Hierarchical Agent Control: A Framework For Defining Agent Behavior

Marc S. Atkin
University of Massachusetts
140 Governor's Lane
Amherst, MA 01003
atkin@cs.umass.edu

Gary W. King
University of Massachusetts
140 Governor's Lane
Amherst, MA 01003
gwking@cs.umass.edu

David L. Westbrook
University of Massachusetts
140 Governor's Lane
Amherst, MA 01003
westy@cs.umass.edu

ABSTRACT

The Hierarchical Agent Control Architecture (HAC) is a general toolkit for specifying an agent's behavior. HAC supports action abstraction, resource management, sensor integration, and is well suited to controlling large numbers of agents in dynamic environments. It relies on three hierarchies: action, sensor, and context. The action hierarchy controls the agent's behavior. It is organized around tasks to be accomplished, not the agents themselves. This facilitates the integration of multi-agent actions and planning into the architecture. The sensor hierarchy provides a principled means for structuring the complexity of reading and transforming sensor information. Each level of the hierarchy integrates the data coming in from the environment into conceptual chunks appropriate for use by actions at this level. Actions and sensors are written using the same formalism. The context hierarchy is a hierarchy of goals. In addition to their primary goals, most actions are operating within a set of implicit assumptions. These assumptions are made explicit through the context hierarchy. We have developed a planner, GRASP, implemented within HAC, which is capable of resolving multiple goals in real time.

HAC was intended to have wide applicability. It has been used to control agents in commercial computer games and physical robots. Our primary application domain is a simulator of land-based military engagements called "Capture the Flag." HAC's simulation substrate models physics at an abstract level. HAC supports any domain in which behaviors can be reduced to a small set of primitive effectors such as MOVE and APPLY-FORCE. At this time defining agent behavior requires Lisp programming skills; we are moving towards more graphical programming languages.

1. INTRODUCTION

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 abstract goals, interleaving planning and execution, distributed control, allowing code reuse within and across domains, and using computational resources efficiently. We have been developing an agent architecture that lets us address these problems, called *Hierarchical Agent Control* (HAC).

HAC can be viewed as a set of language constructs and support mechanisms for describing agent behavior. HAC takes care of the mechanics of executing the code that controls an agent, passing messages between actions, coordinat-

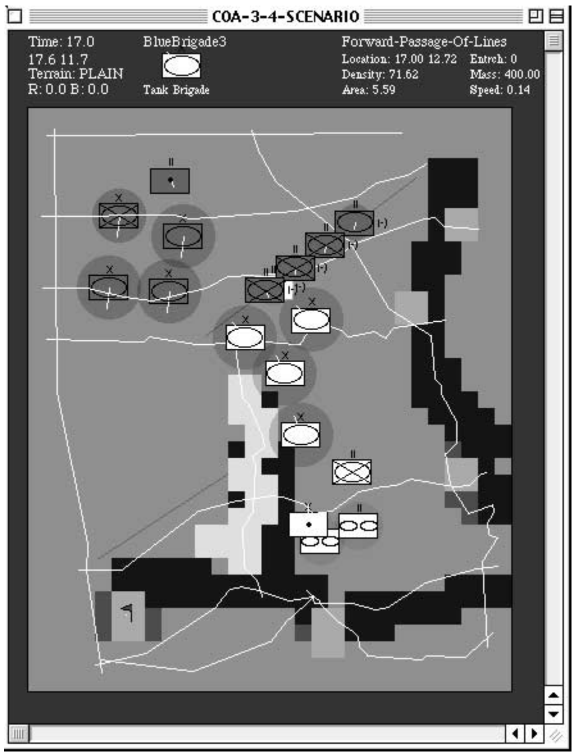

Figure 1: The Capture the Flag domain (CtF). There are two teams; each has a number of movable units and flags to protect. They operate on a map which has different types of terrain. Terrain influences movement speed and forms barriers; terrain also affects unit visibility. A team wins when it captures all its opponent's flags.

