Agent-based Simulation Platforms: Review and Development Recommendations

Steven F. Railsback
Lang, Railsback & Associates
250 California Ave.
Arcata, CA 95521, USA
LRA@Northcoast.com

Steven L. Lytinen
School of Computer Science, Telecommunications, and Information Systems
DePaul University
243 S. Wabash, Room 645
Chicago, IL 60604, USA

Stephen K. Jackson
Jackson Scientific Computing
1081 Fritz Ave.
McKinleyville, CA 95519, USA

Five software platforms for scientific agent-based models (ABMs) were reviewed by implementing example models in each. NetLogo is the highest-level platform, providing a simple yet powerful programming language, built-in graphical interfaces, and comprehensive documentation. It is designed primarily for ABMs of mobile individuals with local interactions in a grid space, but not necessarily clumsy for others. NetLogo is highly recommended, even for prototyping complex models. MASON, Repast, and Swarm are "framework and library" platforms, providing a conceptual framework for organizing and designing ABMs and corresponding software libraries. MASON is least mature and designed with execution speed a high priority. The Objective-C version of Swarm is the most mature library platform and is stable and well organized. Objective-C seems more natural than Java for ABMs but weak error-handling and the lack of developer tools are drawbacks. Java Swarm allows Swarm's Objective-C libraries to be called from Java; it does not seem to combine the advantages of the two languages well. Repast provides Swarm-like functions in a Java library and is a good choice for many, but parts of its organization and design could be improved. A rough comparison of execution speed found MASON and Repast usually fastest (MASON 1–35% faster than Repast), Swarm (including Objective-C) fastest for simple models but slowest for complex ones, and NetLogo intermediate. Recommendations include completing the documentation (for all platforms except NetLogo), strengthening conceptual frameworks, providing better tools for statistical output and automating simulation experiments, simplifying common tasks, and researching technologies for understanding how simulation results arise.

Keywords: Agent-based modeling, individual-based modeling, software platforms

1. Introduction and Objectives

The use of agent-based simulation models (ABMs)—or individual-based simulation models (IBMs)—for research and management is growing rapidly in a number of fields. For example, a steady, sharp increase in the number of ecology publications using IBMs started in about 1990 [1]. This growth is driven primarily by the ability of these models to address problems that conventional system-level models cannot. However, it has been been accelerated by the evolution of theory [2] and strategies [3] for performing science with ABMs and by the growing number and quality of software platforms for agent-based simulation.

SIMULATION, Vol. 82, Issue 9, September 2006 609-623
©2006 The Society for Modeling and Simulation International
DOI: 10.1177/0037549706073695

Software development remains an obstacle to the use of ABMs for many researchers, however. This problem results in large part from an absence of training in software skills in the education of researchers in many fields that use ABMs (e.g., biology, ecology, economics, political science, sociology), and the computer skill that is taught—programming—is not the only one, or even the most important one, needed for developing ABMs [4, 5].

Most of the commonly used ABM platforms follow the "framework and library" paradigm, providing a framework—a set of standard concepts for designing and describing ABMs—along with a library of software implementing the framework and providing simulation tools. The first of these was Swarm [4] (http://www.swarm.org), the libraries of which were written in Objective-C. Java Swarm (http://www.swarm.org) is a set of simple Java classes that allow use of Swarm's Objective-C library from Java.[1] Repast (http://repast.sourceforge.net) was started as a Java implementation of Swarm but has diverged significantly from Swarm. Most recently, MASON [6] (http://cs.gmu.edu/∼eclab/projects/mason/) is being developed as a new Java platform.

These platforms have succeeded to a large extent because they provide standardized software designs and tools without limiting the type or complexity of models they can implement, but they also have well-known limitations. A recent review of Java Swarm and Repast (along with two less-used platforms) ranked them numerically according to well-defined criteria [7]. Criteria were evaluated from documentation and other information about each platform. The review indicated that important weaknesses include the following: difficulty of use; insufficient tools for building models, especially tools for representing space; insufficient tools for executing and observing simulation experiments; and a lack of tools for documenting and communicating software.

The Logo family of platforms has followed quite a different evolution. Their primary purpose has been to provide a high-level platform that allows students down to the elementary level to build and learn from simple ABMs. However, NetLogo (http://ccl.northwestern.edu/netlogo/) now contains many sophisticated capabilities (behaviors, agent lists, graphical interfaces, etc.) and is quite likely the most widely used platform for ABMs. NetLogo includes its own programming language that is simpler to use than Java or Objective-C, an animation display automatically linked to the program, and optional graphical controls and charts. Perhaps because NetLogo is clearly designed for one type of model (Section 3.1) and uses a simplified language, scientists tend to assume it is too limited for serious ABMs. We originally intended to exclude NetLogo as too limited for full treatment in this paper, but found we could implement all our test models (Section 2.1) in NetLogo, with far less effort than for other platforms. A number of serious scientific models have been implemented in Logo platforms; see, for example, An [8]. Of the platforms we review, only NetLogo does not distribute its source code, but its developers have assured us that they attempt to accommodate the need for scientific users to understand exactly how its methods work.

This paper has two objectives. The first is to review and compare platforms now widely used for ABMs: MASON, NetLogo, Repast, and the Java and Objective-C versions of Swarm. The second is to identify development priorities: what general directions in the future development of ABM platforms seem likely to make them more productive?

This review is intentionally from the perspective of the scientist lacking software development expertise but wishing to use ABMs for research. In particular, we have in mind "pattern-oriented" theoretical research [2, 3]: using ABMs of a real system to test and contrast alternative model versions (e.g., alternative rules for individual behavior) by how well they reproduce a variety of patterns observed in the real system. This type of research typically requires moderate to high complexity in the behavior of individuals and in their environment, but often not massive numbers of agents, extremely long simulations, or behaviors that evolve during the simulation. Therefore, our focus is primarily on the "ease of use" issue: how easy is it to implement ABMs and conduct experiments on them? We also briefly address execution speed: while execution speed is not a major concern for many projects (compared to the time for software development and testing), it is a concern for projects using extremely complex simulations or using many model runs to compare alternative model versions, for example, for parametrization or theory development [3].

