In *Working Notes of the Symposium on Integrated Planning Applications, AAAI-95 Spring Symposium Series.*

Segregating Planners and Their Environments

Scott D. Anderson and Paul R. Cohen

Computer Science Technical Report 95-57

Experimental Knowledge Systems Laboratory
Computer Science Department, Box 34610
Lederle Graduate Research Center
University of Massachusetts
Amherst MA 01003-4610

Abstract

By implementing agents and environments using a domain-independent, extensible simulation substrate, described in this paper, agents will have clean interfaces to their environments. This makes it easier for agents to be plugged into other environments that have been similarly defined. If agents can interact with multiple environments, their behaviors and the associated experimental results will be more general and interesting.

This research is supported by ARPA/Rome Laboratory under contract F30602-93-C-0100.

1 Introduction

We argue against a tight integration of planners and their environments. This is not because we think integration is a bad thing, but because we think it's important to run planners in a variety of environments. An *AI Magazine* article [6] describes a number of AI environment simulators, including costs and benefits. By running an agent in a variety of environments, we can hope to get the benefits of all worlds.

To make it easier to mix and match planners and environments, they must have compatible interfaces. The worst case is illustrated in figure 1(a), where planners are tightly integrated with their environments, but completely unportable. The ideal case, which AI should strive towards, is illustrated in figure 1(b), where both the planner and agent interface to a domain-independent substrate. This paper describes a domain-independent substrate that partially addresses these concerns.

2 Discrete Event Simulation

All simulators for AI planning that we are aware of use discrete event simulation. Therefore, planners and environments can meet on the common ground of a domain-independent discrete event simulator. We are designing and implementing such a simulator, called MESS: Multiple Event Stream Simulator.

Environments are implemented by defining event types as classes in the Common Lisp Object System (CLOS). These classes all inherit from an abstract class called **event**, and must provide methods to:

- report the *time* at which the event is to occur. This is so that the event can be properly interleaved with other events.
- *realize* the event, that is, modify the state of the world so that the event "happens."
- *illustrate* the event by executing whatever graphics code is appropriate for the event. (The illustration of an event is separate from the realization because it's common to turn off graphics when running large experiments in "batch mode.")

For example, in a domain like PHOENIX, which simulates forest fires burning in Yellowstone National Park [2,5], one type of event might be the burning of a "fire cell" (a grid is laid over the continuous map to discretely represent the fire). The realization of that event would be to ignite that cell in the array representing the fire. The realization might also check the state of the world to determine whether any agents were in the cell, in order to signal their deaths, if necessary. The illustration of the event might draw a small square of red on the map window on the user's screen. An example of a TILEWORLD event might be the appearance or disappearance of a tile or hole, or the moving of a tile by an agent [8,13]

The simulation literature usually puts simulator designs into one of the following two categories [1, p. 13]:

event oriented
: Each event determines what subsequent events follow from it (that it "causes"). Thus, each event type would have a function to produce the successors of that event; those successor events are scheduled for later realization.

process oriented
: Each event is produced by a process, and that process determines the subsequent events. The process often describes the lifetime of a simulation object, such as a customer.

MESS is process-oriented, because it is convenient to view as processes things like fire, weather, and particularly an agent's thinking. The representations of processes are called *event streams*.

A simulation in MESS is equivalent to a loop in which an "engine" takes the next event from whatever event-stream has it, realizes it, illustrates it (if desired), and loops. The next event is simply the event that is the nearest in the future, as determined by timestamps. Since each event stream produces events in chronological order, the engine can be viewed as merging those streams to yield a single stream in chronological order. The engine is responsible for breaking ties when two events are scheduled for the same time; in addition, events may specify protocols for handling conflicts. The simulation ends when all event streams are exhausted, or when a special "end of simulation" event is realized. Figure 2 depicts the architecture of MESS.

