On-line Planning Simulation

Scott D. Anderson
Spelman College
Atlanta, GA 30314-4399
anderson@auc.edu

Paul R. Cohen
Experimental Knowledge Systems Laboratory
University of Massachusetts
Amherst, MA 01003-4610
cohen@cs.umass.edu

Abstract

MESS is a substrate for building simulation environments suitable for testing plans and on-line or real-time planners. The article describes the design of MESS, how simulations are built and how on-line planners integrate with the substrate. MESS supports *activities*, defined as processes over some time interval, and *interactions* between activities and other simulation events. MESS interfaces with TCL, which is a portable, extensible definition of computation time, enabling MESS to be used for platform-independent simulations of real-time planning. MESS has been used to re-implement the PHOENIX testbed, which simulates forest fires and planning for fire-fighting agents.

The Need for Simulation

As planners become more sophisticated, they will solve increasingly large planning problems involving, for example, the movement and actions of thousands of vehicles, over many hours and under changing conditions. It is extremely difficult to inspect such elaborate plans and determine, for example, their probability of success, the extent to which their goals will be satisfied, and so forth. Nevertheless, such evaluation is critical to a scientific understanding of how and how well a sophisticated planner works.

We believe simulation is necessary to evaluate planners: plans are run many times in the space of conditions that they were meant to handle, and various dependent variables are measured and statistically analyzed. Furthermore, simulators enable the planner to be *on-line*: it can be an agent in an ongoing environment, monitoring the progress of the plan and making additions or corrections as necessary. An on-line planner can even scrap a failing plan or sub-plan and replan (Howe 1993). If the thinking time of the planner is limited, so that there is time pressure on its thinking, the on-line planning becomes *real-time* planning.

A number of simulation environments already exist to support research in on-line and real-time planning (Hanks, Pollack, & Cohen 1993). Some of these simulators are quite domain-specific, such as our own PHOENIX testbed (Cohen *et al.* 1989), which simulates forest fires in Yellowstone National Park. Other examples are TRUCKWORLD (Hanks, Nguyen, & Thomas 1992) and TRAINS (Martin & Mitchell 1994), where trucks or trains move cargo in a graph of depots, cities and towns. Other testbeds are much more domain-independent, such as the MICE testbed (Durfee & Montgomery 1990), in which agents move in a generic gridworld.

With such a plethora of testbeds, there have been many good ideas and much duplication of effort. In addition to implementing the domain dynamics, these testbeds all have to solve the fundamental issues of simulation, such as managing events from many sources and getting them to occur in the correct order. They have to deal with the interface between planners and the environment, and often that interface is not well defined. Will the testbed have multiple agents, and how is their concurrent thinking coordinated? How is thinking time represented and integrated into a discrete event simulation? The solutions are often not as flexible and powerful as the planning community might like. Because of these many design decisions, these testbeds are often not as easily shared as their authors intended. This article describes our work on MESS (Anderson 1995), which we believe captures the best of the common, domain-independent aspects of these simulators, and improves the representation of thinking agents and the measurement of time.

MESS (Multiple Event Stream Simulator) is best described as a simulation *substrate*, rather than a simulated environment in itself. It makes no domain commitment because it works with abstractions called "events," "event streams" and "activities," among others. One builds a simulation environment in MESS by defining the events that happen, thereby changing the state of the world, and defining the event streams that produce those events. The MESS substrate takes care of synchronizing all the events so that the simulation

unfolds in the correct way, with processes interacting as they should. Our goals in designing MESS were (a) domain independence, (b) planner independence, meaning that we pose little constraint on the kind of planner that can be integrated with MESS, (c) extensibility by the user, (d) portability to any Common Lisp platform, and, most importantly, (e) a flexible, platform-independent definition of planning duration, so that real-time simulations will have those properties.

