Capture the Flag: Military Simulation Meets Computer Games

Marc S. Atkin, David L. Westbrook and Paul R. Cohen
Experimental Knowledge Systems Laboratory
Department of Computer Science, LGRC, Box 34610
University of Massachusetts, Amherst, MA 01003
{atkin,westy,cohen}@cs.umass.edu

Abstract

Some of the more complex AI testbeds are not that different from computer games. It seems that both sides, AI and game design, could profit from each other's technology. We go a first step in this direction by presenting a very general agent control architecture (HAC: Hierarchical Agent Control), a toolkit for designing an action hierarchy. It supports action abstraction, a multi-level computational architecture, sensor integration, and planning. It is particularly well suited to controlling large numbers of agents in dynamic environments. We conclude the paper with an example of how HAC was applied to the game we use as our development domain: Capture the Flag.

Introduction

In the Experimental Knowledge Systems Laboratory, we have developed a number of complex simulations. PHOENIX, a system that uses multiple agents to fight fires in a realistic simulated environment, is perhaps the best example (Cohen *et al.* 1989). We have also made efforts to write general simulation substrates (Anderson 1995; Atkin *et al.* 1998). Currently, we are working on creating a system, called COASTER, which will allow army commanders to design and evaluate high level plans ("courses of action") in a land-based campaign. Our domain is a variant of the game "Capture the Flag."

It occurred to us that these simulators were just variations on a theme. Physical processes, military engagements, and a lot of computer games are all about agents moving and applying force to one another (see, for example, (Tzu 1988) and (Karr 1981)). We therefore set ourselves the goal of constructing a domain-general agent development toolkit and simulation substrate. Regardless of the domain, agent designers must face the same kinds of problems: processing sensor information, reacting to a changing environment in a timely manner, integrating reactive and cognitive processes to achieve an abstract goal, interleaving planning and execution, distributed control, allowing code reuse within and across domains, and using computational resources efficiently. Since many game developers are also agent designers, we think they could benefit from this toolkit, too.

This paper will describe a general framework for controlling agents, called *Hierarchical Agent Control* (HAC). Complementing it is a general simulator of physical processes, the *Abstract Force Simulator* (AFS). HAC itself is a general skeleton for controlling agents. It can work with many kinds of simulators or even real-life robots, as long as they adhere to a certain protocol. HAC is written in Common Lisp.

Hierarchical Agent Control

The problem HAC solves is providing a relative easy way to define the behavior of an agent. It can be viewed as 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.

One of the major issues in defining an action set for an agent, and, arguably, one of the major issues in defining *any* kind of intelligent behavior, is the problem of forming abstractions. No agent designer will want to specify the solution to a given problem in terms of primitive low-level actions and sensations. Instead, she will first build more powerful *abstract* actions, which encode solutions to a range of problems, and use these actions when faced with a new problem. If a robot is supposed to retrieve an object, we don't want to give it individual commands to move its wheels and its gripper; we want to give it a "pick-up" command and have the robot figure out what it needs to do.

HAC supports abstraction by providing the mechanisms to construct a *hierarchy* of actions. In the hierarchy, abstract actions are defined in terms of simpler ones, ultimately grounding out in the agent's effectors. Although actions are abstract at higher levels of the hierarchy, they are nonetheless executable. At the same time, the hierarchy implements a multilevel computational architecture, allowing us, for example, to have both cognitive and reactive actions within the same framework (Georgeff & Lansky 1987; Cohen *et al.* 1989).

The main part of HAC's execution module is an action queue. Any scheduled action gets placed on the 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. We will see an example of this later.

The Action Hierarchy

HAC is a *supervenient* architecture (Spector & Hendler 1994). This means that 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 goes a step further in the simplification of the action-writing process, by enforcing that every action's implementation 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.

Let's assume we wanted to build an action that allows an agent to follow a moving target (see Figure 1). If we have a **move-to-point** action (which in turn uses the primitive **move**), writing such an action is fairly easy. We compute a direction that will cause us to intercept the target. Then we compute a point a short distance along this vector, and schedule a child **move-to-point** action to move us there. We leave all the details of getting to this location, including such things as obstacle avoidance, up to the child. The child can send any kind of message up to its parent, including such things as status reports and errors. At the very least it will send a completion message (failure or success). When the child completes, we compute a new direction vector and repeat the process, until we are successful or give up, in which case we send a message to *our* parent.

