SIMULATION OF MULTIPLE TIME-PRESSURED AGENTS

Scott D. Anderson

Department of Computer Science
Spelman College
Atlanta, GA 30314-4399, U.S.A.

ABSTRACT

The paper describes a simulation substrate that allows thinking agents to interact with a world. The world is simulated by standard discrete event simulation, but the timing of an agent's behavior is determined by the amount of computation it performs. Therefore, if an agent thinks a lot about what to do given a situation in the world, the duration of its thinking results in a delay to its subsequent actions. Thus, the thinking of the agent is time pressured. The computation time of the agent is automatically assessed by the substrate in a way that is independent of the computer running the simulation. This is done by implementing the thinking of the agents in a variant of Common Lisp called Timed Common Lisp, in which each function advances a clock by an appropriate, user-specifiable amount of time. This renders agent thinking and behaviors deterministic, making results comparable and replicatable across platforms. The simulation substrate also supports the interaction of continuous activities, in addition to executing discrete, point-like events. This substrate has been used to implement an Artificial Intelligence Planning system that simulates multiple agents fighting forest fires in Yellowstone National Park.

1 INTRODUCTION

The simulation substrate described in this paper was developed to support what is called "Real-Time Planning" within the field of Artificial Intelligence (AI). "Planning" is approximately what it sounds like: given a set of operators (ways the agent can change the state of the world) and a goal (a desired state of the world), the agent derives a partial order of the operators that will accomplish its goal (Allen et al. 1990; Russell and Norvig 1995). The term "real-time" can be confusing, since it doesn't mean hard real-time as the term is used by, say, signal processing engineers. Instead, the research emphasis is on trade-offs between the quality of the plan and the amount of time taken to compute it, or the effort to derive a plan in time for a hard or soft deadline (Stankovic 1988; Bratman et al. 1988). To avoid confusion, this paper will typically describe the agents as being "time pressured."

To be concrete, consider the PHOENIX system (Cohen et al. 1989; Greenberg and Westbrook 1990), which simulates forest fires burning in Yellowstone National Park and agents that fight those fires. Once the fire is started, the agents notice it, plan how to put it out given current environmental conditions, and execute that plan, while also monitoring the weather and the progress on the fire, and re-planning as necessary. This is a time-pressured planning problem because the fire continues to burn while the agents are thinking and acting. An agent that thinks too slowly will find that fires often get big and out of control, so that the agents will be unable to achieve the goal. In a typical PHOENIX simulation, there will be a watch-tower, four bulldozers, one fuel truck, and a helicopter—each of which is an agent—and finally a "fireboss" agent that oversees and coordinates the activities of all the other agents.

The key issue with a simulation like PHOENIX is that the simulation of the fire must continue while the agents are thinking—but by how much? That is, there must be a correspondence between some measure of how much the agent thinks and how much time passes in the simulation (such as how much the fire gets to burn). By varying this correspondence, we can put more or less time pressure on the agents, because they will think slower or faster relative to the passage of time in the environment.

This paper concerns the infrastructure that is necessary for simulating time-pressured agents like the ones in the PHOENIX system. That substrate comprises two components: the Multiple Event Stream Simulator (MESS), which is the simulation system, and Timed Common Lisp (TCL), which is the implementation language for the agents and defines the correspondence between computation and simulation time.

The rest of the paper is organized as follows. Section 2 describes the general design of the MESS substrate and how it coordinates and integrates the multiple agents. Section 3 describes how the activities of agents and environmental processes can interact. Section 4 explains in detail the Timed Common Lisp component, including different ways of modeling computation time. Section 5 describes the PHOENIX testbed, which uses MESS and TCL, and the architecture of the agents, including how they sense, react, plan and communicate.

2 SUBSTRATE DESIGN

At its heart, MESS is a straightforward discrete-event simulator. Events are defined as objects in an object-oriented programming language,[1] where the user defines methods that determine when the event occurs and how it modifies the representation of the world. The "how" code is the *realization* method of the event, and executing that code is called *realizing* the event. The hierarchy of event classes can be used to group kinds of events, such as all the movement events or all the fire events, so that they can be controlled and modified as a group.

