Intelligent Real-time Problem Solving: Issues and Examples

Paul R. Cohen, Adele E. Howe, David M. Hart

COINS Technical Report 90-20

Experimental Knowledge Systems Laboratory
Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003

Acknowledgments:
This work was supported by DARPA-AFOSR Contract 49620-89-C-0121; ONR University Research Initiative Grant N00014-86-K-0764 and a grant from the Digital Equipment Corporation. We thank Michael Greenberg, Dorothy Mammen, Paul Silvey, and David Westbrook for their contributions to this work.

Intelligent Real-time Problem Solving: Issues and Examples

Paul R. Cohen, Adele E. Howe, David M. Hart

COINS Technical Report 90-20

Experimental Knowledge Systems Laboratory
Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003

Acknowledgments:
This work was supported by DARPA-AFOSR Contract 49620-89-C-0121; ONR University Research Initiative Grant N00014-86-K-0764 and a grant from the Digital Equipment Corporation. We thank Michael Greenberg, Dorothy Mammen, Paul Silvey, and David Westbrook for their contributions to this work.

1. Introduction

This report presents our work on real time problem solving (IRTPS). The topic is fundamentally challenging in the sense that it probably cannot be completely addressed within the established knowledge-based and logicist paradigms, but will require methodological, theoretical, and technical developments. Accordingly, this report looks at IRTPS from all these perspectives. Because readers will have different interests, each section of the report is independent of all sections except this Introduction. Section 2 offers a definition of IRTPS. Section 3 describes our real-time testbed, including its current status and portability, and our timetable for making it generally available. Section 4 discusses the architecture we have developed for real-time agents and our near-term research goals. Section 5 is devoted to methodological issues, specifically, how characteristics of environments constrain the design of agents (including an assessment of the pros and cons of simulated environments), how to evaluate IRTPS systems, and the need for analytic models of agent architectures.

The task environment for much of our research is a simulation of forest fires. The task is to control simulated fires by deploying simulated agents, including "smart" bulldozers, fuel carriers, and airplanes. (Smart agents have the simulated physical abilities of, say, bulldozers, and some of the simulated mental abilities of their human operators.) This is a realtime problem in the basic sense that *the environment changes while agents think and act.* If agents think too long, the fires get too big to control. If they don't think long enough, their plans may be flawed and their actions may be less effective. (Section 2 refines this basic definition of the real-time problem.)

The Phoenix system comprises five levels of software:

DES -- the discrete event simulator kernel. This handles the low-level scheduling of agent and environment processes. Agent processes include sensors, effectors, reflexes, and a variety of cognitive actions. Environment processes include fire, wind, and weather. The DES provides an illusion of simultaneity for multiple agents and multiple fires.

Map -- this level contains the data structures that represent the current state of the world as perceived by agents, as well as "the world as it really is." Color graphics representations of the world are generated from these data structures.

Basic agent architecture -- a "skeleton" architecture from which agents, such as bulldozers, airplanes, and firebosses are created. The agent

architecture provides for sensors, effectors, reflexes, and a variety of styles of planning.

Phoenix agents -- the agents we have designed (and are designing) for our own RTPS experiments.

Phoenix organization -- currently we have a hierarchical organization of Phoenix agents, in which one fireboss directs (but does not control) multiple agents such as bulldozers. Each Phoenix agent is autonomous and interprets the fireboss's directions in its local context, while the fireboss maintains a global view. A related project is looking at multiple firebosses and distributed control.

The Phoenix environment (the DES and map level), the basic agent architecture, and Phoenix agents are independent software packages that we offer to other researchers (see Section 3.3). We will offer instrumentation for these components of the Phoenix system by the end of Phase I of the IRTPS initiative.

Our research on IRTPS is part of a larger project whose goal is to develop a sound basis for the design of AI agents. We are analyzing agents in terms of the *behavioral ecology view* shown in Figure 1. This view encourages us to ask how the characteristics of environments (including time) constrain the design and behavior of agents. (We compare this view with the S/E model of Rosenschein, Hayes-Roth and Erman, in Section 5). When we speak of a sound basis for design, we mean the ability to predict how modifying the architecture of an agent will change its behavior in a given environment. Currently, we do this by building models that relate the architecture of an agent to behaviors. The methodological implications of the behavioral ecology view and of modelling are discussed in Section 5.

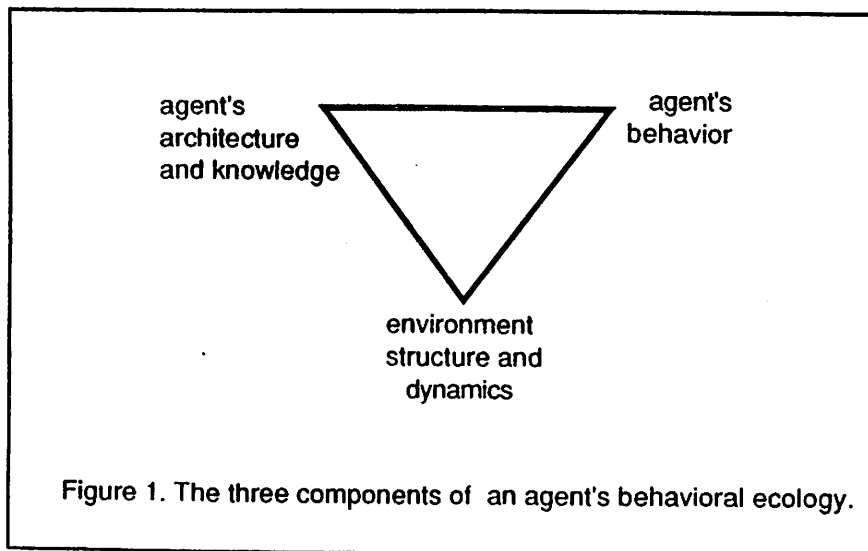


Figure 1. The three components of an agent's behavioral ecology.

2. Definitions.

We begin this section with definitions of real-time problem solving and "the real time problem." Next we examine some terms that are common in the IRTPS literature, such as deadline, predictability, and time scale. These terms are vague, and it is often difficult to tell whether they are intended as descriptions of an agent's environment or its behavior. We propose six classes of terms that should, we hope, reduce the vagueness and ambiguity of previous discussions of IRTPS. Lastly, we show how the behavioral ecology view helps us compare and organize different approaches to IRTPS.

2.1 IRTPS and the Real-time Problem

A crude definition of IRTPS was mentioned in the introduction:

> *The environment changes while agents think and act.*

But this doesn't adequately convey the impact of changes in the environment upon the agents. For example, while a bulldozer thinks about how to avoid a fire, the position of the fire changes, and in some cases the bulldozer can be overrun. A better definition makes explicit the value of problem solving and how it is affected by changes in the environment:

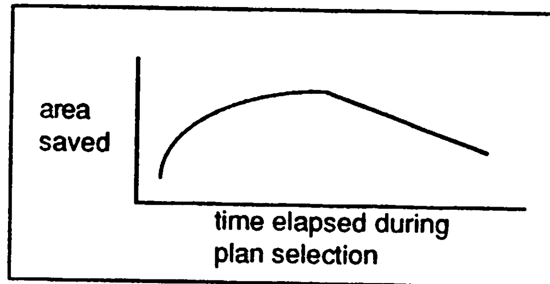
> *The value of problem solving is a function of what the problem solver does --- its thinking and acting --- and one or more parameters in the environment, at least one of which changes during problem solving.*

Note that this definition makes no direct reference to time. This is because time is itself an indirect way of talking about changes in the environment during problem solving, and it is these changes, not the passage of time, that affect the value of problem solving. When we say, "The fire is currently consuming 10 acres per hour," we do *not* mean that an hour is worth ten acres. Time itself has no inherent value. Time provides us a scale on which to measure events that do have value. Consider an analogy to distance. We might say, "As we drive down this street, property values increase by $10,000 a block," but we would not say that a block (e.g., 200 yards) is worth $10,000. Throughout this document we will try to avoid giving the impression that time has any value. We will try to foster the view that time (like distance) is just a scale on which to plot changes in value.

Of course, we can define value to be a function of time, as in real-time operating systems, but typically we measure value in terms of money, acreage, real estate, lives saved or lost, and so on. Unlike time, these value functions may be nonlinear, even discontinuous. This leads to the following definition of the "real time problem":

> *The value of problem solving does not always increase, nor does it always decrease, during problem solving; thus simple strategies such as "work for as long as possible," or "solve the problem as quickly as possible," will generally not maximize value.*

Example: Measuring value in terms of the area of burned forest, we might collect statistics on the relationship between area burned and the amount of time that elapses while the fireboss selects a plan. (For simplicity, we will plot elapsed time against area saved instead of the area burned, so value increases on the y axis:)



Apparently, a little time devoted to plan selection is worthwhile. After a point, however, thinking longer doesn't save more forest. While this "inverted U" could have many causes, the important point is that it seems to be characteristic of real time tasks.