AGENTS'01, May 28-June 1, 2001, Montréal, Quebec, Canada.
Copyright 2001 ACM 1-58113-326-X/01/0005 ...$5.00.

ing multiple agents, arbitrating resource conflicts between agents, and updating sensor values. Although our primary application has been a military simulation system called "Capture the Flag" [2] (see Figure 1 and Appendix A.1), HAC has also been applied to such domains as commercial games, multi-agent simulations, and actual physical robots. Many of the domains HAC has been used with were modeled using a simulation substrate we have created, the *Abstract Force Simulator* (AFS).

In this paper, we will describe the major features of HAC and AFS, using the Capture the Flag (CtF) domain for illustration. Having done this, we also discuss the lessons we learned developing HAC, and future plans for the architecture. The appendix to the paper lists the online movies, which highlight key features of the HAC application.

2. AFS: THE ABSTRACT FORCE SIMULATOR

The purpose of AFS is to allow us to simulate a variety of domains that can be characterized by agents moving and applying force. AFS represents agents abstractly, as "blobs," which have a small set of physical features, including mass, velocity, friction, radius, attack strength, and so on. Blobs can either be modeled as spheres with uniform mass density, or we can use a particle-spring model to represent arbitrary shapes which can deform and redistribute their mass (see also Appendix A.2).

A blob is an abstract unit; it could be an army, a soldier, or a political entity. Every blob has a small set of primitive actions it can perform, PRIMITIVE-MOVE, APPLY-FORCE, and CHANGE-SHAPE. All other behaviors are built from these primitives. We can transform AFS from a billiard ball simulator into a division level military engagement simulator simply by changing the underlying physics. This would involve changing how mass is affected by collisions, what the friction is for a blob moving over terrain, and so on.

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, the properties of this space, such as terrain attributes, are represented as a discrete grid of rectangular cells for reasons of efficiency. 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 the number of blobs in the simulator.

3. HAC: HIERARCHICAL AGENT CONTROL

An important aspect of any agent architecture is how it manages the flow of information between agents and parts of agents. In HAC, we distinguish between three kinds of information: control information, sensor information, and context. Each type of information corresponds to a separate hierarchy in HAC. We will now discuss each in turn.

3.1 The Control Hierarchy

HAC organizes the agent's actions in a hierarchy (see Figure 2). 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

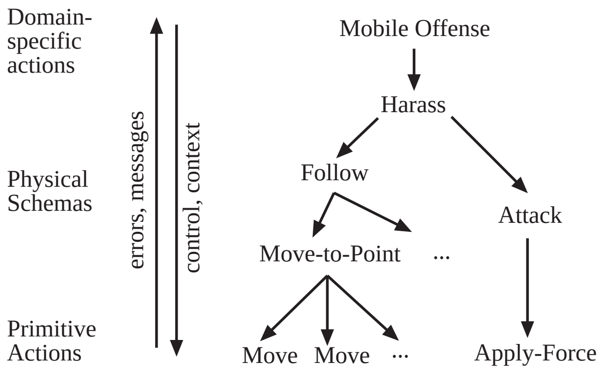

Figure 2: Actions form a hierarchy; control information is passed down, messages are passed up. The lowest level are agent effectors; the middle layer consists of more complex, yet domain-general actions called *physical schemas* [3]. Above this level we have domain-specific actions.

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. They solve more difficult problems, such as path planning, and can react to a wide range of eventualities.

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 [18]. 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.