To provide a basis for the review, (i) we implemented a series of example models in each platform, and (ii) we gained experience teaching these platforms to students and scientists in mathematics and ecology. At the start of the project we had little experience with any of the platforms except Objective-C Swarm.[2] The review is not intended to be quantitative; many of the issues we address would be difficult to represent via numerical scores. Readers should be aware that by focusing on platforms already in widespread use, we ignore the many less well-known (and, likely, more innovative) platforms that could be good choices for some projects.

The platforms we examine continue to be developed and revised. In this paper we consider only the public releases that were current in September 2005: MASON version 10, Repast 3.1, Swarm 2.2, and NetLogo 2.1. Some of our

1. We refer to "Objective-C Swarm" and "Java Swarm" for points specific to one of these two implementations, and to "Swarm" for points applicable to both.

2. Steven Lytinen teaches and uses Java; Stephen Jackson is an experienced programmer of Objective-C Swarm; Steven Railsback is an experienced designer and user, but not programmer, of Objective-C Swarm. Steven Railsback is also on the board of directors of Swarm Development Group, which maintains Swarm. Nothing in this paper is intended to represent the opinions, policies, or interests of Swarm Development Group.

results are already out of date,[3] so we focus mainly on persistent design issues instead of bugs and quirks of the current versions.

2. Methods

2.1 Platform Comparison: Programming Experience

Our main source of information for reviewing and comparing the five platforms was our own experience implementing a series of example models, which we call StupidModel. The name StupidModel was obviously chosen for fun but it also reinforces the recommendation that modeling projects start with a "ridiculously simplified model" [2]. We designed StupidModel mainly as a teaching tool; its simplest version can be taught to beginners in two to four hours, using any of the platforms. (StupidModel is not really a model because it does not represent a real system or problem.)

StupidModel is also intended as a template for real applications; it includes examples of many capabilities typically needed in the ABMs of real systems. The selection of these capabilities was based on a discussion (see Chapter 1 of Grimm and Railsback [2]) of what constitutes an IBM, a review of ecological IBMs (see Chapter 6 of Grimm and Railsback [2]), and our own experiences of designing and teaching ABMs. Many scientific ABMs represent full "life cycles"—reproduction and death of agents—and agents interact with their environment. Agents often have several behaviors scheduled at different times and in different orders (e.g., actions executed by agents in randomized order to represent concurrent, non-hierarchical behavior; actions executed in order of agent size to represent hierarchy), and many models have several distinct types of agent. Stochastic processes are widely used, but goal-oriented decision-making is often essential for realism. Real models do not run indefinitely but stop when specified conditions are met (e.g., to simulate a specific time period for which there are input data from the real system). Also, from the very start, modelers need observability tools from "probes" displaying the state of individuals to statistical summaries of the population. (Unfortunately, these characteristics are generally lacking in the demonstration models distributed with the software platforms—NetLogo being an exception.) The 16 versions of StupidModel we implemented in each platform are summarized in Table 1; the full specifications of StupidModel, and our implementations in each platform, are on the Swarm Development Group's agent-based modeling site (http://www.swarm.org/wiki/Software_templates).

3. Changes since this study was conducted that we have been notified of but not reviewed include: new releases of MASON with many additions, including a variety of charts and graphs; new releases of NetLogo that include non-toroidal spaces and a "batch mode" to run models without the graphical interface; a new release of Repast scheduled for late 2006; and projects underway to reimplement Swarm in the .NET/Mono and OpenStep/GNUstep environments.

Table 1. StupidModel versions

Version	Characteristics Added
1	100 agents ("bugs") distributed randomly into a 100×100 grid space. Bugs have one action: move to a randomly chosen neighbor cell. Bug locations are displayed graphically.
2	A second bug action: growing by a constant amount.
3	Habitat cells that grow food; bug growth is equal to the food they consume from their cell.
4	"Probes" letting the user see the instance variables of selected cells and bugs.
5	Parameter displays letting the user change the value of key parameters at run time.
6	A histogram of bug sizes.
7	A stopping rule that causes execution to end when any bug reaches a size of 1000.
8	File output of the minimum, mean, and maximum bug sizes each time step.
9	Randomization of the order in which bugs move.
10	Size-ordering of execution order: bugs move in descending size order.
11	Optimal movement: bugs move to the cell within a radius of 4 that provides highest growth.
12	Mortality and reproduction: bugs have a constant mortality probability, and reproduce when they reach a size of 10.
13	A graph of the number of bugs.
14	Initial bug sizes drawn from a random normal distribution.
15	Cell food production rates read from an input file; graphical display of cell food availability.
16	A second "species": predator agents that hunt bugs.

We implemented all versions of StupidModel in each of the five platforms reviewed, rigidly following the same model specification. While implementing the same specifications in each platform, we also attempted to understand and follow each platform's unique idioms and conventions; these sometimes had strong effects on software design. Even within these idioms and conventions, there are of course many ways to program the same model, and programming decisions made in early versions affect later versions. We generally used approaches that seemed simple and intuitive, not (for example) attempting to optimize speed. As we built these implementations, we noted the advantages and disadvantages of each platform.

We also evaluated the platforms' facilities for the types of simulation experiment often used in conducting research with ABMs. Simulation experiments such as sensitivity and uncertainty analyses require multiple model runs, including (i) "scenarios" varying inputs such as parameter values and (ii) "replicates", which vary only the pseudo-random number generator seed. Simulation experiments are facilitated by a "batch mode", in which the model is run without graphical interfaces to increase speed.

2.2 Platform Comparison: Execution Speed

While we did not attempt to rigorously compare the execution speeds of the platforms against a formal benchmark, we were interested in seeing which platforms were particularly fast or slow. We timed runs of seven versions of StupidModel over 1000 time steps. The histogram and population graphs (added in versions 6 and 13) were disabled because we could not implement them in MASON (Section 3.3.6). We also ran version 16 in batch mode to examine the effects of screen graphics on execution speed (in NetLogo, we turned off updating of the animation window to approximate batch mode). Runs were timed by having the software print the system clock time at the beginning and end of the run; we report the mean execution time over five replicate runs of each model version. The runs were executed on a Pentium IV computer running the Windows XP operating system. (In Windows, Objective-C Swarm runs within Cygwin, a Unix-like environment; we found the execution time to be only 2% faster when the Objective-C models were run in Linux instead of Windows.) For Java platforms, we timed the interpreted byte code run by the Java Virtual Machine. NetLogo requires a mouse click to run a model after its separate setup method; we included the time taken by a reasonably agile middle-aged professor to accomplish the click.

