EMCSR: pp-pp, year
ISSN xxxx-xxxx
http://conference.emcsr.net

EMCSR
European Meetings on Cybernetics and Systems Research

The Evolution of Agent-based Simulation Platforms: A Review of NetLogo 5.0 and ReLogo

Steven L. Lytinen

School of Computing, College of Computing and Digital Media
DePaul University
243 S.Wabash, Room 645
Chicago, IL 60604, USA
lytinen@cs.depaul.edu,(01) (312) 362 6106

Steven F. Railsback

Lang, Railsback & Associates
250 California Avenue
Arcata, CA 95521, USA
Steve@LangRailsback.com

Abstract: *We review and evaluate two recently evolved agent-based simulation platforms: version 5.0 of NetLogo and the ReLogo component of Repast. Subsequent to the similar review we published in 2006, NetLogo has evolved into a powerful platform for scientific modeling while retaining its basic conceptual design, ease of use, and excellent documentation. ReLogo evolved both from NetLogo and Repast; it implements NetLogo's basic design and its primitives in the Groovy programming language embedded in the Eclipse development environment, and provides access to the Repast library. We implemented the "StupidModel" series of 16 pseudo-models in both platforms; these codes contain many elements of basic agent-based models and can serve as templates for programming real models. ReLogo successfully reimplements much of NetLogo, and its translator was generally successful in converting NetLogo codes into ReLogo. Overall we found ReLogo considerably more challenging to use and a less productive development environment. Using ReLogo requires learning Groovy and Eclipse and becoming familiar with Repast's complex organization; documentation and learning materials are far less abundant and mature than NetLogo's. Though we did not investigate thoroughly, it is not clear what kinds of models could readily be implemented in ReLogo but not NetLogo. On average, NetLogo executed our example models approximately 20 times faster than ReLogo.*

Keywords: agent-based simulation platforms, NetLogo, Repast, ReLogo, template models

Acknowledgement: We thank Jonathan Ozik (Argonne National Laboratory) for assisting us with our model implementations in ReLogo.

1. Introduction

Agent-based models (ABMs) continue to increase in importance and popularity as a way to study complex systems, so the software platforms designed for them are also increasingly important. In 2006 we published a review of five platforms, looking at their conceptual basis, the experience of programming a series of example models, and (informally) execution speed (Railsback *et al*., 2006). Here, we use similar methods to examine two platforms that evolved from those most recommended in our 2006 review. That review identified NetLogo (Wilensky, 1999; http://ccl.northwestern.edu/netlogo) as being especially well designed and documented and easiest to learn and use, while Repast was one of the platforms that we recommended for models that are especially demanding computationally or not well-fitted to NetLogo's conceptual style.

Several new versions of NetLogo have made important changes that address many of the limitations we identified in 2006, though its overall design has been stable. We review the newest

version of NetLogo (v5.0; just released as of February 2012) primarily because NetLogo has become a favorite among scientists new to programming but retains, among some potential users, a reputation as being too restrictive and slow for serious scientific use. We address whether this reputation is deserved.

Repast has changed substantially since our previous review. The Repast "Simphony" version (North *et al*., 2007; http://repast.sourceforge.net), the current version of which was released in 2010, uses a new conceptual approach and is, in important ways, a different platform from previous versions. Part of the "Simphony" version is ReLogo, described by its developers (see http://repast.sourceforge.net/repast_simphony.html) as a "dialect of Logo" (Logo being a computer language designed in the 1960s primarily for education; Papert, 1980). ReLogo is clearly based on NetLogo, as it includes almost all of NetLogo's primitives and many of its graphical interface tools. NetLogo and ReLogo also share a common goal of enabling novice programmers to develop agent-based models. The developers of Repast Simphony strongly encourage users with limited programming background to begin model development with ReLogo (http://repast.sourceforge.net/docs.html), and claim as an avantage of ReLogo is that "Repast Simphony models can be developed in several different forms including the ReLogo dialect of Logo, point-and-click flowcharts translated into Repast Simphony models, Groovy, or Java, all of which can be fluidly interleaved" (North and Macal, 2011, p. 3091).