Note that the implementation of an action is left completely up to the user; she could decide to plan out all the movement steps in advance and simply schedule the next one when the **move-to-point** child completes. Or she could write a reactive implementation, as described above. Note also that every parent declares the set of messages it is interested in receiving, essentially providing *context* for the child. In some cases, a parent might

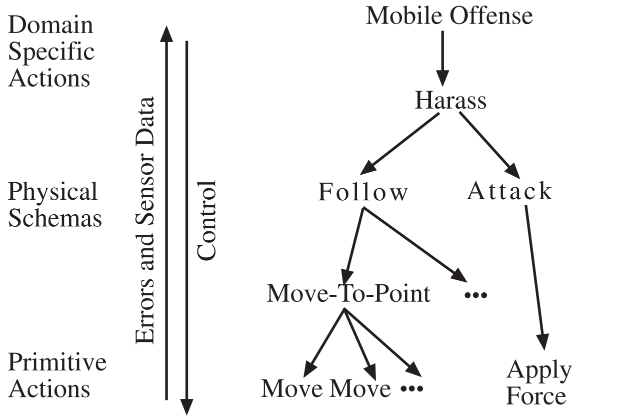

Figure 1: Actions form a hierarchy; control information is passed down, messages and sensor integration occurs bottom-up.

only be interested in whether or not the child terminates. The parent can go to sleep while the child is executing. In other cases, the parent may request periodic status reports from the child, and run concurrently with the child in order to take corrective measures.

The very lowest level of the hierarchy consists of very primitive actions, like **move** and **apply-force**. These actions are little more than low-level robotic effectors; they set the agent's acceleration or attempt to do damage to a neighboring agent. Using these primitives, we build a layer of slightly more complex, yet still domain-general actions (called *physical schemas* (Atkin *et al.* 1998)), such as **move-to-point**, **attack**, and **block**. Above this layer we have domain-specific actions, if needed. It is interesting to note that as you go up the hierarchy, the actions tend to deal with larger time and space scales, and have more of a deliberative than a reactive character. But the transition to these cognitive actions is a smooth one; no extra mechanism is needed to implement them.

An Example Action Definition

In the last section, we described in general terms how actions are defined within HAC. This section will elucidate the process using a concrete example and actual code. HAC provides a number of methods to make the process of writing actions easier. Across actions we must perform the same sort of tasks: generating messages for the parent, advancing the action, etc. In HAC, actions are classes; each action defines a set of methods that address these tasks.

Figure 2 shows the implementation of a multi-agent action, **swarm**, which illustrates many of HAC's features. It is a simple action that causes a number of agents to move around randomly within a circular region. We use the simpler action **move-to-point** to implement this; it is invoked with the construct **start-new-child**. When the agents bump or get stuck, they

```
(defclass* swarm (level-n-action)
  (area                          ;swarm area
   (blobs nil)                   ;blobs (abstract agents) involved in swarm
   ;; storage
   (first-call t)))

(defmethod handle-message ((game-state game-state) (action swarm)
                           (message completion))
  (redirect game-state action (blob (from message))))

(defmethod handle-message ((game-state game-state) (action swarm)
                           (message afs-movement-message))
  (interrupt-action game-state (from message))
  (redirect game-state action (blob (from message))))

(defmethod redirect ((game-state game-state) (action swarm) blob)
  (start-new-child action game-state 'move-to-point
        :blob blob
        :destination-geom
           (make-destination-geom (random-location-in-geom (area action)))
        :messages-to-generate '(completion contact no-progress-in-movement)
        :speed nil
        :terminal-velocity nil))

(defmethod check-and-generate-message ((game-state game-state) (action swarm)
                                       (type (eql 'completion)))
  (values nil))    ;never completes

(defmethod realize ((game-state game-state) (action swarm))
  (when (first-call action)
    (setf (first-call action) nil)
    (loop for blob in (blobs action) do
          (redirect game-state action blob))))
```

Figure 2: Implementation of a multi-agent "swarm" behavior in HAC.

change direction. First, we define the **swarm** action to be a level-n-action. This means it is non-primitive and must handle messages from below as well as pass messages up. We define how we will react to messages from children using the **handle-message** methods. Message handlers specialize on the type of message that a child might send. In the example, we redirect an agent to a new location when the **move-to-point** action controlling it completes. If the **move-to-point** reports any kind of error (all errors relating to movement are subclasses of **afs-movement-message**), such as contact with another agent, we simply interrupt it and redirect the agent somewhere else.

These **handle-messages** methods are invoked whenever a message of the specified type is sent to **swarm**. When this happens, the realize method is also called. In our example, the realize method is only used for initialization: the first time it is called, it sends all the agents off to random locations.

The set of **check-and-generate** methods define the set of messages that this action can send up to its parents. When the realize message is called, the **check-and-generate** methods are invoked. We can specify if they should be called before or after the realize method. The swarm example never completes, and it doesn't report on its status, so it generates no messages.

