Some Issues in AI Engine Design

Marc S. Atkin, Gary W. King, David L. Westbrook and Paul R. Cohen
Experimental Knowledge Systems Laboratory
Department of Computer Science
140 Governor's Drive
University of Massachusetts, Amherst, MA 01003-4610
{atkin,gwking,westy,cohen}@cs.umass.edu

Abstract

Our goal is to build an "AI Engine" akin to the graphics engines that have revolutionized some parts of the game industry. A central issue is finding a general way to specify and control an agent's actions, since it is the behavior of an agent that makes it intelligent. We describe our agent design toolkit HAC (Hierarchical Agent Control), focusing on the lessons learned during its development. HAC is an action-centric uniform language for specifying actions that uses an abstract resource and sensing model. We conclude by identifying some of the guiding principles that have emerged during our work.

Introduction

Graphics engines have revolutionized some parts of the game industry, freeing developers from having to reinvent the wheel every time they start a new graphics-intensive project. If, by analogy, it were possible to design an "AI engine", a set of predefined tools for making games "smart", there might be equally drastic improvement in the quality of game AI.

We have developed a very general agent control architecture (HAC: Hierarchical Agent Control) that provides a uniform way of specifying and controlling agent behavior across a wide variety of continuous, real-time domains. Although our primary application is a military simulation system called "Capture the Flag" (Atkin, Westbrook, & Cohen 1999), HAC can equally well be applied to such domains as commercial games, multi-agent simulations, or actual physical robots (Atkin *et al.* 1998).

HAC can be viewed as a set of language constructs and support mechanisms for describing agent behavior. The primary idea behind it is that while agent behaviors may differ across domains, the architecture that controls them can remain constant. Arguably, specifying *what an agent does*, how it goes about achieving its goals, and how it reacts to its environment and other agents is a large part of what makes an agent intelligent. We therefore believe that some form of behavior description must be integral to any AI engine.

HAC itself and the hierarchy of actions it supports have gone through a number of revisions as we have added features to our simulator and moved to new domains. Throughout this process, we strived to keep HAC as general and as simple as possible. Based on this experience, we are designing a successor system, "HAC II", which aims to simplify the action description process and avoid common problems we encountered. We feel that potential AI engine designers may benefit from learning from our experiences: Why did we change certain features? Why did we prefer one solution over another? What can our development cycle teach us about the issues involved in constructing a general AI engine? In this paper, we will attempt to shed some light on these questions by looking at specific examples of problems that came up.

HAC: Hierarchical Agent Control

HAC is a language for writing agent actions. HAC takes care of the mechanics of executing the code that controls an agent, passing messages between actions, coordinating multiple agents, arbitrating resource conflicts between agents, updating sensor values, and interleaving cognitive processes such as planning.

HAC organizes the agent's actions in a hierarchy. The very lowest levels are the agent's effectors. The set of effectors will depend on the agent and the domain, but typically include being able to move the agent, turn it, or use a special ability such as firing a weapon. More complex actions are built from these primitive ones. An **attack** action, for example, may move to a target's location and fire at it. As one goes up the hierarchy, actions become increasingly abstract and powerful. They solve more difficult problems, such as path planning, and can react to wide range of eventualities.

A hierarchy of sensors parallels the action hierarchy. Just as a more complex action uses simpler ones to accomplish its goal, complex sensors use the values of simpler ones. These are *abstract sensors*. They are not physical, since they do not sense anything directly from the world. They take the output of other sensors and integrate and re-interpret it. A low-level vision system (a physical sensor) produces a black and white pixel array. An abstract sensor might take this image and

Move-to-Point: Move to a specified place on the map; the path planning uses only static terrain features and produces a list of waypoints.

→**Move-with-Waypoints:** Move-with-Waypoints iterates over the points passed in from Move-to-Point and attempts to move to each of them in turn.

→**Move-to-Point-Criteria:** Choose a nearby location to move to; perform local obstacle avoidance. The best location is determined using a local potential field; goals attract whereas terrain and other agents–especially enemies–repel.