In addition to ease of use compared to the rest of Repast Simphony, some potential benefits of ReLogo appear to be: (1) allowing Repast users to take advantage of the many powerful NetLogo primitives and programming concepts such as built-in classes for grid cells, mobile agents, and agentsets; (2) providing a link between NetLogo and Repast, so users familiar with NetLogo can transfer their models to Repast and then use other features of Repast not available in NetLogo; and (3) potentially better execution speed, because ReLogo models are written in the language Groovy, which compiles directly into Java byte code while NetLogo uses its own language that must be interpreted.[1] We review ReLogo because it is so different from previous versions of Repast and promoted by its developers as an "entryway" to Repast, which retains a reputation as a platform better suited for large and complex models. We draw conclusions about how well ReLogo provides these three potential benefits.

While there are several other recent reviews of agent-based modeling platforms (e.g., Castle and Creeks 2006; Nikolai and Madey 2009), we know of none that address NetLogo 5.0 or ReLogo or are based on actual programming experience.

2. Methods

To make this study as comparable as possible to our 2006 review, we use very similar methods. Programming experience was evaluated by implementing a series of ABMs in both platforms, and we informally evaluated execution speed for several of those models.

2.1. Programming experience in implementing StupidModel

In Railsback *et al*. (2006), we evaluated ABM platforms by programming 16 versions of StupidModel, a pseudo-model designed to test common software tasks of agent-based modeling and also to provide a template from which real ABMs can be adapted. (Isaac 2011 provides a very useful explanation and reformulation of these template models.) The simplest version of StupidModel consists of "bugs" moving randomly about a 100x100 world of patches; in later versions, patches grow food which bugs eat, bugs optimize their movement according to food availability, bugs reproduce and die, and in the last version predators are introduced. StupidModel

[1] As of 2006, the NetLogo developers were working on a compiler to translate NetLogo code directly into byte code, by-passing the additional interpretation phase; see Sondahl, Tisue, and Wilensky (2006) for details. In NetLogo v.5, some primitives are compiled while others must still be interpreted (Seth Tisue, e-mail message to author, December 8 2011).

has, since 2006, also been implemented by others in at least six other platforms, including Repast Simphony (http://code.google.com/p/repast-demos/wiki/StupidModel), EcoLab (Standish, 2008), Behavior Composer, Python (Isaac 2011), Xholon (see www.swarm.org/index.php/Software_templates), and metaABM (metaabm.org).

In our current study, we updated the StupidModel specifications of Railsback *et al.* 2006 to clarify a number of small ambiguities. Then we implemented all 16 versions in both NetLogo and ReLogo. The specifications and all code are available at http://condor.depaul.edu/slytinen/abm. During re-implementation, we realized that several parts of StupidModel would be more natural to implement in NetLogo and ReLogo (and probably faster to execute) if changed in small ways. The most prominent example would be to have agents interact with grid cells within a circular instead of square neighborhood. However, to maintain comparability with other implementations and reviews based on them, we did not make such changes. Hence, StupidModel requires a few programming statements that appear clumsy compared to others.

The NetLogo implementations were completely independent of those conducted for our 2006 review of NetLogo version 2, and attempted to make as much use as possible of NetLogo's style and built-in primitives. Because ReLogo is based on NetLogo, the ReLogo implementations of StupidModel were based on the NetLogo code: the same code design and primitives were used as much as possible. NetLogo programming was by S. Railsback, who uses and teaches NetLogo; ReLogo programming was by S. Lytinen, who teaches Java and related languages.

We evaluated NetLogo primarily by how it has changed since the version 2 we evaluated in 2006. ReLogo was evaluated by noting parts of the programs that seemed particularly easy or difficult to program, and important ways that it differs from NetLogo.

2.2. Execution speed

Execution speed was evaluated for seven versions of StupidModel. The model was run for 1000 time steps, with the code using NetLogo's (and ReLogo's) timer primitives to report elapsed time between the first and last time steps. Five replicates of each model were run, each with different random number seeds, and our results report the average execution time of the replicates. The time to initialize models by executing their setup procedure was not included. All model executions were performed on an HP Pavilion Notebook PC running Windows 7, with a four-core Intel 2.13 GHz Intel Processor and 4 GB of RAM. Though the computer had 4 core processors, we only used one during timed model runs so as to avoid interactions between model executions and the use of the CPUs by the operating system.