The inverted U function seems more representative of soft deadlines than hard ones. Value decreases slowly when a soft deadline is missed, precipitously when a hard deadline is missed. In Section 2.2 we formalize and illustrate a hard deadline in the Phoenix environment.

It follows from this definition of the real time problem that an agent must have control of the amount of time it devotes to problem solving. In terms of the previous example, an agent should spend "just enough" time on plan selection---the amount of time that corresponds to the highest point on the curve. Although this picture is an oversimplification (we will discuss some of the complexities later) it does illustrate that if an agent cannot control the amount of time it spends on problem solving, it cannot affect the value of its problem solving.

What does it mean for an agent to control the amount of time it spends on problem solving. First, it does *not* mean that the agent controls the rate at which time passes. We assume this is beyond every agent's control. Instead we mean that an agent can with "one eye on the clock" decide whether to run a process or continue running an interrupted process.

Example: The Phoenix agent architecture provides for multiple *execution methods* to achieve any goal. For example, Phoenix has several path planning methods. Execution methods for a given task require different amounts of time. More precisely, they differ in the amounts of time that are expected to elapse before each terminates. One way that Phoenix agents control the amount of time they spend on problem solving is to base the selection of execution methods on their estimated time requirements.

Estimates figure heavily in this example, and in IRTPS in general. What remains to be seen, in the course of this research, is what characteristics of the environment affect the quality of estimates, and what characteristics of IRTPS architectures affect their dependence on the quality of estimates.

2.2 Describing the Environment and Agents

Although our approach is to design agents for specific environments, we have been content to describe environments at two levels of abstraction, both inadequate for design. We believe this is common in the IRTPS literature.

The levels, with examples, are:

Implementation-specific: When a cell ignites, the simulator figures out when its knights-tour-neighbors are going to ignite. It calculates the rate of spread of the newly-ignited cell to its neighbors, accounting for weather, slope, fuel type, etc.

Apple-pie general: The Phoenix environment is characterized by unpredictable events, real-time constraints, and hard and soft deadlines.

Neither level of description of the environment is appropriate for design. From the first kind of description you can model the structure and dynamics of the environment, so it is genuinely useful. But the second kind of description is actually misleading without a lot of clarification, as the following examples show.

At the IRTPS Workshop[2], the Working Group on Architectures made a list of characteristics of environments:

lots of data

low signal to noise ratio

unpredictable rates at which data arrive (varying quantity of data)

hard and soft deadlines

time-dependent value

spectrum of predictability

---

[2]Pasatiempo, Santa Cruz, California. November 6 & 7, 1989.

incompleteness in data

multiple time scales

combinatoric proliferation of things to attend to

Most of these seem self-explanatory. However, most could be interpreted as descriptions of the agent as well as descriptions of the environment. Take "multiple time scales." Our definition of time scale is the average time between causal event cycles that have value for the agent. Some cycles are very short (e.g., the time between moving into a fire and getting burned) and some are much longer (e.g., the time between a wind shift and the recognition of failure of a fire-fighting plan). But it doesn't make sense to talk about time scales independent of an agent; specifically, independent of the value of events to the agent. Without the concept of value, there's no way to classify the limitless number of causal event cycles, and so the distribution of time scales is uniform. The concept of value enables us to select classes of events—those that have value to the agent—and compute the average length of their causal event cycles. Scale depends on the agent design. It is not an inherent property of environments[3].

Which of the other characteristics listed above is inherent to environments, and which depend on the agent design? For some, both interpretations make sense. When we say "lots of data," we could mean two things: First unlike environments like the blocks world, a lot is happening in the Phoenix environment—there's a lot for the agent to attend to. This seems to be a description of an inherent characteristic of the environment. But we might also mean that in this environment, the agent's sensors can take in more stuff than it can process. So "lots of data" can be interpreted as a characteristic of the environment or as a potential problem for the agent.

Next, consider the "predictability" characteristic. When we say events are unpredictable, we are usually mean "unpredictable for this agent." But, again, we might mean "unpredictable for *any* agent." Once again, we must take care to separate the inherent characteristic of the environment—the one that would constrain any agent—from the potential problem that the environment poses the agent.

The resolution of this ambiguity should provide us with the appropriate level of description of environments for doing design.

There is a predicate called, U_e, that takes environmental events as arguments. $U_e(e)$ means that no agent can predict event e.

There is a predicate U_a that takes environmental events and agents as arguments. If $U_a(e,a)$ then agent a cannot predict event e.

---

[3]We are grateful to Les Gasser for pointing this out.

By definition, $U_e(e)$ implies $U_a(e,a)$ for all a. But it is not the case that $U_a(e,a)$ implies $U_e(e)$. This means, at the very least, that we have to be careful when we say an environment is characterized by unpredictability. More importantly, it points to a gap in our understanding of the agent-environment interaction: If $U_a(e,a)$ and not $U_e(e)$, there must be something about event e that makes it unpredictable to agent a, and if we want to design an agent a' for which Ua(e,a') is false, we have to know why Ua(e,a) is true. To designers, it helps to know $U_e(e)$, but knowing $U_a(e,a)$ doesn't help us fix the problem. We need to know something else. For example, if changes in the position of the fire are unpredictable to an agent, and we view this as a problem, then we need to know why the changes are unpredictable. Two contributing factors may be the limited field of view of agents and the statistical distribution of changes in wind speed and direction. One is an architectural characteristic, the other an environmental characteristic, and together they produce U_a(fire-position,agent).

Terms like "unpredictable" are just shorthand for problems faced by particular agents, not characteristics of the environment, except when they are universally quantified over agents. From the standpoint of design, they don't help us much. Instead we will develop a vocabulary to describe environments in *problem-independent* terms. When we say that an event has a statistical distribution we do not imply anything about the architecture of an agent, or anything about the problem that may arise for an agent as a result of the event having a statistical distribution. We suggest six classes of terms:

Environment characteristics (ECs). These are problem-independent, architecture independent descriptors of the environment. For example, a parameter (say, windspeed) changes aperiodically. A counterexample: Windspeed is unpredictable.

Architecture characteristics (ACs). These are problem-independent, environment-independent descriptors of the architecture. For example, the architecture has a random access memory of limitless capacity; or, the plan selection mechanism is bounded in computation time. A counterexample: the error recovery mechanism exhibits graceful degradation. This is a counterexample because graceful degradation implies something about the problem you are trying to solve.

Problems. A problem is a shorthand for an undesirable behavior, that is, an undesirable interaction between a particular agent and a particular environment. For example, unpredictability is a shorthand for interactions in which, because an agent did not anticipate an environmental event, some negative consequence occured.

Inherent problems. An inherent problem is a problem that we believe all agents face, that is, an undesirable interaction between *any* agent and a particular environment.

Solutions. A solution is a shorthand for a desirable behavior that we, as designers, want to see instead of some undesirable behavior—the

problem. For example, a fast sense-act loop is sometimes a solution to the unpredictability problem.

Solution realizations. A solution realization is one or more architecture characteristics or modifications to architecture characteristics; in short, what we intend to do to the architecture to ensure that the solution (which is a behavior, remember) will occur when we want it to.

In Section 5.1, we illustrate how design of a real-time mechanism proceeds from an informal description of a problem, through a formal description in terms of ECs, ACs and Problems, to solutions and solution realizations.

2.3 The Behavioral Ecology Triangle Organizes and Justifies IRTPS Approaches

We can characterize the dozens of approaches to IRTPS in terms of the behavioral ecology triangle in Figure 1. First, what behaviors do we want from IRTPS systems? Second, what characteristics of the environment make particular behaviors necessary or desirable? Third, what architectural decisions can designers make to achieve the desired behaviors in the given environment?

Example: A desired behavior is for Phoenix agents to meet their deadlines. Two characteristics of the Phoenix environment make this desirable: First, most fires must be contained by the coordinated efforts of several agents. Second, fires spread in such a way that if one agent is very late, the work of others is jeopardized. Another characteristic of the environment conspires against coordinated effort: unpredictable changes in parameters such as wind speed and direction differentially affect the progress of agents. The architectural decisions that allow Phoenix agents to meet deadlines despite unexpected events are discussed in Section 4.2.

The behavioral ecology view provides a framework for organizing the real time literature. For example, we can ask what characteristics of the environment make anytime or approximate processing behaviors necessary or desirable, and what architectural choices are needed to implement anytime or approximate behaviors in particular environments. But we have found that the principal advantage of the behavioral ecology view is that it forces us to justify our design decisions in terms of agents' environments.

Example: It is not uncommon to claim that IRTPS requires a behavior called "graceful degradation." (Other candidates are anytime or approximate behavior). Too often, the next step is to build an architecture that implements this behavior in some environment. This is backwards. The first step must always be to ask whether the environment makes graceful degradation (or anytime, or approximate behavior) necessary or desirable.

