

OUR SUMMER WITH REPAST: FORGING A MODELING AND SIMULATION FOUNDATION

T.A. BERGEN-HILL, *MITRE Corporation, McLean, VA

M.T. MCMAHON, MITRE Corporation, McLean, VA

B.F. TIVNAN, MITRE Corporation, McLean, VA

ABSTRACT

The success of a large-scale agent-based model and simulation (ABMS) depends on finding the right development tools for each phase of its development lifecycle. To that end, the authors spent three months evaluating free agent-based modeling software, with the goal of acquiring a tool set that would start with rapid prototyping and would progress to parallel and distributed runs on a cluster-type computing environment. The products we reviewed during this study include NetLogo, Repast J, Repast Symphony, and ProActive. In this paper, we describe the efforts required to create and maintain various models within each of these products, along with their respective strengths and limitations. We present our results as a progression through a candidate tool set that will serve as our foundation for upcoming ABMS efforts.

Keywords: Repast, NetLogo, ProActive, cluster

INTRODUCTION

During the development lifecycle of an agent-based model and simulation (ABMS), different needs arise at each phase. Initially, a “quick-and-dirty” prototype is required to verify the problem domain with the customer and to preview the simulation’s look-and-feel. After feedback is received, what follows is the formulation of the model and a subsequent semi-robust implementation. Once it is working sufficiently, providing reasonable results for a small input set, the model is pushed to its computational limits and beyond, requiring the eventual use of large-scale computational hardware. The success of the simulation depends on finding the right development tools for each phase.

To that end, the authors spent three months evaluating free ABMS software, noting the steps required to utilize each tool and to transition the simulation to another tool. The goal of this research is to acquire a tool set that would start with rapid prototyping and would progress to a system capable of parallel and distributed runs on a cluster-type computing environment.

Our requirements include the capability of deploying a prototype model developed on a Windows workstation to a distributed-memory Linux cluster. In addition to investigating the issues with cross-platform development (e.g., Windows workstation to Linux workstation), we also discuss the issues with transitioning to parallel models (e.g., Linux workstation to Linux

* Corresponding author address: Tobin A. Bergen-Hill, The MITRE Corporation., 7515 Colshire Drive, McLean, VA 22102-7539; e-mail: tbergenhill@mitre.org.

cluster). Using the parallelism terminology found in North and Macal (2007), the initial goal for the cluster capability is “coarse-grained” parallelism, i.e. a parameter sweep for a trade study or objective function analysis for parameter optimization (Fujimoto 2000, Lin 1994). The eventual goal is a “fine-grained” parallel simulation, i.e., the use of separate processes within the model itself.

In this paper, we describe the efforts required to create and maintain various models within each of these toolkits, along with their respective strengths and limitations. We note which products have the ability to co-exist in a simulation. Where there are conflicts or issues, we investigate whether the problems are intrinsic to the tool itself or whether they stem from the ongoing development of the tool. We present our results as a progression through a candidate tool set that will serve as our foundation for upcoming ABMS efforts, and we discuss the decisions made for selecting the tools. Our hope is that others within the modeling and simulation community who are at the initial stages of a project and who have considered any of these tools can benefit from our experience and recommendations, and that they can apply our template to make their project a success.

Related Work

One of the more similar efforts we’ve found is the HLA_GRID_Repast system from a consortium led by the University of Birmingham (Zhang et al. 2005). As a system for executing large-scale distributed ABMS over the Grid, it is definitely relevant. However, we did not want to be bound by a single protocol for network communication nor by a specific target system. We’re seeking a flexible framework that allows us to choose the network protocol for the situation and to run on a variety of operating systems and hardware configurations.

Another similar effort is the MACE3J testbed. According to Gasser and Kakugawa (2002), it is designed for multiple granularities of parallel execution and can run on a variety of work stations. We recognize that MACE3J is a likely candidate for further evaluation; however, our time constraints for a related, development project restricted the number of toolkits that would receive detailed attention.

RESEARCH

Seeking a progression of toolkits that would lead us from prototype to a massively parallel ABMS, we reviewed multiple existing tools across different development environments and systems. We started with the list of free toolkits cited in the ABMS text by North and Macal (2007), and narrowed it down to a handful of candidates. Before presenting our review notes, we’ll briefly discuss our approach to simulation parallelization.