These experiments first were conducted with the graphical displays active. Both platforms were set to run as fast as they could while updating the display once per time step. (NetLogo allows users to slow a model so display updates are not too fast to observe, and to skip display updates to speed execution.)

We additionally compared execution speed with display updates turned off, by using the platforms' built-in tools for automating simulation experiments such as parameter sweeps (multiple model runs with selected parameters stepped over a range). These tools (called "BehaviorSpace" in NetLogo and "Parameter Sweep" in ReLogo) execute model runs with displays off.

The "unnaturalness" of some elements of StupidModel for NetLogo mentioned in Sect. 2.1 is one of many reasons why our execution speed results should be interpreted cautiously; changes with little effect on models results potentially could strongly affect execution speed. We did not try to optimize execution speed in any way.

3. Results

3.1. NetLogo evaluation

Version 5.0 of NetLogo has few fundamental differences from Version 2, and in fact all the StupidModel code we wrote for Version 2 ran in Version 5.0 without modifications other than those

made by NetLogo's automatic translator. In re-implementing StupidModel, we noted a number of changes in NetLogo relevant to criticisms we made in 2006:

- Default scheduling of an agent action is now for agents to execute one at a time instead of with pseudo-concurrency. This change makes it possible to know the exact order in which events occur and avoids the potential for rather spectacular artifacts of "concurrency" (e.g., Sect. 14.2 of Railsback and Grimm 2012). Now NetLogo by default randomly shuffles any list of agents before they execute an action, but there is also a well-documented way to execute agents in order of any of their variables.
- The graphical interface now provides "Inputs", a widget that allows the exact value of a global variable or parameter to be entered.
- There is now a built-in variable for the number of time steps executed, and primitives for re-setting and reporting it.
- The documentation now clearly says when newly-created agents execute their initial behaviors. Code to create a number of new agents first creates all of them, then has each new agent execute its initial behaviors.
- The dimensions of the space can now be set from the code. That allowed us, for versions 15 and 16 of StupidModel, to read an input file of patch data, determine the dimensions of the space from the file, set the space to those dimensions, and then re-read the file to set its patch variables. This capability does not appear to be present in ReLogo.
- The space can be very easily switched from toroidal to non-toroidal, and primitives that depend on the difference automatically behave appropriately for the current setting.
- The graphical interface not only includes a checkbox to turn off graphics updates, but also provides a "speed controller" that lets the user decide how often the display is updated and, therefore, how much computation resource is available to execute the model instead.
- It is possible to keep code in multiple files and on multiple pages in the code editor, though code still need not be separated by class.
- NetLogo's BehaviorSpace tool was significantly improved by allowing users to select how many of their computer's processors to use. Individual model runs are sent to various processors, but results are all gathered in one output file. Hence, simulation experiments with multiple model runs can be made "parallel" with almost no effort.
- NetLogo is now open source, so users can look at and modify its source code (at https://github.com/NetLogo). Its documentation remains extraordinarily complete and accurate, compared to that of other platforms, so there is also much less need to read source code.

Only one potential improvement suggested in our 2006 review—a stepwise debugger—has not been added to NetLogo.

3.2. ReLogo evaluation

3.2.1. Programming experience

A fundamental design difference between NetLogo and ReLogo is that NetLogo provides a programming language designed specifically for agent-based modeling, while ReLogo models are written in Groovy (Koenig *et al.*, 2007; http://groovy.codehaus.org), a lightly-typed, dynamic object-oriented general purpose programming language. We found it unlikely that ReLogo could be used productively without at least a rudimentary knowledge of Groovy. Thus, ReLogo falls into the category of "framework and library" family of modeling platforms (Railsback *et al*., 2006). There is a trade-off between these two approaches. For novice programmers, the NetLogo language appears easier to learn, since its syntax is simpler than the syntax of an object-oriented language like Groovy. On the other hand, because ReLogo models are written in a general-purpose programming language, it could potentially be easier to write models that require complex actions

by agents which are not captured by the primitives of the modeling platform. One additional potential benefit of ReLogo is the ability to also use Repast Simphony's larger library and wider variety of modeling capabilities. However, it is not immediately clear how this would be done, since Repast Simphony models do not use turtles and patches. (Two fundamental concepts of Logo languages are built-in spatial grid cells known as "patches" and mobile agents known as "turtles".) While Repast SImphony's developers claim that "Repast Simphony models can be developed in several different forms including the ReLogo dialect of Logo, point-and-click flowcharts translated into Repast Simphony models, Groovy, or Java, all of which can be fluidly interleaved" (North and Macal, 2011, p. 3091), there are no examples of such multi-language modelsin either the Repast Simphony tutorials or its model libraries.

