Two Ways to Act

Scott D. Anderson, David M. Hart, Paul R. Cohen
Experimental Knowledge Systems Laboratory
Computer and Information Science Department
University of Massachusetts at Amherst
Amherst, MA 01003
anderson@cs.umass.edu dhart@cs.umass.edu cohen@cs.umass.edu

Abstract

We derive a simple mathematical model of the interaction of two tasks running concurrently on a uniprocessor. Specifically, we model the rate at which two periodic tasks will interrupt each other. The predictions of this model are supported by simulation experiments. From the model, we argue that placing reflexive control and cognitive control on different processors is not justified by the different time-scales over which they act, but is justified by the extent to which the higher priority task dominates the uniprocessor.

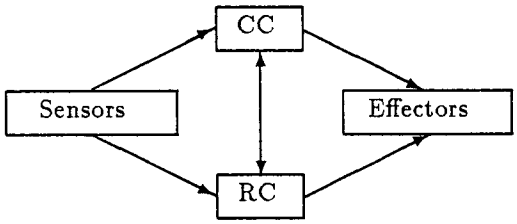

Figure 1: An abstraction of the Phoenix Agent Architecture. The CC is the "cognitive component" and the RC is the "reactive component." Each programs the effectors based on sensor data.

The Phoenix Agent architecture, shown in Figure 1, provides two components that read data from the sensors and program the effectors, thereby providing two sense-act loops for a single agent. One is called the "reflexive component" (RC) and the other is the "cognitive component" (CC). We argue that having two components is justified because of differences between the kinds of tasks assigned to each component and the resulting interruption of one task by another if they were to be assigned to a single component. We then explain how the two components differ and how they are integrated.

1 Why Two Components?

1.1 Interruptions

A Phoenix agent engages in activities that, intuitively speaking, happen on disparate time scales. Examples of activities on short time scales are digging fireline and driving down the road, where the agent must react quickly to the unexpected. Our Phoenix bulldozers can see for 512 meters and drive through softwood at 108 meters per minute. If fire comes into view, they have less than five minutes to stop, or they will be burned up. (In a more complex simulation, this figure would probably be less, because the location of fire would be obscured by smoke, trees, terrain, and so forth.) An example of an activity on a larger time scale is creating and executing a plan to encircle a fire with fireline. A typical indirect attack on a fire can take from 800 to 1200 minutes.

Clearly, 5 minutes is much less than 800–1200 minutes, but what do we precisely mean by "disparate time scales" and why does this imply the need for separate components? Why can't a Phoenix agent drive along the road while planning an attack? As long as the planning activity is interrupted whenever necessary to adjust the driving, why can't both be done concurrently in a single component?

The central issue is the effect of *interruptions*. Consider an agent that must do two periodic tasks concurrently. (Phoenix tasks are typically environment-driven and therefore not periodic, but we consider periodic tasks as a simplification for the purpose of modelling interruption effects.) Task i takes c_i minutes[1] to compute on a particular processor and must be done every d_i minutes. Assume for a moment that at time zero both tasks are ready to run and are at the beginning of their periods. Their periods will overlap in various ways as time progresses, until they return to this initial configuration and the cycle repeats. The length of this cycle is the least common multiple of the task periods, and we will denote it d_{lcm}. Therefore, it is sufficient to analyze the behavior during the d_{lcm} cycle, since that determines the behavior at all longer lengths of time. During d_{lcm} minutes, task i will be executed d_{lcm}/d_i times, and each execution will take c_i minutes.[2] Therefore, both tasks are schedulable on a single processor of the specified type only if

$$\begin{aligned} c_1\frac{d_{lcm}}{d_1} + c_2\frac{d_{lcm}}{d_2} &\leq d_{lcm} \\ \frac{c_1}{d_1} + \frac{c_2}{d_2} &\leq 1 \end{aligned} \qquad (1)$$

The first equation can be read simply as the time of each execution multiplied by the number of executions of each type, and the sum of these must be less than the time available. The sum in the second equation is usually referred to in the scheduling literature as the *utilization* of the processor. Liu and Layland [6] is a standard reference in the scheduling literature: they prove the optimality of the rate-monotonic (RM) algorithm for static scheduling and the earliest-deadline-first (EDF) algorithm for dynamic scheduling of periodic, preemptive tasks. However, Liu and Layland ignore the costs of interruption, although they recognize that it is important in practice; we will not ignore it.

