HAC: A Unified View of Reactive and Deliberative Activity

Marc S. Atkin, David L. Westbrook, and Paul R. Cohen

Experimental Knowledge Systems Laboratory
Department of Computer Science
140 Governor's Drive
University of Massachusetts, Amherst, MA 01003-4610
{atkin,westy,cohen}@cs.umass.edu

Abstract. The Hierarchical Agent Control Architecture (HAC) is a general toolkit for specifying an agent's behavior. By organizing the hierarchy around tasks to be accomplished, not the agents themselves, it is easy to incorporate multi-agent actions and planning into the architecture. In addition, HAC supports action abstraction, resource management, sensor integration, and is well suited to controlling large numbers of agents in dynamic environments.
Unlike other agent architectures, HAC does not conceptually distinguish reactive from deliberative, or single-agent from multi-agent behaviors. There is no pre-determined number of cognitive "levels" in the hierarchy—all actions share the same form and are implemented with the same functions.
GRASP is a multi-goal partial hierarchical planner that has been implemented using the HAC framework. GRASP illustrates two points: Firstly, that the same HAC mechanisms used to write reactive actions can be used to implement a cognitive activity such as planning; and secondly, that the problem of integrating reactive and deliberative behavior itself can be viewed as having to simultaneously achieve multiple goals.
Throughout the paper, we show how HAC and GRASP were applied to an adversarial, real-time domain based on the game of "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 which will allow army commanders to design and evaluate high level plans ("courses of action") in a land-based campaign.

It quickly became apparent that regardless of the domain, agent designers must face the same kinds of problems: processing sensor information, reacting

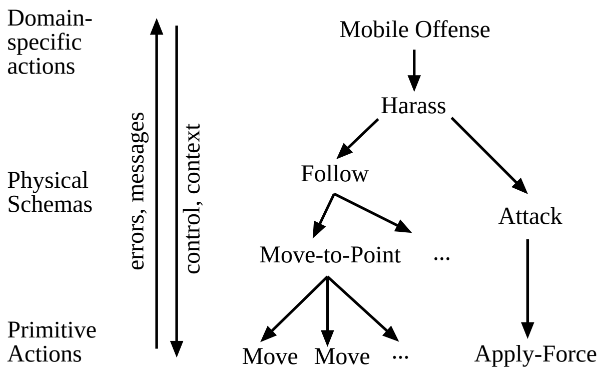

Fig. 1. Actions form a hierarchy; control information is passed down, messages are passed up. The lowest level are agent effectors; the middle layer consists of more complex, yet domain-general actions called *physical schemas.*Above this level we have domain-specific actions.

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. This led to the development of a general framework for controlling agents, called *Hierarchical Agent Control* (HAC). HAC has many unique features—the one we will be focusing on here is its ability to seamlessly integrate reactive and cognitive processes.

HAC: Hierarchical Agent Control

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, coordinating 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" [Atkin *et al.*, 1999] (see Figure 2), HAC can equally well be applied to such domains as commercial games, multi-agent simulations, or actual physical robots [Atkin *et al.*, 1998].

HAC organizes the agent's actions in a hierarchy (see Figure 1). The very lowest levels are the agent's effectors. The set of effectors will depend on the agent and the domain, but typically include being able to move the agent, turn it, or use a special ability such as firing a weapon. More complex actions are built from these primitive ones. An **attack** action, for example, may move to a target's location and fire at it. As one goes up the hierarchy, actions become increasingly abstract and powerful. They solve more difficult problems, such as path planning, and can react to wide range of eventualities.