Mess Design

MESS makes no commitment to a domain but instead supplies the materials to build any domain, namely *events* and *event streams*. For example, the ignition of a firecell is an event in PHOENIX, the appearance of a tile is a TILEWORLD event (Joslin, Nunes, & Pollack 1993; Pollack & Ringuette 1990), and a train traversing a route is an event in TRAINS. Events are defined in MESS using CLOS, where the user supplies code that determines 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, Fox, & Schrage 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 separate processes in the simulation. The representations of processes are called *event streams*. Event streams are also defined using CLOS, 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" various kinds. A very general kind of event stream (ES) is a *function* ES, where a 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.

The MESS engine is so called because it controls all the events and event streams, and it invokes the realization of events. Discrete event simulators go from state to state in discrete steps, which I have called *advancing* the simulation. Figure 2 presents pseudo-code for the algorithm to advance the simulation. Each time

```
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 simulation is advanced, exactly one event is realized.

The event to be realized is whichever is nearest in the future. In a queuing simulation, if we have a customer arrival scheduled for time 18 and a departure scheduled for time 13, the departure must obviously come before the arrival. The simulation literature has several terms for the data structure holding these events; we call it the "pending event list" or PEL. The exact representation used for the PEL is not important here; you may think of it as a totally ordered list of events. When an event is scheduled, it is inserted into the PEL in the correct place; when the simulation is advanced, the first event in the PEL is realized and removed from the list.

In MESS, there can be two kinds of object in the PEL: an event or an event stream (ES). In some ways, an ES can be treated just like an event, because it always has a particular event that is the next event in the stream. If we think of an event as a sheet of paper, an ES is like a pad of paper: it has a bunch of sheets, only one of which shows at a time. The PEL in MESS contains either individual events, or event streams. In practice, in the simulators implemented using MESS, most of the objects in the PEL are event streams.

Let's look briefly at the pseudo-code to see how MESS works. (A more detailed description is available in the first author's dissertation (Anderson 1995).) The primary objective of the engine is to realize events, which we see in the center of the algorithm. If the first thing in the PEL is an ES, the engine must make the

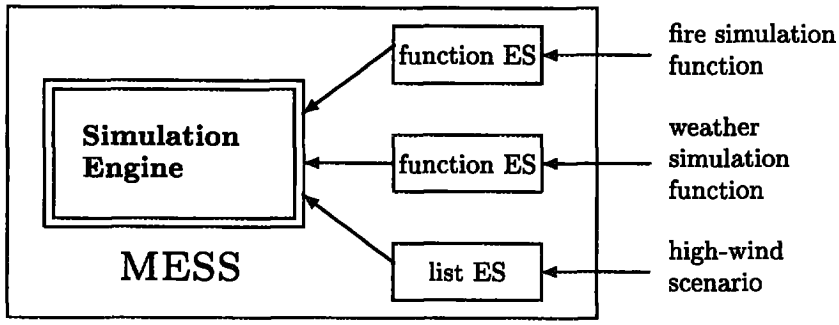

Figure 1: The architecture of the MESS simulation substrate. MESS is structured as a central engine, driving instances of different kinds of event stream (ES). MESS itself is domain independent; the streams listed at the right (weather, fire, scenario) are examples drawn from the PHOENIX domain.

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 testbeds 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. Indeed, the realize and illustrate methods, which operate on events, *must* be specialized, since their default behavior is to do nothing. The peek and pop methods operate on event streams; as mentioned above, several general event stream classes are implemented in MESS already. The user can arrange for particular events to happen during a simulation by using the *list* event stream. The *function* ES classes run a function, supplied by the user, to generate an event either during the peek or pop operation. We've found it straightforward to implement many kinds of processes using just these event streams, but the protocol is designed for extensibility by the implementer of a simulation.

Several minor steps in the pseudo-code deserve mention. The "every event" step executes all the code in a list supplied by the user at the start of the simulation, and so it's easy to 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 "write out" step saves every event to a file, so that a simulation can be analyzed or replayed if desired. Finally, the protocol includes steps to check for interactions and change activities; these are discussed in the next section.

