AN INTRODUCTION TO REPAST SIMPHONY MODELING USING A SIMPLE PREDATOR-PREY EXAMPLE

ERIC TATARA, Argonne National Laboratory, Argonne, IL
M.J. NORTH,* Argonne National Laboratory, Argonne, IL,
and The University of Chicago, Chicago, IL
T.R. HOWE, Argonne National Laboratory, Argonne, IL
N.T. COLLIER, Argonne National Laboratory, Argonne, IL,
and PantaRei Corp., Cambridge, MA
J.R. VOS, Argonne National Laboratory, Argonne, IL,
and the University of Illinois at Urbana-Champaign, Urbana, IL

ABSTRACT

Repast is a widely used, free, and open-source, agent-based modeling and simulation toolkit. Three Repast platforms are currently available, each of which has the same core features but a different environment for these features. Repast Simphony (Repast S) extends the Repast portfolio by offering a new approach to simulation development and execution. This paper presents a model of wolf-sheep predation as an introductory tutorial and illustration of the modeling capabilities of Repast S.

Keywords: Agent-based modeling and simulation, Repast, toolkits, and development environments

INTRODUCTION

Repast (ROAD 2005) is a widely used, free, and open source, agent-based modeling and simulation toolkit with three released platforms, namely Repast for Java, Repast for the Microsoft .NET framework, and Repast for Python Scripting. Repast Simphony (Repast S) extends the Repast portfolio by offering a new approach to simulation development and execution, including a set of advanced computing technologies for applications such as social simulation. North, Howe, Collier, and Vos (2005a and 2005b) provide an overview of the Repast S runtime and development environments.

We use a model of wolf-sheep predation to demonstrate the capabilities of the Repast S toolkit and as an introductory tutorial. While the example is not intended to model real phenomenon, the model's complexity is high enough to illustrate how the user may develop multi-agent models. Spatial and temporal patterns emerge in the model consisting of potentially hundreds of instances of three agent types.

It is important to note that Repast S and its related tools are still under development. This paper presents the most current information at the time it was written. However, changes may occur before the planned final release.

* *Corresponding author address*: Michael J. North, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439; email: north@anl.gov.

THE REPAST S MODEL IMPLEMENTATION BUSINESS PROCESS

As discussed in North et al. (2005a and 2005b), the Repast S model implementation business process is as follows:

- The modeler creates model pieces, as needed, in the form of plain old Java objects (POJOs), often using automated tools.
- The modeler uses declarative configuration settings to pass the model pieces and legacy software connections to the Repast S runtime system.
- The modeler uses the Repast S runtime system to declaratively tell Repast S how to instantiate and connect model components.
- Repast S automatically manages the model pieces based on (1) interactive user input and (2) declarative or imperative requests from the components themselves.

The POJO model components can represent anything, but are most commonly used to represent the agents in the model. While the POJOs can be created using any method, this paper discusses one powerful way to create POJOs for Repast S: the Repast S development environment. However, modelers can use any method—from hand coding to wrapping binary legacy models to connecting into enterprise information systems—to create the Repast S POJO model components.

Regardless of the source of the POJOs, the Repast S runtime system is used to configure and execute Repast S models. North et al. (2005b) detail the Repast S runtime system, the design of which includes:

- Point-and-click model configuration and operation
- An integrated two-dimensional or three-dimensional geographical information system (GIS), and other model views
- Automated connections to enterprise data sources
- Automated connections to powerful external programs for conducting statistical analysis and visualizing model results.

WOLF-SHEEP PREDATION MODEL

We implement a model of wolf-sheep predation (Wilenski 1998) in Repast S as a demonstration of the toolkit's capabilities. This model represents a simple variation of predator prey behavior using three agents: wolves, sheep, and grass. Both the wolves and sheep move randomly on a grid, and the movement has a cost in the form of lost energy. The wolves and sheep need to eat food in order to replenish their energy, and they will die once their energy level reaches zero.