3. A Real Time Testbed.

This section describes describes the Phoenix testbed from several perspectives. Section 3.1 describes how the testbed appears to a user. Section 3.2 focuses on how the discrete event simulator manages simulation time and cpu time for multiple pseudo-parallel processes. Section 3.3 describes three levels of instrumentation for the testbed. Section 3.4 presents a "baseline scenario," that can be run again and again under different conditions to test real-time architectures. Section 3.5 addresses portability issues. The structure and implementation of the testbed is independent of the architecture of Phoenix agents; indeed, we hope that other researchers will use the testbed as an environment in which to test their own agent architectures. For this reason, we will postpone discussing the Phoenix basic agent architecture until Section 4.

3.1 The Appearance and Behavior of the Testbed.

If you watch the Phoenix system run, this is what you will see: A color representation of Yellowstone National Park, in which fires are spreading and several bulldozers, fuel-carriers, and other agents are travelling and cutting fireline. You will see different kinds of vegetation coded by color. You will also see roads and rivers of different sizes, elevation lines, lakes, houses, and watchtowers. Status windows present elapsed time, wind speed, and wind direction. You have full control over the resolution of your view; for example, you can see the entire map at low resolution or just a few acres at high resolution. You can see the environment as it really is, and as it is perceived by one or more of the agents (which have limited fields of view). Figure 2 shows a view of an area of the park, unfortunately not in color (but see [Cohen, 1989] for color pictures). The grey region at the bottom of the screen is the northern tip of Yellowstone Lake. The thick grey line that ends in the lake is the Yellowstone River. The Grand Loop Road follows the river to the lake, where it splits. The Smokey the Bear symbol in the bottom left corner marks the location of the fireboss, the agent that directs and coordinates all others. Two bulldozers are shown cutting fireline around a fire in this figure. Two other bulldozers are parked near the fireboss, along with a plane and a fuel carrier.

The Map level of the Phoenix environment, from which the graphics representations of the environment are generated, is constructed from Defense Mapping Agency data. Because it includes ground cover, elevation, moisture content, wind speed and direction, and natural boundaries, we have been able to construct a moderately realistic simulation of forest fires (but see Section 5 for a distinction between realism and accuracy). For example, real fires and our simulated fires spread more quickly in brush than in mature forest, are pushed in the direction of the wind and uphill, burn dry fuel more readily, and so on. These conditions also determine the probability that the fire will jump fireline and

natural boundaries; and the intensity of the fire (which is coded by color in the simulation) The physical abilities of fire-fighting agents are also simulated accurately; for example, bulldozers move at a maximum speed of 40 kph in transit (on the back of a truck), 5 kph traveling cross-country, and 0.5 kph when cutting fireline.

Recently, we have implemented some realistic weather factors, specifically, lightning strikes which start fires with some frequency, and rain, which affects the moisture and thus the friability of fuels.

Fires are fought by removing one or more of the things that keep them burning: fuel, heat, and air. Cutting fireline removes fuel. Dropping water and flame retardant removes heat and air, respectively. In major forest fires, controlled backfires are set to burn areas in the path of wildfires and thus deny them fuel.

In the past, fire-fighting agents were inexhaustible, but recently we have started to model their consumption of resources. The following example of monitoring fuel levels and refueling conveys the flavor of problem solving within and among Phoenix agents.

Example: Bulldozers monitor their own fuel levels and notify the fireboss when their tank drops below a preset level. Upon reciept of the "I'm low on fuel" message, the fireboss marks the bulldozer with a status of "needs-refueling" and when the bulldozer becomes idle (i.e. completes the segment of fireline it is working on), the fireboss selects a refueling plan for that bulldozer. This involves allocating an available fuel-carrier, calculating a rendezvous point on a road near the bulldozer, telling the bulldozer where to go and who to look for, telling the fuel-carrier where to go, and waiting for an acknowledgement from the bulldozer that it has received fuel. The bulldozer and fuel-carrier then interact through the following process: the bulldozer notices the rendezvous, requests service, and waits for a service-complete acknowledgement from the fuel-carrier. The fuel-carrier arrives at the destination, waits for service requests, queues them up if necessary (not currently utilized), transfers fuel via a pump-effector, terminating when either the bulldozer is not present or leaves, the refueling tank goes dry, the bulldozer's tank is full, or the requested amount is pumped. The fuel-carrier then tells the bulldozer it has finished and the bulldozer in turn tells the fireboss that the refuleing task has been completed.

3.2 The Implementation of the Testbed

Underlying the Phoenix testbed is a discrete event simulator (DES) that creates the illusion of a continuous world, where natural processes and agents are acting in parallel, on serial hardware (currently a Texas Instruments Explorer II Color Lisp Machine, but see Section 3.5). In the simulation, fires burn continuously over time and agents act in concert to control it. Some of these actions are physical, as

in digging fireline and cutting trees. In parallel to these physical actions, agents perceive, move, react to perceived stimuli, and think about what action(s) to execute next.

The DES manages two types of time: cpu time and simulation time. CPU time refers to the length of time that processes run on a processor. Simulation time refers to the ``time of day" in the simulated environment. The illusion of continuous, parallel activity on a serial machine is maintained by segregating each process and agent activity into a separate task and executing them in small, discrete time quanta, ensuring that no task ever gets too far ahead or behind the others. The default setting of the synchronization quantum is five simulation-time minutes, so all tasks are kept synchronized to within five simulated minutes of one another.

The quantum can be increased, which improves the cpu utilization of tasks and makes the simulator run faster, but this increases the simulation-time disparity between tasks, magnifying coordination problems such as communication and knowing the exact state of the world at a particular time. Conversely, decreasing the quantum reduces how ``out of synch" processes can be, but increases the running time of the simulation.

Within the predefined time quantum, all simulated parallel processes begin or end at roughly the same simulation time. Types of tasks differ in how they are ``charged for" cpu time and simulation time. Sensory tasks run for very short intervals of simulation time, after which they are rescheduled; this gives them a high sampling rate compared to the rate at which the world is changing. Effector tasks may use very little simulation time, or the full synchronization quantum. Fire tasks always run for the full synchronization quantum.

All these tasks are allotted as much cpu time as they need by the DES; there is no constant proportionality between the simulation time and the cpu time they require. To see why, note that fires are implemented as cellular automata, so that the cpu time required to calculate the spread of the fire depends on the size of the fire. It may take only a fraction of a second of cpu time to calculate five simulation-time minutes of burning for a small fire, but several cpu seconds to calculate the five simulation-time minutes for several large fires. Similarly, the amount of cpu time required to calculate a few simulation-time seconds of sensor processing depends on the type of sensor being simulated, so there is no constant proportionality between simulated sensor time and cpu time.

In contrast, there is a constant proportionality between the cpu time allocated to cognitive tasks and simulation time. This is because we want to "charge" agents at a fixed rate for thinking. Because cognition and other processes, such as the fire, are simulated parallel processes, they are always allocated the same amount of

simulation time. So when both have run for their allocated times

$$\text{elapsed-simulation-time(cognition)} = \text{elapsed-simulation-time(fire)}$$

as measured by the simulation time clock. Although these processes could take arbitrary amounts of cpu time, it is advantageous to impose a strict relationship between cpu time and the simulation time of the planner and the fire. Thus,

$$\text{elapsed-simulation-time(cognition)} = k * \text{cpu-time(cognition)}$$

and, from the previous expression,

$$\text{elapsed-simulation-time(fire)} = k * \text{cpu-time(cognition)}$$

The advantage of this proportionality is that we now have a way to exert time pressure on cognition. The *real-time knob* is the device that exerts pressure, simply by increasing k. Clearly, we can change k without changing the amount of cpu time allocated to cognition, and when this happens, the net effect is to increase the amount of simulation time allocated to the fire. Because of the strict proportionality between simulation time and cpu time for cognition, the indirect effect of increasing k is to *reduce the amount of simulation time allocated to cognition, relative to the simulation time allocated to the fire*. That is, to increase time pressure on cognition. Currently $k = 300$, which means that one second of cpu time for cognition is matched by five minutes of simulation time for the fire. If we increase k to 600, then the fire is allowed to burn for 10 minutes for every cpu second of cognition time.

Cognitive tasks are allotted a full synchronization quantum each time they run. At times there are not enough cognitive activities to fill a quantum, in which case the task ends and waits to be rescheduled. Some cognitive activities take longer than a full quantum, in which case their internal state is saved between quantum steps.

Example: Imagine it is now 12:00:00 in the simulated world, and an agent is about to begin planning. After one cpu second, simulation time for the agent is 12:05:00. The fire is thus "owed" five minutes of simulation time. But before it runs, the DES runs all sensor, effector, and reflex tasks. After that, it may take 7 cpu seconds to calculate the effects of five minutes of fire. Moreover, simulation time is still 12:05:00, because the agent and the fire are simulated parallel processes. So after roughly eight cpu seconds (one for the planner, negligible time for sensors, effectors, and reflexes, and seven for the fire), we have simulated five minutes of planning and five minutes of fire, and both processes are paused at 12:05:00.