Parallelization Approach

When refactoring a simulation to a distributed architecture, there are numerous tradeoffs to consider. In a shared-memory parallel (SMP) environment, threads can utilize shared memory to exchange data. But in a distributed-memory implementation, agents don’t have access to shared memory and must have an explicit communication facility, which yields model design

overhead as well as runtime communications overhead. Also, the spectrum of parallelization granularity ranges from trade study/parameter sweeps (e.g., batch mode) to simulation-level granularity (e.g., HLA integration) to fine-grained distribution of a single agent. The most suitable level of granularity again is a factor of the desired quantity of network communications verses computational power.

In our investigation, we have kept in mind general parallel design approaches (Buyya 1999): partitioning of a model, inter-process communications, mapping to the hardware, and agglomeration of results. In the modeling and simulation domain, this general approach distills to determining, for a given model, the best approach to partitioning that model in a manner which diminishes run-time overhead (communications), while still enabling distribution across processors in a cluster environment. We describe our partitioning scheme in the Findings section of this paper.

Product Review

The products we reviewed during this study include: NetLogo 4.0 beta, Repast J v3, Repast Symphony Alpha 2, and ProActive v3.2.1. A summary of our notes can be found in Table 1 below. The rest of this section expands on the strengths and limitations of these toolkits. To help contrast the simulation architecture of each toolkit, we provide a comparison of different demonstration models in Appendix A: Demo Model Comparison.

Table 1 Toolkit Comparison Chart.

Toolkit	Strengths	Limitations
NetLogo	<ul style="list-style-type: none"> • Easy-to-use language • Java interface • 2D & 3D display 	<ul style="list-style-type: none"> • Not meant for large scale simulations • No multithread support • No distributed simulation support • No batch run support
Repast J	<ul style="list-style-type: none"> • Handles medium-scale numbers of agents • Contains several useful simulation classes • 2D display • Supports multithreaded events • Supports distributed batch runs 	<ul style="list-style-type: none"> • Designed for single host execution per run, not for distributed simulation • No 3D display • Made obsolete by Repast Symphony
Repast S	<ul style="list-style-type: none"> • Handles medium-scale numbers of agents • Contains several useful simulation classes • 2D & 3D display 	<ul style="list-style-type: none"> • Still in development • No distributed simulation support • Bound to only one development environment (Eclipse) • Currently cannot function with

	<ul style="list-style-type: none"> • Supports multithreaded events 	ProActive in a distributed fashion
ProActive	<ul style="list-style-type: none"> • Supports distributed processes (and consequently large-scale simulations) • Minimally intrusive API • Configure distribution through XML file 	<ul style="list-style-type: none"> • No simulation architecture • No visualization capabilities • Clutters the command line • Makes debugging difficult

The selection of these toolkits over others warrants explanation. One toolkit similar in purpose and structure to Repast is Swarm (Minar et al. 1996); however, we chose to review Repast because of its availability in Java and because of our familiarity with Repast. We also selected NetLogo to review instead of StarLogo (Colella et al. 2001) because of our existing proficiency with NetLogo and because of NetLogo's ability to interface with native Java 1.5 applications.

NetLogo

An ideal rapid-prototyping toolkit for ABMS is NetLogo (Wilensky 1999). We started with version 3.1.4 and migrated to version 4.0 beta 5 halfway through the summer. NetLogo has many strong points. Its script language is easy to use, which is no surprise given that it was originally developed for educational purposes. It features a "Java extensions" interface that links the NetLogo script with external Java routines. The display, which is essential to understanding and debugging the simulation, can switch between 2D and 3D, although the simulation is based on a 2D grid. And there are over 150 models that come with the distribution, to serve as examples.

As a prototype tool, NetLogo is not meant for large-scale simulations. There is no support for multi-threading natively in NetLogo, only through farming the work to threaded Java code. Similarly, NetLogo is not set up to support distributed simulations; its HubNet feature is a form of participatory simulation meant for human interaction, not high performance computing.