Activities and Interactions

Events are "point-like," in that they happen at a moment in time. For example, a customer arrives in a queuing simulation, or a tile disappears in TILEWORLD. However, many kinds of simulations involve things that happen over an interval of time; these are called *activities* in MESS. For example, a train travelling from one station to another would be represented as an activity. Activities are conceptually 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 many simulators, this collision would never be noticed, but MESS keeps track of all current activities and checks for interactions.

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.

The interaction can affect either the activity or the 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. The movement activities of two vehicles might result in a collision, with both activities affected by the interaction. Complex interactions like these are the bane of planners, so it's crucial that we challenge our sophisticated planners with these situations.

Each activity occurrence is implemented as a single object, an instance of a sub-class of an event. This representation allows an easy sharing of information that might be needed for the realizations at the start and finish of the activity. It also yields a single object for specializing the *interaction* function. The engine takes care of "informing" the object that its role as the beginning of the activity is over and it now represents the end of the activity; this is the purpose of the "change activity" step in the pseudo-code in figure 2.

Planners

An on-line or real-time planning agent is integrated into a MESS-based simulation as just another event stream. The agent discovers the state of the simulation by producing sensory events, and it acts by producing effector events. Thus, from the viewpoint of the MESS engine, a thinking agent appears to be the same as any event stream, obeying the same *peek* and *pop* protocol.

Some planners can certainly be implemented using the pre-defined *function* event streams, but because the function is executed from scratch each time, there is no continuous "stream of thought." Therefore, most agents will want to use the pre-defined class of *thinking* event streams. These event streams run the planner as a *co-routine*, switching control back to the MESS engine whenever the planner produces an event, since an event signifies interaction with the simulation, and so the simulation must be brought up to date.

The MESS engine lets the agent ES have its turn when it needs to get the next event from that ES, and the ES runs until it computes an event, whereupon it returns control to the engine. To be precise, an agent event stream gets its turn when it is *popped*, and when it computes an event, the event becomes the pending event in the event stream. The timestamp on the pending event determines when the event is realized and when the ES runs again.

How is the timestamp on the pending event calculated? Note that this is not a question we have considered before. We assumed that the event streams compute the timestamp in domain-specific ways, involving, for example, models of how fast vehicles move or fire spreads. With a thinking ES, we want the timestamp on the event to be determined by the *amount of computation* that has occurred during this turn. That is, the computation of the timestamp on the next event in the agent is a side-effect of its getting a turn to think: the agent thinks until it gives an event to the substrate for realization, and the amount of thinking determines the timestamp of the event.

Thinking time only matters for real-time agents. A planner that is merely on-line may think for as long as it wants. It must therefore determine in some other way when it will get another chance to think. It may, for example, simply get to run every five simulated minutes. While the MESS substrate can easily accommodate on-line agents, it is particularly designed for real-time agents, as the next section on thinking time will show.

