

Notes on Implementing StupidModel in Repast

Steve Railsback and Steve Lytinen

16th February 2006

StupidModel will make you smart!

Abstract

This document provides notes on implementing StupidModel in the Repast 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 teachers and self-taught users understand and explain how models like StupidModel are implemented in Repast.

StupidModel (documented separately) includes 16 versions, each adding or modifying parts of the model. The implementation of StupidModel in Repast is available separately. This implementation is not necessarily an example of good Repast 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 StupidModel software were developed using Repast version 3.1.

1 Basic StupidModel

- The program has two classes: StupidModel is a subclass of Repast's SimpleModel class. StupidBug is not a subclass.
- StupidModel has a buildSchedule method that overwrites the "autostep" schedule approach of SimpleModel. The StupidModel schedule includes an action group for the agents, which currently only executes their "move" method; and an action group to update displays. These are both put in a "allActions" action group, which is necessary to avoid Repast's default execution of actions in random order; putting the agent actions and display actions in the "allActions" group ensures that they are executed in the same order each time step.
- The StupidBug class must implement Repast's Drawable interface, so that the bugs can be drawn on the display.
- A Repast Object2DTorus object (referred to within the StupidBug class as "mySpace") is used to keep track of bug locations.
- Bugs choose their own initial location in their constructor method (StupidBug.stupidBug).

2 Bug Growth

- The StupidBug class now includes a new method “grow”.
- “grow” is added to the model schedule in StupidModel’s buildSchedule method.

3 Habitat Cells and Resource

- Habitat cells require a new class, HabitatCell.
- The space object (mySpace in StupidModel and StupidBug) now holds a habitatCell at every location, not bugs. Each habitatCell has a variable (myBug) that refers to a bug that is in the cell.
- Food growth is implemented in the HabitatCell method “grow”.
- Because the space now contains habitatCell objects, the HabitatCell class must implement the Drawable interface. When the DisplaySurface is updated, it tells habitat cells to execute their “draw” method, which in turn simply tells the bug in each cell to draw itself.
- The draw method in StupidBug now illustrates how color can be used to display the state of individuals: each bug’s color gets more red as the bug gets larger.

4 Cell and Bug Probes

- To make both bugs and cells probeable, a separate space object is required for each. Now, StupidModel includes two spaces (“cellSpace” and “bugSpace”). Both are displayed on the same DisplaySurface (in buildModel). The spaces are added to the DisplaySurface using its addDisplayableProbeable method, instead of the addDisplayable method.
- The StupidBug and HabitatCell classes now implement the interface Named, which provides the name of probed objects.
- The user can now click on habitat cells and bugs in the display window to open a probe to them.
- The variables displayed in a probe are those for which there are “get” and “set” methods. For example, a probe for a habitat cell will display the variable foodAvailable if—and only if—the HabitatCell class includes the methods getFoodAvailable and setFoodAvailable.
- By default, Repast probes are not updated as the model continues to execute. Users will want to override this default so open probes will have their values continuously updated as the model runs. One way is via the Repast Actions tab of the Model Settings window that opens when a model is started: click on “Update Probes”. Another way is by sending a message in the code to the Repast

GUI controller: in the StupidModel software the message “AbstractGUIController.UPDATE_PROBES = true;” has been added to the model’s setup method.

5 Parameters and Parameter Displays

- Parameters are defined by the statement “params = new String[]{...” in StupidModel’s setup method.

6 Histogram Output

- A histogram is created in buildModel, using the Histogram class in Repast. A new parameter MAX_HISTOGRAM_VALUE defines the upper limit of histogrammed bug sizes.
- The histogram is updated each time step by an action added to the schedule’s displayActions group.

7 Stopping the Model

- A method checkEndCondition is added to StupidModel and scheduled.
- The setStoppingtime method in Repast’s SimpleModel (from which StupidModel is subclassed) cannot be used because the model does not know in advance what the stopping time is.

8 File Output

- The schedule’s displayActions group now includes two actions for a recorder object.
- The recorder object is created in buildModel. It is an instance of Repast’s DataRecorder class, which can open an output file and write data recorded from a list of objects. DataRecorder includes a built-in ability to report the mean of some variable over its list of objects, but users must add the code needed to report the minimum and maximum.