→**Primitive-Move-to-Point:** Alter the agent's lower level effectors to actually take the step specified by Move-to-Point-Criteria.

→**Move-with-Waypoints-Monitor:** Terminate Move-with-Waypoints as soon as the agent gets close enough to its goal.

Figure 1: Some of the actions that constitute the **Move-to-Point** hierarchy. Sibling child actions are shown at the same indentation level (both **Move-to-Point-Criteria** and **Move-with-Waypoints-Monitor** are children of **Move-with-Waypoints**).

mark line segments in it. A higher level abstract sensor takes the line segments and determines whether there is a stretch of road ahead. A **follow-road** action can use this abstract sensor to compute where to go next.

HAC executes actions by scheduling them on a queue. The queue is sorted by the time at which the action will execute. Actions get taken off the queue and executed until there are no more actions that are scheduled to run at this time step. Actions can reschedule themselves, but in most cases, they will be rescheduled when woken up by messages from their children. An action is executed by calling its **realize** method. The realize method does not generally complete the action on its first invocation; it just does what needs to be done on this tick. In most cases, an action's **realize** method will be called many times before the action terminates.

HAC is a *supervenient* architecture (Spector & Hendler 1994). It abides by the principle that higher levels should provide goals and context for the lower levels, and lower levels provide sensory reports and messages to the higher levels ("goals down, knowledge up"). A higher level cannot overrule the sensory information provided by a lower level, nor can a lower level interfere with the control of a higher level. Supervenience structures the abstraction process; it allows us to build modular, reusable actions. HAC simplifies this process further by enforcing that every action's implementation (its **realize** method) take the following form:

1. React to messages coming in from children.
2. Update state.
3. Schedule new child actions if necessary.
4. Send messages up to parent.

The next section will give an example of an action hierarchy.

The Move-to-Point Action Hierarchy

In our "Capture the Flag" system there are two teams, Red and Blue, operating on a continuous map. The map terrain influences movement speed. Enemy agents and dynamical map features, such as mountain passes that can be blocked, add to the complexity. Our approach was to combine a static pathfinder, operating on a grid overlayed on the map, with some local obstacle avoidance mechanisms (Reese & Stout 1999). The action hierarchy allows us to cleanly separate the different levels of movement control, as seen in Figure 1. **Move-to-Point** maps out a path, leaving the local obstacle avoidance to **Move-to-Point-Criteria**. By altering the composition of the potential field controlling the agent, we can create very different movement mechanisms. For example, a field that repels enemies strongly creates an agent that flees. A field that is attracted to two targets simultaneously becomes a block (the agent will try to center itself between the two targets).

We found, however, that our static path planning algorithm was both computationally expensive and inflexible, particularly when it came to dynamic obstacles. As simulators become more realistic, the problems agents have to deal with begin to closely resemble those of physical robots. For this reason, HAC II will include coarse pathfinders based on the robot motion planning literature (Latombe 1991).

Note that unlike other agent architectures, HAC does not prescribe the number of action "levels". Although actions lower in the hierarchy will tend to be more reactive, whereas those higher up tend to be more deliberative, the transition between them is smooth and completely up to the designer.

Implementing Formations

Moving agents in formation is hard problem that occurs frequently in games. A formation involves two phases: moving agents to a central location so that they can form up and then moving the agents in formation. Formations need to dynamically adapt to the terrain and situation of the game. For example, a wedge formation may need to become a line to move through a pass. The agents in a formation need to move while taking each others actions into account. For example, it is important that the leader slow down if any of its followers get too far behind. Also, the nearer followers need to slow down so that they do not bump into the leader when the leader slows down.