Repast J

Repast, or the REcursive Porous Agent Simulation Toolkit, is a medium-scale ABMS toolkit that has various forms (North et al. 2006); for our study, we focused on Repast for Java (Repast J) and Repast Symphony (Repast S). Repast J provides an execution environment with an optional GUI for controlling the execution and monitoring of the simulation. It features several useful simulation classes including a discrete-event scheduler, representation of the model space, batch-mode utility classes, and a built-in 2D visualization capability. For agent representations, Repast J has no explicit agent class, but it does offer adaptive behavioral tools.

For our purposes, we used a subset of the Repast J classes:

- *SimModelImpl*: This is a partial implementation of the SimModel interface, through which Repast drives the simulation. Each Repast simulation must provide a SimModel subclass.

- *Schedule*: This serves as a discrete-event scheduler. The simulation can specify the order in which events (or “actions”) are executed.
- *BasicAction*: All simulation events inherit this interface class in order to be stored and executed by the scheduler.
- *SimInit*: This class is responsible for loading and executing the model, with optional GUI controls or batch mode operation.
- *DistributedSimInit*: This experimental stand-in for SimInit uses ProActive to distribute batch runs of the model across a given set of hosts.
- *Random*: This class encapsulates CERN’s Colt random number utilities. An instance of this is placed in SimModelImpl, and the seed can be either generated from a timestamp or explicitly specified for reproducibility.

Both versions of Repast feature a scheduler that supports concurrent actions, simply by specifying a non-zero duration argument. But there are no constructs to support a distributed simulation, only distributed batch runs. The Parameter Wizard makes it easy to specify parameter sweeps for batch runs. It produces a parameter file, which can be passed in on the command line arguments to the SimInit class. SimInit will run the model in batch – but in serial fashion, not simultaneously.

To perform simultaneous distributed batch runs, Repast-J provides an undocumented mechanism that utilizes ProActive. DistributedSimInit establishes the virtual nodes from the ProActive descriptor and creates RemoteBatchController instances on each virtual node. The HomeController class tells the group of RemoteBatchControllers which model to instantiate and run; it also hands them the parameters to use for that run. We’ve noted how to run simultaneous batch runs in Appendix B: Simultaneous Multi-Host Batch Runs In Repast J.

Repast S

Currently in the alpha stage of development, Repast Symphony is a complete redesign of the Repast toolkit. As described by North and Macal (2007), “the Repast S runtime is designed to include advanced features for agent storage, display and behavioral activation, as well as new facilities for data analysis and presentation.”

Repast S makes use of configuration files (model.score and scenario.xml) to specify the roles of the classes in a simulation model. As stated in its preliminary documentation, the central class is the Context, which organizes the agents, denotes the relationships of its members (through “projections”) and holds numerical data in data layers. There is also the ContextBuilder, which does the work of storing information in the Context. The configuration files identifies the ContextBuilder, the agent classes, and which Context to use.

Of the toolkits reviewed, we encountered the most platform-related issues when working with Repast Symphony and Eclipse, the development environment to which Symphony is tied. The case of the file names, ISO encoding of data files, and use of Java annotations (e.g. @override) created compile and run-time issues on Linux systems. Also, Symphony binds us to a specific Java version; on Fedora Core 5, SELinux inhibits the use of Sun’s Java. Most importantly, though, is the different packaging used for the platform-independent distribution of Repast Symphony versus Windows, which uses the intrinsic Eclipse packaging system. Luckily, we found workarounds to these issues to allow us to continue research.

When integrating the ProActive toolkit into our Repast Symphony simulation, we experienced interference from the Repast Symphony runtime environment – something that we didn't encounter with Repast J. First, in order to allow third party JAR files to be used, we had to place them within the directory containing the `repast.simphony.core` package, and we had to explicitly add the names of the JAR files to that package's manifest and build properties. (With Repast J, it was sufficient to include the JARs in the ProActive descriptor's class path variable.) Second, when we configured the simulation to spawn separate processes with their own JVMs from within the simulation, any serialized application-level class that is passed to these processes results in an exception. The spawned Repast S process cannot resolve the temporary ProActive “stub” class. (The Repast J spawned processes had no problems with class resolution.) Finally, we could not locate the means for activating code upon simulation end. This is required in order to properly clean up the ProActive processes. While the `ModelInitializer` interface provides a “hook” that is activated upon startup, no such “hook” exists for shutdown – even overriding `Object.finalize()` did not work.