Figure 2 shows a small part of an action hierarchy. The FOLLOW action, for example, relies on a MOVE-TO-POINT action to reach a specified location. MOVE-TO-POINT will send status reports to FOLLOW if necessary; at the very least a completion message (failure or success). The only responsibility of the FOLLOW action is to issue a new target location

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

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

(defmethod redirect ((game-state game-state) (action swarm) agent)
  (start-new-child action game-state 'move-to-point
                   :agent agent
                   :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 agent in (agents action) do
          (redirect game-state action agent))))
```

Figure 3: Implementation of a multi-agent "swarm" behavior in HAC.

if the agent being followed moves. HAC is an architecture; other than enforcing a general form, it does not place any constraints on how actions are implemented. Every action can choose what messages it will respond to. 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. Unlike other architectures [13, 5], we do not prescribe a preset number of behavioral levels. Parents can run in parallel with their children or only when the child completes.

3.2 An Example Action Definition

This section will elucidate the action-writing process using a concrete example. HAC provides a number of methods to make the process of writing actions easier. Across actions we must perform the same sorts of tasks: generating messages for the parent, executing 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 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 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-message** 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. The swarm example never completes, and it does not report on its status, so it generates no messages.

3.3 The Sensor Hierarchy

The sensor hierarchy provides a principled means of structuring the complexity of reading and transforming sensor information in AFS. Its function is analogous to how the HAC action hierarchy ameliorates the complexity of controlling agents. The sensor hierarchy is grounded by the low level primitives available from the physics simulation: the location of terrain features, the current speed and location of agents on the map and so forth.

Each level in the hierarchy integrates and extends the level below it by compiling the available information and providing additional structure. We call these higher levels *abstract sensors*, because they do not sense anything directly from the world. For example, enemy location information can be combined into a sensor that specifies overall enemy presence; terrain information can be combined into a sensor that specifies passes and movement corridors. Furthermore, these two

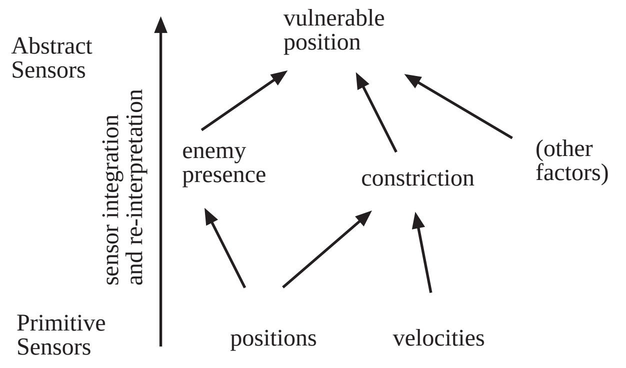

Figure 4: Raw sensor data is transformed into more complex concepts via the abstract sensor hierarchy.

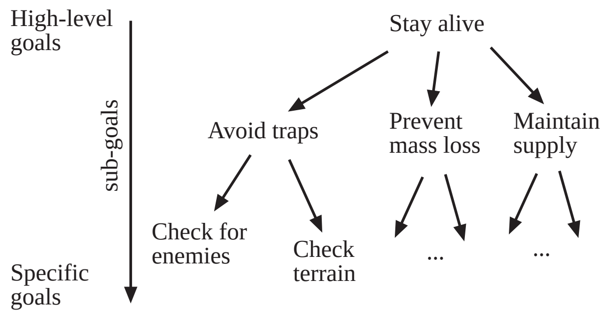

Figure 5: A goal tree for the high-level goal "stay alive." Many such goals exist; each expands into a tree of subgoals. The set of sub-goals appropriate to an action define the *context* under which the action operates.

sensors can be combined to show enemy vulnerability: areas where enemy units are concentrated and cannot move quickly (see Figure 4).

The sensor hierarchy shares the control hierarchy's syntax and structure. Each sensor is analogous to a HAC action. It sends and receives messages and performs sensor computation during its **realize** method. One advantage of this is that the same principles learned in building actions carry over directly when building sensors. This linkage also makes it easy for actions to use sensors as part of their control mechanism. An action can react to a sensor using **handle-message** methods, the same way it reacts to child actions. Each sensor is associated with the set of actions that request it and completes when this set becomes empty.

Like actions, sensors also abide by the principle of supervenience. Higher level sensors integrate and interpret lower level ones but they do not change the lower level information. Lower level sensors provide information to the higher level ones but they do not tell them what to say. One advantage of this is that each level of the hierarchy can be viewed independently without worrying about the levels coming into it or the levels that are using it.