This experiment cannot be considered a rigorous or comprehensive comparison of the execution speeds of the ABM platforms. We were careful to implement each version of StupidModel faithfully in each platform, but we did not attempt to rigorously control all the factors—for example, the numbers of method calls, system calls, or objects created—that can affect execution speed. We believe (but did not attempt to demonstrate) that speed differences were controlled more by the design of each platform and its schedules, spaces, etc., than by differences in our code. In our implementations, we used the most basic built-in classes of each platform, as we would expect a novice user to, with no attempt to optimize execution speed. (Often, major speed improvements can be obtained by careful redesign of the code; one example is to use the "sparse matrix" classes in MASON and Repast instead of the standard grid spaces for models having few objects in the space.) The relative speeds of the platforms varied even among the versions of StupidModel we examined, so it is clear that our results cannot be assumed applicable to all ABMs.

2.3 Review of Development Priorities

To provide the basis for recommending platform development priorities, we identified ABM software issues—factors making platforms more or less productive—from our experience implementing StupidModel and from teaching individual-based modeling. Our experience included teaching a short introduction to NetLogo to a conference workshop; a four-hour introduction to Repast and Java to a small group of Humboldt State University (California) graduate students who had at least minimal experience developing or using ABMs implemented in other platforms; and a week-long class on IBMs, Repast, and Java to graduate students in ecology at the University of Helsinki (Finland). In the two Repast classes, we also used Eclipse (http://www.eclipse.org), an integrated development environment (IDE), to reduce the time devoted to basic Java skills.

3. Results

3.1 Platform Objectives, Philosophies, and Terminology

An unexpected result of this project was discovering the extent to which the design objectives and philosophies of the platform developers differed, and the extent to which these differences are reflected in the platform. Here we summarize our interpretation of the objectives and philosophies of the platforms. This interpretation is based mainly on our experience with the software and documentation, but we have also discussed these issues with the developers of each platform.

Swarm was designed as a general language and toolbox for ABMs, intended for widespread use across scientific domains. Swarm's developers started by laying out a general conceptual approach to agent-based simulation software. Key to Swarm is the concept that the software must both implement a model and, separately, provide a virtual laboratory for observing and conducting experiments on the model. Another key concept is designing a model as a hierarchy of "swarms", a swarm being a group of objects and a schedule of actions that the objects execute. One swarm can contain lower-level swarms whose schedules are integrated into those of the higher-level swarms; simple models have a lower-level "model swarm" within an "observer swarm" that attaches observer tools to the model. The software design philosophy appears to have been to include software that implements Swarm's modeling concepts along with general tools likely to be useful for many models, but not to include tools specific to any particular domain. Swarm was designed before Java's emergence as a mature language. One reason for implementing Swarm in Objective-C was that this language's lack of strong typing (in contrast to, for example, C++) supports the complex-systems philosophy of lack of centralized control; for example, a model's schedule can tell a list of objects to execute some action without knowing what types of object are on the list. Swarm uses its own data structures and memory management to represent model objects; one consequence is that Swarm fully implements the concept of "probes": that is, tools that allow users to monitor and control any simulation object, no matter how protected it is, from the graphical interface or within the code (see Section 3.3.4).

Java Swarm was designed to provide, with as little change as possible, access to Swarm's Objective-C library from Java. Java Swarm was motivated by a strong demand among Swarm users for the ability to write models

in Java, not by the objective of providing Swarm's capabilities as cleanly and efficiently as possible in Java. Java Swarm therefore simply allows Java to pass messages to the Objective-C library, with work-arounds to accommodate Java's strong typing.

Repast development appears to have been driven by several objectives. The initial objective was to implement Swarm, or equivalent functionality, in Java. However, Repast did not adopt all of Swarm's design philosophy and does not implement swarms. Repast was also clearly intended to support one domain—social science—in particular, and includes tools specific to that domain. The additional objective of making it easier for inexperienced users to build models has been approached in several ways by the Repast project. These approaches include a built-in simple model, and interfaces through which menus and Python code can be used to begin model construction.

MASON was designed as a smaller and faster alternative to Repast, with a clear focus on computationally demanding models with many agents executed over many iterations. Design appears to have been driven largely by the objectives of maximizing execution speed and assuring complete reproducibility across hardware. The abilities to detach and re-attach graphical interfaces and to stop a simulation and move it among computers are considered a priority for long simulations. MASON's developers appear intent on including only general, not domain-specific, tools. MASON is the least mature of these platforms, with basic capabilities such as graphing and random number distributions still being added.

NetLogo clearly reflects its heritage as an educational tool, as its primary design objective is clearly ease of use. Its programming language includes many high-level structures and primitives that greatly reduce programming effort, and extensive documentation is provided. The language contains many but not all the control and structuring capabilities of a standard programming language. Further, NetLogo was clearly designed with a specific type of model in mind: mobile agents acting concurrently on a grid space with behavior dominated by local interactions over short times. While models of this type are easiest to implement in NetLogo, the platform is by no means limited to them. NetLogo is by far the most professional platform in its appearance and documentation.

These platforms share many concepts but their terminology for similar concepts differs in sometimes confusing ways. We assembled a list of key concepts and the terms used for them (Table 2; there are subtle differences in concepts among platforms, so the terms in each row are not completely equivalent in meaning).

3.2 Platform Comparison: General Simulation Issues

Here we compare how the platforms deal with several standard issues in discrete event simulation: model structure, scheduling, and random number generation.