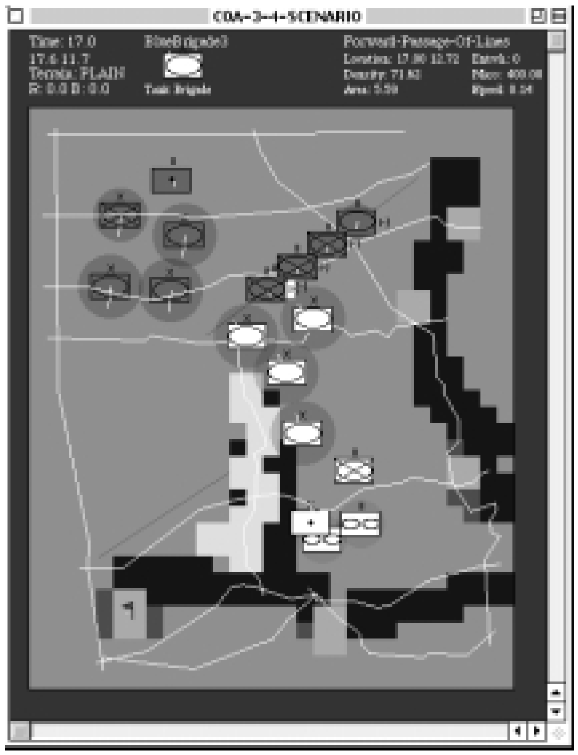

Fig. 2. The Capture the Flag domain (CtF). There are two teams; each has a number of movable units and flags to protect. They operate on a map which has different types of terrain. Terrain influences movement speed and forms barriers; terrain also affects unit visibility. A team wins when it captures all its opponent's flags.

A hierarchy of sensors parallels the action hierarchy. Just as a more complex action uses simpler ones to accomplish its goal, complex sensors use the values of simpler ones. These are *abstract sensors.* They are not physical, since they do not sense anything directly from the world. They take the output of other sensors and integrate and re-interpret it. A low-level vision system (a physical sensor) produces a black and white pixel array. An abstract sensor might take this image and mark line segments in it. A higher level abstract sensor takes the line segments and determines whether there is a stretch of road ahead. A **follow-road** action can use this abstract sensor to compute where to go next. Abstract sensors are used throughout HAC and GRASP to notify actions and plans of unexpected or unpredictable events.

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

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

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

Figure 1 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 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, we do not prescribe a preset number of behavioral levels. Parents can run in parallel with their children or only when the child completes.

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 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 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

```
(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))))
```

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

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 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. The swarm example never completes, and it doesn't report on its status, so it generates no messages.

```
An action or a plan posts a set of goals $G = \{g_1, g_2, ...g_n\}$. This invokes the following process:
1. For every $g_i$:
    1.1 Search the list of plans for those that can satisfy $g_i$.
    1.2. Evaluate each potential plan's pre-conditions and only keep only those whose pre-conditions match.
    1.3. For each remaining plan, estimate it's required resources.
2. Sort $G$ by the priority of $g_i$.
3. candidate_plan_sets := nil.
4. Loop over $g_i$ in order of priority:
    4.1 If only one plan achieves $g_i$, instantiate it (bind unbound variables) and add it to every plan set in candidate_plan_sets; otherwise:
    4.2 If several plans achieve $g_i$, score each one based on:
        • how many resources it uses
        • how many other goals in $G$ it (partially) satisfies
        • other plan-specific heuristics
    4.3 Choose $m$ ($m$ is rarely $> 1$ to limit combinatorics) of the highest scoring plans: $p_1, ..., p_m$
    4.4 Loop over remaining $g_j (j > i)$: if $g_i$ partially satisfies $g_j$, merge $g_j$ into $p_1, ..., p_m$
    4.5 Copy the plan sets in candidate_plan_sets $m$ times; add $p_k$ to copy $k$.
5. Loop over plan_set in candidate_plan_sets:
    5.1 Evaluate plan_set using forward simulation.
6. Execute the plan set (make them child actions of the goal poster) that in simulation, results in a world state with the highest score.
```

Fig. 4. The planning algorithm.

The Planner

We will illustrate how HAC integrates reactive and deliberative actions using a planner, GRASP, as an example. GRASP (General Reasoning using AbStract Physics) is a least-commitment partial hierarchical planner [Georgeff and Lansky, 1986]. Such planners are particularly well suited to continuous and unpredictable domains such as CtF, where the planning 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. In HAC, plan skeletons are implemented as actions that explicitly state the goal they achieve. Every action can be viewed as achieving some goal (for example, the **move-to-point** action achieves the goal of getting to a destination). Planning is necessary when the goal is satisfied by several actions and we have to decide between them.