Abstract sensors also play important and complimentary roles in extensions to AFS. For example, in CtF, we have incorporated a model of defeat that simulates psychological factors such as morale and courage [14]. These factors are significantly affected by perception. If a battalion believes it is isolated, its morale will decrease and probability of surrender will increase. Instead of creating a perceptual system, we use abstract sensors to acquire perceived information of our enemy. This melds well with the military intelligence view of abstract sensors. Furthermore, since abstract sensors do not necessarily provide perfect information, the behavior of our defeat model is more believable.

3.4 The Context Hierarchy and Planner

Even if it is not explicitly expressed, every action is trying to achieve a goal. The MOVE-TO-POINT action is trying to satisfy the goal of getting an agent to a specified position on the map, the ATTACK action is trying to satisfy the goal of damaging another unit. Having actions focus only on their primary goal can sometimes lead to unintelligent behavior. For example, an action that blocks the approach route to a flag is trying to achieve the goal "let no-one pass." If the enemy is able to sneak around some other way, this BLOCK action no longer makes sense. Another example is the MOVE-TO-POINT action. If an agent is moving to a destination and is attacked, it will continue to move, even it would be totally destroyed by doing so.

The problem can be rectified by introducing a notion of *context* to the action. Part of the context of the BLOCK action is the absence of enemies in the area one is trying to protect, part of the context of MOVE-TO-POINT is not to be destroyed. Instead of adding numerous conditional statements to every action that specify all the exceptions to normal behavior, we handle context by having agents satisfy not single, but *sets* of goals.

When actions are initiated, a set of goals can be specified for the agent (or group of agents) executing the action. This goal set defines the context for the action. The set can be viewed as the set of common sense or implicit assumptions an agent should always be considering when trying to achieve the task at hand.

Like actions and sensors, goals are part of a hierarchy. The high-level goal of "staying alive" can be decomposed into more specific goals, depending on the situation (see Figure 5). A hierarchy exists for every such high-level goal. In the general case, agents will be attempting to satisfy goals from several goal hierarchies simultaneously. This interlocking network of goals and actions is what we refer to as the *context hierarchy.*

A mechanism is required to resolve multiple—and possibly conflicting—goals. We have developed a planner, GRASP (General Reasoning using AbStract Physics), that does just that (see [3] for details). GRASP is a least-commitment partial hierarchical planner [12]. Such planners are particularly well suited to continuous and unpredictable domains such as CtF, where the plan space branching factor can be very high. Partial hierarchical planners rely on a library of plan skeletons. Plan skeletons are plans that are not fully elaborated: they may contain unbound variables or subgoals which are not filled in until run-time. Plan skeletons are implemented as actions in HAC. GRASP extends the traditional partial hierarchical planning framework by allowing multiple goals to be associated with a resource or set of resources. These are not simply conjunctive goals; instead, goals are prioritized.

GRASP is invoked whenever multiple goals have to be achieved by one resource, or when many actions (plans) could be used to achieve one goal. Every goal has a priority associated with it; higher priority goals will always be considered first. Plans are retrieved to achieve each goal in the goal set, ordered by priority. Primarily using the heuristic of minimizing resource use, a small set of plan combinations is generated. The plans are evaluated by simulating them and the one resulting in the most favorable future world state is chosen [1]. If problems arise during a plan's execution (because a resource was destroyed and the plan using it cannot succeed without it, for example), an error message is sent to the plan initiator using the HAC messaging mechanism, possibly causing resources to be re-assigned or a complete replan to take place.

Not all goals will be applicable in a given situation ("staying alive" is only relevant when a threat to the agent exists). These *latent goals* are only added to an agent's goal set when their triggering condition is met, simplifying the planning process. A replan is triggered whenever an agent's goals change. If a latent goal should be achieved at any cost, even to the exclusion of other goals, the latent goal's priority can be set to a value higher than that of any other goal.

