MASON: A Multiagent Simulation Environment

Sean Luke
Claudio Cioffi-Revilla
Liviu Panait
Keith Sullivan
Gabriel Balan
Department of Computer Science and Center for Social Complexity
George Mason University
4400 University Drive, Fairfax VA 22030, USA
http://cs.gmu.edu/∼eclab/projects/mason/

MASON is a fast, easily extensible, discrete-event multi-agent simulation toolkit in Java, designed to serve as the basis for a wide range of multi-agent simulation tasks ranging from swarm robotics to machine learning to social complexity environments. MASON carefully delineates between model and visualization, allowing models to be dynamically detached from or attached to visualizers, and to change platforms mid-run. This paper describes the MASON system, its motivation, and its basic architectural design. It then compares MASON to related multi-agent libraries in the public domain, and discusses six applications of the system built over the past year which suggest its breadth of utility.

Keywords: Agent-based modeling, simulation, multi-agent systems, computational social science

1. Introduction

MASON is a single-process, discrete event simulation core and visualization library written in Java, designed to be flexible enough to be used for a wide range of simple simulations, but with a special emphasis on swarm multiagent simulations of many agents (up to millions). We developed the MASON simulation toolkit to meet the needs of computationally demanding "swarm"-style multiagent systems (MAS) research. The system is open-source and free and is a joint effort of George Mason University's Computer Science Department and the George Mason University Center for Social Complexity. MASON may be downloaded at http://cs.gmu.edu/∼eclab/projects/mason/.

Multiagent systems are receiving increasing research attention as affordable computer brawn makes simulation of these environments more feasible. One source of interest has come from social and biological models, notably ones in economics, land use, politics, and population dynamics (e.g., [1-3]). Another source stems from the swarm robotics community, particularly as homeland security and defense interests have bolstered investigations of large numbers of "U∗Vs" (unmanned aerial vehicles, unmanned underwater vehicles, etc.) for collaborative target observation, reconnaissance, mapping, and so on [4-7]. Swarm multiagent simulation is also a common technology in the game and movie industries.

MASON was built from first principles to serve the needs of these research areas. Our design philosophy was to build a fast, orthogonal, minimal model library to which an experienced Java programmer could easily add features, rather than one with many domain-specific, intertwined features that are difficult to remove or modify. To this minimal model we have added those visualization and graphical user interface (GUI) facilities we have found useful for various simulation tasks.

We began work on MASON because we needed a simulation toolkit that made it relatively easy for us to create a very wide range of multiagent and other simulation models and to run many such models efficiently in parallel on back-end cluster machines. Domains to which we intended to apply the simulator ran the gamut from robotics and machine learning to multiagent models of social systems (political science, historical development, land use, economics, etc.). Our previous research in these areas had either relied on a heavily modified robotics simulator (notably TeamBots [8]); a compiled social complexity toolkit such as SWARM [9], Ascape [10], or RePast [11]; or an interpreted rapid-development library such as StarLogo [12], NetLogo [13], or Breve [14].

We needed to run many ($>$100,000) simulation runs to optimize model parameters or perform machine learning in a multiagent problem domain. In such cases, we had to "cook" the simulations on multiple back-end servers (in Linux, Solaris, and MacOS X) while occasionally viewing the results on a front-end MacOS X or Windows workstation. This required speed, the ability to migrate a simulation

SIMULATION, Vol. 81, Issue 7, July 2005 517-527

DOI: 10.1177/0037549705058073

run from platform to platform, and (for our purposes) guaranteed platform independence. Furthermore, we needed to be able to customize the simulator to different multiagent simulation problems and applications. The aforementioned systems did not meet these needs well because they tied the model to the GUI too closely, could not guarantee platform-independent results, or, being written in an interpreted language, were slow. In addition, several such systems, particularly the robotics simulators, were by and large geared to a particular problem domain. Rather than remove special-purpose code from an existing system (potentially introducing bugs), we instead hoped to build on top of a more general-purpose simulator.

Given these research needs, MASON's design goals were as follows:

- A small, fast, easily understood, and easily modified core
- Separate, extensible visualization in 2D and 3D
- Production of identical results independent of platform
- Checkpointing any model to disk such that it can be resumed on any platform with or without visualization
- Efficient support for up to a million agents without visualization
- Efficient support for as many agents as possible under visualization (limited by memory)
- Easy embedding into larger existing libraries, including having multiple instantiations of the system coexisting in memory

There were three design goals we explicitly did *not* make for MASON. First, we did not intend to include parallelization of a single simulation across multiple networked processors. Such an architecture is radically different from a single-process architecture. Second, we intended the MASON core to be simple and small and so did not provide built-in features special to social agents or robotics simulators. We felt such things were more appropriately offered as optional domain-specific modules in the future. Third, although we tried to be reasonably memory efficient, this was not a priority.

We recognize that speed, model detachment, checkpointing and portability, and strong visualization are all common in the simulation community at large. However in the "swarm"-style simulation community, MASON's combination of architecture and these features is essentially unique. In this article, we discuss the architectural design of the system and then detail six applications of MASON presently under way.

2. Architecture

MASON is written in Java to take advantage of its portability, strict math and type definitions (to guarantee duplicable results), and object serialization (to checkpoint out simulations). Java has an undeserved reputation for slowness, and our past experience in developing the ECJ evolutionary computation toolkit [15] suggests that carefully written Java code can be surprisingly fast.

The toolkit is written in three layers: the utility layer, the model layer, and the visualization layer. The utility layer consists of classes that may be used for any purpose. These include a random-number generator, data structures more efficient than those provided in the Java distribution, various GUI widgets, and movie- and snapshot-generating facilities. Next comes the model layer, a small collection of classes consisting of a discrete event schedule, schedule utilities, and a variety of *fields* that hold objects and associate them with locations. This code alone is sufficient to write basic simulations running on the command line.

