,

Simulation for ARPI and the Air Campaign Simulator

Paul Cohen
Scott Anderson
David Westbrook

Experimental Knowledge Systems Laboratory
Computer Science Department, LGRC
University of Massachusetts
Amherst MA 01003-4610
{cohen,anderson,westy}@cs.umass.edu

Abstract

We describe a simulation substrate for building real-time simulators in a Common Lisp environment, and a simulation of Air Campaign Planning implemented in the substrate. The substrate is designed for experimental work: Trials are repeatable and randomness is carefully controlled; all aspects of the simulation are instrumented and integrated with the UMass CLIP/CLASP instrumentation and analysis package. The ACS simulator is for testing campaign-level planners and simulators.

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. Simulation provides a solution: 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.

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. Because of the many design decisions, these testbeds are often not as easily shared as their authors intended. Our MESS substrate (Anderson 1995) captures the best of the common, domain-independent aspects of these simulators, and improves the representation of thinking agents and the measurement of time.

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 simulators in 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 Tile-World 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.

Mess has a central "engine," which interleaves the streams of events that represent different real-world processes. 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 we call ***advancing*** the simulation. Figure 1 presents pseudo-code for the algorithm to advance the simulation. Each time 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. 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). 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 (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 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

```
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 1: Pseudo-code for the MESS engine.

the simulation, so that 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 TILE-WORLD. 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 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. Activities are represented as a single object, 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 1.

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.

Timed Common Lisp for Real-Time Agents

One valuable component of the Mess simulator is Timed Common Lisp (TCL), which has all the functionality of ordinary Common Lisp, but each function advances the clock predictably and appropriately. Most simulators (including our Phoenix system) use elapsed CPU time to assess how long an agent has been thinking. This approach is intuitive and straightforward to implement, but it has drawbacks. 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).) For replicable experiments, for explicit reasoning about time, for deliberation scheduling, and other applications, it is very valuable to have a database of timings for Common Lisp functions (and for user-defined functions). This TCL provides. One can use Mess without TCL, but for real-time applications, one should use TCL. The overhead is minimal (Anderson 1995).

Instrumentation and Data Analysis

Mess is completely integrated with the CLIP/CLASP, an instrumentation and analysis package developed at UMass. With CLIP, one attaches "alligator clips" to pieces of code, for example, messages sent by agents, environment events, or various activities. These alligator clips collect state information periodically or in response to specified events, and dump the data to CLASP for analysis. The CLASP package incorporates dozens of statistical techniques; its design emphasized exploratory data analysis over strict hypothesis testing. After all, most experiments with complex systems are intended to find out how the systems work—the factors that affect performance, singly and in combination—which often involves searching for hidden structure in run-time data. In fact, we have developed a mixed-initiative planning assistant for exploratory data analysis, described in a companion paper in this volume.

One general and apparently very powerful statistical method is called "multi-stream dependency detection, or MSDD. Consider the streams of data flowing from a robot's sensors, a command and control network, the monitors in an intensive care unit, or periodic measurements of various indicators of the health of the economy. There is clearly utility in determining how current and past values in those streams are related to future values. We formulate the problem of finding structure in multiple streams of categorical data as search over the space of dependencies, unexpectedly frequent or infrequent co-occurrences, between complex patterns of values that can appear in the streams. MSDD performs an efficient systematic search over the space of all possible dependencies. There are currently four versions of MSDD. The first version searches dependency space in a best-first manner, guided by a user supplied evaluation function. The second version employs a statistical measure of dependency strength and uses bounds that we derive for the value of that measure to prune huge portions of the search space. When the algorithm terminates, it returns a list of the k strongest dependencies that is equivalent to the list that would be returned by an algorithm that exhaustively searched the space of all possible dependencies. The third version is incremental; it does not require an initial batch of data from the streams (as the first two versions do) but instead incrementally refines hypotheses about where strong dependencies exist as new data is acquired from the streams. The fourth version is a distributed algorithm that runs on networks of machines. All versions of the algorithm employ domain independent heuristics to make the search efficient, and can be augmented with domain specific heuristics to maximize performance on specific problems.

We have successfully applied MSDD to classification problems and to learning rules in a shipping network that relate current states to future pathologies. In the latter, we demonstrated that MSDD can automatically acquire rules that allow an agent to manage the network as effectively as when the agent uses hand-coded domain knowledge. We are currently applying MSDD to the problem of predicting faults in computer networks.

The Air Campaign Simulator

The Air Campaign Simulator (ACS) is being developed at UMass as a testbed for work in the ARPA/Rome Lab Planning Initiative. Currently, ACS implements the Korea Scenario, developed by Doug Holmes at ISX. We will briefly describe the key features of ACS.

Campaign-level Simulation

Activities in ACS are large-scale, abstract, aggregated and at the strategic, campaign level. For example, actions in ACS include **Observe, Supply, Show Force, Blockade** and the like. Actions commit resources, but again, these are aggregated forces, not individual airplanes with tail numbers. Outcomes of actions are similarly general; for example, an attack on a supply route may **disrupt** supply. We do not fly specific aircraft against particular targets on the supply route; we do not model the engagements of these aircraft and their targets; we do not perform simulated BDA. At the campaign level, it suffices to direct a wing against a supply route and assess the change in status of the supply route.

Campaign-level Ontology

Doug Holmes' ontology of campaign-level actions is fully implemented in ACS. One can simulate campaigns manually by selecting actions from this ontology and placing them on the timeline, or, getting closer to the aims of ACS, various ARPI planners can select and have ACS simulate campaign actions. We have augmented Holmes' ontology with various objects, locations, types of terrain and weather, and status variables.

Chains

The concept of center of gravity is very important to campaign planners, and we implement it in ACS with structures called ***chains***. A chain is a causal association between objects; for example, an anti-aircraft battery is "downstream" on a command and control chain from the unit that controls it. We have chains for command and control, logistics, personnel, geography, and other centers of gravity.

Visualization

ACS has various interfaces to show the "campaign at a glance." Chains figure prominently because they identify leverage points in the campaign.

Computing Outcomes

The outcomes of actions such as **Attack, Patrol, Blockade**, and so on are computed by rules that take into account the readiness and morale of the units involved, the weather, and other factors. Currently, outcomes are crude; only four status variables are allowed: operational, degraded, disrupted and non-operational.

Implementation

ACS is implemented in MESS, the simulation substrate described earlier. Currently, it does not exploit much of the functionality of MESS; for example, actions are implemented as events instead of activities, and ACS isn't a real-time simulator. These choices enabled us to produce a first draft of ACS very rapidly; it is available from cohen@cs.umass.edu.

Acknowledgments

This work is supported by ARPA/Rome Laboratory under contract #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.

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.

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.