MESS is a process-oriented simulator (Bratley et al. 1983, p. 13), which means that each event is produced by a process, and that process determines subsequent events. For example, things like fire, weather, and particularly an agent's thinking might each be a separate process in the simulation. The representations of processes are called *event streams*. Event streams are also defined as objects, so that users can add other kinds of event streams if they need a particular way of producing events.

Figure 1 shows the structure of MESS. The simulator has a central "engine," which interleaves the streams of events that represent different real-world processes. These events are drawn from and generated by event streams of various kinds. A very general kind of event stream (ES) is a *function* ES, where a user-supplied function computes the next event upon demand. Another kind of ES is a *list* ES, which produces a pre-defined sequence of events. The MESS engine controls *instances* of these kinds of event streams, one instance for each world process. Figure 2 presents pseudo-code for the algorithm to advance the simulation. Each time the simulation is advanced, exactly one event is realized.

```
Algorithm to Advance the simulation:
    increment event counter
    advance time by head of PEL
    If head of PEL is an event stream
        Set ES to head of PEL
        Peek ES
        Set E to event in ES
    else
        Set ES to nil and
        Set E to head of PEL
    Check for Interaction
    Realize E
    Illustrate E (optional)
    Unless ES = Nil
        Pop ES
    Do Every Event Stuff
    Check Wakeup Time Functions
    Write out E (optional)
    Change Activity
```

Figure 2: Pseudo-code for the MESS engine.

The event to be realized is whichever is nearest in the future. The simulation literature has several terms for the data structure holding these events; we call it the "pending event list" or PEL. In MESS, there can be two kinds of object in the PEL: an event or an event stream. In practice, in the simulators implemented using MESS, most of the objects in the PEL are event streams.

If two events in the PEL are scheduled for the same time, ties are broken in a deterministic way. First, the "priority" of the two events is compared, where the priority is an arbitrary integer specifiable by the user just for this purpose; it defaults to zero. The event with the higher priority is realized first. If there is a tie in priority, the PEL code is structured so that the first event scheduled will execute first; that is, the PEL is a FIFO queue. Currently, the PEL is implemented as an implicit binary heap, but future releases may include better data structures. See Rönngren and Ayani (1997) for a recent comparison of data structures for event queues.

The pseudo-code shows the major features of how MESS works. (A more detailed technical description is available in my dissertation (1995).) The primary objective of the engine is to realize events, which occurs in the center of the algorithm. If the first thing

[1] All of MESS and TCL are implemented in Common Lisp (Steele Jr. 1990). Events and Event Streams are defined using the Common Lisp Object System (CLOS) (Keene 1989). These languages were chosen because Common Lisp is one of the primary languages used in Artificial Intelligence research.

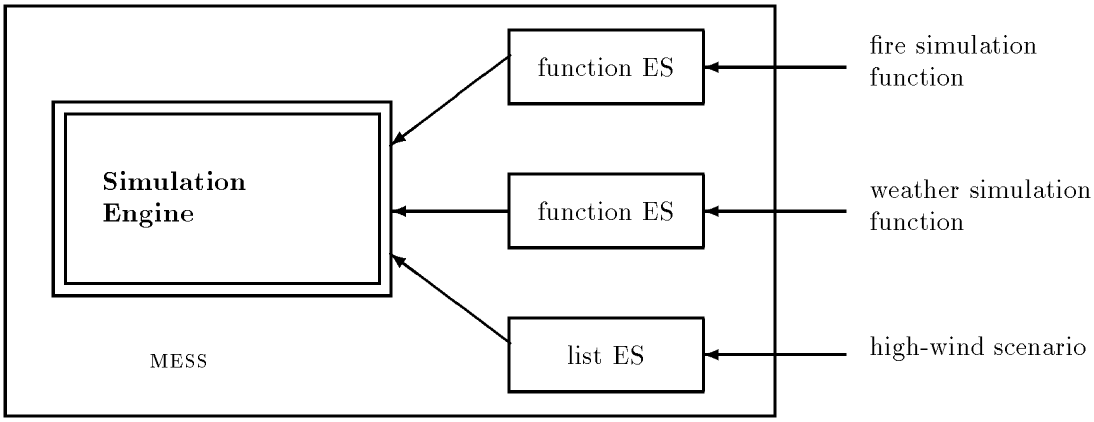

Figure 1: The architecture of the MESS simulation substrate. MESS is structured as a central engine, driving instances of different kinds of event stream (ES). MESS itself is domain independent; the streams listed at the right (weather, fire, scenario) are examples drawn from the PHOENIX domain.