NetLogo and ReLogo both share the notion of "primitives"; i.e., agent/patch actions or other programming constructs which are built in to the language. For some primitives, the difference in syntax between the platforms is negligible. For example, a commonly used primitive in both languages is "neighbors", which reports the neighboring patches to a turtle or a patch. An example of this primitive's use in NetLogo is the following statement, which creates a new local variable containing an agent's neighbors:

```
let npatches neighbors
```

And in ReLogo:

```
def nPatches = neighbors()
```

A key strength of NetLogo is the ability to combine simple primitives such as "neighbors" into compound statements that are very powerful and still easy to read; ReLogo significantly complicates the syntax of such statements. For example, let's say we want to find the neighbors of one of the turtles named "Ralph". Here is the NetLogo statement to do this:

```
let ralph-neighbors [ neighbors ] of one-of turtles with [ name = "Ralph" ]
```

In ReLogo this is written

```
def ralphNeighbors = { neighbors }.of(oneOf(turtles().with({ name.equals("Ralph") })))
```

Among other uses, curly braces in Groovy are used to define *closures*, a concept familiar to computer scientists, but most likely not to the novice programmers that ReLogo is designed for. A closure is an unnamed procedure which is passed one or more parameters and returns an answer. In this case, the closure { name.equals("Ralph") } is passed each of the agents in list turtles(), and returns true or false depending on whether or not a turtle is named Ralph.

Other primitives, which by their nature require more complexity, are considerably different in NetLogo and ReLogo. For example, here is a snippet of code from one of our StupidModels, in which agents execute their procedure "move" in order sorted by their size so that bigger agents move first. In NetLogo, the code is:

```
foreach sort-by [[bug-size] of ?1 > [bug-size] of ?2] turtles
    [ ask ? [ move ] ]
```

Because sorting involves comparing pairs of items, NetLogo uses ?1 and ?2 to refer to the pair. In ReLogo, we wrote this as follows:

```
def sTurtles = sortBy({a,b -> if (a.getBugSize()<b.getBugSize())
                                return 1;
                                 else return -1;},
                      turtles())
for (Turtle t : sTurtles){
        t.move()
}
```

The sortBy primitive in ReLogo takes 2 parameters: a 2-argument (in this case, a and b) closure that returns an integer, and an AgentSet of turtles. The returned integer should be -1, 0, or 1,

depending on whether a>b, a=b, or a<b. (Our code assumes bug sizes are never equal so does not return a 0 to sortBy.)

3.2.2. Documentation

NetLogo is a much more mature modeling platform than ReLogo, and its documentation is far superior to ReLogo's. An extensive User Manual and many example codes are provided with NetLogo. In the NetLogo programming environment, a right-click on any primitive brings up the entry for that primitive in the NetLogo dictionary. Most primitive dictionary entries describe the syntax of the use of the primitive, a description of what the primitive does, and an example of how the primitive is used. For example, here is the NetLogo dictionary entry (slightly modified) for the "max-one-of" primitive:

> max-one-of agentset [reporter]
>
> Reports the agent in the agentset that has the highest value for the given reporter. If there is a tie this command reports one random agent with the highest value.
>
> Example: max-one-of patches [count turtles-here]

A "reporter" in NetLogo is a procedure which returns an answer (i.e., a function). The NetLogo dictionary entry clearly shows that the reporter must be enclosed in square brackets. On the other hand, the ReLogo dictionary tends to be less informative. Here is the corresponding ReLogo dictionary entry for the same primitive:

> maxOneOf(List a, Closure reporter)
>
> Returns the turtle with the largest value when operated on by a set of commands.

The dictionary entry does not describe the syntax of closures in Groovy, and does not give an example of how to use maxOneOf.