9 Randomized Agent Actions

- Repast includes a scheduling method (ActionUtilities.createActionForEachRnd) that appears useful for this modification. However, testing showed that this method only randomizes execution order of the bugs once, not each time step. Instead, a method to shuffle the agent list is added to StupidModel and the schedule.

10 Sorted Agent Actions

- A method to sort the agent list is added and scheduled.
- The sorting method is a Java (not Repast) method `Collections.sort`. It requires each object on the list to have a method called `compareTo`, which compares two objects and reports a 1, 0, or -1 depending on which has the highest value of some variable. See the new `compareTo` method in `StupidBug`.

11 Optimal Movement

- Only the method `StupidBug.move` is modified.
- Repast's `Object2DGrid` class has two methods for finding the neighbors to a cell: `getMooreNeighbors` and `getVonNeumanNeighbors`. Unfortunately, the difference between these can only be found by reading the Repast source code.

12 Bug Mortality and Reproduction

- The code for reproducing is simply added to the `StupidBug.grow` method. (Alternatively, it could have been placed in a new “reproduce” method that would be scheduled after “grow”.)
- Bugs also have a new “survive” method, which compares a random number to the mortality risk.
- The most interesting and confusing part of this version is that bugs cannot just be removed from the agent list when they die (in `StupidBug.die`), and new bugs cannot be added to the agent list when they are produced (in `StupidBug.grow`). Doing so would cause run-time errors because it alters the agent list while the Repast scheduler is looping through it executing its `createActionForEach` methods. This problem must be dealt with in any Repast model that includes reproduction and death. This version uses one common technique: creating separate lists of bugs that have been born and have died (the lists `newBugs` and `deadBugs` in `StupidModel`). Then a new model method `addAndDeleteBugs` is scheduled at the end of a time step; it removes the dead bugs from the agent list and adds the new bugs.
- The code to initialize a new bug (set its location, size, etc.) is in the new class `StupidOffspring`.

13 Population Abundance Graph

- The graph, created as usual in `StupidModel.buildModel`, uses the Repast class `OpenSequenceGraph`. This class collects and graphs data from a list of objects.

Its “record” and “updateGraph” methods must both be added to the schedule’s display actions.

14 Random Normal Initial Size

- This version requires creating a random normal distribution in Repast’s Random class, and calling the distribution to obtain a size for each bug as it is created. These steps are in buildModel.

15 Habitat Data from File Input

- Repast does not have a simple tool for reading input files, so this version uses a new class, DoubleFileGridReader. This class uses Java code to read the input file and provide each set of X, Y, and food production values to the model.
- To keep it as simple as possible, the code in buildModel that uses the input data to initialize grid cells assumes that the first line in the input file has its maximum values of X and Y.
- Because the space is now non-toroidal, the bugSpace and cellSpace objects are now instances of Object2DGrid instead of Object2DTorus.
- The StupidBug move method used in Version 1 tells the bug to randomly move a certain number of cells. In the non-toroidal space used here, that approach would sometimes cause bugs to attempt to move beyond the edge of the space, causing a run-time error. Instead, here the StupidBug move method now uses the space’s getMooreNeighbors method to identify a list of potential destination cells that are within the specified distance. This approach automatically keeps the bug from moving off the edge of the space. However, note that getMooreNeighbors returns the neighbor cells in a Java Vector object, not in an ArrayList; Vectors have different methods than ArrayLists.
- This version requires the bugs to randomly select a destination from among any cells having the same food availability. This was done by randomly shuffling the vector of available cells returned by the “cellSpace.getMooreNeighbors” method, before the bug looks through the list for the best cell. Repast’s SimUtilities class has a handy shuffle method that does this. Note that, according to the Repast API, SimUtilities.shuffle operates on Java lists, but it is used here on a Java vector. That works because Java’s vector class conforms to the Java list interface: methods that use a Java list can also use a vector.

16 Predators

- This change requires a new class, StupidPredator. In the StupidModel class, additions include a new list (predatorList) and space object (predatorSpace) to contain the predators, and a new predator action group on the schedule.