We also attempted to investigate the batch run capability of Repast Symphony on a Linux environment. But, following the instructions provided in the Alpha release, we were unsuccessful. Again, we anticipate that the next release will hopefully address these issues.

ProActive

As described by Baude et al. (2006), ProActive is “a Java-based middleware (programming model and environment) for object and component oriented parallel, mobile and distributed computing.” It provides a minimally intrusive interface to managing distributed object instances. Consequently, it does not have any built-in simulation architecture or visualization capabilities, which makes it an ideal supplementary package to Repast.

To distribute an object, the application uses a ProActive method to turn it into an “active object” which can then be placed on a specified host, or “virtual node”. The application interacts with the object through the normal method calls, as long as the arguments to the methods can be serialized. Return values from objects cannot be immediately accessed, since they are only proxies to the value. The application can check whether the proxy's value has arrived prior to accessing it.

The main configuration file for ProActive applications is called the Descriptor (or PAD), because it describes which machines are available and how many JVMs to run on each. It also describes the class path for the spawned processes. The file is not quite platform independent; on Windows machines, none of the specified paths should contain any spaces.

To support load balancing, ProActive provides the means to migrate active objects between virtual nodes. There is a fair amount of overhead to each active object, however. For large-scale simulations that need to migrate agents, an alternative is to make the agent serializable instead of an active object. Then the agent can be passed between virtual nodes using regular method invocations.

Running a ProActive-enabled application requires the use of extensive scripts that configure environment variables, add many JAR files to the class path, and JVM command-line arguments. Consequently, debugging a ProActive application within an IDE (such as NetBeans

or Eclipse) is difficult because all these run-time parameters must be reproduced. We have yet to be successful in using a debugger with ProActive.

Both distributions of Repast come with ProActive file. However, this version is not current. In the case of Repast-J, which includes ProActive classes within its JAR file, this causes problems on Linux with NetBeans: when specifying the JAR files in the Libraries section of NetBeans, ProActive.jar must appear above repast.jar in order to use the more recent version.

FINDINGS

From the set of toolkits reviewed, we have found two potential paths of progression for an ABMS, as illustrated in Figure 1. The “current” path reflects what is possible with the current state of the toolkits; it is a complete progression from prototype to a large-scale distributed simulation. The “future” path is based on promised capabilities from the Repast Symphony development team.

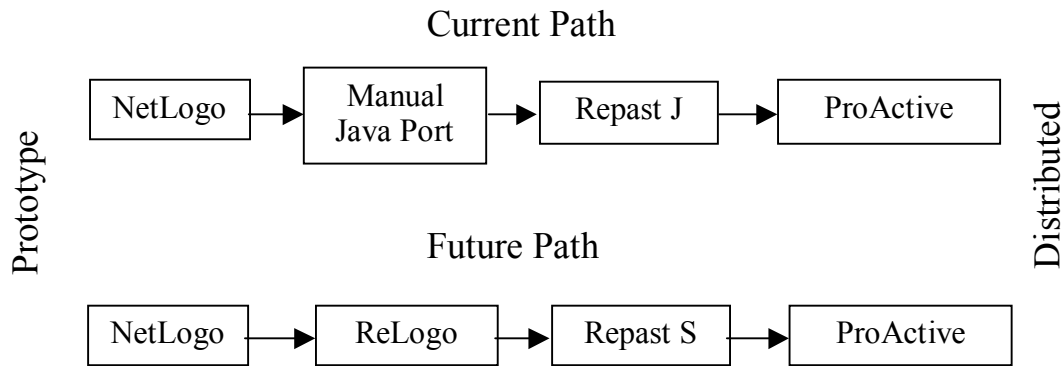


Figure 1 ABMS Development Progression Paths.

Current Path

Using the toolkits in their current state, we have had success with the following progression of development:

- Stage I: Prototype in NetLogo
- Stage II: Pure Java
- Stage III: Repast J simulation
- Stage IV: Distribute simulation via ProActive

The next few sections discuss the transition between these states.