The visualization layer permits GUI-based visualization and manipulation of the model. Figure 1 shows a simplified diagram relating basic objects in the model and visualization layers. For most elements in the model layer, down to individual fine-grained objects in the model, there is an equivalent "proxy" element in the visualization layer responsible for manipulating the model object, portraying it on-screen, and inspecting its contents. A bright line separates the model layer from the visualization layer, which allows us to treat the model as a self-contained entity. We may at any time separate the model from the visualization, checkpoint the model to disk, move it to a different platform and let it continue to run, or attach an entirely different visualization collection. Figure 2 shows this procedure.

2.1 The Model Layer

MASON's model layer has no dependencies on the visualization layer and can be separated from it. A MASON model is entirely contained within a single instance of a user-defined subclass of MASON's model class (`SimState`). This instance contains a discrete event `Schedule`, a `MersenneTwister` random-number generator, and zero or more fields.

Agents and the Schedule. MASON employs a specific usage of the term *agent*: a computational entity that may be scheduled to perform some action and that can manipulate the environment. Note that we do not explicitly state that the agent is physically *in* the environment, though it may be; in this case, we would refer to the agent as an *embodied agent*. Agents are brains which may or not be embodied. MASON does not schedule events to send to an agent; rather, it schedules the agent itself to be *stepped* (pulsed or called) at some time in the future. Hence, MASON's agents implement the `Steppable` interface, as shown in Figure 1. Scheduling an agent multiple times for different functions is easily done with an anonymous wrapper class.

MASON can schedule `Steppable` objects to occur at any real-valued time in the future. Furthermore, the `Schedule` may be divided into multiple *orderings* that further subdivide a given time step: agents scheduled at a given time but in an earlier ordering will be stepped prior to agents scheduled at the same time but in a later ordering. MASON also provides various `Steppable` wrappers

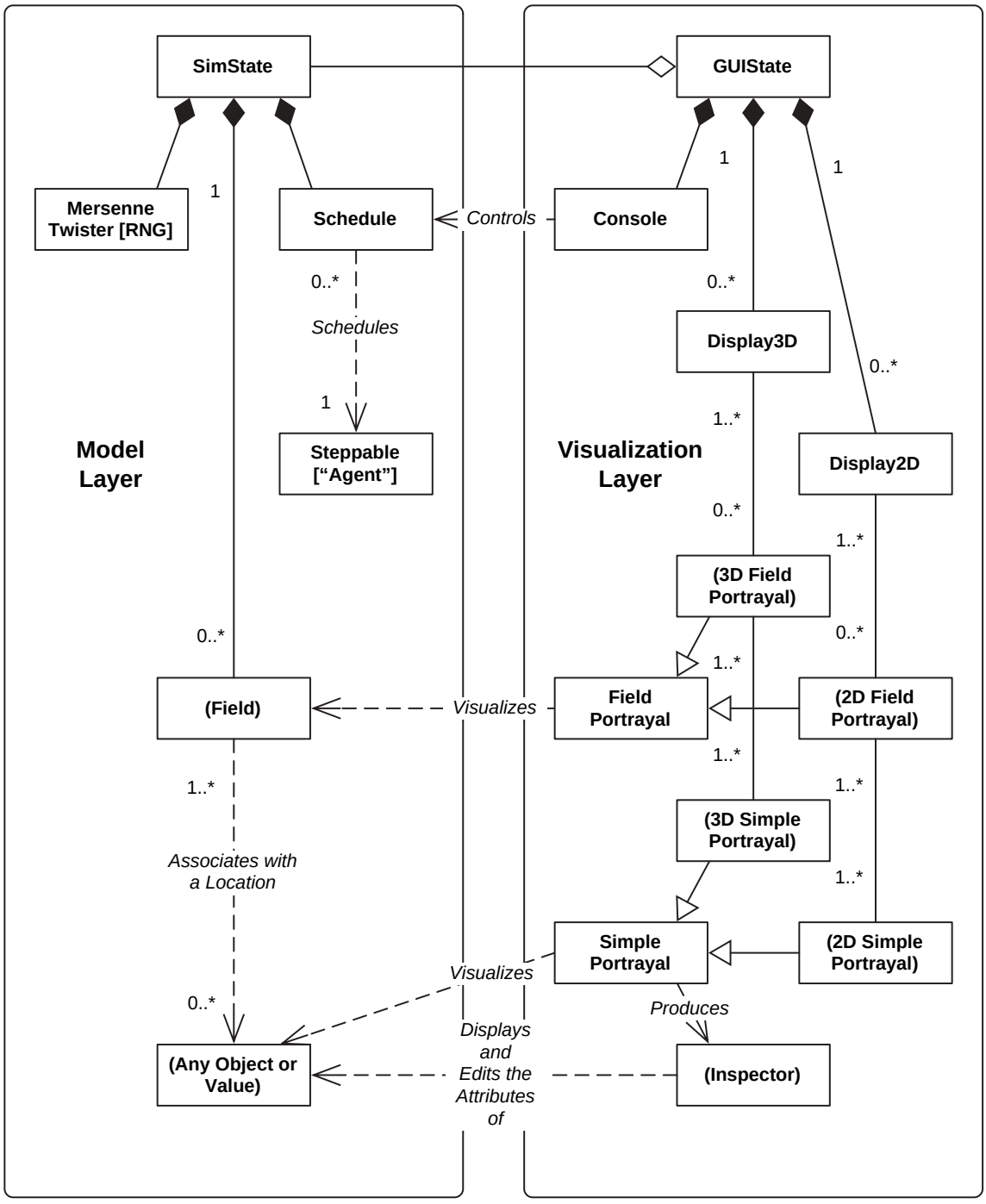

Figure 1. Highly simplified UML diagram of the basic classes in the model and visualization layers. Items in parentheses indicate sets from which numerous class are available.