Before discussing thinking time, let's clarify the integration of planning agents with an example. Consider two computer chess programs playing one another. They play the following game, a classic fool's mate.

1.	P-K4	P-K4
2.	B-B4	B-B4
3.	Q-B3	N-QB3
4.	QxP mate	

For this example, let's fix the times that the agents find their moves as follows.

1.	20	40
2.	70	90
3.	110	140
4.	160	

White starts the game, thinks for 20 time units, and creates an event representing its first move. It uses the MESS function `make-event-during-thinking` to make the event and transfer control back to the engine. Its thinking will be resumed after the event takes place, but the engine must first simulate all events prior to time 20. Consequently, the engine starts Black's event stream, and its co-routine runs from time 0 to time 40, the first 20 of which coincide with White's turn. Given Black's move scheduled at time 40 and White's scheduled at time 20, the engine realizes White's move, and gives White control again, to think of its second move. The transfer of control is depicted in figure 3.

It's possible to define intervals of an agent's thinking as *interruptible* activities, allowing the kind of modeling described in the previous section. Indeed, there is a role for interruptible thinking in the chess example. Suppose that, in their search for a move, the players lower their threshold for move quality when it's their own turn, so that they are more willing to accept a candidate move, thereby decreasing the time until they find a move. To change the threshold, an agent's thinking must be interrupted by the opponent's move. Furthermore, a real chess-playing agent would inter-

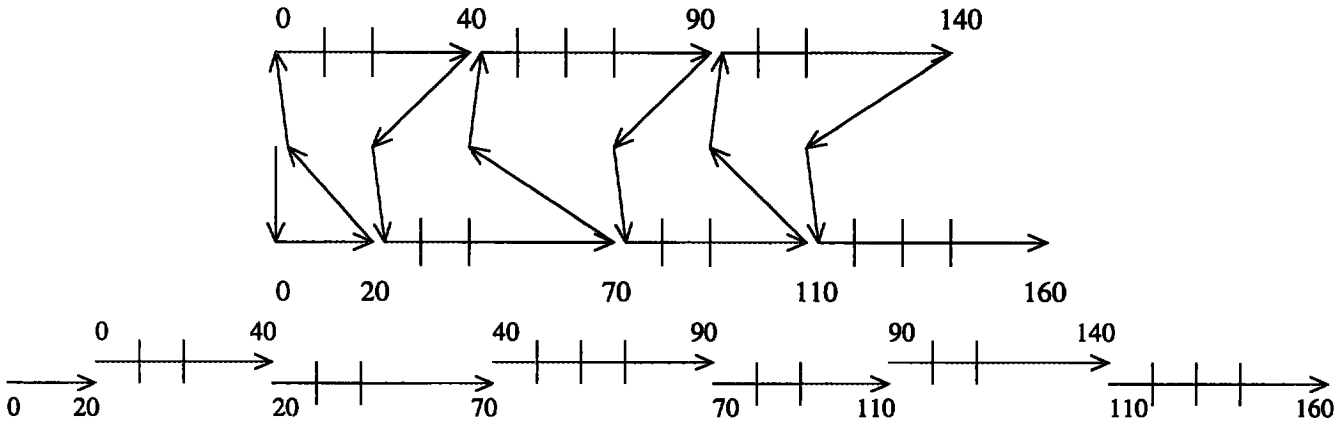

Figure 3: The chess players thinking in parallel. White is below and Black above. Control starts in the center with the MESS engine, which transfers control to White, which thinks for 20 time units and transfers control back to the engine. The engine then transfers control to Black, which thinks from time 0 to time 40 and transfers control back, and so on. The short lines perpendicular to the arrows indicate *interruption points*.

rupt its thinking to observe its opponent's move and take that into account.

In the MESS implementation of this example, when the opponent makes its move, this move interrupts a move-searching loop and changes to another loop that uses a lower acceptance threshold for moves. The interruptible loop is wrapped with a Lisp form marking it as interruptible, and specifying an "interruption handler" to be executed should an interruption occur. Places in the loop where interruptions are allowed are marked with an `ip` form, noting that location as an interruption point. (Requiring the `ip` form avoids problems with critical sections being interrupted, thereby corrupting data structures or control flow.) These interruption points have been depicted with short tick marks in figure 3.

How is interruptible thinking activity implemented? To understand this, we have to take a slightly different view of time, because thinking that happens simultaneously in the real world must happen sequentially in the simulation. The thinking of the chess agents happens as follows:

1. White thinks until it comes up with its first move, at time 20. The move event is scheduled but not realized.
2. Black's interruptible thinking activity starts (with its clock at time 0) and is allowed to think until the interruption occurs at time 20.
3. White's move is realized at time 20, interrupting Black's thought activity.
4. Black switches to its other thinking loop (the one with the lower threshold), and thinks until time 40, when it finds a move. Again, the move is scheduled but not realized.
5. White's interruptible thinking activity starts (with its clock at 20) and thinks until 40, when it is interrupted.
6. Black's move is realized at time 40, interrupting White.
7. and so on, as in the first step.

The MESS engine takes care of orchestrating this, so that agents can be easily implemented.

The Duration of Computation