GRASP is invoked by scheduling a specific action, **match-goal**, as a child. **Match-goal** takes as an argument a goal (or set of goals) to be satisfied. It then consults the plan library, checking which plans satisfy this goal. Using the algorithm outlined in Figure 4, it chooses the best plan (or set of plans). During the matching process, unbound variables in the plan are instantiated—others are instantiated by the plan itself when it runs. When **match-goal** has selected a plan, it terminates, and schedules the best plan as a child action of the action

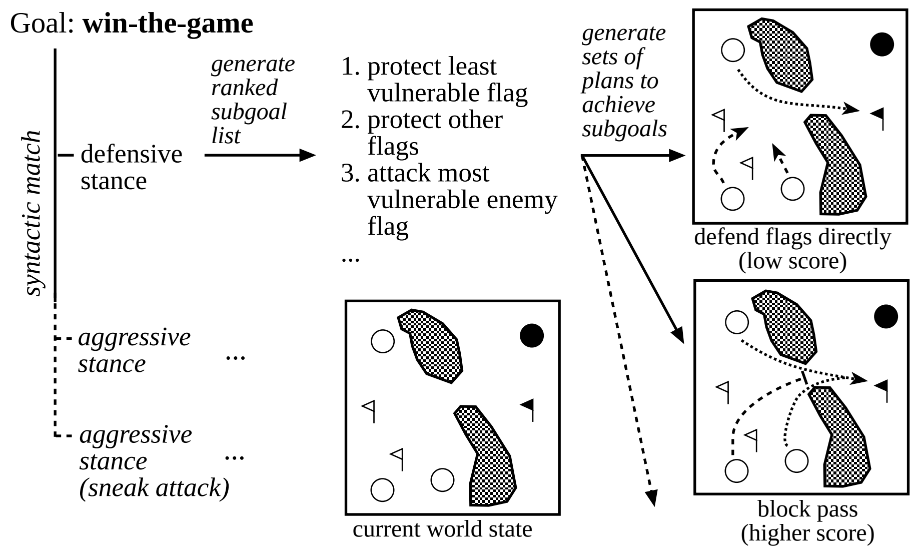
Goal: win-the-game
syntactic match
defensive stance
generate ranked subgoal list
1. protect least vulnerable flag
2. protect other flags
3. attack most vulnerable enemy flag
...
generate sets of plans to achieve subgoals
defend flags directly (low score)
block pass (higher score)
aggressive stance
...
aggressive stance (sneak attack)
...
current world state

Fig. 5. A planning example: White is trying to satisfy the goal **win-the-game**. Several top-level plans match this goal; the example explores what happens when **defensive-stance** is expanded. This plan emphasizes defense, which is reflected in the list of subgoals generated. There are several sets of plans that achieve these subgoals, and many ways to allocate resources to these plans. The planner uses heuristics to prune this set. In the first case, two units are allocated to flag defense, and one is sent out to attack. In the second case, only one unit is needed to block the mountain pass, thus protecting the flags, leaving two units for the attack. This plan set is more likely to succeed and is ranked higher.

that called it. This is completely transparent to the caller: the caller starts the planning process by posting a goal, and ends up with a child action that satisfies this goal. This child action can, in turn, post subgoals using the same process.

If **match-goal** finds no appropriate plan, it fails just like any other action, sending an unsuccessful completion message to the goal poster. If a plan fails *during* its execution, it too can send a failure message to its parent. One way for the parent to react to this message would be for it to try calling **match-goal** again, in effect triggering a re-plan.

In HAC, actions run in parallel with each other. At every tick, the **realize** method of every scheduled action is called. Since planning is usually a time-consuming process, we have implemented **match-goal** so that it spreads its computations over several ticks.

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 uses heuristics in order to achieve the largest set of high priority goals possible.

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.

Figure 5 shows an example of the plan generation procedure. Each goal is prioritized, then plans are generated to achieve each one. Heuristics are used to generate a small number of possible plan sets. If resource problems arise during a plan's execution (because a resource was destroyed and the plan using it cannot succeed without it, for example), a resource 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.

When several plans apply, partial hierarchical planners typically select one according to 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.

Latent Goals

It is instructive to view the problem of integrating reactive and deliberative actions as one of having to achieve multiple goals simultaneously. The issue of multiple goals first came up in CtF, when we found it important that agents should react opportunistically to unforeseen events. If an agent is moving to attack another unit, for example, and notices a flag along the way, it should in most cases interrupt its previous activity to capture the flag, then resume its attack.

