

Notes on Implementing StupidModel in MASON

Steve Lytinen and Steve Railsback

16th February 2006

StupidModel will make you smart!

Abstract

This document contains brief notes on implementing StupidModel in MASON, an agent-based modeling platform. StupidModel is a series of template models designed to illustrate common modeling techniques and features. The notes are intended to help potential users of MASON to understand how models are implemented.

StupidModel (documented separately) includes 16 versions, each adding or modifying parts of the model. The implementation of StupidModel in MASON is available separately. These notes are very brief and focus on differences between MASON and Repast; more detailed notes on the Repast implementation of StupidModel are available. This implementation is not necessarily an example of good MASON programming, but was instead designed to illustrate simple ways to implement basic functions.

Each of the following 16 sections corresponds to a version of StupidModel. These notes and the MASON implementation of StupidModel were developed using version 10 of MASON (current as of September, 2005).

1 Basic StupidModel

- (Nothing noted.)

2 Bug Growth

- Now that bugs have two separate actions, move and grow, the model no longer conforms well to the MASON paradigm of agents having a single action called “step”. Because MASON requires all scheduled actions to be called “step”, yet the move and grow actions are to be executed separately, we had to use two Java anonymous inner classes that each have a method called “step” that actually calls the “move” and “grow” methods of the bugs. Both anonymous classes extend `Sequence`, a built-in MASON class to represent lists of items:

```
// build a sequence that makes all the bugs move
Sequence bugMoveSeq = new Sequence(bugs) {
```

```
public void step(SimState state) {
    for (int i=0; i<steps.length; i++)
        ((StupidBug) steps[i]).move(state);
}
```

- Another change now that bugs have two separate actions is that we must pay attention to MASON's schedule "ordering". Unless we specify an "order" of each scheduled action, MASON will execute all actions in random order. Hence, we give the "move" action an order of 1, and "grow" an order of 2.
- Note that the "step" method passes the model (called "state") to the object being stepped. Therefore, we use the "state" as a communication device, letting it contain variables that all objects need to have the value of.

3 Habitat Cells and Resource

- (Nothing noted.)

4 Cell and Bug Probes

- In MASON, interfaces similar to "probes" in Swarm and Repast are called "inspectors".

5 Parameters and Parameter Displays

- Bug and cell parameters are stored as model variables instead of in the bugs or cells. This approach allows the parameters to be displayed and manipulated from a probe to the model; changing the single parameter variable in the model affects all the bugs or cells.
- A method is added to the UI class called "getSimulationInspectedObject" which returns the Model class ("state") This enables the UI class to display all the model's instance variables for which there are getter and setter methods.

6 Histogram Output

- This version of MASON includes only a "MiniHistogram" class that we found not to be very useful. It requires the programmer to manually calculate an array of values for the histogram bin, and its display is small and difficult to interpret. (DID WE JUST SKIP HISTOGRAMMING?) Newer versions of MASON are expected to include more graphing facilities.

7 Stopping the Model

- We added an item to the model's schedule, which called a "checkIfDone" method which we added to the Model class. In this method, if the terminating conditions were satisfied, we stopped the simulation by calling the schedule's "reset" method. However, we did not find this to be a reliable way of stopping the model.

8 File Output

- MASON includes no facilities for either writing file output or collecting statistics on agents. We opened a file using the Java class "FileWriter" in the model's "buildSchedule" method, and scheduled a repeated action which computed the statistics we wanted and wrote them to the file. We put this code in a new method of the Model class called "outputData".

9 Randomized Agent Actions

- We used the "RandomSequence" class to randomize the order in which bugs move. This works only if the action to be executed is called "step", so we renamed "move" to "step". (Once "move" is renamed to "step", we could alternatively simply "step" the bugs; by default, MASON's scheduler executes step methods in random order.

10 Sorted Agent Actions

- MASON includes a collection class called "bag" that has a method for sorting the objects it contains. However, a "bag" cannot be sequenced (given to the scheduler for execution); only arrays can be sequenced. One option is to use a "bag" of bugs to sort them, then copy the bag's internal array to a different array of bugs, which is then sequenced. Instead, we used the Java "Arrays" class "sort" method.
- As with any sorting method, "Arrays.sort" requires a "Comparator" which (in this case) can compare Bugs and sort them by size. We wrote this code with an anonymous inner class, implementing the "Comparator" interface.

11 Optimal Movement

- MASON's grid space object has a "getNeighbors" method that returns a "bag" collection of neighbor cells. Note that (unlike Repast) this collection includes the center cell for which neighbors are being found.

12 Bug Mortality and Reproduction

- Reproduction requires a method to create new bugs during a model run. We did this by providing a second constructor for the Bug class. This constructor is passed the Bug's parent, so that the new Bug can be placed near the parent in the space.
- In the Repast and Swarm implementations, we maintained lists of dead bugs and newborn bugs, which are removed from or added to the master bug list in a separate "cleanup" method scheduled at the end of a time step. Instead, here we executed the bugs' "grow" and "survive" actions using a temporary list of the bugs; these methods then add new bugs to, or remove dead bugs from, the master bug list. Therefore, no cleanup method is needed. This can be done in MASON because, unlike in Repast and Swarm, the "master list" of agents can be modified during a scheduled action.
- Our implementation subjects new bugs to mortality risk on their first day of life.

13 Population Abundance Graph

- This version of MASON did not include graphing capabilities.

14 Random Normal Initial Size

- We did not implement this iteration, because MASON has no built-in methods for creating normal distributions with variable means and standard deviations.

15 Habitat Data from File Input

- MASON lacks tools for reading file input. We used the same class ("DoubleFileGridReader") that we wrote in our Repast implementation.

16 Predators

- The implementation was similar to the one in Repast, with one exception: as with reproduction, we were able to "kill" a bug by simply removing it from the "master list" of bugs, rather than scheduling clean-up actions to remove dead bugs from the list.