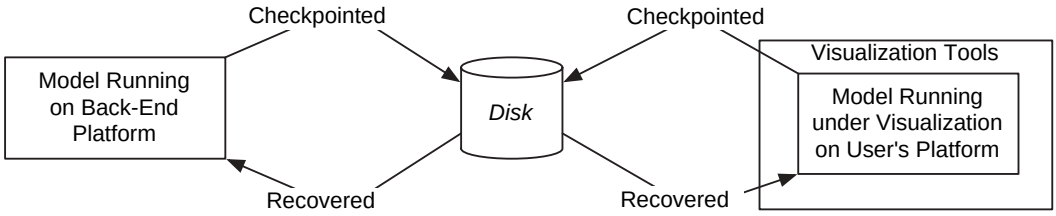

Figure 2. Checkpointing and recovering a MASON model to be run standalone or under different kinds of visualization

that can group agents together, iterate them, perform them in parallel on separate threads, and so on. Agents may be scheduled to run in their own thread asynchronous with the schedule. This thread may run until completion, loop indefinitely, or run until the `Schedule` reaches some later time step.

Fields. MASON's fields relate arbitrary objects or values with locations in some notional space. Some of these fields are little more than wrappers for simple 2D or 3D arrays. Others provide sparse relationships. An object may exist in multiple fields at one time, and in some fields, the same object may appear multiple times. The use of fields is entirely optional, and the user may add additional fields of his or her own devising. MASON provides fields for the following:

- 2D and 3D bounded arrays of objects, integers, or doubles that may be bounded or toroidal and with hexagonal, triangular, or square layouts
- 2D and 3D sparsely populated object grids that are bounded, unbounded, or toroidal and with hexagonal, triangular, or square layouts
- 2D and 3D sparse continuous (real-valued) space, that may be bounded, unbounded, or toroidal
- Networks (graphs), whose edges may be directed or undirected and optionally weighted or labeled

When running the model without visualization, MASON has an intentionally primitive top-level simulation loop. MASON begins by either creating a new `SimState` or loading one from a Java-serialized checkpoint file. MASON then enters the following loop. First, it checks to see if the `Schedule` has any agents remaining to step. If not, or if some maximum time step has been exceeded, MASON exits the loop, finishes the `SimState`, and quits. Otherwise, the `Schedule` advances the time to the minimum agent-scheduled time step, then steps all agents scheduled at that time (sorted by ordering and shuffled randomly within an ordering). If a checkpoint is desired (typically every so many `Schedule` steps), it is done so at this time: asynchronous agents are first requested to pause their threads, then a checkpoint of the entire model is written out, then asynchronous agents resume their threads, and the loop continues.

Agents have full access to the `SimState` and may manipulate its fields, `Schedule`, and random-number generator. MASON imposes few restrictions on the actions they may perform and provides no simplifying protocols for agent design. For example, MASON does not provide a rule language for stipulating agent behaviors. We imagine such things can be included in forthcoming MASON module extensions.

2.2 The Visualization Layer

Objects in the visualization layer may examine model-layer objects only with the permission of a gatekeeper wrapper around the `SimState`, called a `GUIState`. When running with a GUI, it is this class that is responsible for attaching the `SimState` to visualization (or detaching it) and for checkpointing the `SimState` to or from disk. As certain objects in the visualization world need to be scheduled (notably, windows need to be refreshed to reflect changes in the model), such elements may "schedule" themselves with the `GUIState` to be updated whenever the underlying `Schedule` is pulsed but not be scheduled on the `Schedule` itself. This allows the visualization layer to be separate from the model.

In addition to the `SimState`, the `GUIState` also maintains zero or more *displays*, GUI windows that provide 2D and 3D views of underlying fields. Displays operate by holding zero or more *field portrayals*, associating each one with a different field in the model. Each field portrayal is responsible for drawing the field on-screen and for responding to user requests to inspect features of the field. Field portrayals do this by associating *simple portrayals* with individual objects or values stored in the fields. A field portrayal may associate a simple portrayal with a specific object stored in the field, with a class of objects, with all objects in the field, and so on. The user may choose from a number of provided simple portrayals, the user may design the simple portrayal himself, or the object may portray itself instead. Some examples of visualized fields are shown in Figure 3.

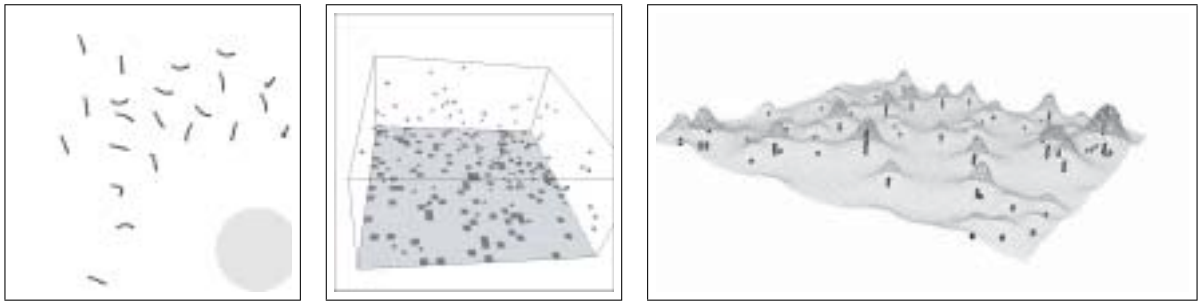

Figure 3. Examples of visualized fields in MASON, showing various forms of 2D and 3D continuous and discrete space

Simple portrayals can also, on request, call up *inspectors* (what SWARM would call "probes") of underlying model objects. Inspectors are GUI panels that allow the user to inspect or modify object parameters. The user can provide custom inspectors for objects or use the basic ones provided (which use Java's Bean Properties facility). Models and fields may also have inspectors. Drawing and inspection follow similar paths: when a display is redrawn, it asks each of its field portrayals to redraw their fields, and in turn, the field portrayals call up simple portrayals to draw elements in the field. Likewise, when a user clicks on a display to request inspection of objects, the display asks the field portrayals to provide inspectors for these objects, and the field portrayals in turn call up the relevant simple portrayals and ask them to provide the inspectors.