MESS defines some classes of event streams that can take a user's code and produce a stream of events; the user can define new kinds of event streams if desired. A simple and general class is a *function* event stream, where the user supplies a function that generates successive events. Another is a *script* event stream, where the user provides a simple list of events that should happen at particular times during the simulation. The most interesting kind of event stream is one that takes agent code and

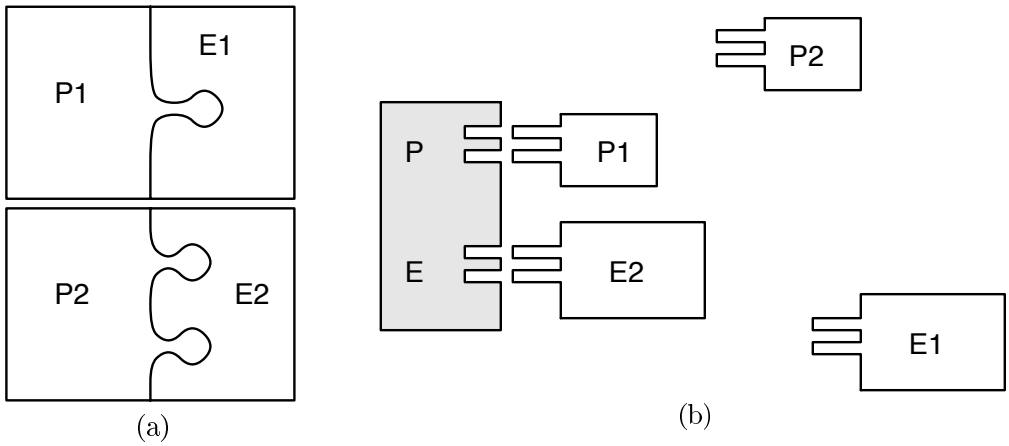

Figure 1: On the left is pair of planners that are tightly integrated with their environments; they are also mutually incompatible. On the right is an ideal case where planners and environments can easily be connected by plugging them into a domain-independent substrate, shown in gray.

executes it, thereby allowing the agent to think and act in the environment. The next section discusses this kind of event stream.

3 Real-time Agents

An agent interacts with the environment by producing events, just like any other event stream. For example, in a gridworld simulator like MICE [3,10,11], an agent might produce movement events such as "up" or "left." The agent might also produce sensor events, which will return information about the state of the world (possibly after a time delay).

Between the times that the agent produces these sensor and effector events, it thinks and plans. Does simulation time pass while the agent thinks? If so, the world processes, such as fires burning or tiles disappearing, continue relentlessly. The agent then faces a tradeoff between thinking time and plan quality: how much time should it spend thinking and what is the expected quality of the resulting plan? When an agent faces *time pressure* on its thinking, we call it a real-time agent. We use "real-time" in the sense that the usefulness of a result depends not only on its intrinsic attributes but also on its timeliness [7,14,15].

MESS is designed to support real-time agents. It does this by requiring agents to be written in a general agent language, which is essentially Lisp except that there is a database of *duration models*. A duration model tells how much time passes while executing some part of the agent's code. Duration models can be tied to Lisp primitives, so that the simulation clock is advanced depending on how much Lisp executes, or they can be tied to higher level functions of the agent, abstracting away from implementation details and substituting a different duration.

The MESS approach to supporting real-time agents differs from majority of real-time AI testbeds, which employ CPU time to measure the amount of agent code executed. For example, the PHOENIX testbed uses CPU time, transforming those units into simulation time units via a constant called the real-time knob. Modifying the setting of the knob varies the amount of time-pressure on the agent by altering its thinking speed relative to the rate of change of the world. In experiments with PHOENIX, we found that CPU time is difficult to measure precisely and repeatably, and it differs between computers, even those of the same CPU type. Consequently, our experiments suffered from unwanted sources of variance. These experiences were shared by the users of TILEWORLD [13], which is why they, too, switched to a platform-independent way of advancing time in agent code [8]. Their solution, however, requires committing to the IRMA agent architecture. Our solution only requires using Lisp.