Stage I to Stage II

Going from NetLogo script to Java code is currently a manual process. Initially, this process can start by using Java extensions from within the NetLogo script for key components of the simulation's business logic. But to go completely to Java, the NetLogo representations of turtles and patches require a Java counterpart, along with the visualization and event loop. The turtle classes will require movement methods, and both the turtles and patches will need NetLogo-like query operations. It turns out that the turtle classes become the simulation's agent classes.

It is debatable whether it is necessary to put the simulation into an intermediate pure Java form, or whether to go directly to using the Repast J classes. It is easier to debug the simulation's logic when in pure Java. However, the port will produce temporary classes that end up being replaced by their Repast J counterparts (e.g. schedule, visuals, and terrain). Working directly in Repast will prevent redundant work.

Once the port is complete, the NetLogo prototype can still be developed, as long as the changes are tracked and reproduced on the Java side. But at some point, modifications must stop flowing down.

Stage II to Stage III

Adapting a Java simulation to the Repast-J framework is relatively straightforward because the requirements are minimal:

- Provide a subclass of SimModelImpl with the begin() and setup() methods overridden
- Schedule events using the Schedule and BasicAction classes
- Fire a stop event when the simulation has finished

Repast J does not have an agent interface class, so there are no restrictions on the agent representation.

With regards to visualization, the simulation could use the 2-D visuals provided by Repast J. In order for agents to be rendered, the agent class must implement the Drawable interface. Also, the SimModelImpl subclass must create some representation of the model space (e.g. Object2DTorus), a DisplaySurface and an Object2DDisplay. Alternatively, the simulation could use a third-party visualization toolkit, such as Java3D. We chose to use Java3D because there are certain advantages to having the viewpoint at the agent level (e.g. obscuration of field of view affecting agent logic).

Another reason to use a separate visual toolkit is to prepare the simulation for a distributed, batch run. The results can be logged to data files and replayed through the visuals. If there's a need to see the visuals during the simulation run, then a communication link can be established between the Repast process(es) and the visual process. But this will introduce a performance penalty.

Stage III to Stage IV

Going to a distributed framework is the most difficult transition to make because of the impact on the simulation's architecture. The simulation engineer must determine how to

partition the execution across different machines while keeping the amount of network traffic to a minimum. The initial impulse is to make the agents themselves become separate processes; but if the agents require constant communication with its neighbors, the overhead of the supporting network communications will outweigh any benefit to be had by distributing the simulation.

Instead, we recommend that the engineer work at a larger level, deciding how to carve out large pieces of the simulation that can operate nearly independently, which we'll call a "domain." Each domain will be responsible for managing all agents within an area – in the same process. To address the boundary condition problem - e.g., when an agent travels between domains – we can learn from parallel battlefield simulations and define the domains geographically (Nicol 1988). Each domain should also write its results to a uniquely named log file. When running on a cluster environment, we recommend writing to a file on the local file system. Otherwise, the cost of writing to a NFS-mounted drive negates the performance benefits of distributed processing.

Since the `SimModelImpl` subclass cannot be serialized and passed to the domains, another class is required to act as a coordination object. Domains would send any needed run-time statistics to the coordination object (in addition to logging data locally); however, this should be kept to a minimum because it is a major performance bottleneck. Domains can also notify the coordination object when they've finished, which can signal Repast J that the run has ended.

When the simulation begins, the following should occur within the `SimModelImpl` subclass's `begin()` method:

- Read in the PAD file
- Activate the virtual nodes
- Create the domains as a `ProActive` group, using the list of available virtual nodes
- Create the coordination object as an active object and pass it to the domains
- Schedule an action whose `execute()` method performs the following:
 - o Check with the coordination object to see whether the domains have finished, and if so, fire a stop event
 - o Display statistics (if enabled)
 - o Update the domains
 - o Schedule another action

Once the simulation ends - either because the domains indicated that they have finished, or because the user presses the stop button - a stop event will be triggered. The `SimModelImpl` subclass needs to handle this event by cleaning up the `ProActive` active objects and killing all the JVMs on the virtual nodes.