Note how simple it was to write a multi-agent action using the HAC methods. HAC is action-based, not agent-based. Writing an action for multiple agents is no different from writing an action for a single agent that has to do several things at the same time (like turning and moving). We envision the different methods for implementing parts of actions as the beginnings of an *action construction language*, and we hope to move HAC in this direction. There would be constructs for combining actions, either sequentially or concurrently. There would be constructs specifying how resources should be used, whether or not something can be used by two agents at the same time, and so on.

Resources

Resources are a very important part of agent control. There are many types of resources: the effectors of each individual agent, the objects in the world that agents use to fulfill their tasks, and the agents themselves. Some resources can only be used by one agent at a time, some resources are scarce, and some resources emerge only in the process of performing some action.

HAC provides mechanisms for managing resources and assigning resources to actions. HAC currently does not contain any general resource arbitration code, but instead assumes that a parent will arbitrate when its children are all vying for the same resources. Actions can return unused resources to their parent, who can then reassign them. Actions can also request more resources if they need them.

The Sensor Hierarchy

HAC not only supports a hierarchy of actions, but also a hierarchy of sensors. Just as a more complex action uses simpler ones to accomplish its goal, complex sensors use the values of simpler ones to compute their values. These are *abstract sensors.* They are not physical, since they don't 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 mark line segments in it. A higher level abstract sensor takes the line segments and determines whether or not there is a stretch of road ahead. A **follow-road** action can use this abstract sensor to compute where to go next.

We use HAC's scheduling and message passing mechanism to organize sensors into a hierarchy, except that it is now sensor-update functions that are being scheduled, not actions, and sensor values that are being passed, not status reports and completion messages. Actions can query sensors or ask to receive a message when a sensor achieves a certain value. These messages allow an action to be interrupted when a certain event occurs, enabling the action to take advantage of unexpected opportunities.

The Planner

As one of its modules, HAC includes a planner. Goals and plans fit seamlessly into HAC's action hierarchy: Scheduling a child action can be viewed as posting a goal, and executing the action that satisfies this goal. Planning is necessary when the goal is satisfied by several actions and we have to decide between them. Simple goals, like moving to a location, are constrained enough that we can write out one good solution. All the actions we have seen so far were simple enough to require only one solution. But particularly as you get higher in the hierarchy, there will be more ambiguity with respect to how a goal should be achieved. Accordingly, goals might have multiple potential solutions.

In HAC, we use the term "plan" to denote an action that satisfies a goal. We introduce a special child action, **match-goal**, that is scheduled whenever an action posts a goal. **Match-goal** will check which plans can satisfy the goal, evaluate them, and choose the best one to execute. If no plan matches, **match-goal** reports failure back to the poster of the goal. Plans themselves may also contain sub-goals, which are satisfied using the same process.

Plans are evaluated by a process of *forward simulation*: Instead of using some heuristic to pick the best plan, we actually use an even more abstract version of AFS to simulate *what would happen* if this plan were to execute. This process is made efficient by estimating the time at which something interesting will happen in the simulation, and advancing the simulation directly to these *critical points* (Atkin & Cohen 1998).

This type of planning, which uses stored solutions

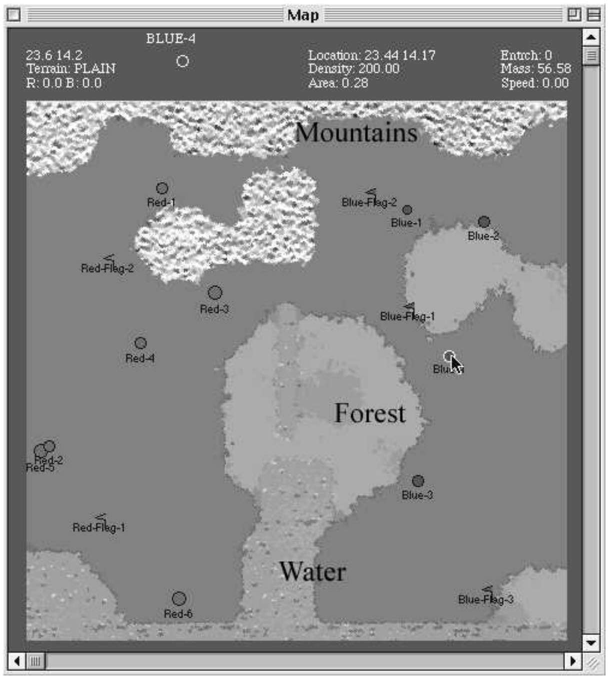

Figure 3: The Capture the Flag domain.