Model structure. Each platform has a standard model structure which is enforced to varying extents by its code and also established via conventions used in documentation and example models. In Swarm, a model usually consists of at least: (i) an "observer swarm" that provides a graphical depiction (animation) of the model, graphs, control panels, and parameter displays; (ii) a "model swarm" that builds and contains the model's agents and other objects, and contains their schedule; (iii) one or more collections of agents; and (iv) one or more space objects representing the agents' environment. MASON uses a structure similar to Swarm's, although the class names are different. One usually defines a user interface (UI) class, which is responsible for displaying the model as it runs, a "model" class which constitutes the simulation model, contains the schedule, and organizes and controls the other model objects, and agent and space classes. Repast's overall structure seems less well defined: some examples use separate model classes, one with and one without graphics, while other examples include display objects within the single model class.

NetLogo's high-level environment almost completely separates the processes of implementing and displaying a model. The user writes the software for behavior of agents and the space's grid cells on a "Procedures" page. Agents are essentially subclasses of built-in "turtles" (mobile agents) and "patches" (spatial cells). On a separate "Interface" page is an automatic animation of agent locations on the space. The schedule is written in a special procedure called "go". Graphs and parameter controllers can be added to the interface via graphical and menu-driven tools, along with simple statements in the software telling the interface when to update. Code controlling the appearance of agents and cells on the animation can be added to the procedures.

Scheduling. A primary purpose of ABM platforms is to control which specific actions are executed when, in simulated time. Models often use discrete, fixed time steps but sometimes use "dynamic scheduling", in which new actions can be generated as the model executes, and scheduled for execution at a specific future time (see, for example, Section 5.10 of Grimm and Railsback [2]). MASON, Repast, and Swarm provide explicit methods for scheduling actions, both in fixed time steps and dynamically. Because it is designed primarily for one type of model, NetLogo provides fewer tools for explicit control of scheduling. By default, the order in which actions are executed is partially controlled by the order in which they appear in the "go" procedure but NetLogo's pseudo-concurrent execution keeps the user from specifying or knowing the exact sequence in which events are executed. However, the pseudo-concurrent execution can easily be deactivated to make NetLogo models use discrete time steps. NetLogo does not appear to provide the ability to schedule an action a specific future time.

Random number generation. All the platforms use the "Mersenne twister" random number generator [9], and optionally allow users to provide their own seed so the

Table 2. Terminology differences among platforms

Concept	Term			
	MASON	**NetLogo**	**Repast**	**Swarm**
Object that builds and controls simulation objects	Model	Observer	Model	Model swarm
Object that builds and controls screen graphics	ModelWithUI	Interface	(None)	Observer swarm
Object that represents space and agent locations	Field	World	Space	Space
Graphical display of spatial information	Portrayal	View	Display	Display
User-opened display of an agent's state	Inspector	Monitor	Probe	Probe display
An agent behavior or event to be executed	Steppable	Procedure	Action	Action
Queue of events executed repeatedly	Schedule	Forever procedure	Schedule	Schedule

sequence of pseudo-random numbers can be repeated. Only Swarm includes a variety of alternative generators.

3.3 Platform Comparison: Programming Experience

In this section we describe the interesting characteristics and differences among the platforms which we identified while implementing the 16 versions (Table 1) of StupidModel. We also discuss the platforms' facilities for non-graphics and automated experiment runs.

3.3.1 Version 1: Mobile Agents

The StupidModel agents exist in a discrete two-dimensional grid. All of the platforms have built-in classes to represent such a grid space. The space is also "toroidal", meaning that cells on one edge are assumed contiguous to the opposite edge. Repast has a class specifically for a toroidal space. MASON has a two-dimensional grid class which can be used for toroidal or non-toroidal spaces, depending on which of its methods are used. Swarm does not have a toroidal space class, so code must be written to detect if a bug is near an edge to allow for the possibility that its motion will carry it across the edge to the other side of the space. In NetLogo, all spaces are toroidal.

MASON, Repast, and Swarm use the same logical, yet somewhat complex, approach to displaying agents in space. First, the user must instantiate a screen display object, which we refer to as an "animation window". Secondly, the user must instantiate a separate "displayer" object that links one or more space objects (as discussed in the previous paragraph) to the animation window. Finally, each object to be drawn must have a "draw" method that specifies its coordinates, shape, and color (in MASON, this draw method is in the "UI" class instead of the agent class). NetLogo, in contrast, automatically draws agents on a built-in animation window; users can customize shapes and colors.

Only NetLogo has a built-in class to represent agents; this class assumes the agents exist in a discrete grid space and automatically provides variables for agent coordinates and keeps track of which cell they are in. In the other platforms, we defined a StupidBug class with instance variables for the x and y coordinates. In NetLogo, agent motion was implemented using a built-in method that moves an agent to a location specified by X and Y coordinates. In the other platforms, agent motion had to be implemented using lower-level operations, which first removed the agent from its location in the grid space, then put the agent in its new location, and finally updated the agent's coordinates.

We used each platform's scheduling mechanism from the very beginning, although version 1 of StupidModel could have been written more simply without full use of scheduling. Actions scheduled for each time step are (i) each bug moving and (ii) the display updating. (The display update is scheduled explicitly so users know, whenever they look at and probe—see Section 3.3.4—the display, exactly where the model was in its schedule when the display was drawn; this can be important for debugging as models become more complex.) Each platform has a built-in method for scheduling repeated execution of a specific action for a collection of agents, but how this is done varies among platforms. In Objective-C Swarm, the scheduling statement simply states which object (or collection of objects) is to be executed and the "selector" for the method those objects execute. In Objective-C, a selector is a standard variable type that represents the name of a method. Java does not include the selector variable type, so Java Swarm uses a selector class. For example, to schedule a list of bugs to execute a method called "move", Java Swarm requires creating a move selector object for the bug class, then placing an action on the schedule which utilizes this selector. The creation of a selector requires a separate programming statement and is slightly awkward. Repast has scheduling statements similar to those of Objective-C Swarm; one simply specifies the name of the method and the object to be scheduled for execution. MASON's scheduling is significantly different, in ways that became important later when more than one action for the same object needed to be scheduled at different times (Section 3.3.2).

It was difficult to confirm that our scheduling specification (bugs move, then the display updates) was met in MASON because the display update is automatic. We were unable to determine from the documentation exactly when the display update is scheduled.