As mentioned in the first section, a number of simulators for real-time planning already exist, so MESS is not unique in this regard. Most of those simulators, however, use CPU time to measure the amount of computation performed by an agent, mapping CPU time into the amount of simulation time that passes while the agent thinks. This approach is intuitive and straightforward to implement, but it has a number of drawbacks. First, it is platform-dependent, so a simulation will run differently on a different CPU, operating system, Lisp implementation, or even a different release of the Lisp compiler. In fact, a simulation will behave differently from run to run even if none of these factors change, due simply to variability in CPU time. (Indeed, this variability can be quite striking (Anderson 1995).) One of the few simulators to avoid CPU time is TILEWORLD, and it's instructive to see why they abandoned it:

> The noise in the data comes, we believe, largely from our decision to use actual CPU-time measurements to determine reasoning time. If we wish to get the cleanest trials possible, we may need to use a time estimate that does not depend on the vagaries of the underlying machine and Lisp system. (Pollack & Ringuette 1990)

Later implementations of TILEWORLD incremented the simulation clock by a fixed quantity for each iteration of the IRMA agent architecture.

MESS also abandons the CPU time approach, but our solution makes no commitment to an agent architecture. Instead, it interfaces with agents implemented in Timed Common Lisp (TCL). In TCL, every primitive of the Common Lisp language has been assigned a duration, defined in arbitrary units, and each primitive advances the clock as a side-effect of its execution. Each agent has a separate clock, and the TCL primitives advance the clock of the agent that is currently thinking. Therefore, the time that an agent thinks is determined by the number and kinds of primitives it executes. In addition, each agent has a parameter that is multiplied by the total of these increments, and by varying this parameter, time moves more or less swiftly for the agent. Consequently, the parameter is called the "Real Time Knob." (The MICE simulator has a similar parameter.)

Timed Common Lisp

TCL is implemented in a Lisp package, allowing it to define a twin to every Common Lisp function. Furthermore, a user can define additional primitives that may be more relevant to the domain and agent. For example, the chess agents might think by executing a `consider-move` primitive and other domain-specific functions. Adding a TCL primitive is easily done by specifying its semantics (using Common Lisp) and a model of its time cost.

Models of the time cost of the functions are stored in a "duration database." The database entry for each primitive specifies (1) a duration model and (2) a proportionality constant. The duration model describes the number of operations performed by the primitive as a function of its arguments and the program state. For example, the `:constant` model specifies that the primitive does one operation regardless of its arguments, hence taking constant time. Another model is `:length`, which specifies that the number of operations equals the length of the primitive's first argument. The `:length` model is used for a number of list-manipulation functions. Many other duration models are implemented in TCL, and this set is extensible by the user, so that domain-specific duration models can be defined, say for anytime algorithms (Boddy & Dean 1989) or deliberation scheduling (Russell & Wefald 1988; 1989). Of course, not every constant-time function takes the same amount of time, and similarly for other duration models. Therefore, the database entry includes a proportionality constant, to be multiplied by the value of the duration model to yield a duration.

Free Operations

When implementing an agent whose thinking time should be "on the clock," we can implement it in the TCL package, and any function we refer to will increment the clock in some way. What if we don't want to increment the clock? Suppose that we are inserting some code to help in debugging the agent, to collect data, or to measure performance or quality. If that code were to affect the behavior of the agent, by affecting the duration of its thinking, the ability to do empirical research would be greatly impaired. This can easily happen if thinking time is modeled by CPU time, and we have seen small variations in thinking time get compounded into markedly deviant behavior later in a simulation (Anderson 1995). TCL is designed to avoid this malady, thereby supporting empirical research.