In general, then, we need some way for the leader action to learn (directly or indirectly) about the positions of the followers and some way for the followers to learn about the position and speed of the leader. There are many ways to implement formations in HAC:

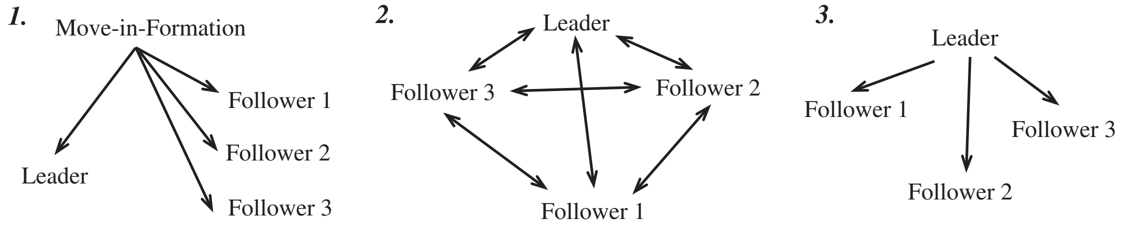

Figure 2: Control flow in three formation implementation approaches.

1 Have **move-in-formation** action that controls the leader and the followers.
2 Create leader and follower actions separately and let them communicate directly with each other so that can all respond appropriately.
3 Give the leader direct information about its followers so that it can control them directly and create movement actions for them.

Since the leader and the followers need to coordinate their actions, it appears that peer-to-peer strategies such as 2 and 3 would work best for controlling formations. However, these techniques both violate our guiding principle of supervenience and also make the code more complex and less flexible. Firstly, moving in formation must handle the deaths of follower or leader agents. The third technique requires that a dying leader pass the mantle on to one of its followers and that this follower take control of the other followers. The second technique also requires each of the cooperating actions to update their communication lists and to collectively choose a new leader. The first technique, on the other hand, provides the easy solution to this problem. We need only to add a simple method to handle the "all-resources-gone" message generated by a child action when its agent dies. This method does the following:

```
(when (follower has died)
  (re-organize formation shape)
(when (leader has died)
  ((select new leader)
   (re-organize formation shape))
(restart child actions)
```

Secondly, **move-in-formation** is the best place to dynamically alter the formation's shape and behavior. This parent action can alter the activity of its children based on abstract sensors. Thirdly, not utilizing the peer-to-peer models makes it easier to base activities on scenarios with incomplete information (when, for example, followers have inaccurate information about the leader's position). Finally, breaking the code into a move-in-formation parent, a leader-move child and zero or more follower-move children increases code modularity. No action is trying to do too much, which means that even though we have added actions, each part of the code is simpler to understand.

Abstract Sensor Uses

We originally introduced abstract sensors into HAC when we realized that sensor information need not follow the information flow defined by the action hierarchy. There may be cases when an agent can sense data that is useful to an action that is not one of its parents.

What did surprise us was the usefulness of abstract sensors. By integrating and interpreting data, they reduce the data glut that would otherwise occur at higher levels of the action hierarchy. They also simplify the actions themselves because they move the sensor retrieval and update code out of the action implementations. An action simply requests that it be notified when an abstract sensor reports a certain value. Many actions may be fed data by one abstract sensor. We have specialized abstract sensors into frequently used sub-classes:

- *Monitors* are abstract sensors that provide information specific to an action rather than about information related to the environment as a whole. Typically, monitors run periodically and send a message to their parent action when some condition becomes true. For example, the **move-with-waypoints** action discussed above uses a monitor to determine when it is close enough to its final location.
- *Action Termination Conditions* are abstract sensors that can be used in a simple logical language to specify the conditions under which an action should terminate or become active. For example, we might want **follow** to terminate when the followee gets too far away from the follower.
- *Latent Goals* are a particular type of goal data structure used by the Capture the Flag planner. They are *latent* because they need not always be considered in the planning process, but only when some condition holds. An abstract sensor notifies the planner when the condition is met. By this mechanism the planner can avoid have to plan for every eventuality, but instead react to unexpected opportunities or pitfalls when they actually arise.