in the PEL is an ES, the engine must make the ES produce an event to realize, which is done by the *peek* operation. (Later, the event is removed from the ES by the *pop* operation.) After the event is realized, the event is *illustrated.* The purpose of realization is to change the state of the simulation, while the purpose of illustration is to modify the graphical user interface (GUI), if any. This separation of realization from illustration aids in running batch simulations, because all the GUI code can be ignored. The separation also helps keep simulations portable, since GUI code is a common source of portability troubles.

The highlighted operations—***peek***, ***interaction***, ***realize***, ***illustrate***, and ***pop***—are all CLOS methods that can be specialized by the user, so that the substrate can be extended to other kinds of simulations. Any object that obeys this protocol can be used by the engine, so different kinds of event streams can be defined by implementing the protocol.

Several minor steps in the pseudo-code deserve mention. The "every event" step, near the end of the algorithm, executes each element of a list supplied by the user at the start of the simulation. By adding code to this list, an experimenter can easily arrange for something to be executed continuously during the simulation. For example, data-collection code is often executed this way. The "wakeup time" step awakens event streams that have been put to sleep for some reason. For example, the fire-simulation ES is asleep when no fire is burning. The optional "write out" step saves every event to a file, so that a simulation can be analyzed or replayed. Finally, the protocol includes steps to check for interactions among events and activities; these are discussed in the next section.

3 ACTIVITIES AND INTERACTIONS

Events are "point-like," in that they happen at a moment in time. However, many kinds of simulations involve things that happen over an interval of time; these are called *activities* in MESS. For example, a train traveling from one station to another would be represented as an activity. Activities are represented as a pair of point-like events, representing the beginning and ending of the activity.

MESS is designed not only to support activities, but also *interactions* between activities and other events, including other activities. Suppose a bulldozer (or other vehicle) is traveling from A to B, while another is traveling on an intersecting course from C to D. In some simple discrete-event simulators, this collision would never be noticed because there is no event at that time, but MESS keeps track of all current activities and checks for interactions.

The interaction can affect either the activity or the intervening event, or both. A rain activity might cancel a scheduled fire-ignition event (which is why the MESS engine checks for interactions before realizing the event). An event representing the firing of a surface-to-air missile might terminate a fighter plane's flight activity and schedule a plane-crash event. The movement activities of two vehicles might result in a collision, with both activities affected by the interaction. An important kind of activity is agent deliberation, which might, for example, be interrupted by a sensory event.

Activities are essentially a kind of event that happens twice. Whenever an activity starts, it is placed on a list by the MESS engine, and it is removed when the activity ends. Each event that happens while the activity is on the list has the opportunity to interact with the activity. This opportunity is implemented via the *interaction* function. The interaction function is a two-argument CLOS generic function, extended by the user, since the semantics of the interaction between the activity and the event is necessarily domain-dependent. While MESS cannot supply the semantics of the interaction, it automates the book-keeping required for interacting activities of agents.

4 THE DURATION OF DELIBERATION

The thinking time of an agent seems straightforward to define: just measure the CPU time of the code. But on what processor? Should the processor matter? What code counts in the computation? How is that code's duration modeled?

An earlier version of the PHOENIX system integrated the thinking of agents with the discrete-event simulation of the environment by advancing the simulation clock depending on the amount of CPU time used by the agent. For example, the default setting in PHOENIX was that one CPU-second corresponds to five minutes of simulation time, so if an agent thinks for 3 CPU-seconds, the fire could burn (and bulldozers move and so forth) for 15 minutes.

This CPU-time approach is standard among AI simulators for time-pressured planning (Anderson 1995). Unfortunately, there are problems with using CPU time, all of which we have suffered while using the original PHOENIX:

Variance: Small, random variations in the measurement of CPU time result in random variation in the behavior of the simulation. This uncontrolled randomness, sometimes called "non-determinism," can make it difficult to replicate particular simulation states, whether for debugging, demonstration, or experimentation.