Wolves prey on sheep and may eat them if the two are located in the same spatial position, thereby increasing the wolf's energy level. Sheep may similarly eat grass if the sheep is located on a patch which contains living grass. Once a sheep eats the grass in its location, the grass needs to regrow before the sheep can eat it again. Repast S models a re-grow rate for grass by counting down after the grass has been eaten in a specific location. Reproduction is modeled by a random process that creates a child from the parent, divides the energy of the parent agent in half, and assigns the energy equally to the parent and child.

REPAST S IMPLEMENTATION

North et al. (2005a and 2005b) provides details on the Repast architecture and general modeling concepts. The Repast S implementation of the wolf-sheep predation model consists of several classes of interest, which include, specifically, the data loader and agent classes.

The context creator class, which implements DataLoader, constructs the main context and returns it to the Repast run environment as shown in Figure 1. Within the user-designed context creator, modelers create a root context and add the desired agents and projections to the context. In this example, we add a Grid projection to the context to model the world using a discrete Cartesian grid.

Next, we add the agents (i.e., the wolf, sheep, and grass) to the context. We obtain the simulation parameters, which may be set in the graphical user interface (GUI), from the run environment to set the initial number of wolves and sheep. Finally, we add a grass agent to each location on the grid and return the context to the run environment.

SIMPLE AGENTS

The three agent classes in the model extend the SimpleAgent class, which contains common methods related to movement and death, for example. The SimpleAgent class also contains Repast-specific `@ScheduleMethod` annotation, which precedes methods to be scheduled. The `@ScheduleMethod` annotation has several options, including the start time and the updated interval. The SimpleAgent step method has an annotation that specifies the method to be scheduled starting at tick 1 and to recur every one tick thereafter (see Figure 2). In this implementation of the model, the SimpleAgent class has an empty step method that we can override, so we can specify the individual step behavior in subclasses.

Movement on the grid is accomplished by the move method, which simply obtains the Grid object from the agent's context and then obtains the agent's coordinates on the grid. The agent moves by randomly selecting one of the nearest eight neighboring grid positions (i.e., the agent's Moore neighborhood). The "die" method occurs when either the agent's energy level reaches zero, or in the case of sheep, when the agent is eaten. We model the death process by simply removing the agent from its context.

The Wolf class shown in Figure 3 has two constructors. We use the first when the wolf is created from a reproduction process. The energy from the parent wolf is passed into the

```
public class PPContextCreator implements DataLoader {
  public Context create(Object creatorID) {
    int xdim = 50;
    int ydim = 50;

    Context<SimpleAgent> context = Contexts.createContext(SimpleAgent.class, creatorID);
    Projections.createGrid("Simple Grid", context,new DefaultGridParameters<SimpleAgent>(
                          new RandomGridAdder<SimpleAgent>(), true, xdim, ydim));

    Parameters p = RunEnvironment.getInstance().getParameters();
    int numWolves = (Integer) p.getValue("initial number of wolves");
    p = RunEnvironment.getInstance().getParameters();
    int numSheep = (Integer) p.getValue("initial number of sheep");

    for (int i = 0; i < numWolves; i++) {
      Wolf wolf = new Wolf();
      context.add(wolf);
    }
    for (int i = 0; i < numSheep; i++) {
      Sheep sheep = new Sheep();
      context.add(sheep);
    }

    AgentCounter counter = new AgentCounter();
    context.add(counter);

    Grid grid = (Grid) context.getProjection("Simple Grid");

    for (int i=0; I < xdim; i++){
      for (int j=0; j < ydim; j++){
        Grass grass = new Grass();
        context.add(grass);
        grid.move(grass, i, j);
      }
    }

    return context;
  }

  public void load(Context context) {
  }
}
```

FIGURE 1 Repast S context creator for the predator-prey model

constructor and is assigned to the child. We use the second constructor for wolves that are created during model initialization, in which the initial energy is randomized. The step method of the Wolf class overrides the SimpleAgent method. Since the SimpleAgent method is already scheduled, the Wolf class step method does not require an additional annotation. The step method consists of the following processes: moving, reducing energy, catching sheep, reproducing, and dying. (The movement and death methods are described in the SimpleAgent class section.)