The `GUIState` also contains a top-level controller GUI window, usually the provided `Console`. The `Console`'s primary function is to allow the user to start/stop/pause/step the `Schedule`, but it also provides the GUI functionality to load and save checkpointed models, to show and hide displays, to view inspectors, and to load additional simulations (each with their own `SimState`s, `GUIState`s, and `Console`s).

Running the model under visualization is a more involved process than without visualization, and not just because things must now be displayed. The underlying model runs in its own thread separate from the GUI's main thread. Since both the model thread and the GUI thread must have access to underlying model data, they enter into a synchronization procedure that guarantees that only one is operating on these data at any given time. The general procedure is as follows. When a `GUIState` is constructed, it creates a `Console`, various displays, and the underlying `SimState`. When the user presses "play" in the `Console` to start a simulation, the `Console` starts the `SimState`, then spawns the model thread. The model thread enters into a loop that it exits only if asked to shut down by the `Console` or if no further agents are scheduled. In this loop, the `GUIState` performs any pre-schedule items, then the `Schedule` advances to the minimum agent-scheduled time and steps any agents at that time, then postschedule items are performed (typically requests to redraw the displays), and finally the thread defers to the GUI thread to give it access to the model before finishing the loop. While the model thread is waiting, the GUI thread can finish redrawing the displays, complete any requests made by the user to inspect the model, and checkpoint out the model (or read a new one in from checkpoint to replace it). When the user presses "stop," the thread is asked to shut down, and the `Console` finishes the `SimState`. When the `Console` is closed, the `GUIState`, `SimState`, and displays are destroyed.

2.3 MASON Usage and Extensions

Because of the separation of the model from visualization, MASON models are usually created in two stages (refer again to Figure 1). First, the author develops the model proper as a self-contained subclass of `SimState`. After this code is completed, the MASON model should be able to run on the command line as a GUI-less application. Next, the author creates a `GUIState` to encapsulate the `SimState`, attaching portrayals and displays. At this point, the simulation can be visualized. The author can create further `GUIState`s to visualize the `SimState` in different ways. A `GUIState` can also be used in combination with a special class, `SimApplet`, to generate online MASON applets. MASON comes with several tutorial and example applications to show how these are done.

The basic MASON distribution provides only those core tools common to most simulation and visualization needs. We have several extensions to MASON available or in development, each of which is distributed as a separate module or online tutorial:

- **Social Network Analysis.** MASON provides only a rudimentary network field. We have developed an extension to MASON that generates a number of useful graph statistics aimed largely at the social network research community.
- **Physics Modeling.** We are actively working on a Java-based 2D physics engine built on top of MASON. In addition, we are examining attaching MASON to a C++ 3D physics engine such as ODE [16].

- **Charting.** MASON has no graphing, charting, or statistical facilities: there are much better open-source facilities available than we can do ourselves, and what is available changes rapidly. We have provided a tutorial showing how to connect MASON to JFreeChart [17] and iText [18] to generate charts and graphs for display in real time and for publication-quality output.
- **Parameterization.** We provide a tutorial showing various approaches to loading MASON simulations from parameter files.

As MASON development continues, we expect further extensions as well, depending on research needs. Possible future directions include OpenGL visualization (in addition to MASON's presently used Java3D), high-level interpreted agent development tools, packet network simulation, and a Geographic Information Systems (GIS) library.

3. Comparison to Other Simulation Environments

MASON's original inspiration came from a desire to reimplement more cleanly some of the problem domains we had constructed in the Teambots [8] simulator. Teambots is an early Java-based lightweight robotics simulation environment that provides minimal physics and robot sensor facilities, a graphic display, and a very simple schedule procedure. Teambots is useful for behavior-based robotics experiments, though its lack of a real physics model is a hindrance. Teambots also makes a hard distinction between the *objects* in the world (including the robots) and the *agents* that drive some of them (such as the robot software). MASON's similar distinction was inspired by this. Teambots makes this delineation because this design allows the experimenter to port software robot behaviors to real robots using provided real-robot application protocol interfaces (APIs). A more recent family of simulators, called Player/Stage [19], has moved toward more realistic robots and environments.

The robotics simulators discussed above are capable but understandably geared to a very specific problem set. We found that implementing nonrobotics multiagent simulations in these simulators involved considerable modifications of the simulators to provide extra-robot functionality or to remove unneeded functionality that would otherwise slow the experiment. Such modifications had a strong likelihood of introducing bugs, particularly given the size and complexity of these simulators.

Compiled Multiagent Simulators. Instead we chose to construct MASON to be usable for a broad scope of lightweight simulation functions, with a general-purpose schedule and fields. In this vein, MASON is most closely comparable to compiled multiagent simulation libraries such as SWARM [9], Ascape [10], and RePast [11]. SWARM is the earliest such system and originally required the user to write in Objective-C and Tcl/Tk. SWARM applications may now be written in Java using special libraries that communicate with Objective-C. Oddly, SWARM does not take advantage of by far the foremost Objective-C system: the OpenStep GUI specification embodied in MacOS X and the open-source GNUstep library. We believe the use of an unusual language but not its primary environment has proven a challenge to SWARM's continued extensibility and maintainability.

To remedy this, RePast was envisioned to reimplement much of the SWARM philosophy entirely in Java or .NET, and it has been the center of considerable community interest in recent years. The RePast distribution has a large footprint: included in the package are neural networks, genetic algorithms, social network modeling, system dynamics modeling, logging, GIS, and graphs and charts.