To assess the cost of interruption, suppose that for a pair of tasks, c_2 and d_2 are much larger than c_1 and d_1. Under any reasonable scheduling algorithm, task 1 will be scheduled in preference to task 2, since otherwise it will miss its deadline.[3]

[1] We arbitrarily choose minutes as the time units for all such variables.

[2] We take the c_i to be fixed. Variance in the actual execution time will affect the interruption rate, but this analysis will concentrate on the effects of static properties of the task set.

[3] Under the RM algorithm, task 1 has a higher rate (the inverse of the period, d_1) and therefore will always preempt

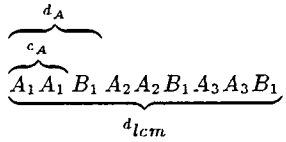

Figure 2: Illustration of how a long task, B, with $c_B = 3$ and $d_B = 9$ is interrupted by a short-period task, A, with $c_A = 2$ and $d_A = 3$. Time moves to the right, with the letters designating which task is running during that time. The subscripts indicate which instance of the periodic task is running.

This means that out of each d_1-minute period, task 1 gets c_1 minutes and task 2 gets at most the remaining $d_1 - c_1$. If $d_1 - c_1$ is shorter than c_2, task 2 will be interrupted. We know that in d_{lcm} minutes we have

$$\left\lfloor \frac{c_2}{d_1 - c_1} \right\rfloor \tag{2}$$

interruptions of *each* execution of task two.

Figure 2 illustrates this formula. We get $\lfloor 3/(3-2) \rfloor = 3$ interruptions of task B. The alert reader will note that some of these interruptions (at most half) can be eliminated by re-ordering. For example, $A_1 A_1 B_1 A_2 A_2 B_1$ can become $A_1 A_1 B_1 B_1 A_1 A_2$, thereby eliminating one interruption of B. Simple RM or EDF scheduling algorithms won't do this, because task A has a higher rate and earlier deadline than B. Also, reordering tends to force some instances of task A together while spreading others apart, which means that the variance of the time between executions will be higher. Increasing the variance may have a deleterious effect on tasks such as sensor sampling.

Again, there are d_{lcm}/d_1 executions of task one, so if we let INT be the cost in minutes of an interruption, equations (1) and (2) yield the following necessary condition for schedulability. Equation (3) extends equation (1) to include the cost of interruptions.

$$\begin{aligned} c_1 \frac{d_{lcm}}{d_1} + c_2 \frac{d_{lcm}}{d_2} + \left\lfloor \frac{c_2}{d_1 - c_1} \right\rfloor \frac{d_{lcm}}{d_2} \mathrm{INT} &\leq d_{lcm} \\ \frac{c_1}{d_1} + \frac{c_2}{d_2} + \left\lfloor \frac{c_2}{d_1 - c_1} \right\rfloor \frac{\mathrm{INT}}{d_2} &\leq 1 \end{aligned} \tag{3}$$

Let us isolate one part of equation (3), the interruption rate:

$$\mathrm{IR} = \left\lfloor \frac{c_2}{d_1 - c_1} \right\rfloor \frac{1}{d_2} \tag{4}$$

The interruption rate is the measure of how scale disparities in the tasks affect performance and is the focus of the following analysis.

To test whether formula (4) correctly calculates interruption rates for particular task sets, we implemented a simple simulator for sets of periodic tasks. To create input task sets, we needed a source of "small" tasks and "large" tasks, corresponding to our intuition that scale difference would determine interruption rate. Small tasks were created by choosing c_1 and d_1 from a fixed normal distribution ($\mu = 10$, $\sigma = 3$), and large tasks were created by choosing from a normal distribution X times larger than the small one ($\mu = 10X$, $\sigma = 3X$). We checked that c was always less than d and that equation (1) held. (We didn't use equation (3), because the simulator assessed no cost for interruptions.) The simulator used the EDF scheduling algorithm and counted the actual number of interruptions. We found that formula (4) is very accurate over a wide range of task sets.