After the simulation has been outfitted with `ProActive`, it must be tested to make sure the functionality has been preserved. We recommend using the following progression of PAD files:

- One virtual node that is using the same JVM as the Repast application
- Multiple virtual nodes that are using different JVMs but on the same host as the Repast application
- Multiple virtual nodes, each on a different host

The final step is adding the ability to perform simultaneous batch runs using Repast J and ProActive. For details on how to accomplish this, see Appendix B: Simultaneous Multi-Host Batch Runs In Repast J.

Future Path

Based on the limited information we've heard about the next release of Repast Symphony, we anticipate that the following progression will be possible:

- Stage I: Prototype in NetLogo
- Stage II: ReLogo
- Stage III: Repast Symphony simulation
- Stage IV: Distribute simulation via ProActive

The next few sections explain the transitions between these states.

Stage I through Stage III

Our speculation is that Repast Symphony will be packaged with a utility dubbed "ReLogo", which will parse NetLogo script and turn it into Repast Symphony Java code. This will make obsolete the manual process of converting NetLogo to Java from the "current" path. The anticipated release of ReLogo is November 2007.

Stage III to Stage IV

As in the "current" path, the process of outfitting a Repast simulation with ProActive constructs is still performed manually. The intent was to demonstrate at summer's end the ability to run a Repast Symphony simulation in a distributed fashion. However, we were unable to successfully invoke methods on active objects that reside in a separate JVM (either on the local host or a remote host).

The test application we used is a modified version of ProActive's n-body simulation. We added the minimum constructs necessary to convert it to a Repast S simulation. For detailed steps on this procedure, see Appendix C: Repast Symphony – ProActive Integration Notes.

NEXT STEP

While we are content with using the "current" path as a template for our ABMS work, we look forward to the streamlined process featured in the "future" path. We hope to see the ReLogo mechanism in the next release of Repast Symphony, and we hope to solve the issues preventing ProActive from reaching its full potential in Repast S.

We wish to explore other ideas to reduce the work involved. Since the use of ProActive centers on properly configured descriptor files, we would like to construct a tool to compose and generate PAD files. Another improvement would be to write a better parameter sweep mechanism in Repast that would vary parameters while running on different groups of hosts; this

would require removing the group name as a parameter and providing it through some other means of input. This custom parameter sweep could also support varying multiple parameters per run, going beyond what is provided in Repast. We are also seeking ways to incorporate ProActive constructs in a semi-automatic fashion, perhaps as custom hooks within the ReLogo conversion process.

REFERENCES

- Baude F., L. Baduel, D. Caromel, A. Contes, F. Huet, M. Morel and R. Quilici, 2006, "Programming, Composing, Deploying for the Grid", *GRID COMPUTING: Software Environments and Tools*, pp. 205-229, London, England: Springer-Verlag.
- Buyya, R., 1999, High Performance Cluster Computing, Volume 1, Architectures and Systems, NJ: Prentice Hall.
- Colella, V. S., E. Klopfer, M. Resnick, 2001, *Adventures in Modeling: Exploring Complex Dynamic Systems with StarLogo*, New York, NY: Teachers College Press.
- Fujimoto, R.M., 2000, *Parallel and Distributed Simulation Systems*, New York, NY: Wiley And Sons
- Gasser, L. and K. Kakugawa, 2002, MACE3J: Fast Flexible Distributed Simulation of Large, Large-Grain Multi-Agent Systems, *The First International Joint Conference on Autonomous Agent and Multiagent Systems, Vol. I*, pp. 745-752, New York, NY: ACM Press.
- Lin, Y., 1994, Parallel independent replicated simulation on a network of workstations, *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*, pp. 73-80, New York, NY: ACM Press.
- Minar, N., R. Burkhart, C. Langton, and M. Askenazi, 1996, The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations, Working Paper 96-06-042, Santa Fe, NM: Santa Fe Institute.
- North, M. and C. Macal, 2007, *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*, New York, NY: Oxford University Press.
- North, M.J., N.T. Collier, and J.R. Vos, 2006, "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit," *ACM Transactions on Modeling and Computer Simulation*, Vol. 16, Issue 1, pp. 1-25, New York, NY: ACM Press.
- Nicol, D.M., 1988, Mapping a battlefield simulation onto message-passing parallel architectures, *Distributed Simulation*, pp. 141-146, San Diego, CA: Society for Computer Simulation International.
- Wilensky, U., 1999, *NetLogo*, Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University.