3.3 Instrumentation of the Testbed

The Phoenix testbed is designed to support experiments with a variety of IRTPS architectures---not only our Phoenix agent architecture. Currently it has been instrumented to some extent, and much more instrumentation is planned. In this subsection we describe three levels of instrumentation suggested by Nort Fowler. Low level metrics are largely hardware-dependent estimates of how the software system is utilizing the hardware. Middle level metrics give us a fine-grained picture of how a specific architecture behaves over time; for example, we can measure the communication overhead among agents, the time required to respond to significant changes in the environment, the amount of time spent in error recovery, the ability of scheduling algorithms to meet deadlines, and so on.are specific to the agent architecture. High level metrics are domain specific. They record features of the environment that are affected by the agents, such as acreage burned by fires, and consumption of resources.

Any researcher who implements a new agent architecture in the Phoenix environment will have to define middle-level metrics, because these are architecture specific, but probably won't have to define high and low level metrics. For example, Phoenix agents maintain a timeline of pending actions, and we need to know the average latency between posting and executing an action. An agent architecture implemented as a blackboard system may instead look at the scheduling of tasks on an agenda. Most of our middle level metrics are for the Phoenix agent architecture, not for unanticipated other architectures.

Currently, the following instrumentation is complete or nearly so:

Low Level Instrumentation

Run time , Cpu time , Disk wait time, Time since last run , Idle time, Utilization, Overall. Each of these is graphed against time.

We also provide an interface to the Explorer performance metering tools which work at the function call level and provide for each function the: Number of calls , Average run time, Total run time, Real time, Memory allocation, Page faults

Middle Level Instrumentation

Statistics on cpu utilization by the cognitive component of each agent to see the actual profile of real-time response, graphed against time

The latency between when actions on the timeline become available for execution and when they are executed.

Metrics on sensor and effector usage

Metrics on reflexes

The goal of these metrics is to compare the utilization of cognitive and other resources in different scenarios (see Section 3.4). Each scenario will contain important events (e.g., a fire is detected). We will graph these metrics against time and annotate the graphs at the points that the significant events occured, so we can see how the agent architecture responded.

High Level Instrumentation.

Fire destruction is currently measured by amount and type of forest, houses, and agents burned.

Resource allocation is currently measured by amount and type of agents employed to fight the fire, gasoline consumed, fireline cut, distance traveled, and time required to contain the fire.

3.4 Baseline Scenarios

One advantage of studying IRPTS in a simulated environment is the ability to run the same environmental scenario again and again while modifying aspects of the agent architecture (see Section 5). We have recently implemented the ability to define *scripts*, which include the type and number of available agents, and guide the environment through a series of changes in conditions such as windspeed and other weather conditions. We also have the ability to introduce stochastic factors into scripts, such as lightning strikes. Besides scripts, we will soon be able to provide baseline statistics on events such as rates of spread of fires in different conditions.

Scripts play an important role in evaluating IRTPS systems, and comparing IRTPS architectures. By design, scripts can force an agent to confront virtually any IRTPS issue. Here is a simple script that raises the six IRTPS issues that were discussed in the original IRTPS Initiative:

Materiel:

One fireboss to coordinate the activities of three bulldozers

Bulldozers can move 6.5 kph in softwood, 56 kph on road, .5 kph while building line

One watchtower at location approx 42500x37500

Environment:

A fire of radius 700 km starting at coordinates approx 45000x46250,

Starting wind speed and direction 3 kph from the south

Environmental Changes:

At time 2 hr, wind changes to 10 kph from the NW, threatening buildings---the base and lodge

Since it involves a burning fire, the script requires agents to produce relevant output in a timely fashion. The particular scenario includes an environmental

change (asynchronous with the reasoning system) that invalidates previous input, necessitating the detection of a new threat to higher priority areas and a redirection of ongoing reasoning in order to protect them. To handle this scenario, a system must reason efficiently and effectively about temporal processes, namely the expected progress of a fire under particular environmental conditions and the abilities of a limited number of agents to take steps, over time, toward putting out the fire.

We must add that this script confounds the current implementation of Phoenix agents. They are currently incapable of redirecting their efforts to save the base and lodge.

3.5 Portability.

The Phoenix system runs on color and monochrome Texas Instruments Explorers and MicroExplorers. We can package everything together (including support-code for the frame system, grapher, EKSL utilities, etc.) as needed. We are making progress on the documentation.

The entire Phoenix system is designed to be modular, so fellow researchers can use the components they want. The smallest self-contained module, and the most basic, is the Phoenix environment. This includes the DES, the Map layer, and the user interface. We are confident that a researcher could take this code and build his or her own agents to interact with it. However, we have not done this in our own lab, so we cannot be sure.

Above the environment are three additional levels of software---the Phoenix basic agent architecture, our own Phoenix agents, and the organizational structure that holds among our Phoenix agents. The basic agent architecture is a skeleton with hooks for sensors and effectors, reflexes, and a cognitive component (see Section 4). Some weeks ago the entire lab went through the exercise of defining a new type of Phoenix agent (an airplane) given only the basic agent architecture, and we are confident other researchers can do the same. There are really two aspects to defining a new agent. One is mostly bookkeeping: We define frames for the agent that describe its physical abilities, so that the DES knows how it behaves over time. We also define frames that add instances of the new agent to a script. This is the easy part. The hard part is defining the cognitive abilities of the agent. In terms of the basic Phoenix agent architecture, this means defining plans, execution methods, a cognitive scheduler, and other architectural components discussed in Section 5.

Of course, the Phoenix environment does not and should not care about the cognitive component of a new agent, other than to schedule its processes to guarantee the illusion of simultaneity with other agents and environmental

processes. Thus, it is relatively easy to tell the Phoenix environment about the physical abilities of new agents, as in the examples above, and unnecessary to tell the environment how the cognitive components of the agents work. We hope this will make it easy for researchers to use the Phoenix environment, or the environment and the basic agent architecture, to design and test their own agents.

4. Toward a Solution: The Phoenix Agent Architecture

A uniform agent architecture is shared by all agents. This architecture is the structure of the agent, the "hardware" that dictates the fundamental faculties and limitations of the agent. The structure endows and bounds acuity, speed of response, and breadth of action. The structure constrains what an agent can do, but not what it does. Specific methods control what the agent does. Control methods determine what to do and how to do it. This dichotomy between structure and control is reflected in this subsection and the one following it. Section 4.1 describes the agent architecture and Section 4.2 focuses on techniques for real-time problem solving. For a more detailed description of these components, see [Cohen, 1989])

4.1 Phoenix Agent Architecture

The agent architecture has four components. *Sensors* perceive the world. Each agent has a set of sensors, such as fire-location (are any cells within my radius-of-view on fire?) and road-edge (in what direction does the road continue?). *Effectors* perform physical acts such as moving or digging fireline. *Reflexes* are simple stimulus-response actions, triggered when the agent is required to act faster than the time-scale for the cognitive component. An example is the reflex of a bulldozer to stop if it is moving into the fire. The *cognitive* component performs mental tasks such as planning, monitoring actions, evaluating perceptions, and communicating with other agents. Although every agent has these components, each component can be endowed with a range of capabilities.

Sensors get input from the world (fire simulation and map structures). Their output goes to state memory in the cognitive component, and also to the reflexive component (triggering instant responses in the form of short programs to the effectors). For example, a bulldozer sensor that detects fire within its radius-of-view updates state memory automatically. If the detected fire is in the path of the bulldozer, the emergency-stop reflex is also triggered. Effectors are programmed by the cognitive component and by reflexes. Their output performs actions in the world. In the preceding example, the emergency-stop reflex would program the movement-effector of the bulldozer to stop. If the fire were not too close, the cognitive component might then step in and program the movement effector to start moving parallel to the fire. If the cognitive component also programmed the blade effector to put the blade in the down position, the bulldozer would not only

maintain a safe distance from the fire, but it would also build fireline as it moved. Sensors and effectors are first-class objects whose interactions with other components and the world are implemented in Lisp code. Reflexes, as mentioned, are triggered by sensory input, which causes them to program effectors to react to the triggering sensation. They are implemented in production-rule fashion, with triggering sensations as their antecedent clauses and effector programs as their consequents. Because they respond directly to the environment and so must keep up with it, sensors, effectors, and reflexes operate at the same time scale as the simulation environment and are synchronized as closely as possible within the discrete event simulator.

The cognitive component receives input from sensors and sends programs to the effectors to interact with the world. It is responsible for data integration, agent coordination, and resource management, in other words, most problem solving activity. This component operates in larger time slices than the others, thus reducing the overhead of context switching, but increasing the possibility of reasoning with outdated information.