Once we were sure that the formula was fairly accurate, we analyzed it to find the factors that determined the interruption rate. In particular, we wanted to know the effect of the "scale" difference in the task set. Given the way the task sets are chosen, the expected value of both c_2 and d_2 is $X\mu$. Of course, they are subject to the constraint from equation (1); nevertheless, we will approximate formula (4) as follows:

$$\mathrm{IR} = \left\lfloor \frac{X\mu}{d_1 - c_1} \right\rfloor \frac{1}{X\mu} \tag{5}$$

Observing that for any α, $\lfloor \alpha \rfloor = \alpha - \epsilon$, where ϵ is between 0 and 1, we reasoned as follows:

$$\begin{aligned} \mathrm{IR} &= \left\lfloor \frac{X\mu}{d_1 - c_1} \right\rfloor \frac{1}{X\mu} \\ &= \left(\frac{X\mu}{d_1 - c_1} - \epsilon \right) \frac{1}{X\mu} \\ &= \frac{1}{d_1 - c_1} - \frac{\epsilon}{X\mu} \end{aligned} \tag{6}$$

Clearly, the quantity $d_1 - c_1$, often called *laxity*, is a second important factor. Note that laxity is *not* the same as the utilization of the higher-rate task, which is c_1/d_1. We will notate the laxity as Y in our equations:

$$\mathrm{IR} = \frac{1}{Y} - \frac{\epsilon}{X\mu} \tag{7}$$

We now have an equation simple enough to understand and make predictions from. We predict a strong effect of laxity (increasing Y should sharply decrease IR) and a smaller effect of scale (increasing X should increase IR until the $\epsilon/X\mu$ term vanishes). We also predict no interaction between these two factors—their effects should be purely additive.

To test these predictions, we ran a number of simulations of randomly chosen task sets and performed an analysis of variance (ANOVA). The results are shown in Table 1. Note the strong effect of Y ($p < .0001$) and the complete lack of an interaction effect. The effect of X, however, was not significant. Given our analysis, this suggests that X was already too big to show an effect, so we ran another set of tests, varying X over slightly smaller values. Table 2 shows these results. Note that X is now highly significant ($p < .0042$), even though its range has changed only slightly.

Unfortunately, Table 2 also shows a slight interaction effect. One explanation is that the linear regression model which underlies the ANOVA test is not applicable to equation (7) because of the hyperbolic effect of each factor. Therefore, we ran another experiment using factors which are the reciprocal of the original factors. That is, equation (7) was rewritten in terms of factors X' and Y' to get:

$$\mathrm{IR} = Y' - X' \frac{\epsilon}{\mu} \tag{8}$$

The ANOVA results for this experiment are given in Table 3. We see that X' and Y' are highly significant factors, and that

task 2. Under the EDF algorithm, task 1 will typically have an earlier deadline, again because of the shorter period, and preempt task 2.

source of variation	df	sum of squares	mean square	F	P
X	3	.006	.002	1.558	.2012
Y	4	.062	.016	11.471	.0000
XY	12	.009	.001	.580	.8563
error	180	.243	.001		

Table 1: ANOVA: $X \in \{3,4,5,6\}, Y \in \{2,4,6,8,10\}$

source of variation	df	sum of squares	mean square	F	P
X	3	.015	.005	4.564	.0042
Y	4	.065	.016	14.521	.0000
XY	12	.025	.002	1.869	.0408
error	180	.201	.001		

Table 2: ANOVA: $X \in \{2,3,4,5\}, Y \in \{2,4,6,8,10\}$

source of variation	df	sum of squares	mean square	F	P
X'	3	.033	.011	5.127	.0021
Y'	3	.076	.025	11.649	.0000
$X'Y'$	9	.025	.003	1.261	.2632
error	144	.313	.002		

Table 3: ANOVA:
$X' \in \{1.0,.71,.42,.125\}, Y' \in \{.5,.37,.23,.10\}$

there is no interaction effect. The predictions of our analysis have largely been confirmed by these experiments.

Our intuitive justification for having two sense-act loops was that scale differences caused high interruption rates, and that was costly enough to warrant two components. We were surprised to find that scale is only a small factor, and that laxity is more important. While intuition might have told us that low laxity is bad, we now know that changing the task set to one with different laxities produces a highly nonlinear effect on the interrupt rate. We also know that a small X can reduce the interrupt rate and that changes in X when X is small produce noticeable changes in the interrupt rate. On the other hand, large values of X will cause the second term to go to zero, so the interrupt rate will be maximized for that value of Y. Furthermore, changes in X when X is large will produce no noticeable change in the interrupt rate. With respect to the design of Phoenix, this means that the scale difference we pointed to before (5 minutes versus 800–1200) maximizes the interrupt rate and that a small change to this scale difference will be ineffective in reducing the interrupt rate.