Wilensky, U., 2002, *NetLogo Mousetraps model*, <http://ccl.northwestern.edu/netlogo/models/Mousetraps>, Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University.

Zhang, T., G. Theodoropoulos, R. Minson, S. Turner, W. Cai, X. Yong, and B. Logan, 2005, Grid-aware large scale distributed simulation of agent-based systems, European Simulator Interoperability Workshop (Euro-SIW) paper 05E-SIW-047, Toulouse, France.

APPENDIX A: DEMO MODEL COMPARISON

The comparison of various demonstration models distributed with the toolkits can be found in this appendix. These examples supplement the discussion of the toolkits found in the Product Review section.

NetLogo MouseTrap

One model that comes with three of the toolkits is the MouseTrap simulation, which originally came from the Swarm toolkit (Minar et al. 1996). The version for NetLogo (Wilensky 2002) is simply represented by three functions:

- setup: resets the traps (i.e. the patches) and the balls (i.e. the turtles)
- go: controls the triggering of the traps and introduction of new balls
- move: move the ball in a random direction up to a specified maximum distance

Repast J MouseTrap

The Repast-J representation of MouseTrap takes the form of three classes:

- *MouseTrapModel*: a subclass of *SimModelImpl*. This class manages the simulation, houses the simulation space and display, and tracks the current number of active balls. The *begin()* method constructs the model out of MouseTrap classes. During the run, its *scheduleTrigger()* method schedules a *TriggerAction* for each trap that fires.
- *MouseTrap*: the representation of the “agent”. The *trigger()* method randomly chooses which neighboring traps are triggered by the balls launched by the trap, passing them to the model’s *scheduleTrigger()* method.
- *TriggerAction*: a subclass of *BasicAction*, this holds a reference to a MouseTrap instance. The *execute()* method triggers the trap and updates the display.

Repast S MouseTrap

The Repast S representation of MouseTrap includes the following core classes:

- *MouseTrap*: This is the agent representation, as designated in the model.score file. Its *trigger()* method is similar to the Repast J version.

- *MouseTrapsCreator*: This is the ContextBuilder for the demo and is designated as such in the model.score and scenario.xml files. It creates a grid projection for the context and fills it with MouseTrap agents. It also adds a data layer to the grid projection.
- *TrapInitializer*: This ModelInitializer subclass triggers the first trap. The scenario.xml file identifies this as the model's initializing class.
- *TrapTrigger*: Implementing the IAction interface, this class is similar to the TriggerAction class in the Repast J version.

ProActive n-Body

Unfortunately, there was no MouseTrap simulation that came with ProActive. Instead, we'll use an n-body simulation to demonstrate how a ProActive ABMS is structured. Below is a description of the main classes used in the n-body simulation:

- *Domain*: This class manages each Planet in the simulation, calculating the forces exerted on it by its neighbors. It relies on distributing the resulting sum to all other Domains (and their planets). It then notifies the Displayer to update the planet's position. This is one of the "active objects" in the simulation.
- *Start*: This class reads the PAD file and activates the virtual nodes, then creates the Displayer, the Planets and their Domains. It launches the Domain processes on the virtual nodes, using the ProActive group construct. When the Domains finish the specified number of iterations, they notify this class.
- *Displayer*: Acting as a wrapper for the planetary display, this class is handed to each Domain by the Start class. Like the Domain, this is also an active object.
- *Planet*: This is the "agent" of the n-body simulation, and has several physical properties to regulate its motion.