Platform-dependence: The simulation behaves differently from one computer to another. This exacerbates the variance problem and adds unwanted noise to data from large experiments in which trials are run on many different machines.

Interference: Inserting code to record or print data, say for debugging, demonstrations, or analysis, affects the CPU time of the code, which in turn affects the behavior of the simulation. This is something like the Heisenberg uncertainty principle in physics: the act of observing the code affects the code. While the Heisenberg principle may be true in the real world, it is hardly convenient for experimental scientists.

Essentially, all these troubles are "noise" that comes from using CPU time. Consequently, we looked for another way of measuring how much computation an agent has done, one that gives us the ability to replicate simulation states.

4.1 Duration Modeling

The basic idea for modeling a computation's duration is to advance the clock by some amount for each "primitive" that is executed. If these increments depend only on the code that is executed and not the computer system and compiler, the duration of the code will be invariant. What remains is to decide what a primitive is and how the increments are determined.

4.1.1 Low-level Models

A "low-level" primitive is a primitive of the Common Lisp language, such as `first`, `+`, or `find`. Using low-level models retains much of the flavor of the CPU-time approach, because the duration is tied quite tightly to exactly what code executes. Timed Common Lisp, or TCL, shadows every primitive of Common Lisp, so that the functions have the same semantics but also advance the clock by a certain amount. Programming in TCL is exactly as in Common Lisp—the two are essentially the same from the programmer's viewpoint. The difference is that TCL primitives advance the clock.

Of course, the clock should not be advanced by the same amount for each TCL primitive. Indeed, it should not even be advanced in the same way. For example, `first` should advance the clock by a small constant; most primitives fall into this category, although the constants are all different. Functions like `+`, on the other hand, should advance the clock depending on the number of arguments they get. The duration of a function like `member`, which searches a list for some element, should depend on the length of the list. For a function like `make-array`, the duration should depend on the number of elements in the array.

TCL defines about two dozen classes of such *duration models*. A duration model is some measure of the amount of work a primitive does. This measure is then multiplied by a coefficient to yield the actual duration of the primitive with that duration model. Duration models are entirely analogous to the "big-O" notation of complexity theory. A constant time function like `first` has a duration model that is $O(1)$, while a function like `*` has a duration model that is $O(n)$ where n is the arity (number of arguments) of the function. The duration model of `sort` is $O(n \log n)$, where n is the number of elements to be sorted.

By using these low-level models, TCL can report numbers that seem like CPU time, but are noise-free and can be replicated on any Common Lisp platform, since TCL runs on any Common Lisp.

4.1.2 High-level Models

The fundamental operations of an agent's mind need not be reduced to the primitives of Common Lisp—we can define duration models at a higher level. For example, a chess-playing agent might define "evaluating a board position" or "generating a move" as a fundamental "cognitive primitive." The duration of some chess deliberation is then $O(f(n, m))$, where m and n are the number of these higher level primitives; for example, m could be the number of board positions evaluated and n could be the number of moves generated, and f is some arbitrary function of those numbers, determining the duration of a move.

Of course, as with the primitives of Common Lisp, we don't want to confine ourselves to constant-time models. TCL allows duration models to be defined as arbitrary functions of the primitive's arguments and the computational state of the system. The model can even be pseudo-random, if that's desirable.

An agent that is thinking about its own thinking time (for example, it might decide "I'd better stop planning and go fight the fire" or "I'll think about it for ten minutes and then stop") needs a simple, declarative representation of how long its thinking will take. High-level cognitive primitives can help here, especially since the duration models are stored in a TCL database that is accessible to the agent. If the duration model is not a simple constant, the agent can still try to predict how long the computation will take by guessing at the aspects of the simulation state used by the duration model. For example, it might be reasonable to guess at the number of board positions that will be evaluated during a move. It certainly seems easier to guess at that number than to guess at the amount of CPU time that the move would take. Of course, a historical approach can also be used, where the durations that occurred on previous runs are used for prediction; these historical durations can also be stored in the TCL database.