1.2 Computer and Human Precedents

So far, we have analyzed only periodic tasks, even though Phoenix tasks are usually not periodic. We believe, though, that the force of the argument holds more generally. Aperiodic tasks can be characterized by their relative magnitude, perhaps by their expected arrival rates and expected computation time. We can also assess the extent to which smaller, higher priority tasks will saturate a processor, even though it may not be a simple ratio of computation time to period.

We certainly see the need for parallel components in other domains. In computer systems architecture, we see autonomous I/O units with their own processors, largely because of speed disparities between the I/O device and the CPU. In such architectures, there can be many concurrent components. Consider also the architecture of the human central nervous system (CNS). There is ample evidence of autonomous components and concurrent processing ability. Indeed, it was by analogy with the human CNS that the Phoenix architecture was designed. We do not claim that the Phoenix agent architecture is a model of the human CNS, but we believe that the factors of scale and saturation may explain and justify human and computer architecture as well as the Phoenix agent architecture.

2 How the Components Work[4]

The RC contains a collection of reflexes, which are associations between sensors and effectors mediated by two simple functions, one for triggering reflex execution based on sensor readings (the trigger function) and one for changing effector settings based on those sensor readings (the response function). Trigger functions are simple functions of the sensor readings, such as whether the value exceeds a threshold or equals some value. Response functions make simple changes to effectors, turning them on or off or making minor parameter adjustments. These functions rely only on currently available sensor readings; reflexes retain no persistent state information other than the values of their control parameters. These parameters define reflex sensitivity and are set by the CC.

After a sensor executes, trigger functions, which link the sensor to particular reflexes, are executed to determine whether the associated reflexes should be activated. A trigger function may rely on values from more than one sensor, in which case it simply checks the most recent readings for the critical sensors. When activated, the reflex executes the response function to change effector settings. For example, when the *sense-road-heading* sensor has a value different from that of the *sense-agent-heading* sensor, the *follow-road* reflex changes the *heading* parameter of the movement effector to maintain the road heading.

While the RC has a simple internal structure, the CC is more complex, with numerous subcomponents and control mechanisms. It plans by a method we call lazy skeletal expansion. By "lazy," we mean that commitment to a precise course of action is delayed as much as possible, thereby maximizing the opportunity for environmental change to be taken into account. Deferred commitment is accomplished by interleaving three basic activities: find, expand, and execute. *Find-plan* actions are placed on the *timeline* (the agent's agenda of pending tasks) as part of plans (to defer commitment to a particular plan or action) or in response to exceptional conditions (as noted by messages from the reflexive component or from other agents). These actions use their context with the timeline to search the plan library for skeletal plans appropriate for the context and the current state of the world. For example, if the find-plan action is to get an agent to the fire, the context includes information about the type of fire-fighting plan it is part of, the techniques being used to fight the fire, and the coordination needed with other agents, in addition to the location of the fire. *Expand-plan* actions instantiate the plan's network of actions on the timeline. *Execute* actions calculate variable values, manage resources and control the agent's interactions with the world. As the timeline actions are executed, plans and actions are incrementally added, sensors and effectors are activated, and the agent pursues the plan. Actions become eligible for execution only when the

[4]This section is gratefully drawn from Howe and Cohen [5].

siblings that precede them have already executed; even then, execution may be deferred until information about the state of the world is available. Because plans are combinations of primitive actions and plan expansion actions, action and planning are interleaved.

3 What They Know

The RC is quite limited in its knowledge, mostly because that knowledge is either useless or dangerous. Because the RC is only looking for simple conditions, such as some sensor reaching a critical level, it has no need of the contents of the plan library, since the plan library comprises mostly fire-fighting plans and plans that are irrelevant to the RC. Some plans are indeed relevant, such as planning a path to a safe location; in fact, that is the very plan that the CC uses when the RC has alerted the CC that fire is encroaching. However, that plan can take many minutes to run, thereby preventing the RC from doing high-rate tasks that keep the agent alive. Another example is that the RC has no access to the agent's map of Yellowstone. This information is certainly relevant, as it might help the RC notice and avoid obstacles that are marked on the map, long before they are visible. However, availability of the map might tempt the RC to do long-running searches, which would affect its ability to keep up with its other tasks.