The Phoenix cognitive component directs its own actions by adding prospective actions onto the timeline, a structure for reasoning about the computational demands on the agent, then selecting and executing these actions one at a time. Actions may be added in response to a change in environmental conditions (e.g., a new fire) or as part of the computation of other actions (e.g., through plan expansion). Every action that the cognitive component accomplishes is represented on the timeline with its temporal relations to other actions and resource requirements (e.g., processing time and necessary data). The cognitive scheduler decides which action to execute next from the timeline and how much time is available for its execution.

Actions may perform calculations, search for plans to address particular environmental conditions, expand plans into action sequences, assign variable values, process sensory information, initiate communication with other agents, or issue commands to sensors and effectors. These actions are represented in skeletal form in the plan library. Actions are described by what environmental conditions they are appropriate for, what they do, how they do it (the Lisp code for their execution, called the execution methods), and what resources and data, environmental and computational, they require. A plan is a special type of an action. It includes a network of actions related by their data references and temporal constraints.

Planning is accomplished by adding an action to the timeline to search for a plan to address some conditions. When the search action is executed, it selects an action or plan appropriate for the conditions and places it on the timeline. If this new action is a plan, then when it is executed it expands into a plan by putting its sub-actions onto the timeline with their temporal inter-relationships. If it is an action, it instantiates the requisite variables, selects an execution method (there

may be several with differing resource requirements and expected quality of solution), and executes that method. We call this style of planning *skeletal refinement with lazy expansion.* Plans are represented as shells that describe what types of actions should be executed to achieve the plan but do not include the exact action or its variable values until it is executed. Delaying expansion allows the expanded plan to address more closely the actual state of the environment during execution.

This planning style is common to all agents in the Phoenix planner, though it is flexible enough so that agents with a variety of cognitive capabilities are possible. For example, the fireboss has far more sophisticated methods for gathering and integrating information than the bulldozer does. It can direct the actions of the bulldozers, while the bulldozers can only make requests of the fireboss. However, the fireboss, unlike the bulldozers, does know how to get out of the way of the fire because it does not work close to the fire.

Creating a different type of agent requires defining a cognitive component. One can optionally define a set of programmable sensors and effectors (of arbitrary complexity) and add a set of reflexes to handle situations that require instant response by the agent. To create sensors and effectors, the simulator must be told rates of action under varying environmental conditions, range of perceptions, and other physical capabilities. Creating reflexes involves describing the triggers, the expected output from sensors, and the response, the programming for the effectors. The default cognitive component consists of plans, which are networks of actions available to the agent and tailored to situations in the environment, and methods which describe how to execute the actions. Creating a new cognitive component with the same structure as that described here involves defining a new plan library.

Several design decisions in the Phoenix agent architecture have been made specifically to facilitate real-time control. One important decision is to incorporate both reflexive and cognitive abilities in agents, enabling agents to respond reflexively to events that occur quickly, while responding more deliberately to resource management and coordination problems on a longer time scale. The combination of a reflexive and cognitive component accounts for time scale mismatches inherent in an environment that requires micro actions and contemplative processing. Micro actions, such as following a road and keeping out of the immediate range of the fire, involve quick reflexes and little integration of data. Contemplative processing, such as route planning, involves long search times and integration of disparate data such as available roads, terrain conditions, and fire reports. This horizontal decomposition ensures that the agent can perform reflex actions to keep it from danger and maintain the status quo, while also performing more contemplative actions. This strategy for responding to disparate demands of the environment is advocated by Brooks, and Kaelbling; although in both cases, they chose more levels of decomposition for their domains. Our agent architecture, in effect, combines two different planning components:

one highly reactive, triggered by specific environmental stimuli and operating at very small time scale, and the other slower and more contemplative, integrating large amounts of data and concerned with resource management and coordination.

Another design feature that facilitates real-time control is the timeline and its single representation for all actions. Because prospective actions share a uniform representation on the timeline, all problem solving actions have access to the same memory structures and can be monitored and allocated resources using the same mechanisms. All problem solving tasks are subject to the same constraints with respect to resource allocation: how much time is required, what information gathering resources are required, and what data is necessary. This framework allows new cognitive capabilities to be integrated easily by defining their requirements within the action description language and relying on the timeline and its supportive scheduling mechanisms to temporally arbitrate their allocation.

Lazy skeletal expansion also facilitates real-time control. Plans are only partially elaborated before the agent acts. This deferred commitment exploits recent information about the state of the world to guide action selection and instantiation. Completely deferred commitment, such as in reactive planning, is probably not tenable when agents or actions must be coordinated or scarce resources managed. The integration of planning and acting in Phoenix is designed to be responsive to a complex dynamic world by postponing decisions on exactly what action to take, while also grounding potential actions in a framework (skeletal plans coordinated on the timeline) that accounts for data, temporal and resource interactions.

4.2 Real-Time Control in the Agent Architecture

How does a Phoenix agent respond to real-time pressure? One approach is to control processing requirements. This enhances the flexibility of actions and the sophistication of control decisions. Providing alternative execution methods for timeline entries ensures a range of choices that vary in their timeliness. Different scheduling strategies for managing the actions on the timeline provide greater responsiveness to real-time constraints. Another approach is an expectation-based monitoring technique that reduces the overhead of monitoring while providing early warning of plan failure. Earlier warning of plan failure affords the planner more time to adjust and more flexibility in possible responses. These approaches are discussed below.

4.2.1 Control of Processing Requirements

Processing requirements can be controlled in two ways: by controlling how much time is used by individual actions and by controlling the overall distribution of time across all actions. Approximate processing and anytime algorithms are methods

for controlling how much time is used by individual actions. In these methods, processing time is traded against quality or correctness of solution to satisfy time constraints that could not be managed under rigid processing demands. In Phoenix, these methods are alternative execution methods. Execution methods, as introduced in Section 4.1 are lisp code that performs the cognitive actions. Each cognitive action may may be executed by one of several execution methods, with differing time requirements and so differing solution expectations. The Phoenix planner delays the choice of an action's execution method until the cognitive scheduler selects the action for execution, thereby allowing the scheduler to select a method suited to existing time constraints. By postponing the ultimate commitment of cognitive resources until a choice must be made, those resources can be allocated judiciously.

Alternative execution methods are particularly useful in actions that incur potentially high computation costs with predictable results, such as path planning. Phoenix uses an A* algorithm to calculate paths for bulldozers. It searches the two-dimensional map representation of the world for the shortest travel time path between two points. It expands the current best path incrementally, searching each unobstructed neighboring cell for the best next step. The algorithm is parameterized to work at multiple levels of resolution, so that search steps could range from 128 meters up to 8 kilometers. A small search step, 128 meters, yields the shortest path, requiring the least travel time for the bulldozer. However, this resolution requires the most computation (i.e., cognitive resources). The largest search step, 8 kilometers, typically yields a longer path, which requires more travel time, but can be calculated quickly, consuming less computation time. At times it even fails to find a solution, since there are bottlenecks in the map that don't appear at large search steps. Each of these resolutions constitutes a different execution method for calculating a path, alternative methods which trade-off cognitive-time for quality of solution.

The cognitive scheduler controls the overall distribution of cognitive processing time across all actions. At each time step, it selects the next action from the timeline to execute, chooses an execution method for the action, and executes it. Thus, the scheduler is key to controlling the responsiveness of the cognitive component to real-time constraints. The current version of the scheduler for Phoenix is rudimentary and considers only a short horizon for scheduling decisions. It selects the next action for execution based on timeline ordering, action priority and the amount of time an action has been waiting for execution. A more sophisticated scheduler is being designed now.

4.2.2 Sophisticated Monitoring Through Envelopes

Just as we can explicitly represent the movements of an agent through its physical environment, so can we represent its movement through spaces bounded by failure or other important events. These spaces are called *envelopes*. Typically, one dimension of an envelope is time, and the others are measures of progress.

For example, imagine you have one hour to reach a point five miles away, and your maximum speed is 5 mph. If your speed drops below its maximum, for even a moment, you fail. As long as you maintain your maximum speed, you are*within your envelope*. The instant your speed drops below 5 mph, you *lose* or*violate* your envelope. This envelope is *narrow*, because it will not accomodate a range of behavior: any deviation from 5 mph is intolerable. Most problems have wider envelopes. Indeed, real time systems should be designed to ensure that narrow envelopes are the exception, not the rule.

The following problem illustrates a wider envelope. A bulldozer has one hour to travel five miles, as before, but its maximum speed is 10 mph. It starts slowly (perhaps the terrain is worse than expected). After 40 minutes it has travelled just two miles. It can still achieve its goal, but only by travelling at nearly maximum speed.

Clearly, if the agent waits 40 minutes to assess its progress, it has waited too long, because an heroic effort will be required to achieve its goal. In Phoenix, agents check their envelopes at regular intervals, hoping to catch problems before they get out of hand. One near-term research goal is to develop a theory of envelopes that will tell us when and how often they should be checked.