(plans) that are not fully elaborated, is known as *partial hierarchical planning* (Georgeff & Lansky 1986). Due to the flexibility within the plans, it is particularly well suited to dealing with dynamic environments. In addition, the fact that we have pre-compiled solutions for certain types of problems cuts down enormously on the amount of search we would otherwise have to do. Finding ways to keep the branching factor low is a primary concern in continuous domains such as ours; this is why we don't use a generative planner. In many domains, there are only a small number of reasonable ways to react to any given situation. We typically only generate a small number of high level plans, but spend a lot of time evaluating them. We also interleave planning with execution; **match-goal** is an action like any other, and must share its computational resources. Even if a particular plan takes a long time to evaluate, lower level reflexive processes will still operate. The higher the plan is in the hierarchy, the longer is will typically take to generate. This is balanced by the fact that higher cognitive levels deal with larger time and space scales, and are usually less sensitive to a delay in their starting time.

An Example Domain: Capture the Flag

We have been developing a dynamic and adversarial domain, "Capture the Flag," in which to test our underlying simulator, HAC, and the planner. There are two teams, Red and Blue. Some of the units on each team are of a special type, "flag"; they cannot move. The objective for both teams is to capture all the opponent's flags. Figure 3 shows one of the randomized starting positions for this domain. Notice that we use different types of terrain. Some of the terrain, such as mountains and water, cannot be traversed by units, which gives us the opportunity to reason about such concepts as "blocked" and "constriction."

We have constructed an action hierarchy based on the primitives **move** and **apply-force** that allows us to move to a location, attack a target, and defend a unit. We can also block passes or intercept hostile units. The top-level actions are more domain-specific and concern themselves primarily with the allocation of resources. They generate a list of tasks, such as "attack a specific enemy" or "defend a flag," and then construct a list of actions that achieve these tasks. Each task might be achieved by more than one action (a flag could be defended by placing units around it or by intercepting the units that are threatening it, for example). We prefer actions that achieve multiple tasks. Different combinations of actions give rise to different high-level solutions to the ultimate goal of capturing the opponent's flags. For example: If attack tasks are weighted more strongly than defense, the resulting strategy is more aggressive.

Once a plan has been generated, it is evaluated by forward simulation. Forward simulation also takes into account how the opponent might respond. The best plan, determined by a static evaluation function of the resulting map and placement of units, is chosen and executed.

Summary and Discussion

Both computer games and real-world AI applications can profit from one another. We have introduced HAC as toolset for controlling agents in any domain. Part of HAC is a planner that is able to cope with the complexities of a continuous, dynamic, real-time domain. The planner's distinguishing feature is that it evaluates plans by efficiently simulating ahead in a more abstract space.

These are the issues HAC addresses:

- Reactive and cognitive processes are integrated seamlessly.
- Agent control, action execution, planning, and sensing are all part of the same framework
- HAC is a modular system; it can be used to control agents in simulation or real robots. Supervenience enables us build re-usable action modules.
- Efficient multi-agent control and simulation.
- Planning in continuous domains.

We believe that architectures like HAC, which were designed by AI researchers, yet are applied to problems that have many of the same characteristics as computer games, could be the basis for moving AI technology into computer game design. This would help address a common complaint people have about strategy games, namely that the computer player isn't very capable and makes stupid mistakes that can be easily exploited. HAC also makes the process of defining agent behaviors when writing a new game easier. In turn, we believe that technologies emphasized in computer game design, including real-time graphics, transparent multiplayer support, customizability of the game world and interface, could be put to good use in improving AI systems' performance and appeal.

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

Anderson, S. D. 1995. *A Simulation Substrate for Real-Time Planning.* Ph.D. Dissertation, University of Massachusetts at Amherst. Also available as Computer Science Department Technical Report 95–80.

Atkin, M. S., and Cohen, P. R. 1998. Physical planning and dynamics. In *Working Notes of the AAAI Fall Symposium on Distributed Continual Planning*, 4–9.

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.

Cohen, P. R.; Greenberg, M. L.; Hart, D. M.; and Howe, A. E. 1989. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine* 10(3):32–48.

Georgeff, M. P., and Lansky, A. L. 1986. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation* 74(10):1383–1398.

Georgeff, M. P., and Lansky, A. L. 1987. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 677–682. MIT Press.

Karr, A. F. 1981. Lanchester attrition processes and theater-level combat models. Technical report, Institute for Defense Analyses, Program Analysis Division, Arlington, VA.

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.

Tzu, S. 1988. *The Art of War.* Shambhala Publications.