A wolf may catch a sheep to eat if a sheep exists on the same grid coordinate as the wolf. The wolf agent obtains the Grid object from its context and scans through the objects at its location. If there is a sheep at the wolf's location, the wolf eats the sheep and increases its energy

```
public class SimpleAgent {
  private int x;
  private int y;

  @ScheduledMethod(start = 1, interval = 1)
    public void step() {
    // override by subclasses
  }

  public void move() {
    Context context = ContextUtils.getContext(this);
    Grid grid = (Grid) context.getProjection("Simple Grid");
    GridDimensions dims = grid.getDimensions();
    GridPoint point = grid.getLocation(this);

    x = point.getX() + (int) Math.round(2*Math.random() - 1);
    y = point.getY() + (int) Math.round(2*Math.random() - 1);

    grid.move(this, x, y);
  }

  public void die(){
    Context context = ContextUtils.getContext(this);
    if (context.size() > 1)
      context.remove(this);
    else
      RunEnvironment.getInstance().endRun();
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }
}
```

FIGURE 2 The SimpleAgent class

according to the gain set in the parameters list. Repast S models reproduction by a random process which generates a random number, and if the number is less than the wolf reproduction rate, a new wolf is created. The energy level of the parent is halved and given in equal parts to the parent and child.

Sheep behavior is very similar to wolf behavior and is specified in the Sheep class as shown in Figure 4. The only major difference between the Wolf and Sheep classes is the manner by which they acquire energy. The sheep scan their location for grass agents and, if the grass is alive, the sheep eat the grass and increase their energy levels. When the grass is eaten, the grass agent's state is switched from alive to dead.

The Grass class shown in Figure 5 again overrides the step method of SimpleAgent. It has a single constructor which randomizes the countdown timer for re-growth and sets the initial state of the grass to alive or dead. The step method simply checks whether or not the grass is dead. If it is, the step method reduces the re-grow timer *or* switches the state of the grass to alive if the timer has expired. If the grass is alive, the step method does not execute any code.

```
public class Wolf extends SimpleAgent {
  private double energy;

  public Wolf (double energy){
    this.energy = energy;
  }

  public Wolf(){
    Parameters p = RunEnvironment.getInstance().getParameters();
    double gain = (Double) p.getValue("wolf gain from food");
    energy = Math.random() * 2 * gain;
  }

  @Override
  public void step() {
    Context context = ContextUtils.getContext(this);

    move();

    energy = energy - 1;  // Reduce energy

    // Catch sheep
    int x = getX();
    int y = getY();

    Parameters p = RunEnvironment.getInstance().getParameters();
    double gain = (Double) p.getValue("wolf gain from food");

    Grid<SimpleAgent> grid = (Grid<SimpleAgent>) context.getProjection("Simple Grid");

    Sheep sheep = null;
    for (SimpleAgent agent : grid.getObjectsAt(x,y)){
      if (agent instanceof Sheep)
        sheep = (Sheep) agent;
    }
    if (sheep != null){
      sheep.die();
      energy = energy + gain;
    }

    // Reproduce
    p = RunEnvironment.getInstance().getParameters();
    double rate = (Double) p.getValue("wolf reproduce");

    // Spawn
    if (100 * Math.random() < rate){
      energy = energy / 2;
      Wolf wolf = new Wolf(energy);
      context.add(wolf);
    }

    // Death
    if (energy < 0) die();

  }
}
```