Ascape is a multiagent simulation toolkit inspired by the Sugarscape model [2]. Ascape tries to be as rule oriented as possible within a Java framework: agents have rule-based behaviors that fire based on specific environmental conditions, and these agents are grouped into larger structures that have their own behaviors and fire their subsidiary agents in a user-specified order. This framework simplifies model development in some cases, but it also imposes considerable constraints on simulation design as a whole, particularly on simulations requiring arbitrary event handling.

Like MASON, these three multiagent system toolkits all provide graphical visualization, inspection of simulation objects, stochastic event ordering, and the generation of various forms of media. But there are some important differences. Stemming from its design goal as a general-purpose agent simulation environment, MASON provides 3D fields, visualization of both 2D and 3D fields in 3D, and somewhat more sophisticated and flexible 2D visualization. MASON is also somewhat faster than the toolkits described above both in underlying model and in visualization, and it is expressly designed to produce duplicable results if necessary.

But most important, MASON is capable of separating the model from visualization dynamically. While some of the above frameworks can run "headless," this is generally an either/or proposition, and furthermore, migration of a headless process from one machine to another is generally not possible, much less visualization of the headless process mid-run (except through analysis of its statistical output). These systems lack these features largely because they were originally designed for single-shot models that the experimenter would construct, run once, and then analyze. MASON instead was designed to be executed a very large number of times on different machines as part of a model optimization procedure.

Interpreted Multiagent Simulators. Another approach is to build a simulation toolkit in which the user manipulates the world through an interpreted, online programming language such as Logo. By eliminating the compile-run cycle, the experimenter is free to make small changes in the

code, even at runtime, to experiment with its effects. This is the design philosophy behind StarLogo [12] and later NetLogo [13]. These systems provide basic functionality similar to SWARM but impose a modified version of Logo as the experimenter's model implementation language. A related simulator is Breve [14], which provides 2D and 3D worlds with which the user may manipulate objects using the ODE physics engine and a proprietary language called Steve.

These simulators offer many benefits for rapid prototyping by enabling immediate feedback on code changes, encouraging tweaking of the model mid-run, and (in Breve's case) wrapping a powerful physics environment with a simple, easy-to-learn library. Furthermore, in theory, an interpreted-language design can more easily be ported mid-run from platform to platform and to dynamically add and remove visualization tools. The primary downside of these simulators is that, for our purposes, they are slow. The language features that make them so attractive for rapidly building a model also make them less appropriate for complex simulations with long runtimes. In addition, simulations built with these tools tend to be bound by the constraints imposed by their respective graphical interfaces. For example, Breve generally assumes visualization of a single space, and NetLogo constrains simulation tools to fit within a single window.

4. Applications

MASON has existed for 2 years at George Mason University, but we have already used it for a number of simulation tasks ranging from micro-air vehicle coordination and virus propagation to models of collective behavior in simple societies. Here we will mention a few of interest. We describe these systems primarily to demonstrate the depth of applicability of MASON. The models shown use a wide range of MASON features, including its square and hexagonal grids, sparse discrete fields, continuous fields, network facilities, 2D and 3D environments, real-valued and discrete schedules, added charts and graphs, and capability to be embedded in a larger external toolkit.

Four of these simulations—cooperative target observation, ant foraging, urban traffic, and "wetlands"—are computationally intensive and require many runs in batch. It is for such tasks that MASON is particularly well suited. Two simulations—network intrusion and anthrax propagation—were originally written in other simulation packages and were ported to MASON to take advantage of agent inspection features and as tests of porting difficulty.

Network Intrusion and Countermeasures. Network intrusion is an agent-based model designed to study computer network security issues, first developed in Ascape and then ported to MASON by an inexperienced MASON developer to test the difficulty and speed of porting to the new system (with, we felt, very positive results). The current version models a network of 2500 computer systems connected via two overlaid grid topologies: IP address space (or physical space) and remote login space. In these spaces live two kinds of agents: computer systems and one or more hackers. Each computer system has a set of security policies implemented when the system is believed to be compromised. A computer may be classified as secure, threatened (in the sense that a nearby computer has been compromised), compromised at a user level, or compromised at the super-user level. The parameters of the model allow one to understand the effects of changes in security policies as well as the effects of changes in hacker behavior. Figure 4 shows a snapshot of a simulation.

Urban Traffic Simulation. We have developed a lightweight urban traffic simulation in MASON to examine traffic flow from a multiagent perspective and hope to apply derived algorithms to other environments such as packet routing. A small simulation is shown in Figure 5. The simulation uses a network field in MASON with intersections as nodes and roads as graph edges. Cars and traffic lights are scheduled using MASON's real-valued time schedule. Cars beginning travel along a road are scheduled to appear at the intersection at the end of the road at some time in the future, depending on road length and car speed. When a car reaches an intersection, it is placed in another queue to wait at the intersection's stoplight. While a light is green, some N waiting cars are allowed through the intersection at a given time step.

Using this simulation, we are investigating how to maximize both global and per car mean travel time, variance in wait time, and other factors. Of particular interest to us is how the system can adapt to "smooth out" sudden unexpected floods of traffic (after a sporting event ends, for example), in addition to handling regular "rush-hour" style floods.

Cooperative Target Observation in Unmanned Aerial Vehicles. In recent experiments [20], we examined the effectiveness of various algorithms that direct mobile UAV agents (called *observers*) to collectively stay within an "observation range" of as many randomly moving targets as possible. Observers and targets live in a sparse continuous 2D or 3D field in MASON. The cooperative target observation environment is shown in Figure 6. We used this environment to examine "tunably decentralized" cooperative algorithms, where by changing a parameter, we could gradually shift the algorithm from one global decision-making procedure to individual per agent procedures. We examined two such algorithms for controlling the observers based on K-means clustering and hill climbing.