Action Idioms

Over the course of HAC's development, we have written many actions. These actions have been improved and updated as the application domain changed. Actions share many common elements, and in the interest of streamlining the action design process, we have made *before*:

```
(realize for formation-move
   (if (first-time?)
     (start leader)
     (for each follower
        (start them up with a position in formation))
   (elseif ; not first time
     (for each child
       (if (completed? child)
         (push child completed-children))
       (if (failed? child)
         (push child failed-children))
       (if (or (unavailable? child) ; dead, stuck, etc.
               (member child failed-children))
          (pick an appropriate formation)))
     (if (completed? leader)
       (set leader-completed 'true))
     (if (failed? leader)
       (set leader-failed 'true))
     (if (or (unavailable? leader) ; dead, stuck, etc.
             leader-failed)
       (pick a new leader from explicitly passed
            list of resources)
       (promote the new leader)
       (pick an appropriate formation))
     (if (check-for-new-terrain?)
       (pick an appropriate formation))
     (if (calculate-enemy-threat?)
       (pick an appropriate formation)))
   (if all children have died
     (tell your parent))
   (reschedule this realize for next tick))
```

after:

```
(define formation-move which inherits
    from child-class)

(realize for formation-move ; assumes that leader
      ; and followers have already been decided on
   (start leader)
   (for each follower
      (start them up with a position in formation)))

(register abstract sensors for formation move
   (leader loss)
   (child loss))

(handler for leader loss for formation-move
   (select-resource of type leader) ; resources are
             ; automatically available from parent
   (promote the new leader)
   (pick an appropriate formation))

(handler for child loss for formation move
   (pick an appropriate formation))

(handler for terrain changes for formation move
   (pick an appropriate formation))

(handler for enemy threat for formation move
   (pick an appropriate formation))
```

Figure 3: **Formation-move** actions before and after the action idiom implementation.

more and more of these *action idioms* part of the HAC support mechanisms. Eventually, we would like writing actions in HAC to be like putting together a structure from building blocks.

Every action checks for messages from its children and generates messages to its parents. These functions are performed by the methods **handle-message** and **check-and-generate-message**, respectively, which are specialized on the type of event to be processed. Typically these methods run before the main body of the **realize** method, but **check-and-generate** can take an optional argument to run afterwards. **Handle-message** handlers are also used to process messages sent by abstract sensors.

We often found that many **realize** methods did a lot of one-time initializations on the their first invocation, so we added a construct to deal with this. We also found that certain types of child–parent configurations were a lot more common than others. We introduced language features to support these common idioms: Child actions can be specified to execute sequentially or in parallel; an action can complete when any child completes, they all complete, or when some other condition is met; actions can use no, one, or multiple resources.

The issue of resource management quickly became paramount in the design of intelligent actions. HAC contains a number of mechanisms for managing resources and assigning resources to actions. When children are invoked, they are passed a specified subset of the parent's resources. Mechanisms exist to steal resources from one action and assign them to another, to notify an action when a resource is no longer available, and to find resources of a certain type. Some resources can only be used by one agent at a time, some are consumed, and some emerge only in the process of performing an action. In HAC II we would like to model resources at the level of agent effectors, allowing us to assign parts of agents to different tasks.

Figure 3 shows many of these action idioms in practice using a pseudo-code version of the **formation-move** action as an example. Without the specialized handlers, we have one big **realize** method filled with if-statements. Assigning resources and dealing with their disappearance is also handled much more cleanly.

Towards an AI Engine—Lessons Learned

We view HAC as a first step in the development of a general-purpose AI engine for use in games. Although building agents and controlling agent behavior is one of the key issues in this endeavor, such an AI engine should also provide access to many of the other AI techniques that have been developed in recent years. This includes modules for planning, scheduling, search, simulation, vision, speech, and path planning in 2D and 3D worlds, as well as some simple pattern recognition and information retrieval tools.