FIGURE 3 The Wolf class

```
public class Sheep extends SimpleAgent {
  double energy;

  public Sheep(double energy){
    this.energy = energy;
  }

  public Sheep(){
    Parameters p = RunEnvironment.getInstance().getParameters();
    Double gain = (Double) p.getValue("sheep gain from food");

    energy = Math.random() * 2 * gain;
  }

  @Override
  public void step() {
    Context context = ContextUtils.getContext(this);

    move();

    energy = energy - 1;  // Reduce energy

    // Eat Grass
    int x = getX();
    int y = getY();

    Parameters p = RunEnvironment.getInstance().getParameters();
    double gain = (Double) p.getValue("sheep gain from food");

    Grid<SimpleAgent> grid = (Grid<SimpleAgent>) context.getProjection("Simple Grid");

    Grass grass = null;
    for (SimpleAgent agent : grid.getObjectsAt(x,y)){
      if (agent instanceof Grass)
        grass = (Grass) agent;
    }
    if (grass != null && grass.isAlive()){
      grass.setAlive(false);
      energy = energy + gain;
    }

    // Reproduce
    p = RunEnvironment.getInstance().getParameters();
    double rate = (Double) p.getValue("sheep reproduce");

    // Spawn
    if (100 * Math.random() < rate){
      energy = energy / 2;
      Sheep sheep = new Sheep(energy);
      context.add(sheep);
    }

    // Death
    if (energy < 0) die();

  }
}
```

FIGURE 4 The Sheep class

```
public class Grass extends SimpleAgent {
  private int countdown;
  private boolean alive;

  public Grass(){
    Parameters p = RunEnvironment.getInstance().getParameters();
    int regrowTime = (Integer) p.getValue("grass regrowth time");

    countdown = (int) (Math.random() * regrowTime);

    if (Math.random() <= 0.5) alive = true;
    else alive = false;
  }

  @Override
  public void step(){
    if (!alive){
      if (countdown <= 0){
        Parameters p = RunEnvironment.getInstance().getParameters();
        int regrowTime = (Integer) p.getValue("grass regrowth time");

        alive = true;
        countdown = regrowTime;
      } else {
        countdown--;
      }
    }
  }

  public boolean isAlive() {
    return alive;
  }

  public void setAlive(boolean alive) {
    this.alive = alive;
  }
}
```

FIGURE 5 The Grass class

MODEL EXECUTION AND OUTPUT

Once the essential model classes are created, the user may load them into the Repast S runtime shown in Figures 6 and 7. The Repast GUI has a series of buttons along the top menu bar for model loading; setup; playing and pausing; stepping; stopping; and resetting. The tree in the top left pane displays its model components graphically and is interactive, allowing users to create and add various components to the model. The lower left pane shows the table of simulation parameters that users can modify during model execution. Parameters would typically include global model data, such as the initial number of agents or properties that are common to classes of agents, rather than properties associated with an individual agent instance.

The right side of the GUI may contain one or more graphical representations of model data, including time-series charts; bar charts; and two- and three-dimensional grid and network projections. Figure 6 shows the population of sheep over time, starting at tick 0. The oscillations in the population are typical of the dynamic behavior observed in predator-prey systems.