3.2.3. Development environment

The NetLogo development environment is very intuitive for a novice user (Figure 1). NetLogo has a "syntax checker" button (the button with a check mark icon) that the user can click to ensure that code in the model is syntactically correct. This can almost instantly identify most errors in code without running the model.

In contrast, ReLogo is packaged within the Eclipse integrated development environment (http://www.eclipse.org). Eclipse is sophisticated and very powerful, but for a novice there is a steep learning curve for learning Eclipse (as is suggested by the large number of textbooks devoted to the Eclipse IDE; see, for example, McAffer and Lemieux, 2005) as well as Groovy and the ReLogo primitives. While we recommended Eclipse as an environment for developing with "framework and library" platforms (Railsback et al 2006), the complexity of ReLogo in Eclipse stands in stark contrast to NetLogo.

Another usability disadvantage of ReLogo results from Groovy being a loosely typed language. The Groovy compiler in Eclipse (at least as downloaded from the Repast Simphony web site) cannot identify many of the kinds of programming errors that are identified in NetLogo by its syntax checker (or identified by Eclipse in Java code). In ReLogo, many syntax errors are only identified at runtime. For example, if a class does not contain a method that is called in a ReLogo model, the mistake is not identified by the Groovy compiler, as is illustrated in the following code:

```
ask(turtles(), { hunt() } )
```

The compiler cannot determine if turtles class contains a hunt method, since in fact that AgentSet returned by turtles() may all be members of some subclass of Turtle, (e.g., Predator). Thus, any error can only be detected at runtime.

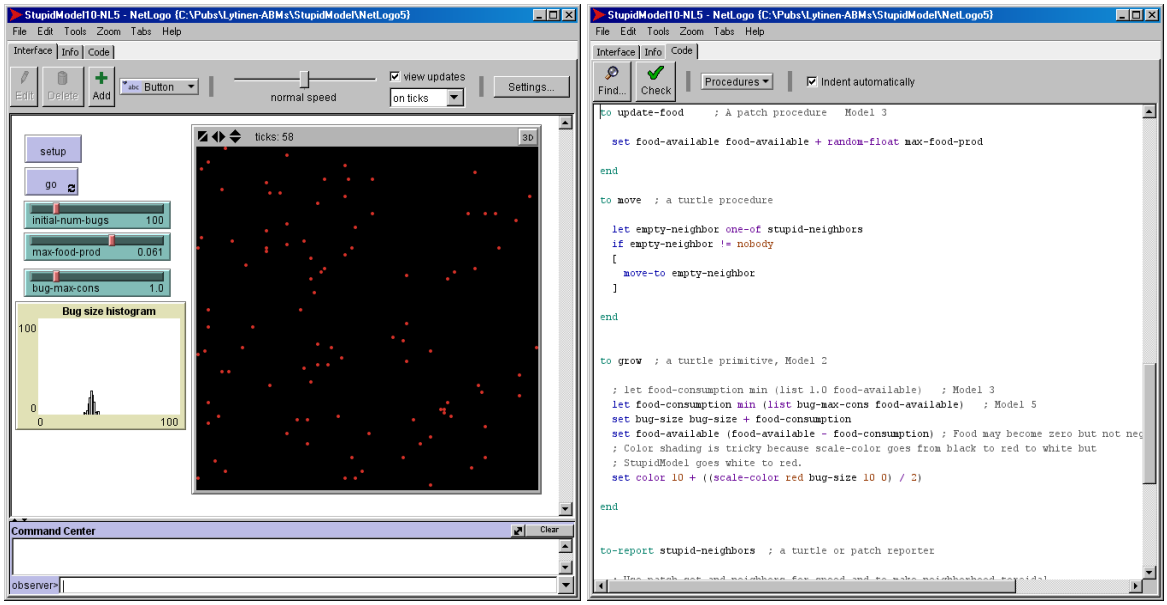

Figure 1. NetLogo's graphical interface and code tabs.

Similarly, incorrect use of a variable name (i.e., it is not associated with the appropriate class) in some circumstances is not caught by Eclipse or the compiler and will only be flagged as an error at runtime. Our experience with ReLogo is that it takes half to a full minute to start running a model to expose the errors. Hence, having errors caught only at runtime negates much of the benefit of Eclipse and significantly slows code development.