Making these tools general enough to be easily and widely applicable will certainly be a challenge. Here are some of lessons *we* learned while moving HAC to its successor system, HAC II (in no particular order):

- *Enforce modularity.* Although hardly anyone will argue that modularity in the design of actions is desirable, abiding to it requires a great deal of discipline. As the formation example demonstrated, there are many ways to solve a problem. We found that modularity was helped by adhering to the principle of supervenience and by giving common action idioms to the designer. One question that arises is whether the same idioms are appropriate at all levels of the hierarchy, since lower levels deal with simpler actions and smaller time and space scales.
- *Don't force one approach on the game designer.* This principle manifests itself in many places. One example is that although HAC does enforce an action hierarchy, it places very few other constraints on the design of actions. Most any control scheme could be implemented in HAC. Another example is the need that emerged for many different path planners, for both continuous and discrete, and for both static and dynamic domains.
- *The action, not the agent, is central.* By having our hierarchy be one of *tasks that have to be accomplished*, it was very easy to incorporate multi-agent actions and planning into our architecture.
- *Resources are first class objects.* Initially overlooking the importance of resource management was probably one of the main lessons we learned. Since our hierarchy is organized around *actions*, resources become the agents that perform the actions. There are now many mechanisms in HAC to pass resources to children, to select certain kinds of resources, and to react to resources becoming unavailable. In fact, even sensor data can be viewed as a resource: abstract sensors manipulate data resources, whereas plans and actions manipulate agents.
- *Control and Sensing are conceptually separate.* We found that separating sensing from control conceptually, while still using the same uniform language to implement the sensors, simplified our code enormously. Abstract sensors became a very general tool for organizing data and for solving any problem that involved having to react to some event in the world. In fact, we are discussing folding our current message passing model into one that is completely event-based. Events would be used to pass control information and sensor data around the hierarchies from parent to child, child to parent, peer to peer, and between actions and sensors.

One of the more exciting directions we are moving in with HAC is generalizing it so it can be used to control physical robots. It is interesting that modern real-time games and robotics place similar demands on an architecture. The current engine uses a centralized queue and imposes no constraints on the CPU time used by an action. The new engine will be operating in a real-time, decentralized environment, and will need to to deal with widely varying time scales, from microseconds to days. The currently used, centralized action queue will be replaced by a more general mechanism that simply forwards events to the appropriate action, whether it is a local action running on the same piece of hardware or an action running remotely.

HAC is a framework for writing actions. It is appropriate for any application where you want to define intelligent behavior for an agent. Most game genres require such behavior, be it for a monster in a first-person shooter, the computer opponent in a strategy game, or fellow team mates in a sports game. HAC is very lean; we estimate that in our "Capture the Flag" application, HAC itself (which does not include the actions the game designer has written, only the architecture overhead), uses less than 1% of the CPU time and memory.

HAC's interface to a domain consists of a set of low-level sensors and effectors for every agent. In the case of a game, these would be specified by the game engine. From these basic elements, the action writer can build up arbitrarily complex behavior. Currently writing actions requires programming skills, but one of the goals of HAC II is to explore how simple we can make this process.

Acknowledgments

This research is supported by DARPA/USAF under contract numbers N66001-96-C-8504, F30602-97-1-0289, and F30602-95-1-0021. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of the Defense Advanced Research Projects Agency/Air Force Materiel Command or the U.S. Government.

References

Atkin, M. S.; Westbrook, D. L.; Cohen, P. R.; and Jorstad, G. D. 1998. AFS and HAC: Domain-general agent simulation and control. In *Working Notes of the Workshop on Software Tools for Developing Agents, AAAI-98*, 89–95.

Atkin, M. S.; Westbrook, D. L.; and Cohen, P. R. 1999. Capture the flag: Military simulation meets computer games. In *Working Notes of the AAAI Spring Symposium on AI and Computer Games*, 1–5.

Latombe, J.-C. 1991. *Robot Motion Planning*. Dordrecht, The Netherlands: Kluwer.

Reese, B., and Stout, B. 1999. Finding a pathfinder. In *Working Notes of the AAAI Spring Symposium on AI and Computer Games*, 69–71.

Spector, L., and Hendler, J. 1994. The use of supervenience in dynamic-world planning. In Hammond, K., ed., *Proceedings of The Second International Conference on Artificial Intelligence Planning Systems*, 158–163.