Ant Foraging. We have recently examined how to augment ant-like robot swarm behaviors with pheromones to perform "central place food foraging," in which agents

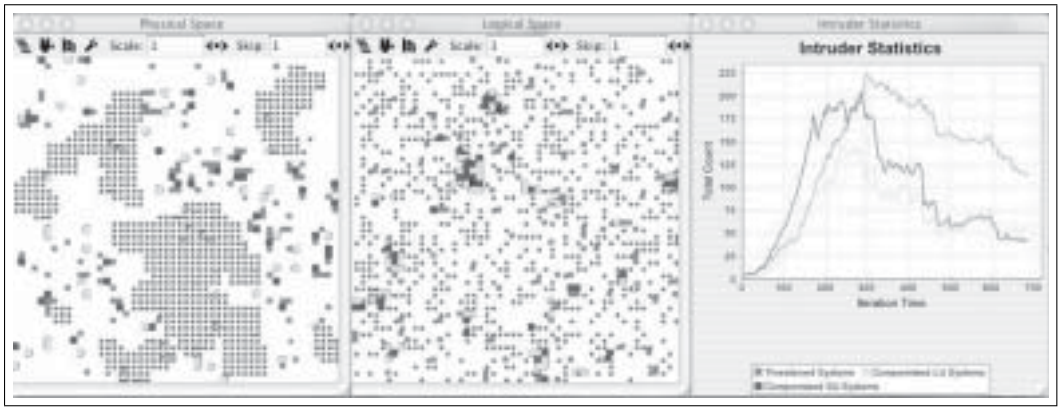

Figure 4. Network intrusion model: the physical (left) and logical (center) spaces, together with statistics on intrusions and compromised systems (right)

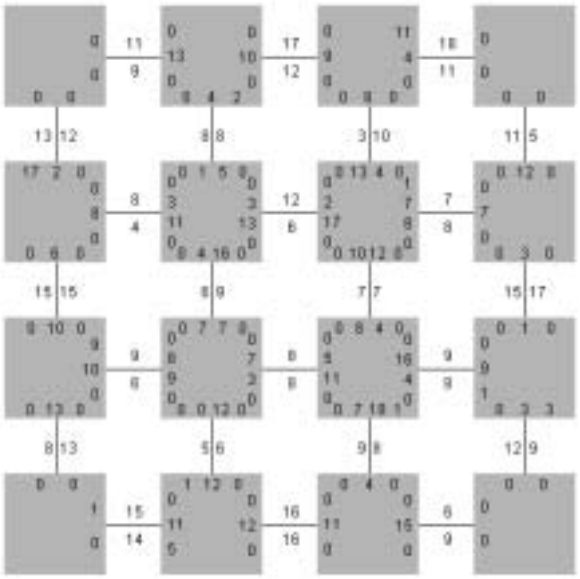

Figure 5. A small 16-intersection urban traffic simulation. Gray squares are intersections, and lines are two-way roads. Numbers indicate how many cars are traveling on roads or waiting to perform various actions at intersections.

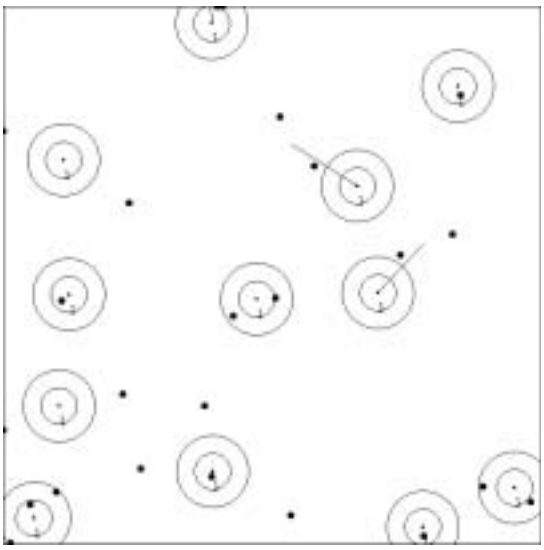

Figure 6. Cooperative target observation model. Small doubly circled dots are observers. Outer circles are their observation ranges. Large dots are targets. Straight lines connect observers with newly chosen destinations.

leave a nest to search for food, then return to the nest laden with food. Figure 7 shows a typical 100 $\times$ 100 cell environment with 1000 ants, a nest (bottom right), a food source (top left), and two large elliptic obstacles. The ants cooperatively discover and optimize a minimum-length trail. Our experiments in this environment [21] suggested that pheromones bear a strong resemblance to utility value functions found in dynamic programming and reinforcement learning.

This environment was implemented using 2D sparse grids and value grids in MASON and is notable in that, in some cases, MASON served as a subsidiary object within the ECJ evolutionary computation system [15]. This allowed us to use ECJ to optimize ant behaviors: ECJ would iteratively consider a candidate ant behavior, then fire up MASON to test the behavior in simulation and assess its quality. More details on these experiments are reported in Panait and Luke [22].

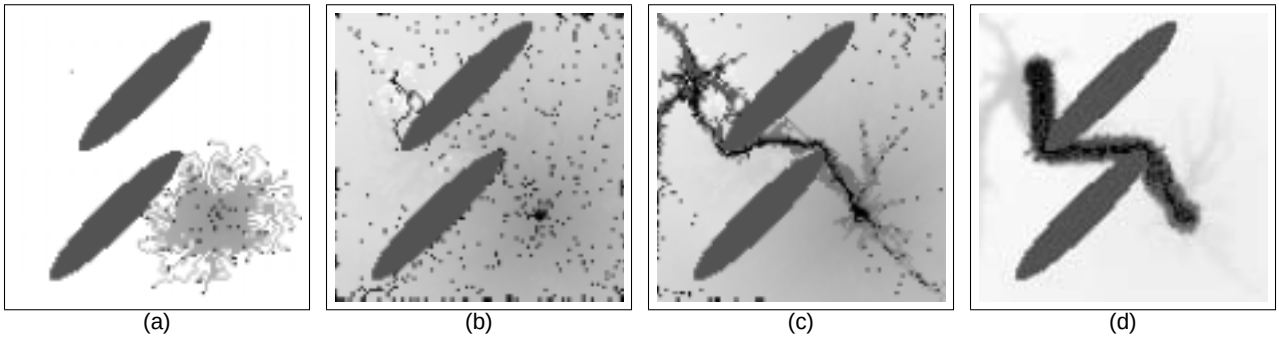

Figure 7. Foraging sequence with two obstacles. Nest is in bottom-right quadrant, and food source is in top-left quadrant. Left to right: (a) ants leave the nest while depositing a pheromone. (b) Ants discover the food source and begin return to nest along the pheromone while depositing a second pheromone. (c) Trail is established. (d) Ants perform trail optimization.