The CC has access to all the agent's knowledge, which encompasses the plan library, the timeline, the map, envelopes, and communications from other agents (for more on envelopes, see Hart et al. [4]). In principle at least, the CC can compute anything. Since the RC takes care of keeping the agent alive, even in the face of environmental change, the CC is free to pursue other goals, in particular, fighting fires. Firefighting typically involves setting deadlines for tasks to be done, so the CC is, in fact, under time pressure, somewhat as the RC is. The difference, though, is that the CC's deadlines are largely self-imposed, since the fire-fighting environment, in the large, tends to have soft deadlines. Letting the fire burn for another five minutes or even another hour will typically make little difference in its final size or effects, because most fires move slowly.[5] However, many plans of attack commit the firefighters to digging particular firelines by a particular deadline, and this gives the agent a constraint on how long it can take to do something. On the other hand, if the constraint proves to be a problem (perhaps because the fire has increased its speed), the plan can be aborted and re-planned with new commitments.

4 How They Interact

The authority relationship of the RC and the CC is asymmetrical, mostly stemming from the difference in their knowledge. Essentially, the RC is fast but somewhat stupid, while the CC is smart but somewhat slow. Therefore, the CC "programs" the RC to execute and monitor actions that the CC cannot do itself. Programming the RC consists of turning reflexes on and off and adjusting other parameters, such as its sensitivity to change.

When the RC notices a problem, it usually communicates it to the CC by placing an action on the CC's timeline indicating the nature of the problem. Occasionally, however, the RC can deal with the problem by itself. For example, if there is an obstruction in the path to be driven, the RC can drive around it without having to notify the CC. But for the most part, the RC does as the CC has "programmed" it.

[5] For example, under reasonable environmental conditions, fire spreads at less than 300 meters per hour.

5 Conclusion

We have attempted to justify a particular design decision of the Phoenix agent architecture. We began with an intuitive argument about the effects of scale differences between tasks on interruptions. Our formal analysis, however, led to a different assessment of the impact of scale differences and the discovery of another, more significant factor—saturation of the processor by smaller tasks. We tested and confirmed the predictions of this model using a simple simulator.

Two directions can be pursued with this work. The first is to generalize the argument to other kinds of architectures, and possibly to other architectural decisions. The second is to analyze the disadvantages of splitting tasks between multiple components. For instance, what is the cost of the extra communication and "programming" between the CC and the RC? By this kind of modeling, analysis and testing, we hope to put agent design on firmer ground than intuition, which has sometimes proven treacherous.

6 System Status

The Phoenix system is a complete and well-tested simulation of Yellowstone National Park and the firefighting agents, such as bulldozers, watchtowers, helicopters and firebosses, that act in that world. For a system architecture overview, see Cohen *et al.* [1]. There is always room for improvement, but at this point we consider it primarily as a testbed for our ideas; enhancing the complexity of the world and the capabilities of the agents is secondary. The Phoenix system has run thousands of scenarios, where a scenario consists of setting fires, either at random locations or under user control, and allowing the agents to try to put them out. A scenario usually lasts for between forty and a hundred hours of simulation time. As of this writing, the system can simulate around twenty scenarios before the machine fails due to exhausted swap space, for reasons we have not yet been able to discover.

We have a technical report [3] available that documents the testbed, and we would be interested in having other research groups test their agents in the Phoenix world.

Acknowledgements

This research was supported by AFOSR under the Intelligent Real-Time Problem Solving initiative, contract #AFOSR-91-0067, by DARPA-AFOSR contract #F49620-89-C-00113, and by the Office of Naval Research under a University Research Initiative grant, ONR #N00014-86-K-0764, and the Texas Instruments Corporation.

We thank Adele Howe, Keith Decker, and Paul Silvey for their help in developing this paper. We also thank John A. Stankovic and Krithi Ramamritham for critically reading an earlier version of this paper and contributing their knowledge of real-time computing.

References

[1] 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):34–48, Fall 1989.

[2] Thomas Dean. Planning, Execution, and Control. In *Proceedings of the DARPA Knowledge-Based Planning Workshop*, Austin, Texas, December 1987.

[3] Michael Greenberg and David L. Westbrook. The Phoenix Testbed. Technical Report 90–19, Department of Computer and Information Science, University of Massachusetts, Amherst MA 01003, 1990.

[4] David M. Hart, Scott D. Anderson, and Paul R. Cohen. Envelopes as a Vehicle for Improving the Efficiency of Plan Execution. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 71–76, 1990.

[5] Adele E. Howe and Paul R. Cohen. Responding to Environmental Change. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 85–92, 1990.

[6] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.