3.2.4. Code vs. XML vs. interface dialogs

In NetLogo, there is generally a clear-cut, intuitive division between portions of a model written in code vs. those portions developed via menus and dialogs on the interface. Generally speaking, procedures and variables are written as code, under the “Code” tab in the NetLogo user interface. GUI Widgets such as sliders, monitors, inputs, graphs, and histograms are defined under the “Interface” tab using menus and filling in dialogs. While this makes it easy to develop a user interface for a model, this approach has the disadvantage that there is no single place where one can look to see everything that the model contains. (In fact, version 5.0 provides the less clear-cut option of writing code to update plots inside the plot widgets instead of on the code tab.) Thus, there is no sense of having the “source code” for a NetLogo model.

This division of labor becomes somewhat further blurred when experiments are conducted in NetLogo using the BehaviorSpace tool. In BehaviorSpace simulation experiments, operation of the widgets on the user interface is taken over by BehaviorSpace. A BehaviorSpace experiment is created by filling out a dialog to define parameter sweep values, outputs to be reported, and code to be performed when a model run is initialized and finished.

In ReLogo, a model is broken up into at least two parts: Groovy code and XML files. While this general approach of a mixture of code and XML does follow current system specification

techniques, we feel that the breakdown between these components of the model is not as intuitive as in NetLogo. In ReLogo, Groovy code is written to perform the execution tasks and to define some of the variables. XML files are used to define other parameters in a model, such as the size of the model's space. But other characteristics of the space, such as whether or not it should be toroidal, are again defined in Groovy code. Similarly, the definition of User Interface widgets is also done programmatically. We find this division between Groovy code and XML to be suboptimal.

3.3. Execution speed comparison

ReLogo consistently took 20 times longer or more to run our models than did NetLogo when the model display was turned on (Table 1). Turning off the model display caused both platforms to run somewhat faster, with NetLogo running 3-20% faster and ReLogo running 3-30% faster. As would be expected, the largest gains in speed are for the simplest models (versions 1 and 3 in Table 1) and the smallest gains are for the most complex model (version 15), since proportionally more time is devoted to updating the display in simple models. This suggests a relatively constant time overhead in model display, independent of model complexity.

Although we did not investigate execution speed differences between ReLogo and Repast Simphony models implemented in Java in our current study, our timing results are in stark constrast to the results of our previous review (Railsback *et al.*, 2006), in which we reported that NetLogo was 2-3 times slower than Repast for the simpler versions of StupidModel, 3-4 times slower for versions 15 and 16, and approximately 8 times slower for version 16 with the display turned off.

In a separate effort, students at the University of Michigan's Center for the Study of Complex Systems have re-implemented StupidModel in Repast Simphony using Java (http://code.google.com/p/repast-demos/wiki/StupidModel). In future work we intend to investigate execution speeds of these implementions compared to our ReLogo and NetLogo implementations.

The somewhat unexpected result that NetLogo v.5 is much faster than ReLogo is probably due at least in part to NetLogo implementing its primitives more efficiently than they are in ReLogo. We did not attempt to compare the platforms' implementation of their primitives by reading source code. However, we did conduct some simple experiments replacing complicated primitive statements with simpler low-level code; such changes tended to make ReLogo faster and NetLogo slower, which is consistent with the assumption that NetLogo primitives are implemented efficiently. Because NetLogo is its own language, its compiler may be able to perform optimizations that are impossible in ReLogo. Another contribution to NetLogo's speed is the aforementioned partial compilation of NetLogo code directly into Java byte code. Another potential explanation for the speed difference is that the Windows version of NetLogo is packaged with a server version of the Java virtual machine, which is faster than the default virtual machine that Eclipse typically uses. However, our experiments with NetLogo using both server and default virtual machines indicate that the server versions explain only a 10-20% increase in execution speed.

4. Conclusions and Recommendations

4.1. NetLogo

Through a steady series of revisions, the current version 5.0 of NetLogo has addressed all of the critiques and "wish list" items in our review of version 2 (Railsback et al. 2006), with the exception of adding a stepwise debugger. Many of these changes (Sect. 3.1) substantially improved NetLogo's usefulness as a scientific tool; a key example is ability to set up and execute simulation experiments with ease and speed using BehaviorSpace. Yet NetLogo's core concepts and overall design have remained stable.