4. THE HAC APPLICATION

HAC's core simulator and planner were developed using ANSI Common Lisp [20]; the graphical user interface uses Lisp extensions specific to Macintosh Common Lisp (MCL). The system runs under MacOS 8.x or 9.x and requires a G3 or better Macintosh with at least 64 Megabytes of RAM. Map displays and visualizations require displays with thousands or millions of colors at a resolution of at least 1024 x 768. HAC supports a text based network socket interface that can be used both for network play and to control other simulators that support the same interface.

AFS is appropriate for any agent-based simulation of the physical world that can be modeled abstractly using primitives such as MOVE and APPLY-FORCE. HAC is appropriate for any application where you want to define intelligent behavior for an agent in a hierarchical manner. In particular, we have used the same framework to model a military adversarial planner, a Pioneer 1 robot simulator and to control and plan in commercial games such as Battlezone and Dark Reign. HAC is very lean; we estimate that in our "Capture the Flag" application, HAC itself (which does not include the actions the 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. From these basic elements, the action writer can build up arbitrarily complex behavior. Currently writing actions requires programming skills, but one of our goals is to explore how simple we can make this process.

5. LESSONS LEARNED

The development of HAC is an ongoing process. This section summarizes the lessons we learned while developing HAC and some of our plans for future work.

- *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 more and more of these *action idioms* part of the HAC support mechanisms. For example, 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. Eventually, we would like writing actions in HAC to be like putting together a structure from building blocks. Ideally, this would be done using a graphical user interface, making the process more accessible to a non-programmer.
- *Modular actions and sensors.* Designing modular actions requires a certain degree of discipline on the part of the action designer. HAC helps by adhering to the principle of supervenience and by giving common action idioms to the designer. Being able to offload action contingencies to the planner through the multiple goal resolution mechanism also facilitates modularity.
- *Flexibility in action implementation.* 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. For example, we have used HAC to implement a subsumption architecture in our AFS-based robot simulator.
- *The action, not the agent, is central.* By having our control 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.* The issue of resource management quickly became paramount in the design of intelligent actions. 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. Initially overlooking the importance of resource management was probably one of the main lessons we learned. Since our hierarchy is organized around *actions*, resources are the agents performing 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.
- *Three separate hierarchies.* 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. Dealing with multiple goals at the agent level was a natural extension of having to achieve multiple simultaneous goals in the domain.

One of the more exciting directions we are moving in with HAC is generalizing it so it can be used to control physical robots. We would like to model resources at the level of agent effectors, allowing us to assign parts of agents to different tasks. It is interesting that robotics and modern real-time simulators place similar demands on an architecture. The current HAC engine uses a centralized queue and imposes no constraints on the CPU time used by an action. Future engines will be operating in a real-time, decentralized environment, and will need 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.

6. SUMMARY AND RELATED WORK

This paper has introduced HAC as a domain-general agent design tool. These are the issues we believe HAC addresses well:

- Agent control, action execution, planning, and sensing are all part of the same framework.
- Resources are explicitly modeled.
- Actions are not monolithic entities that always run to completion. Actions send messages about their status, completion (either successful or unsuccessful), or problems. They can be and often are interrupted or rescheduled.