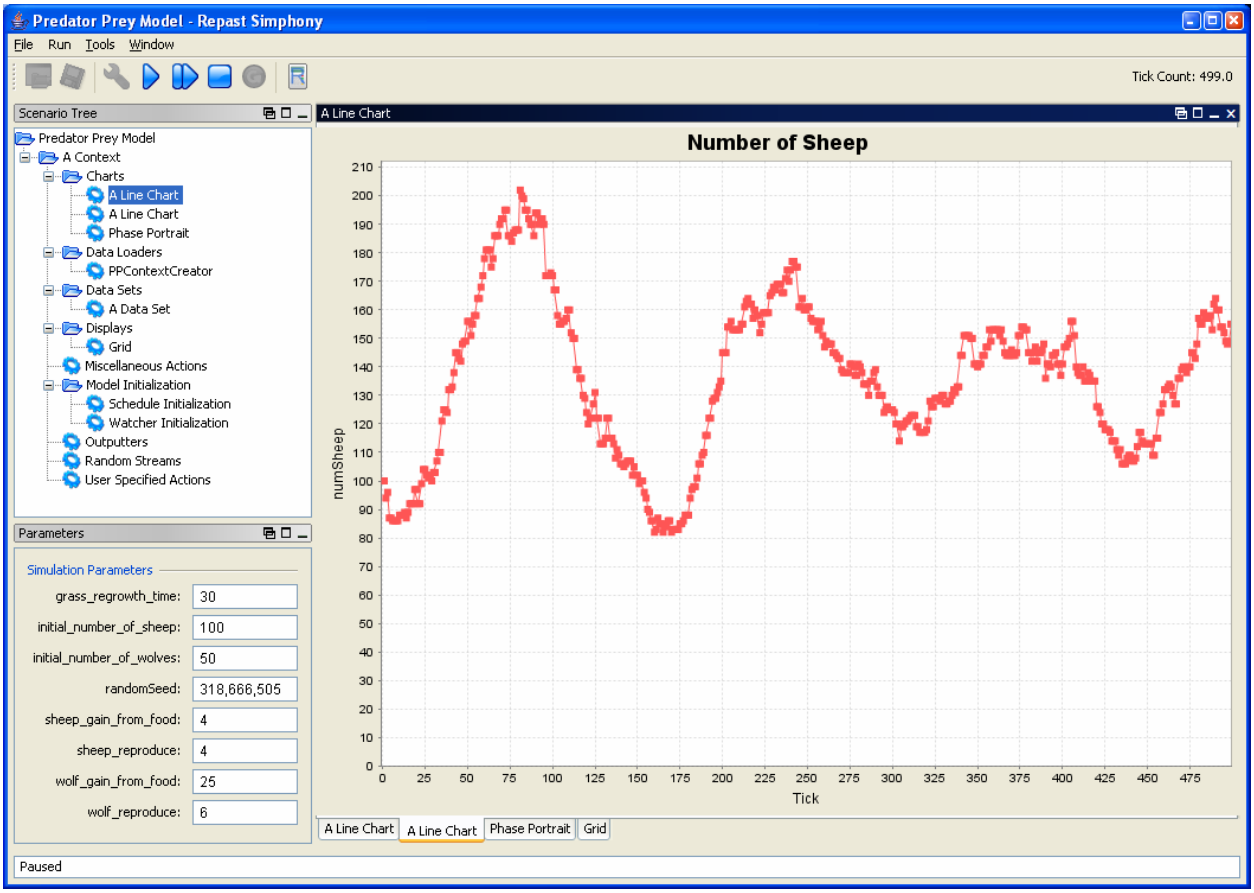

FIGURE 6 A series chart displaying the sheep population over time

Figure 7 shows a three-dimensional display of the grid projection with the wolves and sheep represented by three-dimensional objects. We organize the grass agents on a plane such that one grass agent is positioned at each discreet grid location. The model shown here uses a discrete 50 by 50 grid, with 100 initial sheep and 50 initial wolves, whose spatial positions are randomly determined during the model initiation.

Modelers can update displays at regular user-specified intervals or whenever a move event causes the spatial position of one of the agents to change. Modelers can also undock the two- and three-dimensional displays from the Repast GUI to provide multiple simultaneous model displays. Data may also be saved via one of several types of data logging "outputters." The user may create or modify existing data using these outputters in the GUI model tree. North et al. (2005a and 2005b) provide additional detail on these features.

CONCLUSIONS

The Repast S runtime is a pure Java extension of the existing Repast portfolio. Repast S extends the Repast portfolio by offering a new approach to simulation development and execution. The Repast S development environment is expected to include advanced features for