Anthrax Propagation in the Human Body. The interaction between pathogens and infected hosts is usually investigated using laboratory and live studies. But for some diseases, such as inhalation anthrax, live studies are not possible due to their deadly effects. After examining laboratory studies, the spread of anthrax in human organs was modeled as a series of discrete events that map out a time course for infection in the human body. Different systems in the human body that play a role in inhalation anthrax were modeled as spatial entities to show how the anthrax disease flows through the body. The dynamics of these interactions was implemented using 2D sparse grids, one per system. The model also displayed statistics on the interactions of the systems and on the patient's health and disease state.

The anthrax model was developed originally using SWARM in Objective-C but was rewritten in its entirety in MASON to take advantage of MASON's speed and its control and inspection features. The individual performing the port had no previous knowledge of MASON at all but reported that the port was fairly easy as MASON has a similar scheduling mechanism to SWARM. Figure 8 shows before-and-after screenshots showing an anthrax panel as displayed in SWARM and in MASON.

Wetlands: A Model of Memory and Primitive Social Behavior. Using the MASON wetlands model [23, 24], we investigated the effect of memory, forgetfulness, and simple hierarchical group organization on the emergent patterns of agent interactions in a primitive human society. Groups of agents look for food, which is generated by a moisture layer in the simulated landscape, and seek shelter when they get too wet. In addition, groups of the same culture share information about food and shelter location to mimic some minimal social in-group versus out-group behaviors. The system was implemented using MASON's hexagonal grid facilities in multiple layers, as shown in Figure 9.

5. Conclusion

In this article, we presented MASON, a multiagent simulation library written in Java. MASON is fast and portable, has a small core, and produces guaranteed duplicable results. MASON is also designed to completely separate the model from the visualization dynamically or reattach it, migrate the simulation to another platform in the middle of a run, and provide visualization in 2D or in 3D. We also showed six applications of MASON, highlighting the broad applicability of the toolkit. Two of the applications are ports of previous simulation models from Ascape and SWARM.

We plan to position MASON as a principled foundation for future multiagent simulation systems to build upon. MASON is free open source under a BSD-style license and is available at http://cs.gmu.edu/∼eclab/projects/mason/.

6. Acknowledgments

Funding for MASON development has been provided by the GMU Center for Social Complexity and by DARPA/IXO/PCAS grant no. 200748. Our thanks to Ken De Jong and Jayshree Sarma for their assistance in the development of the article. Thanks also to additional MASON developers: Christian Thompson is developing the physics engine, and Daniel Kuebrich wrote applications and Quicktime support. Thanks also to application writers for their assistance: the network intrusion model was written by Elena Popovici, the anthrax model was written by Jayshree Sarma and Elena Popovici, and the wetlands model was written by Sean Paus. Earlier versions of this article were presented at the SwarmFest04, Ann Arbor, Michigan, and at the RC33 Workshop of the International Sociological Association, Amsterdam. We thank Nick Gotts, Nigel Gilbert, Gary Polhill, Klaus Troitzsch, and Scott Moss for their comments. Last, we must blame

Figure 8. Panels from the anthrax propagation model: (left) original SWARM model and (right) MASON replication