Scheduling the display update was not trivial in Repast because Repast by default executes actions in random order each time step. We had to override this default to ensure that the agent "move" action was always executed before the display was updated. To do so, we created a sequential subschedule (called an "ActionGroup" in Repast) in which first the bugs moved, and then the display was updated. This subschedule then was placed as the only item on the overall schedule.

In NetLogo, actions are scheduled simply by including them in a special procedure called "go". In this case, the "go" procedure includes a statement telling the bug agents to execute their "move" procedure, then a statement telling the display to update.

3.3.2 Version 2: Agent Growth

In version 2, a second action—grow—is added for the bug agents. Adding a second agent action to the schedule was trivial in all platforms except MASON. MASON's complication was a result of its scheduler's use of a standard software design called the *template method design pattern* [10]. All actions scheduled in MASON must act on objects that implement an interface called *Steppable*. (In Java, abstract classes are often written as interfaces. This is the case with *Steppable*; however, in other object-oriented languages the same design would be accomplished using an abstract class.) This interface requires a *hook method* called "step"; any object to be scheduled must have a method called "step", and only the step method can be executed by the MASON scheduler. This design was used to avoid the execution time required to look up which code to execute when an action is specified by a selector, as in Swarm and Repast; the MASON scheduler always knows to execute a method named "step".

Use of this template method design pattern makes it easy to schedule the agents to each perform all of its actions at once, simply by placing a sequence of method calls into the agents' step method. For example, if in StupidModel we had wanted each bug to move and then grow before the next bug acted, then we could have written "step" to call the methods "move" and then "grow". However, for models requiring agents to have distinct actions scheduled separately, this template design pattern is not convenient. Because we wanted all of our StupidBugs to move before any of them grew, in essence we needed two "hook" methods: one for moving and one for growing. Producing the desired sequence of actions is therefore done by building wrappers (internal classes implementing *Steppable*) around the "move" and "grow" methods for our agents. This complex programming device is certainly difficult for Java novices.

In version 2 we also programmed the bug's color to reflect its size, grading from white to red as the bug size ranges from 0 to $\geq$10. In Repast and MASON, we found it most straightforward to use the built-in Java Color class. Swarm requires (and MASON and Repast allow) users to define a "colormap": an indexed range of colors. In the observer swarm, we built a map of 64 colors scaling from white to blue; in their draw method, the bugs calculate an index between 0 and 63 from their size and use this index to obtain the appropriate color. MASON includes a built-in tool for creating these color gradients in its colormap.

NetLogo provides a color-scaling primitive: we only needed to specify that a bug's color is red, shaded by bug size over a range of 0 to 10.

3.3.3 Version 3: Habitat Cells

In MASON, Repast, and Swarm, the easiest way to implement habitat cells—for this version—was to create a habitat cell class and put cell objects, not bugs, into the grid space. The cells have an instance variable for the bug they contain, if there is a bug in the cell. The bugs obtain information about their habitat (food availability) from the cell having the same coordinates as themselves. The display updates by sending a "draw" message to the cells, which simply tells their bugs to draw themselves.

NetLogo makes version 3 easier in two ways. First, the built-in grid space automatically contains habitat cells ("patches") and is designed to track agents and spatial resources separately, so both bugs and habitat cells exist in the same space. Secondly, agents automatically have access to the instance variables of the cell they occupy; thus, in StupidModel the amount of food available in a cell is available to a bug object as though the data were one of the bug's instance variables.

3.3.4 Version 4: Cell and Bug Probes

"Probe" displays of an object's variables that can be opened from a model's animation window are built in to all five platforms. In MASON, Repast, and Swarm, the displayer automatically generates probes to the objects in its underlying space objects; the user can add code for customizations such as which variables are included in the probe display, which mouse button opens which type of object (bug versus cell), and whether the probes are updated each time step if left open as the model runs.

In MASON and Repast, variables can be probed only if the user provides "getter" and "setter" methods. In Java Swarm, variables can be probed only if they are declared public. However, in Objective-C Swarm, probes have access to all instance variables and methods of any object, even if they are declared to be private or constant. Thus, no additional code must be written for a probe to display all of an object's data. (In Objective-C Swarm, in fact, probes can be used within the code instead of getter and setter methods.)

Although adding probes to an animation window is relatively easy in all platforms, probing both cells and bugs required a major change in MASON, Repast, and Swarm software. For both to be probeable, bugs and cells need separate displayer objects that put them on the animation window, and separate underlying space objects. Therefore, in version 4 bugs exist in one grid space object ("bug space") and the habitat cells exist in a separate "habitat space".

Having two spaces in the model increased the complexity of interaction among objects. Now bugs have to know about both their own space and the habitat space, because the amount of food in its habitat cell determines how much the bug can grow. In addition, the bug's grow action has to update the habitat space: the amount of food in the cell decreases by how much the bug has eaten.

Facilitating communication among objects in a "pure" object-oriented language such as Java is a common difficulty for novice programmers. Because Java lacks global variables, the most common way to facilitate object-to-object communication is to pass objects via the constructor, and then to store the objects in an instance variable. For example:

```
// Repast version 3
public class StupidBug {

   private Object2DTorus bugSpace;
   private Object2DGrid habitatSpace; ...
   public StupidBug(Object2DTorus aSpace,
   Object2DGrid bSpace) {

      bugSpace = aSpace;
      habitatSpace = bSpace;
      ...}

}
```

Instances of the "StupidBug" class can then communicate with the bug and habitat spaces via these instance variables. Our teaching experience has been that novice Java programmers become confused by the need to pass objects to a constructor in order to facilitate communication between objects.

MASON provides a more understandable approach for communication among objects. Its "hook" methods ("step") always take one parameter, namely the SimState object that represents the overall model. Then, inside its step method, an object can communicate with other objects in the model via the model's "getter" methods: the model object serves as a communication device by holding information needed by other objects. For example:

```
// MASON version 3
public class StupidBug {
   // wrapper for the grow method has
   been omitted for simplicity

   public void grow(SimState state) {

      StupidModel theModel = (StupidModel)
      state;
      ObjectGrid2D habitatSpace =
      theModel.getHabitatSpace();
      ...
      // Now the bug determines food
      availability from habitatSpace
      }
}
```

In this case, the bug's grow method needs information about food availability from the habitat space, so from the model it obtains a local variable for the habitat space.