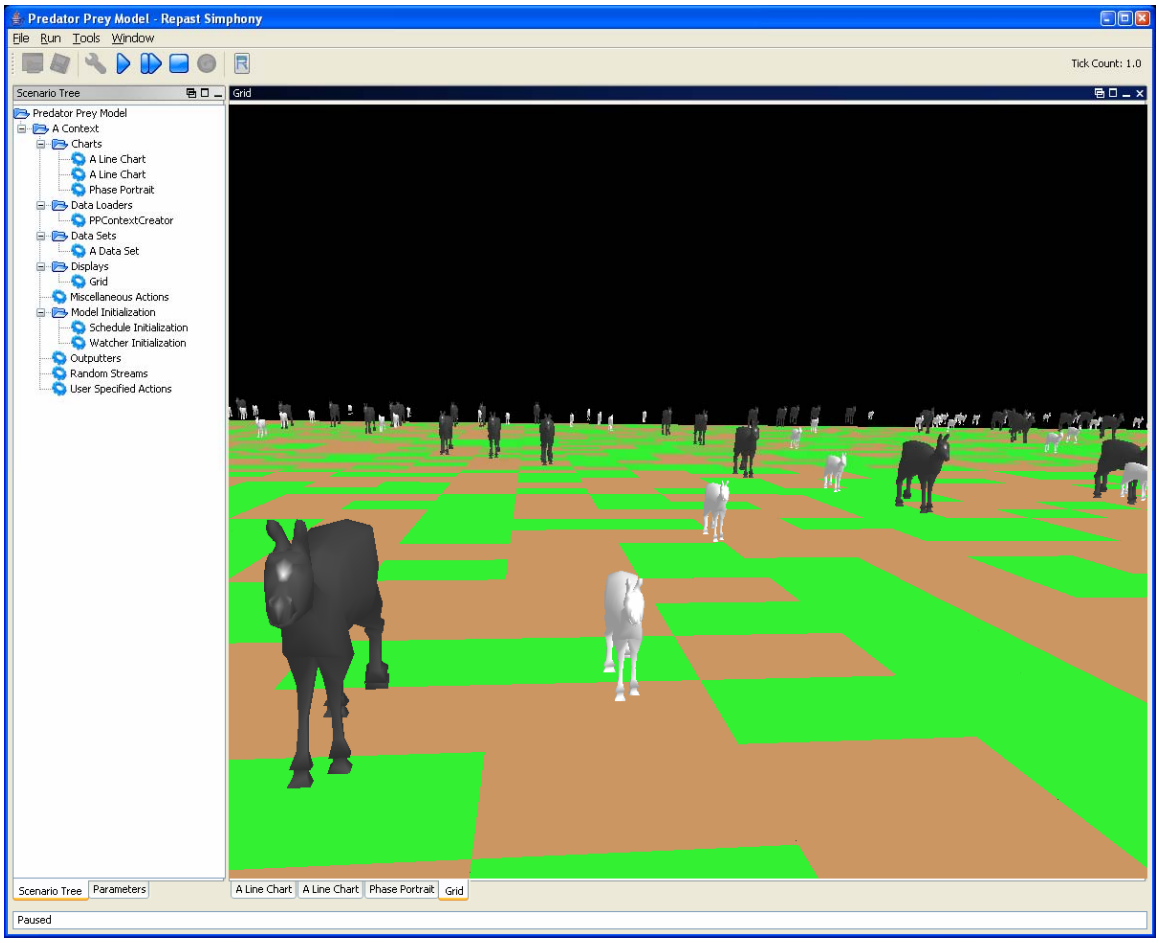

FIGURE 7 3D display of model grid. Sheep are light and wolves are dark. Light and dark squares represent living and dead grass, respectively.

agent behavioral specification and dynamic model self-assembly. Any plain old Java object can be a Repast S model component. This paper presents an introductory tutorial and illustration of the modeling capabilities of Repast S using a simple model of wolf-sheep predation. We described a specific implementation of the model in Repast S with three agent classes by using detailed model source code.

ACKNOWLEDGMENT

The authors wish to thank David L. Sallach for his visionary leadership in founding the Repast project and Charles M. Macal for his sustaining involvement in the project. This work is supported by the U.S. Department of Energy, Office of Science, under contract W-31-109-Eng-38.

REFERENCES

North, M.J., T.R. Howe, N.T. Collier, and J.R. Vos, 2005a, "The Repast Simphony Development Environment," in C.M. Macal, M.J. North, and D. Sallach (eds.), *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, ANL/DIS-06-1, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15.

North, M.J., T.R. Howe, N.T. Collier, and J.R. Vos, 2005b, "Repast Simphony Runtime System," in C.M. Macal, M.J. North, and D. Sallach (eds.), *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, ANL/DIS-06-1, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15.

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* **16**(1):125, ACM (January): New York, NY.

Repast, ROAD (Repast Organization for Architecture and Design), 2005, Repast Home Page, Chicago, IL; available at http://repast.sourceforge.net.

Wilensky, U., 1998, "NetLogo Wolf Sheep Predation Model," Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL; available at http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation.