AFS and HAC: Domain-General Agent Simulation and Control

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

Abstract

We present two systems for simulating and designing agents, respectively. The first, the Abstract Force Simulator (AFS), is a domain-general simulator of agents applying forces; many domains can be characterized in this way. The second, Hierarchical Agent Control (HAC), is a general 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. Together, AFS and HAC provide a very general framework for designing and testing agents.

Introduction

We have set ourselves the goal of constructing a domain-general agent development toolkit. 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.

The best possible solution to any of the above problems will depend to some extent on the domain. As a consequence, we wanted our toolkit to have enough flexibility to allow any one of many solutions to be implemented. We provide the agent designer with different tools for dealing with a problem, but it is up to her to decide which tool to use.

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). AFS can be used to simulate many different domains, as will be described in the next section. 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 provides the mechanics for sensor management, action scheduling, and message passing. The actual implementation of actions and sensors is provided by separate and interchangeable modules, as is the planner. Additional modules might be task specific, such as the language generation module we have used. Figure 1 illustrates the system decomposition. AFS and HAC are written in Common Lisp.

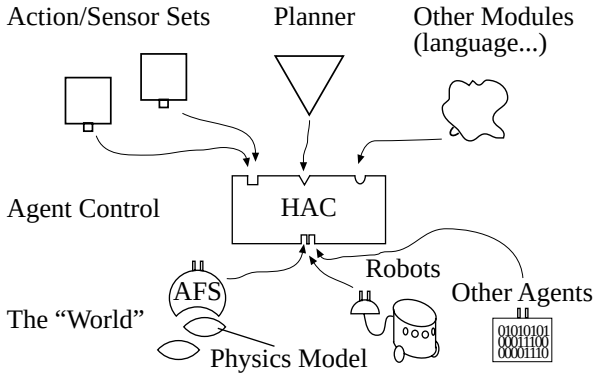

Figure 1: The AFS/HAC system modules and organization.

We will now describe AFS, which in many ways motivated the need for a general agent control architecture, before returning to HAC.

The Abstract Force Simulator

Physical Schemas

It occurred to us some time ago that many of the simulators we had been writing really were just variations on a theme. Physical processes, military engagements, and games such as billiards are all about agents moving and applying force to one another (see, for example, (Tzu 1988) and (Karr 1981)). Even the somewhat abstract realm of diplomacy can viewed in these terms: One government might try to ***apply pressure*** to another for some purpose, or intend to ***contain*** a crisis before it spreads.

Furthermore, it became clear that there is a common set of terms upon which all the above processes operate. Some examples are **move**, **push**, **reduce**, **contain**, **block**, or **surround**. Collectively, we refer to these terms as ***physical schemas.*** If moving an army is conceptually no different than moving a robot, both

these processes can be represented with one **move** action in a simulator. We believe that people think and solve problems in terms of physical schemas. An example: A person notices his sink is leaking, and considers what to about it. He realizes that there are cracks in the sink. The way to solve the problem is to plug the cracks. Now confront this person with a military problem: Hostile forces are moving across a mountain range into friendly territory. What should be done about it? If the person understands that military forces can behave like water, and that the cracks are passes in the mountain range, he can prevent the flow by making the passes untraversable.

Based on these ideas, we have developed a simulator of physical schemas, the Abstract Force Simulator (AFS). It operates with a set of abstract agents, circular objects[1] called "blobs," which are described by a small set of physical features, including mass, velocity, friction, radius, attack strength, and so on. A blob is an abstract unit; it could be an army, a soldier, a planet, or a political entity. Every blob has a small set of primitive actions it can perform, primarily **move** and **apply-force**. All other schemas are built from these actions. Simply by changing the physics of the simulator (how mass is affected by collisions, what the friction is for a blob moving over a certain type of surface, etc.), we can turn AFS from a simulator of billiard balls into one of unit movements in a military domain.

Simulator Mechanics

AFS is a simulator of physical processes. It is tick-based, but the ticks are small enough to accurately model the physical interactions between blobs. Although blobs themselves move continuously in 2D space, for reasons of efficiency, the properties of this space, such as terrain attributes, are represented as a discrete grid of rectangular cells. Such a grid of cells is also used internally to bin spatially proximal blobs, making the time complexity of collision detection and blob sensor modeling no greater than linear in terms of the number of blobs in the simulator. AFS was designed from the outset to be able to simulate large numbers (on the order of hundreds or thousands) of blobs.

The physics of the simulation are presently defined by the following parameters:

- Blob-specific attributes:
  - maximum acceleration and deceleration
  - friction of the blob on different surfaces
  - viscosity and elasticity: do blobs pass through one another or bounce off?
- Global parameters:
  - the effect of terrain on blobs
  - the different types of blobs present in the simulation (such as blobs that need sustenance).
  - the damage model: how blobs affect each others' masses by moving through each other or applying force.
  - sustenance model: do blobs have to resupplied in order to prevent them from losing mass?

AFS is an ***abstract*** simulator; blobs are abstract entities that may or may not have internal structure. AFS allows us to express a blob's internal structure by composing it from smaller blobs, much like an army is composed of smaller organizational units and ultimately individual soldiers. But we don't have to take the internal structure into account when simulating, since at any level of abstraction, every blob is completely characterized by the physical attributes associated with it. Armies can move and apply force just like individual soldiers do. The physics of armies is different than the physics of soldiers, and the time and space scales are different, but the main idea behind AFS is that we can simulate at the "army" level if we so desire—if we believe it is unnecessary or inefficient to simulate in more detail.

Since AFS is basically just simulating physics, the top-level control loop of the simulator is quite straightforward: On each tick, loop over all blobs in the simulator and update each one based on the forces acting on it. If blobs interact, the physics of the world will specify what form their interaction will take. Then update the blob's low-level sensors, if it has any. Each blob is assumed to have a ***state reflector***, a data structure that expresses the current state of the blob's sensory experience. It is the simulator's job to update this data structure.

Hierarchical Agent Control

Since AFS is such a general simulator, we require an appropriately general method for controlling the blobs within it. This control scheme is HAC (***Hierarchical Agent Control***). Whereas the physics modeled in AFS define ***how*** an a blob's actions unfold in the world, HAC defines ***what*** the blob should do. One way to look at HAC is as a toolset for designing a hierarchy of actions, goals, and sensors. Another way is as an execution module for the many actions that are running concurrently within a blob and across blobs.

Although we have primarily been using HAC in conjunction with AFS, HAC is by no means tied to a specific simulator, and, unlike AFS, could be applied to domains that are not easily viewed as abstract physical processes. We are currently working on using HAC to control actual mobile robots (Pioneers). All HAC requires to work with ***any*** given agent is an interface to the state reflector for that agent (i.e., a way to access the low-level sensory data the agent collects) and an implementation of a set of low level action primitives, like **turn**, **move**, and **push**.

[1] Eventually, blobs will be able to take any shape, and deform and redistribute their mass. Computing the physical interactions of arbitrarily-shaped objects is time-consuming; as a first pass, we intend to make blobs elliptical.

One of the major issues in defining an action set for an agent, and, one might argue, 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 lets us abstract 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 multi-level 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 a blob to follow a moving target (see Figure 2). 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.

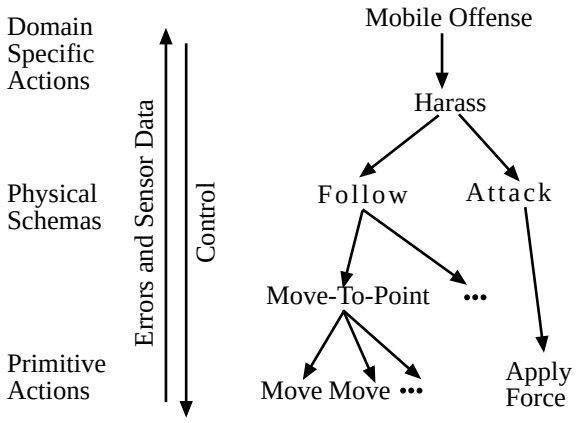

Figure 2: Actions form a hierarchy; control information is passed down, messages and sensor integration occurs bottom-up.

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 totally reactive implementation, as described above. Note also that every parent declares the set of messages it is interested in receiving. In some cases, a parent might 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, such as interrupting the child.

The very lowest level of the hierarchy consists of very primitive actions, things like the aforementioned **move** and **apply-force**. These actions are little more than low-level robotic effectors; they set the blob's acceleration or attempt to do damage to a neighboring blob. Using these primitives, we build the layer of physical schemas, which consists of actions such as **move-to-point**, **attack**, and **block**. Above this layer we have domain-specific actions, if needed. It is interesting to

```
(defclass* swarm (level-n-action)
  (area                          ;swarm area
   (blobs nil)                   ;blobs 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 3: Implementation of a multi-agent "swarm" behavior in HAC.

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 3 shows the implementation of a multi-agent action, **swarm**. It is a simple action that causes a number of blobs 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 blobs bump or get stuck, they 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-messages** methods. Message handlers specialize on the type of message that a child might send. In the example, we redirect a blob 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 blob, we simply interrupt it and redirect the blob 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 blobs 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 blob-based. Writing an action for multiple blobs is no different from writing an action for a single blob 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. The sensor hierarchy is not yet fully implemented, so we will provide only a short description here.

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.

Since the lowest level sensors are associated with blobs moving around on the map, and higher level actions use abstract sensors that in turn use the values produced by these low level sensors, the structure of the sensor hierarchy will often mirror that of the action hierarchy. The two hierarchies cannot be merged, however. Suppose we have an abstract sensor that uses data from within a spatial region. This sensor uses all the data produced by blobs operating within this region, regardless of whether the blobs are all performing the same ***action***. Thus, the two hierarchies might not correspond exactly.

We are implementing a simple blackboard architecture that allows us to index sensor information based on location and type. We will still use the 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 will use the blackboard to query sensors. An action can also ask to receive a message when a sensor achieves a certain value. Currently, actions do this checking themselves. 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. How do goals and plans fit into HAC's action hierarchy? Quite simply: Scheduling a child action can be viewed as posting a goal, and executing the child 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 gets 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.***

This type of planning, which uses stored solutions (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.

Partial hierarchical planning meshes very well with the idea that people understand and reason about the world in terms of physical schemas. Viewed at the level of physical schemas, there really are only a few different ways to solve a problem. For example, if A and B are point masses, A can cause B to move by i) pushing it, ii) asking it to move (if it is an intentional agent), or iii) initiating movement in B. These separate solutions can be written down as plans that satisfy the goal "make B move". The exciting thing about planning at the physical schema level is that the plans you use are not limited to just one domain. If you can figure out what "move" and "push" ***mean*** in a domain, you can use your old plans.

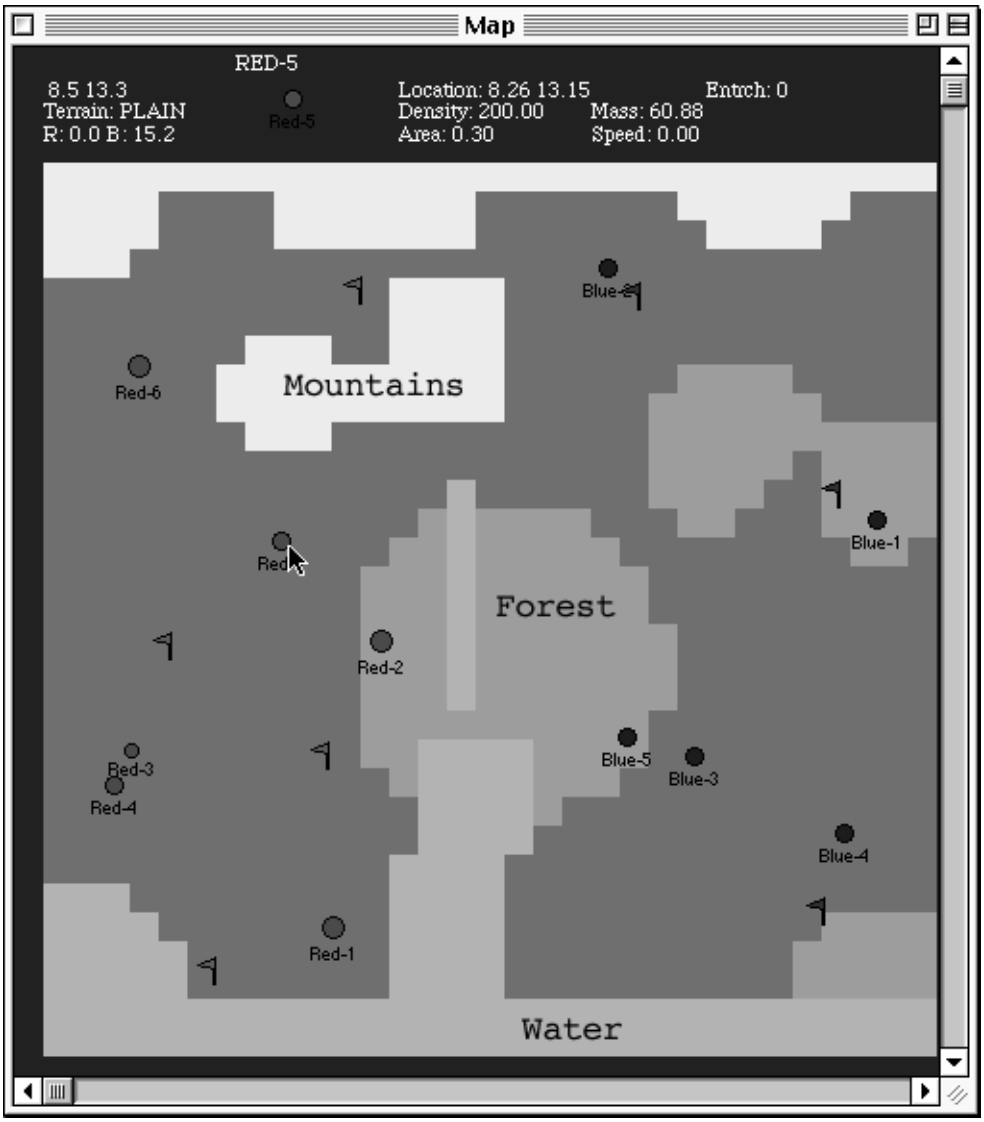

Figure 4: The Capture the Flag domain.

Since **match-goal** is a child action, just like any other, interleaving planning and execution is a straightforward process. Let's assume **match-goal** takes a certain amount of time, t, to finish evaluating all the plans that match. All this means is that it will take t time units before a plan starts executing. All other actions that the parent scheduled during this time interval will continue to execute normally. And since every action must be written to respond to failures or unexpected events in its children, so must a goal poster respond to the failure of one the matched plans. This is how replanning is initiated.

An Example Domain: Capture the Flag

We have been developing a dynamic and adversarial domain in which to test AFS, HAC, and the planner. This domain is based on the game of "Capture the Flag." There are two teams, red and blue. Some of the blobs 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 4 shows one of the randomly generated 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 blobs, which gives us the opportunity to reason about such physical schemas 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 blob. We can also block passes or intercept hostile blobs. 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 an specific enemy" or "defend a flag," and then construct a list of physical schemas that achieve these tasks. Each task might be achieved by more than one physical schema (a flag could be defended by placing blobs around it or by intercepting the blobs that are threatening it, for example). Depending on how tasks are weighted and what physical schemas are chosen to achieve these tasks, one arrives at different high-level solutions to the ultimate goal of capturing the opponent's flags. If attack tasks are weighted more strongly than defense, one might end up with a more aggressive strategy.

The different strategies correspond to different plans for achieving the goal "win-capture-the-flag." They are evaluated within match-goal 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 blobs, is chosen and executed.

Our goal for this domain is to show how one can reason with physical schemas, and how this reasoning fits in with our general HAC architecture. Beating the computer is already non-trivial, and as our plans become more refined, it will only get harder. Furthermore, the actions we use and a lot of the reasoning we do during planning is not specific to the Capture the Flag domain. In fact, since we already had implementations of **move-to-point** and **attack** from other domains, it took us only two weeks to implement this scenario and a first version of the plans that solve it. This speaks for the power and flexibility of HAC.

Summary and Discussion

This paper has introduced AFS as a general simulator for any domain that can be described as agents moving and applying force to one another, and HAC as toolset for controlling such agents. We set out not to build a domain-specific agent design tool, but to only provide a general framework that helps the designer, and leaves the actual implementation up to her. These are the issues we believe HAC addresses well:

- 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.
- Easy transference of knowledge across domains through the use of physical schemas.

These are the issues we believe still need work:

- Communication *between* agents in the hierarchy, without going through a parent.
- An automatic domain to physics mapper.
- A fully implemented agent sensor model.
- A specification of an *action construction language*.

Acknowledgements

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

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. *IEEE Special Issue on Knowledge Representation* 74(10):1383–1398.

Georgeff, M. P., and Lansky, A. L. 1987. Reactive reasoning and planning. In *AAAI 87*, 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.