In GRASP, we handle this by introducing the concept of a *latent goal*. A latent goal is one that is not active all the time, but only when some condition holds. In order to prevent the planner from having to consider many goals, most of which are not applicable at any given time, we make these conditional goals latent. They are only considered by the planner when their triggering condition becomes true.

Let us introduce the following terms:

- goals: static symbols that a number of plans can satisfy
- latent goals: goals that become active when a condition holds
- sentinels: an abstract sensor that monitors the world to check for the applicability of a latent goal.

When a latent goal is created for an agent (or group of agents), a sentinel is set up. When the sentinel activates, the associated latent goal is added to the set of goals these agents must consider. The latent goal now behaves exactly like

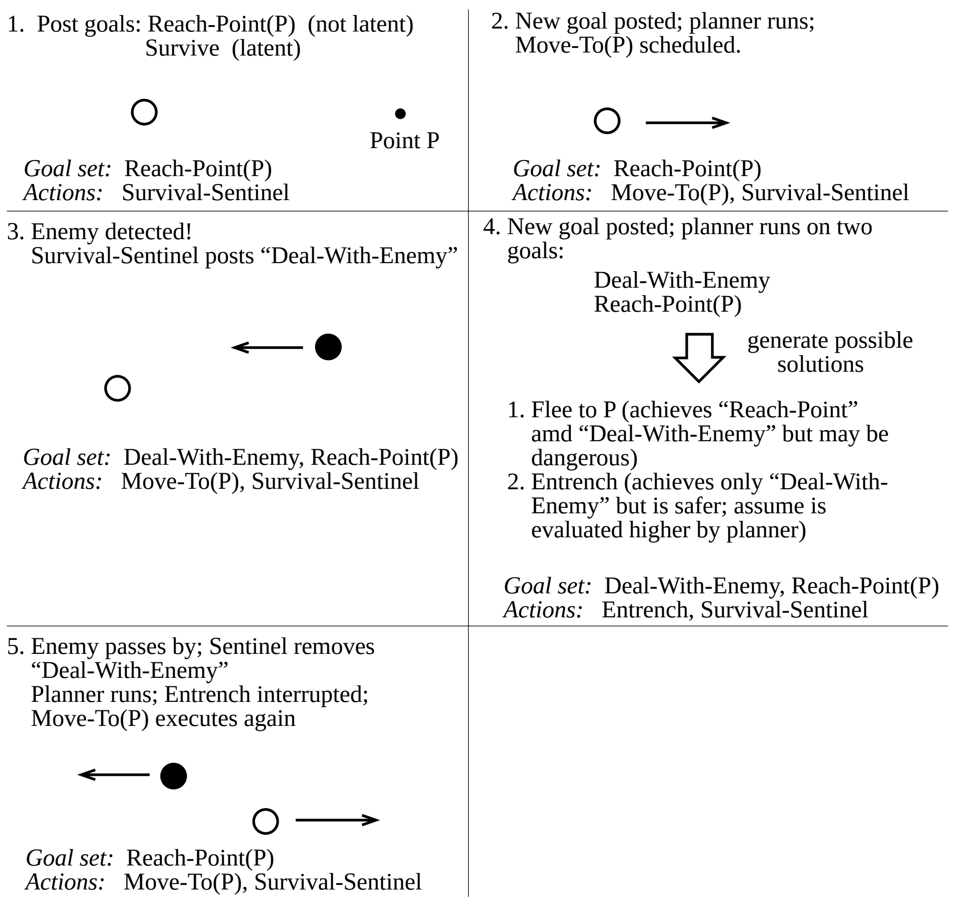
1. Post goals: Reach-Point(P) (not latent)
Survive (latent)
Point P
Goal set: Reach-Point(P)
Actions: Survival-Sentinel
2. New goal posted; planner runs; Move-To(P) scheduled.
Goal set: Reach-Point(P)
Actions: Move-To(P), Survival-Sentinel
3. Enemy detected!
Survival-Sentinel posts "Deal-With-Enemy"
Goal set: Deal-With-Enemy, Reach-Point(P)
Actions: Move-To(P), Survival-Sentinel
4. New goal posted; planner runs on two goals:
Deal-With-Enemy
Reach-Point(P)
generate possible solutions
1. Flee to P (achieves "Reach-Point" amd "Deal-With-Enemy" but may be dangerous)
2. Entrench (achieves only "Deal-With-Enemy" but is safer; assume is evaluated higher by planner)
Goal set: Deal-With-Enemy, Reach-Point(P)
Actions: Entrench, Survival-Sentinel
5. Enemy passes by; Sentinel removes "Deal-With-Enemy"
Planner runs; Entrench interrupted;
Move-To(P) executes again
Goal set: Reach-Point(P)
Actions: Move-To(P), Survival-Sentinel