- HAC is a modular system; supervenience enables us build re-usable action modules.
- Latent goals allow unforeseen events to be exploited.
- HAC is organized around tasks that have to be accomplished. Resources then become the agents that implement actions.

Peer-to-peer communication can be implemented in HAC using the messaging mechanism, but we have seldom found a need to do this. HAC does not include built-in mechanisms for agents to advertise their goals or for negotiating teams (as opposed to other architectures, where this is one of the primary foci, for example [21]). Our philosophy is that even peer-to-peer communication requires some kind of context, at the very least an agreement on the communication protocols involved. In HAC, parent actions provide this context explicitly. Parents are primarily responsible for coordinating their children. In the absence of a pre-established action hierarchy, one could imagine a mechanism that creates parents on the fly for agents that want to cooperate.

The GRASP planner integrates a number of new and old ideas to deal with continuous and adversarial domains in real-time. It builds upon the established notion of a control hierarchy, used in many agent architectures and hierarchical task network planners (e.g., [22, 6]). The idea of reasoning using procedural knowledge has also been used in a number of other systems, including PRS [11], PRS-Lite [17], RESUN [4], PHOENIX [5], the data analysis system AIDE [19], and in languages for reactive control such as RAP [7], XFRM [16] and PROPEL [15]. The APEX architecture also attempts to manage multiple tasks in complex, uncertain environments, placing particular emphasis on the problem of resolving resources conflicts [10].

Although many systems reason about multiple concurrent goals, GRASP is unique among partial hierarchical planners in that it places much of the burden of resolving these goals on the planner, using the availability of resources as its primary heuristic. Unlike PRS and RAP, for example, GRASP does not require the designer of actions (tasks) to anticipate every possible event interaction. Plans that react to unforeseen events can be kept conceptually separate from those that are implementing longer term goals.

Like PRS, HAC allows for the specification of blocking and non-blocking children (child actions that run in sequence with their parents or in parallel), and like later versions of RAP [8], success and failure are treated like any other message, and do not implicitly determine the flow of control between actions.

HAC and GRASP use the same representation for actions at all levels of the hierarchy, and also for plans and sensors. Contrast this with the majority of current agent control architectures, e.g. CYPRESS [23] and RAP [9], which distinguish between procedural low-level "skills" or "behaviors" and higher level symbolic reasoning. Different systems are often used to implement each level (CYPRESS combines SIPE-2 and PRS, for example). HAC does not conceptually differentiate between discrete actions and continuous processes, nor does it limit the the language used to describe them. Although we provide macros and functions to streamline the behavior writing process, all the power of the Lisp programming language can be used in any action or plan.

Acknowledgments

This research is supported by DARPA/USAF/ISO under contract numbers F30602-99-C-0061 and F30602-97-1-0289. 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.

7. ADDITIONAL AUTHORS

Additional authors: Brent Heeringa (University of Massachusetts, email: `heeringa@cs.umass.edu`), Andrew Hannon (University of Massachusetts, email: `hannon@cs.umass.edu`), and Paul R. Cohen (University of Massachusetts, email: `cohen@cs.umass.edu`).

8. REFERENCES

[1] M. S. Atkin and P. R. Cohen. Using simulation and critical points to define states in continuous search spaces. In *Proceedings of the 2000 Winter Simulation Conference*, pages 464–470, 2000.

[2] M. S. Atkin, D. L. Westbrook, and P. R. Cohen. Capture the Flag: Military simulation meets computer games. In *Proceedings of AAAI Spring Symposium Series on AI and Computer Games*, pages 1–5, 1999.

[3] M. S. Atkin, D. L. Westbrook, and P. R. Cohen. Domain-general simulation and planning with physical schemas. In *Proceedings of the 2000 Winter Simulation Conference*, pages 1730–1738, 2000.

[4] N. Carver and V. Lesser. A planner for the control of problem solving systems. *IEEE Transactions on Systems, Man, and Cybernetics, special issue on Planning, Scheduling, and Control*, 23(6):1519–1536, November 1993.