Agents check *failure* envelopes, which tell them whether they will absolutely fail to achieve their goals, and*warning* envelopes, which tell them that they are in jeopardy of failure. Typically, there is just one failure envelope but many possible warning envelopes. To continue the previous example, the bulldozer would violate a warning envelope if its average speed drops below 5 mph, because this is the speed it must maintain to achieve its goal. Violating this envelope says, "You can still achieve your goal, but only by doing better than you have up to this point." These concepts are illustrated in Figure 3 . The failure envelope is a line from "30 minutes" to "five miles," since the bulldozer can achieve its goal as long as it has at least 30 minutes to travel five miles. The average speed warning envelope is a line from the origin to the goal, but the bulldozer violated that envelope immediately by travelling at an average speed of 3 mph. In fact, it moved perilously close to its failure envelope. The box in the upper right of Figure 3 illustrates that the agent can construct another envelope from any point in its progress. In this example, the new envelope is extremely narrow.

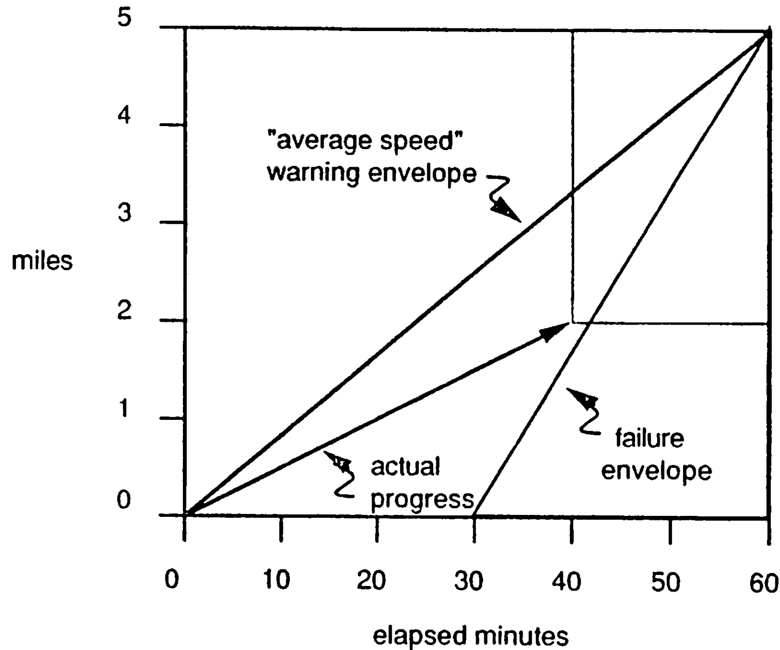


Figure 3 . Depicting actual and projected progress with respect to envelopes

Agent Envelopes and Plan Envelopes. We distinguish between the envelopes of individual agents and those of multi-agent plans. In Phoenix, plan envelopes are maintained by the fireboss agent, who coordinates several subordinate bulldozers. Because the environment changes, global plans may be put in jeopardy even if agents are making progress that, from their local perspective, is well within their envelopes. Figure 4 illustrates plan envelopes as they are currently implemented in Phoenix: The leftmost illustration represents the current state of the fire, its projected boundaries after one and two hours, and the firelines that three bulldozers are expected to cut. By projecting where the fire will be, then adding some slack time, the fireboss anticipates that the last of these lines will be cut an hour before the fire reaches it. On the right of Figure 4, we see the actual progress of the fire: After one hour, it has grown less than expected, so the amount of slack time grows (bottom of Figure 4) and the plan stays well within its one hour slack time envelope. But during the next hour, the fire grows more rapidly than expected; so rapidly, in fact, that the slack time envelope is violated. Sometime during this interval, the fireboss will check the plan envelope and discover that it is violated. It then replans and typically sends one or more additional bulldozers to help out.

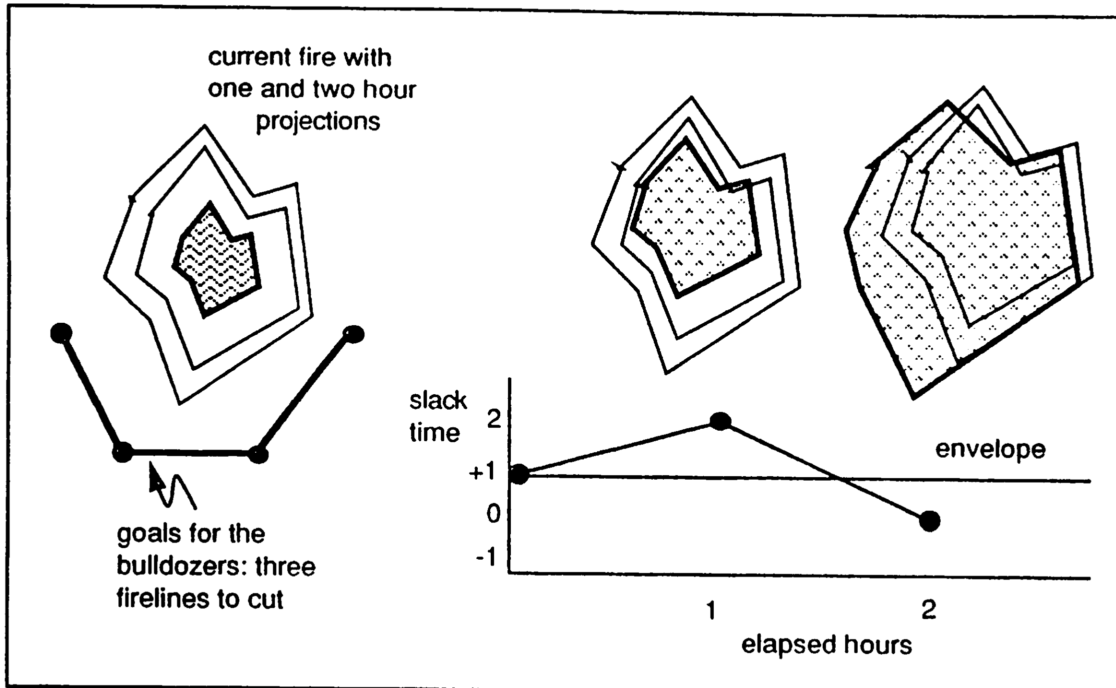


Figure 4. A plan envelope for maintaining slack time.

The Utility of Envelopes. A planner can represent the progress of its plan by transitions within the plan's envelopes. Progress, failures and potential failures are clearly seen from one's position with respect to envelopes, whereas this information is not always apparent from one's position in the environment.

Envelopes function as early warning devices in two ways. First, explicit warning envelopes alert the planner to developing problems. Second, failure envelopes can tell an agent it has failed long before its allocated time has elapsed. In Figure 3, for example, the agent knows it has failed as soon as it crosses the envelope. A third kind of early warning has yet to be implemented: Just as a planner can project the course of events in its environment, so it can project its progress within its envelope and, particularly, when an envelope might be violated. A simple projection method is extrapolation. For example, if we checked the envelope in Figure 4 after 75 minutes we would see a "downward" trend. By linear extrapolation we could estimate when the envelope would be violated. Of course, the downward trend may reverse, or level out. But sometimes it will be worthwhile to have the projected time of envelope violation despite its uncertainty.

Envelopes integrate agents at different levels of a command hierarchy: A fireboss agent formulates a goal and a corresponding envelope, and gives them to a subordinate bulldozer agent with the following instructions: "Here is the goal I want you to achieve. I don't care how you do it, and I don't want to hear from you unless you achieve the goal or violate the envelope." The bulldozer then works independently, not monitored by the fireboss. It figures out where to go, how to avoid obstacles, and how to keep clear of the fire, until its goal is achieved or its envelope violated. Meanwhile, the fireboss is free to think about other agents, other

goals, or to replan if necessary. Envelopes grant subordinate agents a kind of autonomy, and grant superordinate agents the opportunity to ignore their subordinates until envelopes are violated.

We have yet to develop cognitive scheduling mechanisms to take full advantage of envelopes. The design of these mechanisms is motivated by the following questions: How often should envelopes be checked? Should we adopt a fixed interval or a dynamic one, and if the latter, what execution methods will determine when to check next? When should agents project envelope violations and how should they use the projections? Given that checking a plan envelope, or projecting progress with respect to it, may involve collecting and integrating information from the environment and all the participating agents, the cognitive overhead of of these activities can be considerable and must be carefully scheduled.

5. Methodological Issues.

Our overriding research goal is to develop a sound basis for the design of AI agents. AI is a kind of design. We don't design graphics, or VLSI circuits, or mechanical devices: we design intelligent agents. The agents are evaluated by how they behave. Their behavior is determined by their environments and their architectures. Once we adopt this view, we see immediately that we do not know enough about the relationships between agent architectures, behaviors, and environments (the corners of the behavioral ecology triangle in Fig. 1) to design intelligent agents in a principled way. For example, we cannot even precisely define the characteristics of environments (Sec. 2.2), much less behaviors. And we cannot answer the question, "How would the behavior of this AI program, in this environment, change if you change its architecture this way: ... ?" But until we can answer this question, AI system design will remain ad hoc.