Fig. 6. Latent goal example, showing how an agent's goal and action sets change over time. The white agent has the goal of reaching point P (Step 1). Along the way, an enemy is sighted, prompting the agent's "survival" sentinel to post a goal to deal with the enemy (Step 3). The planner is invoked and attempts to achieve both these goals (Step 4). The solution that achieves both (fleeing) is considered too dangerous; instead, the agent entrenches. After the enemy has passed, the "Deal-With-Enemy" goal is removed, the planner runs again, and the agent resumes moving to P (Step 5).

a goal in a normal multi-goal set. When the triggering condition is no longer met, the latent goal is removed. 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. Figure 6 illustrates this process.

Latent goals are not the same as conditional goals, used in many procedural languages. Conditional goals are created within a plan; the plan itself checks for their applicability and coordinates them with other goals being achieved. GRASP, on the other hand, places the burden of resolving latent goals on the *planner*. A plan need not know about the latent goals that may pop up during its execution.

Latent goals provide a mechanism for specifying the *context* under which an agent operates. The set of latent goals 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. In the same vein, reactive actions can be viewed as plans that achieve short-term goals. Frequently, these short-term goals arise unexpectedly (for example, an "obstacle avoidance" goal may be generated in the context of a **move-to-point** action). By extending GRASP to handle multiple goals, short-term goals can be reacted to without compromising the plans that are still executing to achieve the longer term ones.

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.
- By having our hierarchy be one of *tasks that have to be accomplished*, it was very easy to incorporate multi-agent actions and planning into our architecture. Resources then become the agents that implement actions.

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., [Wilkins, 1988; Currie and Tate, 1991]). The idea of reasoning using procedural knowledge has also been used in a number of other systems, including PRS [Georgeff and Ingrand, 1989], PRS-Lite [Myers, 1996], RESUN [Carver and Lesser, 1993], PHOENIX [Cohen *et al.*, 1989], the data analysis system AIDE [St. Amant, 1996], and in languages for reactive control such as RAP [Firby, 1987], XFRM [McDermott, 1992] and PROPEL [Levinson, 1995]. The APEX architecture also attempts to manage multiple tasks in complex, uncertain environments, placing particular emphasis on the problem of resolving resources conflicts [Freed, 1998].

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 [Firby, 1994], 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 [Wilkins *et al.*, 1995] and RAP [Firby, 1996], 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 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, 1995] Scott D. Anderson. *A Simulation Substrate for Real-Time Planning.* PhD thesis, University of Massachusetts at Amherst, February 1995. Also available as Computer Science Department Technical Report 95–80.

[Atkin *et al.*, 1998] Marc S. Atkin, David L. Westbrook, Cohen, and Jorstad. AFS and HAC: Domain-general agent simulation and control. In *Working Notes of the Workshop on Software Tools for Developing Agents, AAAI-98*, pages 89–95, 1998.

[Atkin *et al.*, 1999] Marc S. Atkin, David L. Westbrook, and Paul R. Cohen. Capture the flag: Military simulation meets computer games. In *Working Notes of the AAAI Spring Symposium on AI and Computer Games*, pages 1–5, 1999.

[Carver and Lesser, 1993] Norman Carver and Victor 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.

[Cohen *et al.*, 1989] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele 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.

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

[Firby, 1987] R. James 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.

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

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

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

[Georgeff and Ingrand, 1989] Michael P. Georgeff and Francois Felix 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.

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

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

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

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

[Myers, 1996] Karen 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.

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

[St. Amant, 1996] Robert 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.

[Wilkins *et al.*, 1995] David E. Wilkins, Karen L. Myers, John D. Lowrance, and Leonard P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

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