[5] P. R. Cohen, M. L. Greenberg, D. M. Hart, and A. E. Howe. Trial by fire: Understanding the design

requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, Fall 1989. also Technical Report, COINS Dept, University of Massachusetts.

[6] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[7] R. J. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, Seattle, Washington, 1987.

[8] R. J. Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 49–54, 1994.

[9] R. J. Firby. Modularity issues in reactive planning. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, pages 78–85, 1996.

[10] M. Freed. Managing multiple tasks in complex, dynamic environments. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 921–927, Madison, WI, 1998.

[11] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 972–978, Detroit, Michigan, 1989. AAAI Press, Menlo Park, CA.

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

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

[14] B. Heeringa and P. R. Cohen. An underlying model for defeat mechanisms. In *Proceedings of the 2000 Winter Simulation Conference*, page 933, 2000.

[15] R. Levinson. A general programming language for unified planning and control. *Artificial Intelligence*, 76(1-2):319–375, 1995.

[16] D. McDermott. Transformational planning of robot behavior. Technical Report YALEU/CSD/RR #941, Yale University, New Haven, CT, Dec. 1992.

[17] K. L. Myers. A procedural knowledge approach to task-level control. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, pages 158–165, 1996.

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

[19] R. St. Amant. *A Mixed-Initiative Planning Approach to Exploratory Data Analysis.* PhD thesis, University of Massachusetts, Amherst, 1996. Also available as technical report CMPSCI-96-33.

[20] G. L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition, 1990.

[21] M. Tambe, J. Adabi, Y. Al-Onaizan, A. Erden, G. A. Kaminka, S. C. Marsella, and I. Muslea. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110(2):215–239, 1999.

[22] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, 1988.

[23] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

APPENDIX

A. ONLINE DEMOS

We have prepared five short movies showcasing various aspects of HAC and AFS, running in the Capture the Flag domain. These movies are available on the World Wide Web at *http://eksl.cs.umass.edu/research/ctf/*. A short description of each follows.

A.1 The ‘Capture the Flag’ Interface

War games in Capture the Flag are played in a virtual world of “nearly three dimensions”: In addition to latitude and longitude we have elevation, although all hills are the same height. Opposing land and air units attempt to capture each other’s flags. They can exploit or be hindered by terrain features such as hills, rivers, and terrain types such as forest and swamp. The dynamics of engagements are influenced by models of psychological factors—fear, morale, fatigue and the like.

Opposing units can be controlled by humans or by the GRASP planner, and a common configuration is for humans to play against GRASP. A natural interface allows human players to direct their units on the battlefield, and the units themselves are capable of intelligent reactive behavior to carry out directives without constant supervision (e.g., one can direct a unit to attack another and it will figure out how best to get there, which formations to adopt, and so on). Actions available to units include: move to a location, occupy (defend) a location, retain a flag, block, follow and assume, follow and support, forward passage of line, direct attack, indirect attack (e.g., by artillery).

Capture the Flag provides land and air combat units of the following types: Tank, Mechanized Infantry, Light Infantry, Cavalry, Artillery, and Aviation. The smallest units in Capture the Flag are battalions, the largest are divisions. Units have limited sensors, and limited knowledge of the battlefield. Terrain influences visibility.

A.1.1 Game Control

The Capture the Flag game is controlled via a palette of standard commands. These include commands to move the simulation forward one tick at a time, to let the simulator run continuously, and to stop the simulator. There are also controls that allow the player to save the state of the game at any time and then return to a previously saved state. Furthermore, there are several menu commands for viewing visualizations and controlling network play.

A.1.2 Agent Control