In fact, design is one of six research activities implied by the behavioral ecology model. Here is the complete list:

Prediction: How will behavior be affected by changing the architecture of the agent or its environment? For example, how will behavior be affected by changing the size of short-term memory, or by changing the mechanism by which long term memory is accessed? How will behavior be affected if the environment "speeds up," so that events that took N seconds now take N/2 seconds?

Explanation: Why does a particular behavior (presumably unexpected) emerge from the interaction between an agent and its environment? For example, why does an agent that combines long-term, goal-directed behavior with short-term reactive behavior sometimes exhibit something like an approach-avoidance conflict---dashing first toward a goal, then away from it, but getting nowhere in the long run?

Design. What architectures will produce a particular set of behaviors in particular environments? For example, what architectures will enable an agent to respond to events in the environment that occur at very different time scales?

Environment analysis: What aspects of the environment most constrain agent design? What is our model of the environment?

Generalization: Whenever we predict the behavior of one agent in one environment, we should ideally be predicting similar behaviors for agents with related architectures in related environments. In other words, our theories should generalize over architectures, environmental conditions, tasks, and behaviors.

Functional relationships: What knowledge do we need to answer questions in these classes? What are the functional relationships between the architecture of an agent and its behavior?

Both the behavioral ecology model and the S/E model of Rosenschein, Hayes-Roth, and Erman (see their paper in this volume) explicitly acknowledge the relationships between architecture the environment, and behavior. Rosenschein et al. denote the architecture and environment S and E, respectively; and characterize behavior as a sequence of state changes called a run. Furthermore, Rosenschein et al. seem to implicitly subsume, in what they call measurement and evaluation, some of the research activities above. But because neither the S/E model nor the behavioral ecology model make predictions, it is premature to compare them except to note some apparent differences in emphasis.

Rosenschein et al. view the "S/E boundary" as flexible, so that sometimes the environment can be made responsible for an activity that, in other circumstances, we might require of the agent. For example, with the general vision problem currently unsolved, we might construct an environment that "preprocesses" sensory data for the agent, thus moving the S/E boundary inward, toward the agent, bypassing the need for sophisticated sensors. This example suggests a small apparent difference between the S/E model and the behavioral ecology model: whereas the S/E model seems to assume a simulated environment, the behavioral ecology model does not. Although the Phoenix project uses a simulated environment, our principal research tasks (prediction, design, explanation, etc.) do not *presume* a simulated environment. It isn't clear yet whether the principal research tasks of Rosenschein et al. presume a simulated environment.

This raises the methodological question of whether one should use simulations at all. Some researchers insist that the subtleties of real environments are "lost in translation" to simulated environments. This is to some extent a straw man, because we don't view simulations as accurate representations of the real world. (In fact, we recently got into trouble by claiming that the Phoenix environment is an accurate simulation of forest fires.[4]) But it is important to distinguish *realism*

---

[4]See Letters to the Editor, AI Magazine, Vol. 10 No. 4.

and *accuracy*. Realism is necessary for our research; accuracy is not. Here are some examples of the distinction: In a realistic simulation, processes become uncontrollable after a period of time; in an accurate simulation, the period of time is the same as it is in the real world. In a realistic simulation, agents have limited fields of view; in an accurate simulation, agents' fields of view are the same as they are in the real world. In a realistic simulation, the probabilities of environmental events such as wind shifts are summarized by statistical distributions; in an accurate simulation, the distributions are compiled from real-world data. When possible, we use accurate data; for example, in Phoenix we use Defense Mapping Agency data of elevation, ground cover, and so on, and the fire dynamics are derived from U.S. Forest Service manuals (NWCG Fireline Handbook, 1985). But the goal of our research is not to accurately simulate forest fires in Yellowstone National Park. It is to understand the design requirements of agents in realistic environments—environments in which processes get out of hand, resources are limited, time passes, and information is sometimes noisy and limited.

With this in mind, we see that simulations have several advantages:

Control. Simulators are highly parameterized, so we can experiment with many environments. For example, we can change the rate at which wind direction shifts, or speed up the rate at which fire burns, to test the robustness of real-time planning mechanisms. Most important, from the standpoint of our work on real-time planning, is the fact that we can manipulate the amount of time an agent is allowed to think, relative to the rate at which the environment changes, thus exerting (or decreasing) the time pressure on the agent.

Repeatability. We can guarantee identical initial conditions from one "run" to the next; we can "play back" some histories of environmental conditions exactly, while selectively changing others.

Replication. Simulators are portable, and so enable replications and extensions of experiments at different laboratories. They enable direct comparisons of results, which would otherwise depend on uncertain parallels between the environments in which the results were collected.

Variety. Simulators allow us to create environments that don't occur naturally, or that aren't accessible or observable.

Interfaces. We can construct interfaces to the simulator that allow us to defer questions we'd have to address if our agents interacted with the physical world., such as the vision problem. We can also construct interfaces to show things that aren't easily observed in the physical world; for example, we can show the different views that agents have of the fire, their radius of view, their destinations, the paths they are trying to follow, and so on. The Phoenix environment graphics make it easy to see what agents are doing and why.

Let us return now to the comparison of the S/E and behavioral ecology models. We noted that the former model represents behavior as "runs," sequences of state transitions (or as measures over runs), whereas the behavioral ecology model is

inspecific about how to represent behavior. On the other hand, the behavioral ecology model is quite specific about the causal relationships that hold among the environment, the agent architecture, and the agent's behavior. The behavioral ecology model comes from biology; it regards the architecture as analogous to the genotype and the behavior as analogous to the phenotype. And it assumes that selection operates on the phenotype. Thus, there is no direct causal link between an agent's environment and its architecture; rather, the environment ensures that behaviors are differentially rewarded, so the architecture must be modified to produce "good" behaviors. This, then, is what we mean by a good architecture— one that produces behaviors that are good in a particular environment.

It's important to know whether such behaviors can be generated by design, that is, by intentional modifications to the architecture, or whether they must evolve by search. Advocates of emergent behavior often take the latter view. They say that one cannot generate the phenotype from the genotype; one cannot predict how a moderately complex architecture will behave. This has important practical and methodological implications for IRTPS. Do we build IRTPS systems "top down," by assembling components that are predicted to behave in particular ways, and damn the emergent behaviors? Or do we build them "bottom up," by assembling components incrementally and empirically, waiting for desired (and undesirable) behaviors to emerge? In fact, we mix the approaches in proportions determined by the degree to which behaviors can be predicted from architectures (or components of architectures). Moreover, this degree of predictability is determined in part by the desired precision or scale of the predictions. If you want to know the precise number of cpu seconds that a process will run, you are probably out of luck. But if you want to know the upper bound runtime, it may be possible. You probably can't know the exact location of a fire ten minutes from now, but you can certainly draw a circle that has a high probability of circumscribing the fire. Thus, the question of whether behaviors can be generated by design depends intimately on how precisely we want to specify and predict the behaviors.

This brings us to a final methodological issue: evaluation. Let us first ask, What is being evaluated? Whether an architecture exhibits "timely" behavior? or exhibits a good tradeoff among several desired behaviors? Whether the behaviors are exhibited in a sufficiently wide range of environments? Whether we can predict when the behaviors will and will not occur? Whether we understand the functional relationships between architecture and behavior well enough to design an agent that will exhibit desired behaviors in a new environment? All of these should be evaluated. More pointedly, evaluation *cannot* stop with the demonstration that a system "works," however sophisticated the demonstration! We must take at least two more steps: We must attempt to show *why* the solution works (or doesn't work). This is uncommon, but essential if we are to make progress as an engineering field. The third step is to show why any solution with such-and-such abstract characteristics must work (or not work). This requires models of the behaviors and environments under study.

5.1 An Example of Design for IRTPS

We will briefly illustrate the previous points, and the terminology in Section 2.2, with the example of the design of Phoenix's cognitive scheduler. It's current cognitive scheduler is very weak. We are designing another one that achieves many of the behavioral goals of IRTPS. In the terms of Section 2.2, this seems to imply that we should list the problems and inherent problems, describe the relevent ECs and ACs, and after analyzing how the problems arise out of the interactions between ECs and ACs, we propose solutions and solution realizations. In fact, this seems to be an idealization. Instead we start with an *informal* description of some problems, and then hunt around for ECs and ACs that we believe account for the problems. The result is a formal description of the problems in terms of ECs and ACs. Then we generate solutions and solution realizations.

Here is an example of the first steps.