Using a high-level model also allows for a new class of simulation experiments on agents in which the durations of different cognitive primitives are independently controlled. For example, if "move generation" and "board evaluation" are two cognitive primitives in a chess agent, we can modify the duration model for one primitive independently of the other to see the effect on the agent's behavior and performance. That is, as it becomes relatively slower to think about A than B, what does this do to the quality of the agent's behavior? In principle, one can also alter the duration model for low-level Lisp functions independently, but there are no interesting research questions posed by that manipulation. By moving to high-level primitives, one can ask sensible questions about duration/quality tradeoffs.

4.2 Non-interfering Code

So far, we've described how the clock advances as each primitive executes. What if we don't want the clock to advance? Suppose, for example, we put in a `print` statement either to debug or demonstrate the program's behavior. We don't want that insertion to affect the behavior of the simulation. With a CPU-time approach, it can be hard to turn off the clock, but with TCL it's trivial. Any code that shouldn't advance the clock is wrapped in a `free` form. For example, the following reports what the agent is thinking about without affecting its thoughts or their duration:

```
(defun think ()
  ...
  (free (format t "Thinking about ~s~%"
                current-thought))
  ...)
```

This ability is particularly important in extensive simulations, where we may want to log the agent's thoughts and actions, so that the simulation can later be analyzed, either for debugging or hypothesis testing. We certainly want the behavior of agents to be independent of the experiments performed on them.

4.3 Overhead

What are the disadvantages of using TCL? There are no notational disadvantages, since it looks just like Common Lisp and requires no commitment to a particular agent- or cognitive-architecture. The advancing of the clock, however, does entail an inevitable overhead. Quite simply, the code is doing more work. Therefore, there will be some slowdown of the user's code.

It's difficult to make any blanket statements about how much slowdown there will be without knowing the kind of code and duration models. The speed will depend partly on the level of the primitives that the code uses. For example, if the code is "low-level" code that does a lot of operations like `car` and `cdr`, each of those primitives now has an associated increment of the clock. For such simple functions, incrementing the clock is a significant slowdown. On the other hand, a function like `sort` is barely slowed down by measuring the size of its input (n) and incrementing the clock by $cn \log n$, where c is the duration model coefficient. If the user defines cognitive primitives at a higher level, the overhead may be even less. In addition, unfortunately, the speed also depends on the quality of the Lisp compiler.

Timings have been done using the worst case of low-level duration models. One set of timings were done using Gabriel's Lisp benchmark programs (Gabriel 1985) and these indicate that using TCL appears to increase execution time by between 20 and 120 percent. (The results varied widely over the roughly two dozen programs in Gabriel's benchmark suite.) On average, the overhead is about 50 percent. Informal experience with simulations in the Phoenix system suggest the overhead for TCL seems to be on the order of 20 percent.

5 PHOENIX

In the late 1980s, the Experimental Knowledge Systems Lab (`http://eksl-www.cs.umass.edu/`), at the University of Massachusetts at Amherst, embarked on a research program into Real-Time Artificial Intelligence. The goals of the project were to challenge AI agents with difficult problems to solve under time pressure. The domain that was chosen was fighting forest fires, since it involved changing conditions over long and short time scales, communication and cooperation among agents, and tradeoffs between planning time and plan quality (minimizing measures such as forest burned and agents killed). The project was called Phoenix, a term that comprises the simulator system and simulated environment (Yellowstone National Park) as well as the agents.

5.1 Agents

Each of the Phoenix agents has the same agent architecture (although it is possible to substitute a different agent architecture). This architecture comprises two components: the cognitive component and the reactive component.

The reactive component has a number of simple situation-action rules that are frequently executed so that the agent can react to environmental conditions. For example, while a bulldozer is traveling, it will have reactions to maintain its speed and direction, to check for fire, and to come to a quick halt if something dangerous like fire or a lake appears in its path.

The cognitive component does planning and other kinds of thinking activities for the agent. It can consult a map of the park, a database of facts about fires (such as how fast they burn under various conditions) and facts about agents (such as how fast bulldozers can move, or how much fuel they use), and a library of skeletal plans, which can be retrieved and adapted to the current situation. The cognitive component maintains an agenda of actions it intends to do (or has done—this history is useful when plans need to be repaired). In the standard agent architecture, the cognitive component repeatedly selects an executable action from the agenda (an action is executable when all its preconditions have been executed) and, if possible, executes the action. If the action can't be executed, the plan library is searched for some plan that will accomplish the action, the plan is instantiated, and placed on the agenda to be executed later. This method is of interleaving planning (by instantiation from a library) and execution of actions is called "lazy skeletal refinement."