Our finding that NetLogo executed models considerably faster than ReLogo certainly contradicts NetLogo's widespread reputation as slow. The speed advantage over ReLogo could largely be a

result of NetLogo being relatively stable over a number of years, giving its developers the chance to continually refine its efficiency.

Table 1. Execution time comparison. Number of agents and runtimes represent the mean of 5 runs executed in each platform.

Description of StupidModel version	Number of agents	NetLogo runtime (secs)	ReLogo runtime (secs)	Speed ratio (NetLogo / ReLogo speed)
Version 1: The world has 100 x 100 patches. At each tick, each bug moves to a randomly chosen vacant patch with x and y co-ordinates ±4 from current location.	100	6.3	173	27.7
V. 3: Patches grow food. After moving, bugs eat food from their patch, up to a maximum rate of consumption. Turtles grow in size based on consumption.	100	9.1	189	20.9
V. 11: Bugs select patch with maximum food availability. Larger bugs move first.	100	9.7	191	19.7
V. 12: Bugs produce up to 5 offspring (and die) when they reach a certain size.	140	13.5	280	20.7
V. 15: The world size is 256 x 112. Patch food growth rates are read from a file.	1330	103	2509	24.3
V. 16: 200 predators now eat bugs. At each tick, predators move to an immediately neighboring patch that contains a bug, provided no other predators are in the patch, then eat the bug.	729	65.1	1516	23.3

4.2. ReLogo

ReLogo is the latest in a series of directions taken by the Repast project. Since our 2006 review, the original Java version and a Python-language version with “drag and drop” model building appear to have been abandoned in favor of the new Java and Groovy-based “Simphony” version that includes ReLogo. The main critiques of Repast we made in 2006—high complexity not mitigated by either strong organization or good documentation, and lack of a conceptual framework—still seem valid.

A NetLogo user would notice a number of differences upon trying to use ReLogo:

- In addition to understanding the NetLogo primitives, a working knowledge of Groovy (and possibly Java) is necessary;
- The Eclipse debugger can be very useful, but working in Eclipse is far more complex, and less efficient in many ways (especially using Groovy rather than Java), than the NetLogo environment;
- The necessity to understand and keep track of low-level programming issues such as Eclipse projects and Java/Groovy packages, classpaths, dependencies, “import” statements, etc. is new and challenging; and

- ReLogo requires following object-oriented programming conventions such as having code for each “class” (turtles, patches, the observer, etc.) in separate files and not having global variables.

4.3. Potential benefits of ReLogo to Repast users

In Sect. 1 we speculated that two motivations for the development of ReLogo were to allow Repast users to take advantage of NetLogo’s primitives and concepts, and to facilitate the transfer of models from NetLogo to Repast so unique features of Repast can then be used. In addition, ReLogo’s developers imply that Repast Simphony models can be developed with a fluid interleaving of ReLogo and Java-based Repast Simphony modules. Are these benefits provided by ReLogo?

We found ReLogo to be successful in the sense that it does implement NetLogo’s fundamental concepts (a world of grid patches and turtle agents) and primitives. This would be beneficial, compared to just using NetLogo, if ReLogo and Repast could then be used to do things that seem impossible, or very clumsy, in NetLogo. Common examples of such things are multiple spaces and non-square spatial units; the trout model of Railsback et al. (2009) represents multiple sections of river as separate spaces, each made up of rectangular (or in later versions, polygonal) cells. While such models could almost certainly be implemented in Repast, it is not clear that ReLogo would be helpful. Many of the NetLogo primitives and concepts are specialized for a single square-grid world and do not make sense in other contexts. Hence, ReLogo may have limited usefulness for models that could not easily be implemented in NetLogo.

Another successful aspect of ReLogo is its tool for importing NetLogo models and translating them into ReLogo code and XML files. In fact, we found (given the state of ReLogo documentation) one of the most effective ways to learn how to program ideas in ReLogo was to first program them in NetLogo and then have the translator convert the code to ReLogo.