In NetLogo, probes to both agents and cells are built into the animation window and require no code.

3.3.5 Version 5: Parameters and Parameter Displays

All of the platforms have some built-in probe-like facility for displaying and altering model parameters—which may specify initial conditions or control behavior of model objects—at the start of a model run. Parameters are usually treated as instance variables of the model class. In Repast, parameters to be displayed at startup are selected simply by listing them in a statement in the model's "setup" method and by providing getter and setter methods for each. MASON automatically provides access to any model variables with getter and setter methods. Swarm requires the user to create and activate a probe display object for the model swarm.

NetLogo allows users to control parameters before and during a model run via graphical "slider" controls on its interface page. To add a slider for a model parameter, the programmer simply drags a new slider onto the interface and uses its menu to hook the slider up to the desired model variable. While sliders are easy to add and convenient for exploring wide ranges of parameter values, they are less convenient for specifying an exact parameter value.

3.3.6 Version 6: Histogram Output

Histograms are particularly useful for ABMs because they can output the full distribution of some characteristic over all the agents. In version 6, we added a histogram of bug size. Repast and Swarm have built-in histogram classes that are relatively easy to use. MASON does not yet provide such a class.

In Java Swarm, the histogram class (and several other graphics classes) is complicated to use because the "createBegin – createEnd" object creation paradigm of Objective-C Swarm is incompatible with Java constructor methods. To work around this incompatibility, Java Swarm requires instantiating two separate objects in a non-intuitive process that is not documented well.

In NetLogo, histograms are created using drag-and-drop and a menu, on the interface page. Then, a simple code statement specifies when the histogram is updated.

3.3.7 Version 7: Stopping the Model

Stopping when a specified condition is met is an example facility for scientific ABMs that is often neglected in simple demonstration models. All platforms have a simple statement by which the model's code can stop its own execution.

3.3.8 Version 8: File Output

Scientific models invariably produce output for later analysis; for ABMs, this output typically includes summary statistics on the agents. A variety of statistics is needed to capture the variability among agents—simple averages are not sufficient. Two platform capabilities support this need.

File output tools. Objective-C Swarm and Repast provide built-in classes to facilitate output of data to a file, and data recording actions can be scheduled just like any other action, so that they take place at known times. Java Swarm and MASON do not provide file writing tools, so a Java class for file output must be used. NetLogo provides simple primitives for opening and writing to files, although their ability to format and control output is limited.

Statistical calculations. Swarm has a powerful tool for collecting summary statistics on a collection of model objects in its "Averager" class. NetLogo also includes primitives that provide all common statistics. Repast's "DataRecorder" provides only an average. MASON lacks tools for summary statistics. NetLogo was the only platform lacking a built-in variable for the number of time steps executed, so we had to add one to label output.

3.3.9 Version 9: Randomization of Bug Movement

This version illustrates how to randomize the order in which a collection of agents executes some scheduled method. Randomized scheduling is often used to avoid artifacts of execution order; for example, in later versions of StupidModel, bugs that execute their "grow" method earlier have access to more food than bugs executed later. All of the platforms provide the ability to randomize the order of actions. MASON and Swarm make this the easiest, via scheduling methods that pseudo-randomly shuffle the order of the agents each time step. Repast does not provide a scheduling method which randomizes the execution order of a collection of agents; users can instead use a Repast class to shuffle the agent list. This shuffle must also be included in the schedule so it is repeated each time step.

NetLogo documentation does not describe exactly how the scheduler orders execution of the agent list, and there is no simple method to randomize execution order. However, the NetLogo documentation does provide an easily followed example of how to combine several primitives in a complex way to effectively randomize execution of an agent action.

3.3.10 Version 10: Size-ordering of Bug Movement

In contrast to randomized scheduling of an action, some models use ordered execution to represent a hierarchy: bigger agents execute their actions before smaller ones so they have access to more resources. Therefore, version 10 illustrates how to sort a collection of agents by some attribute (size) prior to execution of an action.

All of the platforms provide a facility for sorting a list of agents, most often relying on the underlying sorting libraries of their implementation languages. Repast and MASON assume that the modeler will use a sort method from either the Java Collections or Arrays class. Swarm provides a wrapper for a C library tool called QSort. For NetLogo, we had to modify the example approach for randomizing execution order (Section 3.3.9) so it instead sorted by decreasing bug size.

3.3.11 Version 11: Optimal Movement

In this version, the model specification calls for the bugs to move to the unoccupied habitat cell, within a limited distance, having the most food. This is an example of goal-oriented decision-making by agents.

All platforms except Swarm provide methods to find all cells within a specified distance in a grid space. (In fact, MASON and Repast provide a variety of methods for different definitions of "neighbor"—although the definitions are not documented in Repast.) The user must then write code to loop through the list of neighbor cells to find the one with the most food. (Starting in version 15, the list of neighbor cells was shuffled randomly to avoid artifacts of multiple neighbor cells having exactly the same food availability.)

3.3.12 Version 12: Mortality and Reproduction

We added mortality and reproduction to the "grow" method of the bugs: when a bug reaches a size of 10, it produces offspring and dies. Introducing births and deaths into StupidModel complicated it considerably. First, it is important to exactly specify what should happen when an agent is born: should an agent begin executing its actions during the time step it was born in, or in the next tick? If an agent dies, is it immediately removed from the space so its location is available for other agents? Also, how do we prevent the schedule from executing actions for newly dead agents? The platforms vary considerably in whether they handle these sorts of details or leave them to the user.

In Repast and Swarm, the list of agents cannot be modified in the middle of a scheduled action. A bug created during its parent's grow method cannot be immediately added to the bug list, and the dead parent cannot remove itself immediately from the bug list, because the scheduler is actively stepping through the bug list. We avoided this limitation in a typical way, adding new bugs to a separate "new agent" list and adding dead bugs to a "dead agent" list. Then a separate method must be scheduled later to add the new bugs to, and remove the dead ones from, the master bug list.