MESS differs from commercial simulation software primarily in allowing the execution of arbitrary code to determine how much time passes in a process. Commercial software usually requires describing the events in a process and the *delays* between each event, where the delays are given as explicit constants. In a MESS agent, the delay is determined by the amount of computation, or thinking, that the agent does, where the thinking is implemented by ordinary Lisp code. Thus, MESS gives the researcher a great deal of flexibility in how to implement an agent, while maintaining a realistic measure of its

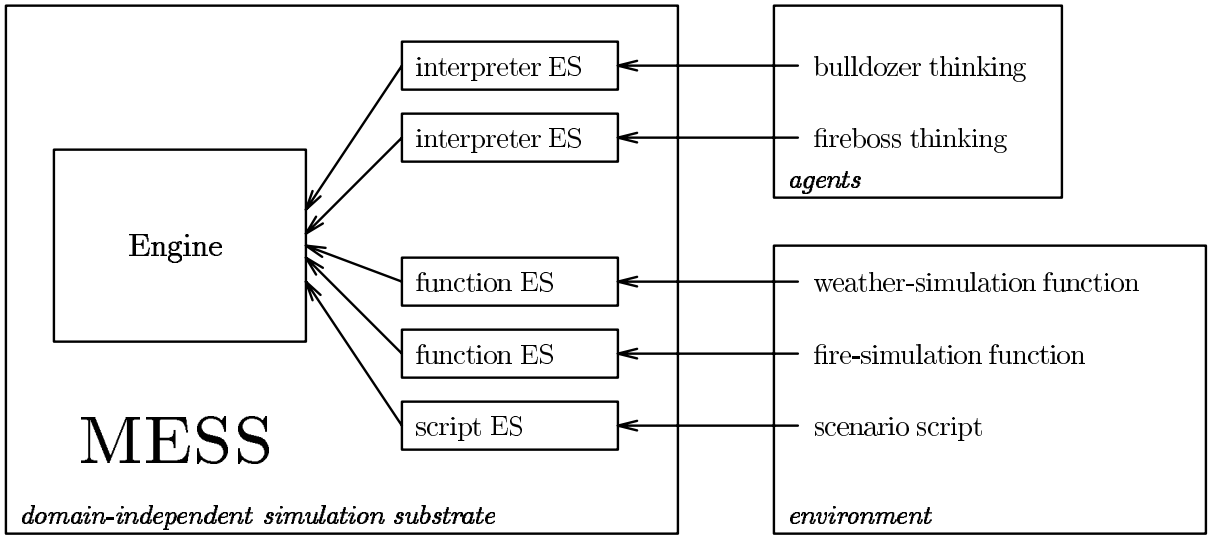

Figure 2: The design of MESS: the central engine realizes events from instances of different kinds of event stream (ES). The arrows starting at the ES boxes indicate events flowing into the engine to be realized. Arrows pointing to the ES boxes indicate the mapping from domain-dependent agents and environment processes to the domain-independent MESS substrate. An agent's thinking is modelled by an event stream that runs the agent's code to produce events.

computation time.

Figure 2 shows two agents implemented using MESS. The code that defines the agents is domain-dependent, but the ability to execute that code is provided by MESS via an event stream that interprets agent code. For efficiency, the agent code is compiled as much as possible. The interpreter event stream executes the agent code, saving state as necessary to allow the execution of other event streams to be interleaved. Thus, all event streams progress concurrently.

The duration database also allows introspection by the agent, so that it can be aware of the computation time of its thinking activities and take that into account when planning and scheduling. The database thereby supports research in deliberation scheduling.

4 Status

MESS is currently under development. An initial implementation will be available by early spring of 1995. In addition, the PHOENIX testbed will be re-implemented atop the MESS substrate. This port is being done for several reasons:

- to take advantage of the reduction in variance, the improvement in experimental control and the ability of the agent to introspect about its computation time;
- to work out the details of how agents and environments interface to the substrate
- to provide an initial library of classes of events, event streams, and other objects
- to make the PHOENIX testbed available on stock hardware, both for those who want to work with it directly and as a starting point for those wanting to build similar environments.

The re-implementation of PHOENIX will probably be available in late spring of 1995.