Compared to the cognitive component, the reactive component is limited: the reactions cannot consult the map, knowledge base, or agenda. This limitation is because the reactions must be quick and the limitation also helps to ensure that they are. If a reaction rule notices a problem that it doesn't have an immediate solution for, it places the problem on the cognitive component's agenda. For example, the reactive component can stop the agent from moving into a lake, but it can't plan a path around the lake, so it places that planning problem on the agenda.

5.2 Communication

Communication is integral to the Phoenix system, because the agents must work together to put out the fire. For example, the fireboss instructs bulldozers where to dig, and bulldozers request refueling when necessary. Currently, the communication is done in a very simple way, where the sender executes a function giving the name of the receiver and the text of the message, and the simulation system inserts the message into a queue that is read by the receiver's cognitive component as part of its "main loop" (the loop that executes actions on the agenda). Thus, the communication is much like email ought to be: reliable, noiseless and essentially instantaneous, but there is no guarantee when the receiver will read the message. It would be straightforward to modify the simulator to delay or mislay messages, but adding noise to a message would require a very different message processing ability of all the agents.

5.3 Sensing

Sensing is done by the reactive component, since most reactions are triggered by sensory events, such as seeing fire. The reactive component records the sensation on the agent's personal map of the environment, which is consulted by the cognitive component when it needs to know what is going on. Currently, sensing is done by simply accessing relevant parts of the true, "real world" map of the park. Thus, the agent always sees accurately, within its limited field of view, which is taken to be a circle of fixed radius around the agent. The sole exception to the accuracy of the sensing is by the watch-tower agent, which regularly scans its very large field of view, but probabilistically sees only a fraction of the cells. Therefore, it may miss a small fire on the first or second scan, but eventually the fire will be seen and reported. This adds a small amount of realism to the simulation, at small cost. As with communication, it would be possible to add sensor error, at the cost of improving the agents' sensing abilities.

6 STATUS

The MESS and TCL components currently run in any Common Lisp platform. These components make up a simulation substrate, in the sense that they are domain-independent, but domain simulations can be built on top of them.

The PHOENIX simulation system has been implemented in Common Lisp and builds on the MESS and TCL foundation. A graphical user interface is available as well, implemented in CLIM.

All of the above can be retrieved by anonymous FTP from `ftp.cs.umass.edu/pub/eksl`.

Currently, MESS is implemented for sequential execution on uniprocessor computers, and, therefore, so is PHOENIX. (The original Phoenix was implemented using multiple threads on a uniprocessor, resulting in events happening out of order by up to five simulation minutes, since that was the "quantum" for process swapping.) MESS does not currently implement any mechanisms for parallel simulation.

7 EXPERIMENTAL FRAMES

Simulators like PHOENIX allow for several kinds of experiments on cognitive agents.

First, the elimination of CPU noise from the simulator make it easier to run large experiments in which the simulation runs might be distributed over a number of platforms, with varying CPUs, memory, number of users, and so forth. If the simulator relied on CPU time, these other factors would have to be dealt with. (In a pilot experiment with the first version of PHOENIX, statistical analysis of the data showed that the machine a trial was run on was a significant factor.) While these factors might be controlled or eliminated in other ways, it makes experiments more difficult to run and analyze.

Secondly, the elimination of CPU noise allows experiments in which particular world states are replicated many times. One use for such experiments would be to test the importance or effectiveness of an agent's decisions in a given situation: replicating the situation makes it possible to try different decisions a significant number of times. A PHOENIX experimenter might, for example, be interested in whether the "indirect attack" fire-fighting method is superior to the "direct attack" method, so the experimenter can run pairs of trials in which the simulation is identical up to the moment when the agent chooses one of these two plans. (For this reason, there is a mechanism in PHOENIX to force the selection of a particular plan whenever it is a candidate.) Such experiments would have the same advantage over the CPU-time simulator that the matched-pairs t-test has over a t-test between groups.

