set up a Monitor to watch for a customized exit strategy, and either have the monitor exit directly, or have another situation communicate back that it is time to exit.

Architecture

Although I refer to this system as a type of "Abstract State Machine," it in no way follows rigorously the classical ASM syntax. In fact, I was not aware of the ASM design methodology until well into the second year of development of the system.

However, the concepts of ASM use have dovetailed so nicely with my system's design that we have essentially

been using the ASM method for dealing with our AI since the implementation of this new system.

We begin by building "ground models" to form requirement specifications for new features that the game will require. We then gradually build up these models into a full solution with iterative passes that slowly increase in complexity and specificity. Also, the rule-based nature of ASMs lends well to expert-knowledge filled environments like professional sports. Sporting games have not only well documented rules of play, but well documented individual player statistics and tendencies, which form the basis of "secondary" simulation rules. In this way, we can start construction of new AI logic quickly, without having to consider the minor details of implementation. We can then later add as much depth to any part of the AI as we see fit and have time for.

Like an ASM, situations in this system can describe states that themselves are not really points in the state space, but rather form algorithmic relations between variant areas of state space. The situational structure provides both the transitional rules (by way of the structure itself and the interruptability rules) as well as the behavior at each state within the situations.

In a way, both ASMs and my structure provide a sort of combination of State Graph and Decision Tree. Since rules can be used at any point to delineate either state transitions or selection criteria, you can freely "switch" between encoding logic in a decision tree format with logic embedded in state machine style format, depending on the author's needs at the time. Abstract situations give the author pure decision tree ability, creating deep logical structures that are parsed to separate out complex game state conditions. Interruptability flags, polling policies, and control over scoping issues give designers a huge amount of leeway as to how the situation states connect to one another.

Finally, the use of role inheritance greatly increases the author's ability to build coherent, richly structured behavior. By allowing the situation builder to re-use old roles and/or build upon old role designations, he can sequence very complex chains of behavior in an intuitive and straightforward manner. Authoring classical state based behaviors that reach a decent level of complexity can become quite difficult; the author has to think ahead, and construct all the substates inside of a larger parent state that can contain all the intermediate variables and roles necessary for inheritance (or else incur a fairly sizable refactoring task). In this system, the author can just build situations as he normally would, and when he wants to inherit from a parent, he can do so with a few mouse clicks.

System Usage

The system was designed for a few things: to enable game designers the ability to directly develop complex AI systems, and to not overly limit the power of the system by dumbing it down. Overall, the system was very successful. In only a few years, the system has gone from being primarily a tool for programmers to something that most of the team uses for a large number of AI-related activities. Designers are currently almost completely in charge of the primary game AI. Artists use the system directly to do simple, art driven AI for groups of agents like the crowd. Non-gameplay programmers use the system for presentation, sound, and camera tasks.

The system uses a mixture of debugging information. In-game diagnostics and visual debugging info is complemented by logging and a full game recording and playback system. While the game is running, you can pause, rewind the action, and then bring up debug information pages that will show you what situations are currently active, as well as what situations were tested on each tick, and why they failed to fire. This last year we also put in a system to update situations on the fly, so that authors could change a situation in the editor, and send it directly to the development kit while the game is running. This allowed for much more rapid iteration on situations, since it cut out some of the turn-around time associated with packing up data files, and lengthy restarts of the game.

Debugging can also be achieved from within the system itself. Much like putting special code into a function to help debug a twisty logical construction, situation designers are able to construct "helper situations" that will force the game into specific setups, or allow other backdoors to get quickly to particular game conditions. Then, key situations can be tested for much more easily then waiting until the specifics might happen during a live game. These helpers are then discarded once the situation is operating as wanted.

The only parts of the system that are still currently "code-based" are the perception system and the low level animation helpers (which in our game are called Smart Targets and Motion Controllers). Perceptions are in-game variables that can be accessed by the rest of the system; things like "distance to the ball" or "am I in the back half of the court" and the like. Smart targets are structures that manage positions a particular behavior is headed towards, with automatic avoidance, occupancy concerns, and specialty placement algorithms controlled by code. Motion controllers are special functions that help with low level custom animation issues. Note that there are currently plans to data drive both the perceptions and the smart targets using the situation system.

Benefits of the system

The system allows for very rapid creation of complex synchronized AI behavior chains. The combination of easy, inherited dynamic role binding and the vast array of options allow almost any AI problem multiple solutions. The first year the system was in place, I implemented the first round of AI behavior using the system myself. The total XML file size for the core game AI was around 700k. The next year, a designer was assigned to learn the system,

and I moved into more of a support role (as far as the core AI was concerned, I was really working more on extending and debugging the system). The AI file size doubled in only 6 months of designer time. The year after that, three more designers were assigned, and the AI content grew to almost six times its original size. On top of that, an additional megabyte or so of AI data was produced for other elements in the game.

The ability to tweak polling policies and the like give the system an amazing ability to tune for performance. The first year, we were able to make the system run in one-third the CPU time with only two days of tuning, and without any degradation of AI game performance. Now that the designers know some general rules of thumb about scheduling as well as limits on the number of situations we can check per tick of execution, they can tune the game themselves for performance using the in-game debugging tools.

Drawbacks of the system

The Situation editor has quite a hefty learning curve. With all the available options and settings, it takes approximately a week of instruction and plenty of examples and documentation to really get a programmer to use the system with competency. Designers took a bit longer as you might have guessed. However, considering that the system is powerful enough that, like c++, it is almost trivially easy to author constructions that do absolutely nothing useful; we have had huge success in getting non-technical staff to embrace classical programming concepts like logical constructs, scoping issues, and inheritance. In fact, within a year or so of working within the system, some designers were asking for exotic new polling policies and priority systems as they constructed elaborate setups that had never occurred to us before.

However, this brings up a point. The system is very powerful. It is essentially a full blown visual "programming language" for developing the game AI. As such, we've had to deal with many of the problems associated with green programmers, except with our designer staff. Teaching good programming practices to our designers has become part of my job. Like stubbing out behavior trees with default behavior and *then* adding higher priority specialization, so that AI controlled players will always have something to do. Even the intricacies of using version control software, including XML based merging and diffing of files to make sure that they're not checking in garbage. We have even begun to notice that many of the designers suffer from beginning programmer maladies such as the "I want to re-do everything myself" syndrome. It has been interesting finding out that programmers aren't alone when it comes to many of these types of professional behaviors.

As with any heavily data driven system, debugging is always an issue. Without a dedicated debugger, we rely on in-game tools for the majority of our efforts. This means the standard hit for any data driven system, in that you are not only working on maintenance of your product, but are now also linked to the maintenance of the AI tool chain. Time budgets must now be split between tool upgrades, and other coding areas that need attention. However, by keeping the system very open ended and powerful, we have escaped the vast majority of major overhauls that would have been needed had we limited the tool in the beginning (in the name of simplicity). Over the last two years, the tools have really only changed in small ways, as special circumstances have come up where the designers wanted specific new features from the system.

Conclusions

This flavor of free form state tree has vast potential for game development. It provides intuitive authoring of chains of behavior through a state based interface, while at the same time overcoming some of the limitations of typical FSM or HFSM systems by extending the paradigm with role inheritance and polling/exit policies that allow for a stunning amount of control over program flow.

Overall, the system has been a huge boon to our current product, and the extended system is being used in other products that we are working on in-house. We are continually trying to make the user interface of our editing tools better, as well as improving the debugging systems and author help techniques. I feel that on the whole I have learned a great deal about creating highly extensible AI systems, while at the same time gaining valuable insight into how to get non-technical staff into the game of AI development.