5 Conclusion

We have described a domain-independent substrate for real-time AI simulators. Environments are defined using event types and event streams supplied by MESS or by extending the existing set of types via object-oriented programming. This ability to extend and specialize existing environment implementations will allow greater sharing of environments among researchers. (The MICE testbed also uses this idea, although at a higher level, since it commits to a four-connected gridworld: agents can only move up,

down, left and right.) Agents are defined by writing Lisp code that implements thinking, sensing and acting, with the latter two abilities implemented by generating events. The duration of thinking code is entirely machine-independent (hence noiseless and repeatable) and entirely under the control of the user.

Our approach does not solve all compatibility issues, since agents necessarily think in terms of the environment they intend to operate in. Consider just the agent's and environment's model of space. There are gridworlds, such as MICE and TILEWORLD, graphworlds, such as TRAINS [9] and TRUCKWORLD [6, 12], and worlds with continuous space, such as PHOENIX or ARS MAGNA [4]. A PHOENIX agent cannot be put into TILEWORLD and work, because its movement commands are all wrong. Nevertheless, we believe that our approach will eliminate many of the problems in running an agent in a different environment.

It's important to test agents in different environments because only then can we know how general our results are: Is an agent effective only in certain kinds of environments? Which ones? What features of the environments help or hinder the agent's performance? We believe agents should be well integrated into their environments, but let's also keep them somewhat segregated.

Acknowledgments

This work is supported by ARPA/Rome Laboratory under contracts F30602-91-C-0076 and F30602-93-C-0100 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 hereon. The authors are also grateful to one of the anonymous reviewers for their helpful comments.

References

[1] Paul Bratley, Bennett L. Fox, and Linus E. Schrage. *A Guide to Simulation*. Springer-Verlag, 1983.

[2] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, Fall 1989.

[3] Edmund H. Durfee and T. A. Montgomery. MICE: A flexible testbed for intelligent coordination experiments. In L. Erman, editor, *Intelligent Real-Time Problem Solving: Workshop Report*, Palo Alto, CA, 1990. Cimflex Teknowledge Corp.

[4] Sean P. Engelson and Niklas Bertani. Ars Magna: The abstract robot simulator manual, version 1.0. Technical Report 928, Yale University, New Haven, CT, October 1992.

[5] Michael Greenberg and David L. Westbrook. The Phoenix testbed. Technical Report COINS TR 90–19, Computer and Information Science, University of Massachusetts at Amherst, 1990.

[6] Steve Hanks, Martha E. Pollack, and Paul R. Cohen. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 13(4):17–42, 1993.

[7] Eric J. Horvitz, Gregory F. Cooper, and David E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the Eleventh International Joint Conference on Artificial Itelligence*, pages 1121–1127, 1989. Detroit, Michigan.

[8] David Joslin, Arthur Nunes, and Martha E. Pollack. TileWorld user's manual. Technical Report 93-12, Department of Computer Science, University of Pittsburgh, 1993. Contact `tileworld-request@cs.pitt.edu`.

[9] Nathaniel G. Martin and Gregory J. Mitchell. A transportation domain simulation for debugging plans. Obtained from the author, martin@cs.rochester.edu, 1994.

[10] Thomas A. Montgomery and Edmund H. Durfee. Using MICE to study intelligent dynamic coordination. In *Second International Conference on Tools for Artificial Intelligence*, pages 438–444. IEEE, 1990.

[11] Thomas A. Montgomery, Jaeho Lee, David J. Musliner, Edmund H. Durfee, Daniel Damouth, Young-pa So, and the rest of the University of Michigan Distributed Intelligent Agents Group. MICE users guide. Technical report, Department of Electrical Engineering and Computer Science, University of Michigan, March 1994.

[12] D. Nguyen, Steve Hanks, and C. Thomas. The TRUCKWORLD manual. Technical Report 93-09-08, University of Washington, Department of Computer Science and Engineering, 1993. Contact `truckworld-request@cs.washington.edu`.

[13] Martha E. Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189. American Association for Artificial Intelligence, MIT Press, 1990.

[14] Stuart J. Russell and Eric H. Wefald. Principles of metareasoning. *Artificial Intelligence*, 49:361–395, 1991.

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