APPENDIX B: SIMULTANEOUS MULTI-HOST BATCH RUNS IN REPAST J

This appendix discusses how to utilize Repast J and ProActive to perform simultaneous batch runs. The main difficulty is to prevent each run (which itself uses multiple hosts) from executing on remote hosts that are currently executing other runs. To accomplish this, the DistributedSimInit class can be used in combination with a special parameter sweep:

1. Make multiple virtualNode mappings in the XML file:
 - "remote": these are the nodes that will have the SimModelImpl subclass
 - "WorkersN": these are the groups of nodes that will perform the work; N is the number of nodes in the "remote" group
2. Add a property: vnMapping; this will indicate which virtual node mapping for the model to use for its workers
3. Define a property sweep file where the inner-most sweep is the vnMapping property; have it cycle through the different WorkersN values.
4. In the model's main() method, have it instantiate a DistributedSimInit class. Call its open() method giving it the PAD filename, the model class name, and the sweep file.

Since `vnMapping` is a parameter that is being swept, and since the sweeping mechanism only varies one parameter per run, the same parameters will be run across all workers sets, which limits its usefulness. However, each run will use a different random seed, though – based on time.

`HomeController` creates a persistent `DataWriter`, but does not specify its file name. After all the batch runs occur, it tells `DataWriter` to write the current time at the end of the file; since the filename is null, the exception occurs. To work around this, create a `DataRecorder` in the model's `begin()` method. This will automatically set the filename of `DataWriter`, so when the batch runs finish, no exception will be thrown, and everything is cleaned up properly.

APPENDIX C: REPAST SIMPHONY – PROACTIVE INTEGRATION NOTES

The following notes reflect the steps needed to incorporate Repast Symphony and ProActive into an ABMS. They are geared towards a Windows environment.

- Repast S (and Eclipse) must be stored in a path that does not have spaces in it. For example: `c:/program files/RepastS` will cause problems with ProActive, but `c:/projects/RepastS` is fine.
- Create a new project in Eclipse according to the write-up
- In the `model.score` file, point the Base Path at `../../repast_workspace/PROJECT_DIR`
- Add attributes only to the `model.score` file; nothing else is required by RepastS
- Have the main model file inherit `ContextBuilder`, and provide a `build()` method
- To the project, add external libraries for ProActive:
 - o Right-click on the project and choose `Build Path->Add External Archives...`
 - o Browse to the ProActive directory and add the following:
 - `ProActive.jar`
 - `ic2d.jar`
 - o Repeat and browse to `ProActive/lib` and add the following:
 - `xercesImpl.jar`
 - `bouncycastle.jar`
 - `javassist.jar`
 - `jsch.jar`
 - `log4j.jar`
 - o Repeat and browse to `ProActive/lib/components` and add the following:
 - `fractal.jar`
 - o Copy these ProActive JARs to `RepastS/repast.simphony/repast.simphony.core/lib`
 - o Add `fractal.jar`, `xercesImpl.jar` and `bouncycastle.jar` to the RepastS runtime component

- In Eclipse, locate the repast.simphony.core project and open the MANIFEST.MF file
- Of the various tabs for the manifest, choose Runtime
- In the Classpath section, click New... then enter lib/your_lib.jar and click OK.
- NOTE: This will add it automatically to the MANIFEST.MF file as well as the build.properties file
- In Eclipse, open the plugin_jpf.xml file in the repast.simphony.core project
- At the end of the <runtime> section, add a line for your JAR file, e.g.:
 - <library id="your_lib" path="lib/your_lib.jar" type="code" />
- Add a PROJECT.launch file to your project
 - Click on the pull-down menu for the Run button and choose Run...
 - Select the "SimpleHappy" run configuration and click the Duplicate button
 - Rename it to your project
 - On the Arguments tab:
 - Set the Program Arguments to "../..../repast_workspace/nbody/scenario.rs" (without the quotes)
 - Set the VM Arguments to "-Djava.security.policy=file:c:/projects/proactive/scripts/proactive.java.policy" (without the quotes)
 - [Note: adding -Dlog4j.configuration=file:c:/projects/proactive/scripts/proactive-log4j actually prevents the log from appearing in the console]
 - On the Common tab, click the Browse button next to "Shared File" and choose your project
 - Click the Apply button, then Close

This will allow you to run the simulation with a single virtual node running on the local JVM. When the active objects are placed in a different JVM, the active objects are created without problems. But when an active object is passed from the main process to the other process, ProActive throws a ClassNotFound exception. The error still appears even when the application classes are explicitly added to the classpath section of the PAD file.