Querying the Database

Agents will want access to the duration database, so that they can reason using the durations of various primitives. For example, a scheduler or meta-reasoner will need this information. This is easily done with the TCL function `primitive-duration`, which needs to know the value of the Real Time Knob and the arguments to the primitive. The arguments are necessary because, in general, the duration of a primitive can be defined to depend on its arguments. If it does not depend on its arguments, they may be omitted when using `primitive-duration`. The function then returns the time cost of the primitive. This ability to be aware of the duration of its reasoning can allow a real-time agent to adjust its planning to the time pressure of the environment.

Performance

There is a practical side to MESS and TCL: how efficient are they? While the bulk of time spent in running a simulation is spent in executing domain-specific code, we still want to minimize the overhead spent in the engine and in TCL.

Engine Performance

We can measure the performance of the MESS engine (that is, the **advance** function describe above) in events

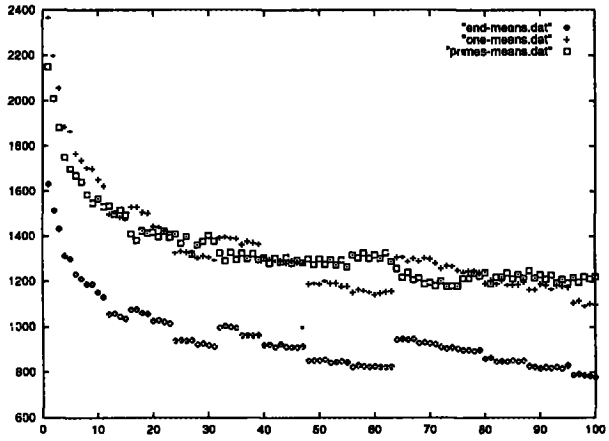

Figure 4: The speed of the engine, plotting mean performance in events per second.

per second. Much of what the **advance** function does is fixed-cost overhead. Managing the PEL, however, depends a lot on the user's simulation code, particularly (1) the number of event streams, which determines the size of the PEL, and (2) the pattern of timestamps produced by the event streams, which determines where they will be inserted into the PEL. Sets of trials were run varying both these factors, with the number of event streams varied from one to a hundred, and the timestamps varied over three abstract patterns:

- events are periodic and with the same period in all streams, so events are typically scheduled in the middle of the PEL.
- events are periodic but with different periods in over the event streams, so events are inserted in different places in the PEL.
- events are periodic but the times are staggered so that events are always inserted at the end of the PEL. This is the worst case for our PEL representation (a heap).

We would expect the first two cases to be representative of MESS's performance in typical use.

The data for the three experiments are presented in figure 4, which plots the mean for each condition. In the "average" cases, the performance with a hundred event streams is over 1000 events per second. This graph clearly shows the lower performance in the condition that we expected to be the worst case. The graphs also show the diminishing decline behavior that we would expect of an engine whose performance is dominated by the $O(\log n)$ PEL representation. We can expect that MESS will continue to scale well to larger simulations.

Duration Performance

TCL's advancing of the clock entails an inevitable overhead, because 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 simple functions like `car` and `cdr`, incrementing the clock is a significant slowdown. On the other hand, high-level functions such as `move-evaluation` in the chess domain is barely slowed down by incrementing the clock. In addition, unfortunately, the speed also depends on the quality of the Lisp compiler—a good compiler can open-code much of the incrementing code using type-specific arithmetic instructions.

We can, however, take timings of standard benchmark programs, to get an idea of how much TCL slows the code down. We compiled and executed Gabriel's suite of benchmarks (Gabriel 1985) in both plain Common Lisp (Harlequin Lispworks on a DEC Alpha) and in TCL. As expected, the overhead varied across the suite, but the cost of using TCL appears to be 20 to 120 percent, depending on the benchmark program. Naturally, we hope that realistic AI programs will tend more towards the lower end of the range. Remember that all of Gabriel's benchmarks use fairly low-level functions, and so we can view these results as an upper bound on the cost of using TCL.

Status