In MASON, however, we dealt with these problems by copying the list of agents and scheduling the "grow" action on this temporary agent list. When an agent died or was born, it was immediately removed from or added to the master agent list. Interestingly, this approach was

facilitated by the complicated coding needed to overcome MASON's requirement that only actions called "step" can be scheduled (Section 3.3.2).

In contrast to the clumsy and complex methods to handle agent reproduction and death in the other platforms, NetLogo provides simple methods for removing and adding agents, automatically updating the agent list and dealing with the scheduling issues. However, it is not clear from the NetLogo documentation exactly *how* these tasks are done; for example, does a new agent execute its actions on the same time step that it was created?

3.3.13 Version 13: Population Graph

For all platforms, adding a line graph for bug population was nearly identical to adding a histogram (Section 3.3.6).

3.3.14 Version 14: Random Normal Initial Size

This version illustrates the common technique of drawing random values from a specified statistical distribution. All the platforms except MASON provided simple tools for defining and using normal distributions. Swarm and Repast provide extensive random libraries: Bernoulli, binomial, exponential, log–normal, Poisson, etc., distributions. NetLogo provides most of these distributions. We did not implement the random normal bug size in MASON because MASON provides only an undocumented Gaussian distribution, which presumably returns values from a standard normal distribution.

3.3.15 Version 15: Habitat Data Read From a File

The introduction of "real" habitat data brought several challenges, discussed separately.

Reading input data. In Repast, MASON, and Swarm, we needed to write code to read file input from scratch. NetLogo has a simple file reader that was easily used.

Scaling space objects to file data. In many scientific models, the space dimensions depend on what input data are used, so are best determined from the input file. In Swarm and Repast, we could readily (using our own code) read the input file, determine the space dimensions, and then use those dimensions to create the space and spatial display objects. In MASON, it was not clear how this can be done, because of MASON's separation of its "UI" display elements from the rest of the model and the way in which the UI interacts with the model. In MASON's example models, the UI always makes its display before building the rest of the model, so display dimensions cannot be determined by the model. As a result, the most straightforward way to write version 15 of StupidModel in MASON was to read the input file twice: first while creating the display (to obtain the space dimensions) and again while building the model itself.

NetLogo does not allow the space size to be determined from a file, or by the code at all. Instead, the dimensions of the space must be set manually via a menu on the user interface.

Non-toroidal spaces. We made the space non-toroidal to be realistic. It was easy to change to a non-toroidal space in Swarm, Repast, and MASON, either by switching to a built-in non-toroidal space class (Repast) or by no longer performing the toroidal calculations (Swarm and MASON).

NetLogo spaces are inherently toroidal and there is no easy way to make them non-toroidal. Instead, we had to modify the "move" action of bugs to prevent them from crossing over a space boundary.

3.3.16 Predators

This version adds a second type of agent to the model: predators that hunt bugs. This required, for all platforms, adding a new class for predators and scheduling their "hunt" action. NetLogo lets users define "breeds", different types of agent. Each breed has its own built-in agent list and can have different actions scheduled separately from other breeds. Hence, for this version we created predators as a separate breed from bugs. All breeds are automatically displayed on the animation window.

Predators are allowed to occupy the same cell as a bug, so in MASON, Repast, and Swarm we could not simply put the predators in the same space object used by bugs—the simple grid space classes we used allow only one object per cell. We could have modified the bug space by putting a list object in each cell and storing bugs and predators on the list; MASON and Repast also provide alternative grid spaces that can contain more than one object per cell. These approaches have the disadvantage of requiring (e.g., to see if there is a bug in a cell) code to go through the list of objects in the cell and test whether any of them are bugs. Instead, we used a straightforward solution: adding new space and display objects to contain and display the predators.

3.3.17 Support for Simulation Experiments

NetLogo has a built-in "BehaviorSpace" tool for automating experiments. Using a menu, the user can define scenarios by specifying values of one or more global variables, set the number of replicates for each scenario, and specify stopping rules to end each model run. Repast has a menu-driven tool ("Multi-Run") similar to NetLogo's BehaviorSpace in design, but its performance and documentation were not yet reliable. Repast also has an experiment manager ("batch mode") controlled via an input file ("parameter file"). Swarm and MASON have no tools for automating experiments. Swarm's design can help users write their own: programmers can design an "experiment swarm" to initialize and execute a series of model swarm runs.

We did not test Repast's "batch mode" but inferred from documentation that it bypasses Repast's graphical

control panel. The Repast convention for bypassing user-programmed graphical displays (which we did implement) appears to be writing a separate version of the model code in which displays are not created. The situation is similar in Swarm: batch mode (which in Swarm simply means non-graphics) is invoked by a run parameter that tells the code whether control panels and displays should be created, but some programming is required. The Swarm convention is to write a "batch swarm", which executes the model swarm without creating any graphics; observer and batch swarms are both attached to a model and the run-time parameter determines which to execute. MASON's graphical control panel lets users switch displays off and on at will during a model run; in addition, a separate batch version of the model is easily made by adding a simple "main" method to the model class and running it instead of the UI class. To run a NetLogo model without graphic updates, users can (i) manually switch off the display of agent locations in space at any time and (ii) disable graphs by removing their update statements from the code.

3.4 Platform Comparison: Execution Speed

The time required to execute 1000 time steps of various versions of StupidModel varied widely (Table 3), but some results were unexpected. MASON was the fastest platform for all but the simplest version of StupidModel. Repast took 1–54% longer than MASON to execute the various versions, with the smallest difference for version 16, the most computationally intensive. NetLogo run times were three to five times those of MASON, but NetLogo was as fast as MASON (and faster than Repast) for version 1.

Swarm was by far the fastest platform for version 1 of StupidModel. For the remaining versions, however, Swarm was surprisingly slow: Objective-C Swarm execution was seven to 14 times longer than MASON, and Java Swarm 19–31 times longer. Profiling (a compiler option that determines the time spent in each method) version 3 of Objective-C StupidModel indicated that no particular methods or programming techniques were inordinately slow. The fact that Swarm went from fastest to slowest when we increased the numbers of message calls and objects may indicate that it can be fast for models with fewer interactions and more computation, and slow for models with more method calls and less computation.