In interactive mode, Capture the Flag lets the player control her units via a simple interface resembling those used in computer games. To tell an agent to do something, the user first clicks on the agent and then clicks again somewhere else on the board. Capture the Flag automatically chooses the most typical action depending on the agent be-

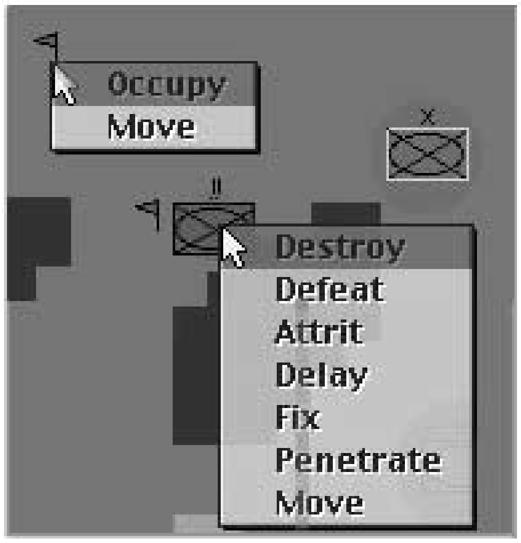

Figure 6: Context sensitive menus

ing given the command and what is under the mouse on the second click. For example, if the player clicks on open terrain, the unit will move to it; if the player clicks on an enemy unit, the unit will attack it; and if the player clicks on a friendly unit, the unit will provide it with support. If the player holds down the control key while clicking, Capture the Flag will display a menu with all of the available actions (Figure 6). This convention allows for both simple and sophisticated play using the same general interface. Lastly, multiple units can be given a command simultaneously by selecting the first unit and then holding down the shift key while selecting the others.

A.2 Particle-Spring Model for Blobs

This clip illustrates our particle-spring model for non-uniform blobs. Four features of the model are demonstrated. The first feature is the ability to change a blob's shape by reconnecting springs. The blobs start out in the delta formation, and form new shapes just by altering the configuration of their springs. Notice that the original particles move to the closest spot in the new formation and then the mass is distributed evenly among all the particles in the blob.

The second feature is the effect terrain has on blobs. You will notice a large blob in the circle formation pass through a small opening, distorting and slowing down in the process.

The next demonstration is of force application between blobs. Notice the two blobs are in delta formations, and that they lose mass at the point of overlap. As the mass is lost in the two particles, mass is redistributed from the other particles in the respective blobs to compensate.

Finally, blobs are split and then merge back together. Once again, mass distribution is the key to accomplishing these two tasks. Merging is much like applying force, but instead of losing mass when two particles overlap, one particle adds its mass to the other particle, and mass flows from the remainder of the blob to the point of overlap.

A.3 Abstract Sensors

The Abstract Sensor hierarchy provides a principled means of structuring the complexity of reading and transforming sensor information in AFS. Abstract sensors can realistically and effectively play the role of military intelligence. Depicted is a scenario where knowledge of enemy location leads to a successful defense of a friendly flag. Specifically, an artillery unit uses information provided by a friendly scout unit, through abstract sensors, to destroy three attacking infantry battalions.

A.4 Reactive and Deliberative Behavior

HAC blurs the distinction between deliberative planning and reactive action. This movie shows the FOLLOW action using planning to move towards and then engage a red enemy unit. While moving, FOLLOW must continually react to changes in the red unit's position and its own progress. HAC allows the control to flow easily between movement, path planning, and reaction to the external environment without requiring the architecture to have a preset number of cognitive levels.

A.5 Planning

In the Capture the Flag domain, winning involves coordinating multiple subgoals: protecting your own flags, thwarting enemy offensives, choosing the most vulnerable enemy flag for a counter-attack, and so on. Each requires resources (units) to be accomplished. Sometimes one resource can be used to achieve several tasks. For instance, if two flags are close together, one unit might protect both. Or advancing towards an opponent's flag might also force the opponent to retreat, thus relieving some pressure on one's own flags.

GRASP is a partial hierarchical planner, extended to handle multiple simultaneous goals. GRASP generates an approximate plan quickly, filling in details as it is executed. Partial hierarchical planners typically decide between plans based on heuristic criteria. GRASP instead performs a qualitative simulation on each candidate plan (or plan set). Potential plans are simulated forward, then a static evaluation function is applied to select the best plan. The static evaluation function incorporates such factors as relative strength and the number of captured and threatened flags of both teams to describe how desirable the resulting world state is. During plan evaluation, the opponent's actions (and planning process) is also simulated, resulting in minimax search for the best plan.