MESS and TCL are fully implemented. Indeed, they've been used to re-implement the PHOENIX testbed, so that it now runs portably on Common Lisp implementations. The PHOENIX graphical user interface is implemented using the Common Lisp Interface Manager (CLIM). We hope that other researchers will try their planners in the PHOENIX environment. Failing that, we hope that they will use MESS and TCL for implementing new simulation environments. Having a common substrate will make it easier for planning researchers to share simulators and planners, and we looking forward to more comparative, empirical work.

All of this software is available by anonymous FTP from `ftp.cs.umass.edu` in clearly named subdirectories of `/pub/eksl/`. Documentation is included, as are well over a dozen "miniatures": simple MESS-based simulations that demonstrate how particular features are used. For example, several variations on the chess example described above are implemented as miniatures and distributed with MESS.

Summary

MESS is a domain-independent substrate for implementing simulation environments, which we feel are necessary for evaluating complex plans and planners. MESS makes no commitments to a domain or a planner architecture, so it should be usable by anyone working in Common Lisp. MESS includes support for representing ongoing activities and implementing interactions between activities and concurrent events.

MESS combines with TCL to support empirical research in real-time planning because TCL provides a platform-independent "virtual machine" for executing the Lisp code that implements an agent's thinking. TCL is flexible and extensible, so that duration models can be modified, and new domain- or planner-dependent duration models can be defined.

MESS captures much of the important core functionality of a simulator for real-time planning, relieving researchers from implementing this core. Simulation environments built using MESS will have a clear interface between agent and environment, making it much easier to plug in a different agent or environment and test hypotheses about relative behavior and performance. It's for these reasons that we believe MESS will be helpful to the planning community.

Acknowledgments

We gratefully acknowledge the assistance of David L. Westbrook in the design and implementation of MESS, TCL and the new implementation of PHOENIX. We thank David M. Hart for clarifying this prose.

This work is supported by ARPA/Rome Laboratory under contract numbers F30602-93-C-0100 and F30602-95-1-0021. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory or the US Government.

References

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

Boddy, M., and Dean, T. 1989. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 979–984. Detroit, Michigan.

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

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

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

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

Hanks, S.; Nguyen, D.; and Thomas, C. 1992. The new Truckworld manual. Technical report, Department of Computer Science and Engineering, University of Washington. Forthcoming. Contact `truckworld-request@cs.washington.edu`.

Hanks, S.; Pollack, M. E.; and Cohen, P. R. 1993. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine* 13(4):17–42.

Howe, A. E. 1993. *Accepting the Inevitable: The Role of Failure Recovery in the Design of Planners.* Ph.D. Dissertation, University of Massachusetts at Amherst. Also available as Computer Science Department Technical Report 93–40.

Joslin, D.; Nunes, A.; and Pollack, M. E. 1993. TileWorld user's manual. Technical Report 93-12, Department of Computer Science, University of Pittsburgh. Contact `tileworld-request@cs.pitt.edu`.

Martin, N. G., and Mitchell, G. J. 1994. A transportation domain simulation for debugging plans. Obtained from the author, `martin@cs.rochester.edu`.

Pollack, M. E., and Ringuette, M. 1990. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 183–189. American Association for Artificial Intelligence.

Russell, S., and Wefald, E. 1988. Decision-theoretic control of reasoning: General theory and an application to game playing. Technical Report UCB/CSD 88/435, UC Berkeley.

Russell, S. J., and Wefald, E. H. 1989. On optimal game-tree search using rational meta-reasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 334 340. Detroit, Michigan.