Informal description: The plan selection mechanism may take too long to find a plan. As a result, the fire may burn too much area, or may become uncontrollable. (Note that this is intentionally vague, to show how we formalize the problem description in terms of ECs and ACs.)

Environment characteristics: What is going on in the environment that could contribute to the problem, as informally described above? Let's concentrate on one thing, the spread of the fire. We want to model this in a way that allows us to firm up the informal problem description. Suppose we model the spread of the fire as an exponential process analogous to compound interest and population growth. Then, the perimeter of a fire after t time units is:

$$p = p_i(1 + r)^t$$

Here, p is the perimeter of the fire, p_i is the initial perimeter of the fire, r is the percentage increase in the fire perimeter every time unit, and t is the number of time units that have elapsed.

Architecture characteristics. For now, we will list just two characteristics: It takes a period of time, d, to generate a plan and get the bulldozers to the fire to begin implementing the plan. And once at the fireline, bulldozers dig at a constant rate.These are both oversimplifications, but useful, as we shall see.

Now we can say more formally what the problem is:

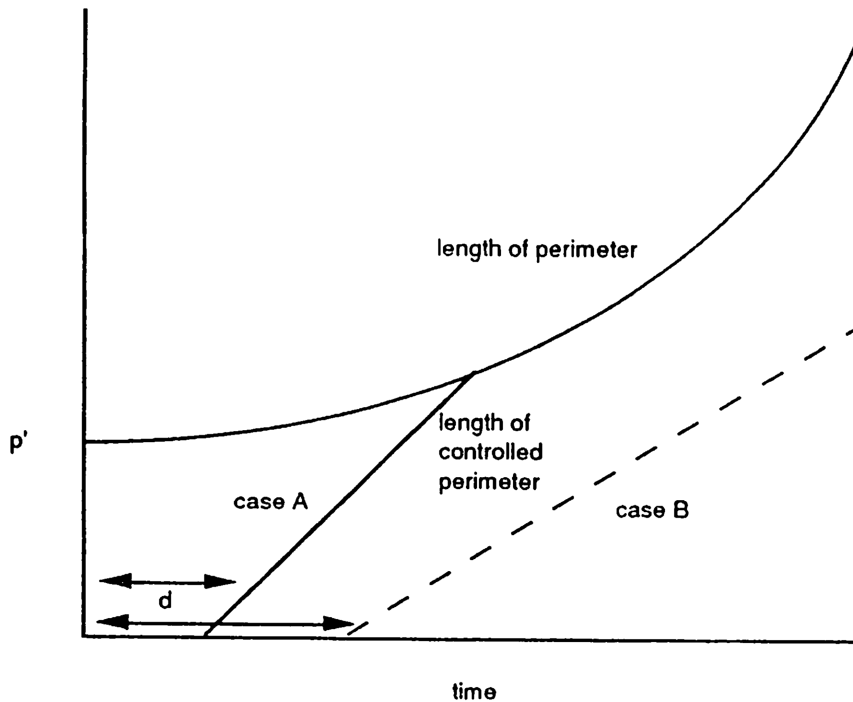


Figure 5

The curved line represents p, the length of the perimeter, as a function of time. It begins not at the origin, but at a point that represents the initial perimeter of the fire (e.g., its size when detected). We assume for simplicity that the steepness of the curve is described by one parameter, r, which captures factors such as wind speed and fuel type. Obviously, a more complex model could be generated if needed. After some delay, d, a plan is detected and some bulldozers are dispatched and then arrive at the fire. They begin cutting fireline at a constant rate, so the length of the controlled perimeter increases at a linear rate determined by the number of bulldozers. We show two possibilities, case A and case B.. In case A (solid line) the bulldozers arrive at the fire sooner, and in greater numbers than in case B (dashed line). In fact, in case A the fire is controlled, whereas in case B it is not. We know this because the line for case A intersects the line for the perimeter, which means that at some point, the length of the controlled perimeter equals the length of the fire perimeter; or, all the fire perimeter is controlled. In case B, this doesn't happen.

Before we can rephrase the informal problem description more precisely, we need to know what affects the parameters represented in the diagram above. This will tell us what we control as designers, what the system itself controls as an autonomous agent, and what the environment alone controls.

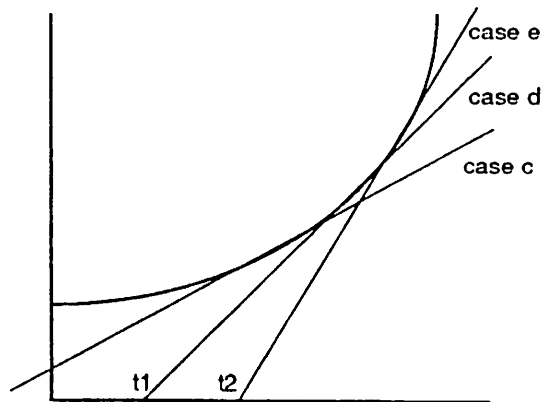
p' — This could be reduced if the fire was sighted earlier. The lower limit on the speed with which the fire is sighted is an AC that we control. It depends on things like how big a fire must be before it is noticed, how often the watchtowers look, how long it takes them to report their findings, how long it takes the fireboss to notice, etc. Most of these ACs have lower limits that we control, and actual values that the agent controls.

r — This parameter, which determines the steepness with which the perimeter increases, is an EC.

d — as with p',we control the lower limit on d, and the actual value is controlled by the agent.

slopes of "controlled perimeter" lines — this has an upper limit that we control (by controlling the number of available bulldozers) and an actual value that the agent controls, by controlling the number of bulldozers committed to the fire. .

At this point, we can begin to give formal descriptions to problems. For example, what is a *deadline*? In general, a deadline is a point at which the value of problem solving changes, usually downward. Consider a hard deadline for the plan selection process. In the previous diagram, this is represented as an upper limit on d. Consider three cases, denoted c, d, and e in the following diagram:



In case c, the slope of the "controlled perimeter" line is shallow because, say, it corresponds to a plan that involves only two bulldozers. Moreover, the deadline for the institution of the plan *has already passed*. You can see this by shifting the line for case c to be tangent to the "perimeter" line. By the same operation you can see that, for case d to succeed, the bulldozers (more of them than in case c, hence the steeper slope) must be at work before t1; and for case e to work, they must be busy by t2. Any delay longer than t1 or t2 shifts the respective lines to the right, ensuring that they will never intersect the perimeter line and the fire will never be controlled.

Now we can be more precise about the problem: For given values of r and p' (assuming the fire has just been sighted), find a plan that is expected to contain the fire and that can be instituted before its deadline.

Note that the original problem, a failure to get plans ready in time, has been formalized in the context of an agent model and an environment model. Moreover, a common IRTPS term has been defined in these contexts. One might argue that, in the process, we have taken a nice, general term like "deadline" and replaced it with something that is so specific to Phoenix as to be unusable. We believe we have done exactly the opposite. Not only have we made a vague term precise, but we have also identified a very general functional relationship or "rule" associated with the term: Imagine that the perimeter of the fire grows linearly, not exponentially. Then the notion of deadline illustrated in Figure 5 would not exist. If a process F grows linearly, and another linearly-growing process B is trying to control it, then a comparison of the growth rates of F and B will tell us whether B will succeed, and when it will succeed (assuming the growth rates don't change). If B grows faster than F, then it will control F eventually. The only effect of delaying the onset of B is to delay the control of F. On the other hand, if F grows superlinearly, as in Figure 5, and B grows linearly, then a delay does not merely delay the event in which B controls F, it may make that event impossible (as shown by the dashed line in Figure 5). we believe this is a very general phenomenon, and thus a very general interpretation of "deadline": A deadline is the point at which a linear process becomes incapable of catching—at any time in the future—a superlinear one. Obviously this can be generalized to functions of other orders—a sublinear process trying to catch a linear one, and so on.

As we evaluate the Phoenix project, we will certainly ask whether it plans in a timely way, whether it meets deadlines, balances cognitive load, exhibits graceful degradation, and so on. But the most telling evaluation will be whether we have been able to engage in the specialization-generalization process illustrated above: Whether we gave terms such as "deadline" precise interpretations in terms of Phoenix ECs and ACs and then generalized them again, as we did when we said a deadline is a point at which one process becomes incapable of catching another.

Phoenix. Cohen, Howe and Hart. November 1989.

References:

Cohen, P. R., Greenberg, M. L., Hart, D.M., Howe, A. E. 1989.Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments. AI Magazine, Vol. 10, No. 3. pp. 32-48. Fall, 1989.

Cohen, P.R. and Howe, A. E. 1988. How Evaluation Guides AI Research. AI Magazine 9(4) 35—43.

NWCG Fireline Handbook 410-1, National Wildfire Coordinating Group, Boise, Idaho, Nov 1985.

Rosenschein, S. J., Hayes-Roth, B., and Erman, L. D. Notes on Methodologies for Evaluating IRTPS Systems. In this volume.