4.4. Recommendations

NetLogo is now a powerful tool widely used in science and we recommend it strongly, especially for those new to modeling and programming but also for serious scientists with software experience. NetLogo was chosen by Railsback and Grimm (2012) as the platform for their textbook on scientific agent-based modeling. There are, however, some model designs that NetLogo (and ReLogo) is not well suited for; in our experience, such models are relatively rare.

There are few situations in which we would recommend ReLogo. After developing some familiarity with Groovy, we found ReLogo easier for implementing StupidModel than Repast was. (We implemented some versions of StupidModel in Repast Simphony.) However, the question remains whether there is any real benefit in programming a model in ReLogo or translating it from NetLogo, instead of just using NetLogo. One potential benefit is the ability to then use specialized Repast libraries, but users should check very carefully that (1) the same functions are not available in NetLogo (or in NetLogo’s many extensions) and (2) the Repast libraries will work with models using ReLogo’s structure and primitives.

An open question is whether ReLogo provides an advantage over NetLogo in potentially enabling ReLogo users access to a wider variety of tools and paradigms provided by the Repast Simphony framework. The currently available tutorials on Repast Simphony and ReLogo (http://repast.sourceforge.net/docs.html) are separate documents, and suggest no way to integrate the use of ReLogo and Java-based Repast approaches in a single model. We could find no evidence in the literature of anyone having written a model with a mixture of ReLogo code Java-based Repast code. Therefore, a tutorial on how to do this interleaving would be greatly beneficial.

Our execution speed experiments are not sufficient to conclude how NetLogo 5.0 compares to other platforms such as Repast Simphony. However, they (along with the ease of using

BehaviorSpace on multiple processors) indicate that execution speed is not necessarily an important reason to avoid NetLogo.

References

Castle, C., and Crooks, A. (2006). Principles and concepts of agent-based modelling for developing geospatial simulations. Working paper 110, Centre for Advanced Spatial Analysis, University College London.

Isaac, A. G. (2011). The ABM template models: A reformulation with reference implementations. *Journal of Artificial Societies and Social Simulation* 14(5), http://jasss.soc.surrey.ac.uk/14/2/5.html.

Koenig, D., Glover, A., King, P., Laforge, G., and Skeet, J. (2007). *Groovy in Action*. Greenwich, Connecticut: Manning Publications.

McAffer, J., and Lemieux, J. (2005*). Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications.* Boston, MA: Peason Education.

Nikolai, C., and Madey, G. (2009). Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation* 12(2). http://jasss.soc.surrey.ac.uk/12/2/2.html.

North, M., and Macal, C. (2011). Product design patters for agent-based modelling. In Jain, S., Creasey, R., Himmelspach, J., White, K., and Fu, M. (eds.), *Proceedings of the 2011 Winter Simulation Conference* 3087-3098.

North, M., Howe, T., Collier, N., and Vos, R. (2007). A declarative model assembly infrastructure for verification and validation. In S. Takahashi, D.L. Sallach and J. Rouchier, (Eds.), *Advancing Social Simulation: The First World Congress*. Heidelberg: Springer.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.

Railsback, S. F., Lytinen, S. L., & Jackson S. K. (2006). Agent-based simulation platforms: review and development recommendations. *Simulation 82*(9), 609-623.

Railsback, S. F., and Grimm,V. (2012). *Agent-based and individual-based modeling: A practical introduction*. Princeton: Princeton University Press.

Sondahl, F., Tisue, S., and Wilensky, U. (2006). Breeding faster turtles: Progress toward a NetLogo compiler. In *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects* (pp. 5-16). Chicago, USA.

Standish, R. (2008). Going stupid with EcoLab. *Simulation 84*(12), 611-618.

Tisue, S., and Wilensky, U. (2004). NetLogo: A simple environment for modelling complexity. Presented at *The International Conference on Complex Systems*, Boston Massachusetts, May 2004.

Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, Illinois.

About the Authors

Steven Lytinen

Steve Lytinen (PhD, Yale University) is a professor in the School of Computing, College of Computing and Digital Media at DePaul University. His research expertise is in artificial intelligence and agent-based simulation.

Steven Railsback

Steve Railsback (PhD, University of Bergen, Norway) is an adjunct professor in the environmental modeling program, Department of Mathematics, Humboldt State University; and an environmental engineer with Lang, Railsback & Associates. His expertise is in individual-based ecological modeling and river management.