Finally, the TCL component allows fine-grained control of the deliberation time of the agent. By using high-level duration models, the experimenter can selectively modify the thinking time of various aspects of the agent. For example, what is the effect of selectively slowing down an agent's ability to predict the growth of the fire? Can it compensate, say by allowing more slack time in its plans? Can it switch to different kinds of plans, ones that don't require as much prediction? The PHOENIX agents are not currently capable of such reasoning, but may be able to in the future.) This kind of experiment could help a researcher concentrate on aspects of the agent's cognition that will best improve its performance.

8 CONCLUSION

The goals of the PHOENIX project were to investigate issues in Real-Time Artificial Intelligence; that is, agents acting intelligently under time pressure. Consequently, it was important to define a correspondence between the amount of thinking that an agent does and the passage of time in the simulation. The initial implementation of PHOENIX used CPU time for that correspondence, but that suffered from problems of platform-dependence (the same simulation would run differently on different machines), variability (simulations could not even be reliably replicated on a single machine) and interference (adding code to instrument the simulation would affect its behavior). These problems are not only practical impediments to debugging, but they prohibit, for example, experiments in which simulation states are replicated.

To provide noise-free integration of thinking agents, the MESS and TCL components were implemented. MESS integrates multiple thinking agents and environmental processes (such as fire and weather) into a extensible, object-oriented discrete-event simulator and supports interactions among the activities of the agents and world processes. TCL is targeted to thinking agents and defines a platform-independent mapping between amount of computation and simulation time. Furthermore, this mapping is under user control in many ways, so that the duration of an agent's thinking (or aspects of that deliberation) can itself become an experimental variable.

The time pressure in PHOENIX comes not from artificial deadlines, but a necessary response to changing conditions in a dynamic world. Consequently, sensing, reacting, planning and communication are all essential to a PHOENIX agent. An agent must notice and react to changes in the world, plan ways to deal as team member with large problems for which individual reactions are insufficient, and communicate those plans to other agents. Simulation testbeds are the only effective way to implement, debug and test such time-pressured agents, since they must be empirically run and tested under controlled and repeatable conditions.

ACKNOWLEDGMENTS

This work was supported by ARPA/Rome Laboratory under contracts F30602-93-C-0100 and F30602-95-1-0021, and by NTT Data Communications Systems Corporation. The U. S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notice contained herein.

REFERENCES

Allen, J., J. Hendler, and A. Tate (Eds.) 1990. *Readings in Planning*. Morgan Kaufmann.

Anderson, S. D. 1995, February. *A Simulation Substrate for Real-Time Planning*. Ph. D. thesis, University of Massachusetts at Amherst. Also available as Computer Science Department Technical Report 95–80.

Bratley, P., B. L. Fox, and L. E. Schrage. 1983. *A Guide to Simulation*. Springer-Verlag.

Bratman, M., J. Israel, and M. Pollack. 1988. Plans and resource-bounded practical reasoning. *Computational Intelligence 4*: 349–355.

Cohen, P. R., M. L. Greenberg, D. M. Hart, and A. E. Howe. 1989, Fall. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine 10*(3): 32–48.

Gabriel, R. P. 1985. *Performance and Evaluation of Lisp Systems*. MIT Press.

Greenberg, M. L. and D. L. Westbrook. 1990. The Phoenix testbed. Technical Report COINS TR 90–19, Computer and Information Science, University of Massachusetts at Amherst.

Keene, S. E. 1989. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley.

Rönngren, R. and R. Ayani. 1997, April. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation 7*(2): 157–209.

Russell, S. and P. Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall. ISBN 0-13-103805-2.

Stankovic, J. A. 1988, October. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer 21*(10): 10–19.

Steele Jr., G. L. 1990. *Common Lisp: The Language* (second ed.). Digital Press.

AUTHOR BIOGRAPHY

SCOTT D. ANDERSON is an assistant professor in the Computer Science Department at Spelman College. He received a B.S. in Computer Science and a B.A. in English Language and Literature from Yale University in 1983. He earned a M.S. and a Ph.D. from the University of Massachusetts at Amherst. His research interests is in the empirical evaluation of planners in complex, time-pressured environments. He is a member of the American Association for Artificial Intelligence (AAAI) and the Association for Computing Machinery (ACM).