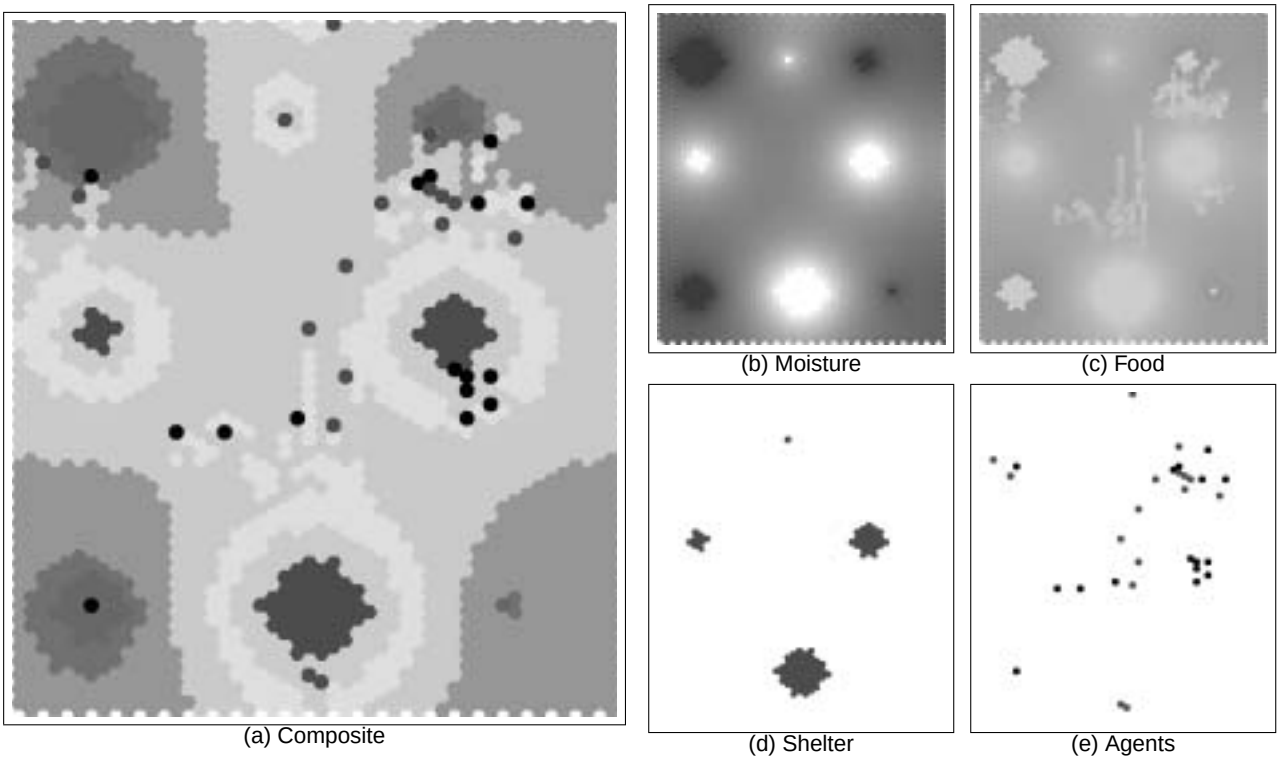

Figure 9. Wetlands initial visualization and layers. Composite visualization (a) consisting of moisture layer (b), food sites layer (c), shelter sites layer (d), and agents layer (e)

John Grefenstette for coming up with the MASON name. It stands for “Multi-Agent Simulation of Networks” . . . or “Neighborhoods” . . . or something like that. Ns are hard.

7. References

[1] Axtell, Robert. 2001. Social science as computation. Technical report, Center for Economic and Social Dynamics, Brookings Institution, Washington, DC.

[2] Epstein, Joshua M., and Robert Axtell. 1996. *Growing artificial societies: Social science from the bottom up*. Cambridge, MA: MIT Press.

[3] Gilbert, Nigel, and Klaus G. Troitzsch. 2005. *Simulation for the social scientist*. 2nd ed. Buckingham, UK: Open University Press.

[4] Bassett, Jeffrey K., and Kenneth A. De Jong. 2000. Evolving behaviors for cooperating agents. In *Proceedings from the Twelfth International Symposium on Methodologies for Intelligent Systems*, edited by Z. Ras, 157-65. New York: Springer-Verlag.

[5] Fernandez, F., and Lynne Parker. 2001. Learning in large cooperative multi-robot domains. *International Journal of Robotics and Automation* 16 (4): 217-26.

[6] Parker, Lynne. 2002. Distributed algorithms for multi-robot observation of multiple moving targets. *Autonomous Robots* 12 (3): 231-55.

[7] Parker, Lynne E. 2003. The effect of heterogeneity in teams of 100+ mobile robots. In *Multi-robot systems: Vol. II. From swarms to intelligent automata*, edited by Alan C. Schultz and Lynne E. Parker, 205-15. Dordrecht, the Netherlands: Kluwer.

[8] Balch, Tucker. 1997. TeamBots simulation and real robot execution environment. http://www-2.cs.cmu.edu/∼trb/TeamBots/

[9] Johnson, Paul, and Alex Lancester. 2000. Swarm User Guide: Swarm Development Group.

[10] Parker, Miles. 1998. Ascape. http://www.brook.edu/dybdocroot/es/dynamics/models/ascape/

[11] Collier, Nick. 2001. Repast: An agent based modelling toolkit for Java. http://repast.sourceforge.net

[12] Resnick, Michael. 1994. *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds*. Cambridge, MA: MIT Press. http://education.mit.edu/starlogo/

[13] Wilensky, Uri. 1999. Netlogo. http://ccl.northwestern.edu/netlogo/

[14] Klein, Jon. 2002. BREVE: A 3D environment for the simulation of decentralized systems and artificial life. In *Proceedings of Artificial Life VIII, 8th International Conference on the Simulation and Synthesis of Living Systems*. http://www.spiderland.org/breve/

[15] Luke, Sean. 2004. ECJ 11: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/

[16] Smith, Russell. 2004. The open dynamics engine. http://ode.org

[17] Gilbert, David. 2004. JFreeChart. http://www.jfree.org/jfreechart/

[18] Lowagie, Bruno, and Paulo Soares. 2004. iText Java PDF generation library. http://www.lowagie.com/iText/

[19] Gerkey, Brian, Richard T. Vaughan, and Andrew Howard. 2003. Player/Stage robotics simulator. http://playerstage.sourceforge.net

[20] Luke, Sean, Keith Sullivan, Liviu Panait, and Gabriel Balan. 2005. Tunably decentralized algorithms for cooperative target observation. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi Agent Systems.* Edited by F. Dignum et al., 911-17. ACM.

[21] Panait, Liviu, and Sean Luke. 2004. A pheromone-based utility model for collaborative foraging. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-2004)*.

[22] Panait, Liviu, and Sean Luke. 2004. Evolving ant foraging behaviors. In *Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*.

[23] Paus, Sean. 2003. Floodland: A simple simulation environment for evolving agent behavior. Technical report, Department of Computer Science, George Mason University, Fairfax, VA.

[24] Cioffi-Revilla, Claudio, Sean Paus, Sean Luke, James Olds, and Jason Thomas. 2004. Mnemonic structure and sociality: A computational agent-based simulation model. In *Proceedings of the Agent 2004 Conference on Social Dynamics.*

Sean Luke *is assistant professor in the Department of Computer Science at George Mason University, Fairfax, Virginia.*

Claudio Cioffi-Revilla *is professor of Computational Social Science and director of the Center for Social Complexity at George Mason University, Fairfax, Virginia.*

Liviu Panait *is a PhD candidate in the Department of Computer Science at George Mason University, Fairfax, Virginia.*

Keith Sullivan *is a PhD student in the Department of Computer Science at George Mason University, Fairfax, Virginia.*

Gabriel Balan *is a PhD student in the Department of Computer Science at George Mason University, Fairfax, Virginia.*