For version 16, batch mode significantly improved the speed for the Java platforms. For MASON and Repast, turning off graphic output reduced execution time by 75%; for Java Swarm, the reduction was 35%. Batch mode did not speed up Objective-C Swarm, presumably because its run time is dominated by non-graphical operations. In batch mode, MASON and Repast were nearly identical in execution time.

An analysis of model size versus execution time shows some interesting differences among platforms. Version 3 of StupidModel adds 10,000 habitat cells to the 100 bugs; for versions 12 and higher, the number of bugs emerges from mortality and reproduction (the time-averaged number of bugs is reported in Table 3); and versions 15 and 16 have even more habitat objects. For MASON and Repast, execution time appears to depend mainly on the number of bugs in the model, with minimal influence of the number of habitat cells. However, the execution time of NetLogo and Swarm appears influenced by the number of habitat cells: execution time jumps between versions 1 and 3. Possible reasons for this difference are the overhead in NetLogo and Swarm for object creation and its corresponding memory management, or higher overhead in method calls (the versions with more objects also use more method calls).

3.5 Key Issues in Agent-based Modeling Software

Our experience in using and teaching agent-based modeling platforms revealed the following issues that affect their productivity. This section is not intended as a further comparison of platforms but instead to inform users and developers about factors that can make platforms more useful.

The framework and library paradigm is good—but the framework is important. The software paradigm pioneered by Swarm—a framework of concepts, such as swarms, collections, actions, schedules and observers, and the software implementing them—was designed to give scientists a standardized way to think about and describe their models and then to translate the model description into working software [4]. This paradigm seems intuitive to people already thinking about systems in an individual-based way: what types of agents are in the system, what behaviors do they execute and when, and how do we observe what emerges? Another important advantage of the paradigm is that it does not appear to limit the type of model that can be implemented; our experience has been that scientists avoid platforms that appear customized for a particular type of model because they assume it will be difficult to implement their own—inevitably unique—model. Using a standard programming language, such as Java, undoubtedly helps convince users that the framework and library platforms will allow them to do whatever they want. In contrast, platforms that are menu-driven (e.g., RepastPy; AgentSheets) or use domain-specific primitives (e.g., Mobidyc [11]) have not caught on because they clearly restrict the modeler.

Compared to Swarm, Repast and MASON seem more like libraries and less like frameworks, which makes the transition from ideas to working simulator more difficult. MASON and Repast allow models to be organized in a Swarm-like framework, but their documentation and teaching materials do not emphasize the conceptual aspects of model design. MASON's "model" and "modelWithUI" concepts (Table 2) at least provide a standard framework for separating the model and graphical observers (as Swarm's "model swarm" and "observer swarm" concepts do), but Repast, in its software design and documentation, seems to require or encourage little standard-

Table 3. Number of bugs, total number of objects (bugs, habitat cells, and predators), and execution time (mean of five observations) for seven versions of StupidModel and version 16 in batch mode

Version	Number of Bugs	Number Objects	Execution time (s)				
			MASON	NetLogo	Repast	Java Swarm	Objective-C Swarm
1	100	100	7.9	8.1	12	6.1	2.2
3	100	10,100	10	30	12	379	68
8	100	10,100	9	31	12	176	70
11	100	10,100	11	49	13	245	85
12	150	10,150	12	63	16	292	95
15	1760	29,800	113	417	124	2984	1528
16	1660	29,900	122	314	123	2338	1029
16 batch	1660	29,900	30	229	31	1543	1000

ization. It would be easier to teach these platforms (and even to motivate students to bother with them) if their libraries were more clearly linked to a well-defined, standard conceptual framework.

Platform complexity is a major concern. All the platforms are large, with dozens of classes and hundreds of methods. This complexity is intimidating and makes it difficult to find what one needs or to determine whether a tool for some particular task is available. However, these classes provide useful tools, so smaller is not necessarily better. Unfortunately, the platforms too often suffer from problems that aggravate the negative effects of their complexity, as follows.

- Lack of a clear philosophy and decision process for what will or will not be included.
- Software not in well-organized packages or libraries.
- Lack of complete documentation. Users should not have to read source code to get a basic idea of how a platform's methods work.
- Failure to use common design patterns widely. For example, only Swarm's classes for collections of objects (lists, arrays, maps) are designed so that any class operating on collections (for scheduling, data collection, graphing, etc.) can use any kind of collection.

IDEs such as Eclipse are very useful. Although Eclipse is itself complex, it was helpful from the start for students learning Java and Repast together. Particularly valuable are instant identification and help with syntax errors, menu tools for mundane tasks such as adding getter and setter methods, and the debugger—not only for testing code but also for understanding how an ABM executes (e.g., is the scheduler really randomizing the order in which agents execute?). NetLogo's integral development environment provides many of these benefits and is a major contributor to NetLogo's ease to use.

Scientific modelers need scientific tools, from the start. As agent-based modeling matures as a research approach, tools for scientific analysis are needed [12]. We found that students building scientific models need statistical output (e.g., maximum, minimum, variance of agent properties) after only one day's work. Statistics often need to be broken out by agent categories (e.g., age, sex). These capabilities are necessary to give users a view of simulation results that is more complete than graphical displays yet easy to generate and analyze. None of the platforms provides a complete tool for statistical output. Experiment managers that automate execution of multiple scenarios and replicates are also essential.

Understanding causality is an unfulfilled need. Modelers are constantly faced with the problem of figuring out why unexpected results arose in an ABM and whether the results are useful or only the consequence of mistakes. Probes and debuggers are helpful, but only for observing a few individuals or a small part of the software. Software tools for understanding causality in full-scale simulations are generally lacking. We find optional output files providing details of a particular part of the model useful, but additional tools need to be invented.

Is Java the right language? There are many differences between Java and Objective-C; our experience indicates that the following are important.

- **Syntax and object typing.** Objective-C code can be much shorter and easier to read than Java, in part because of its messaging syntax. Another reason is Java's strong typing: almost any use of an object requires telling the compiler the object's type. Because these platforms use generic classes extensively, programmers must frequently use Java's "casting" operator. Casting makes code harder to write and read and is a common frustration to beginners. With Objective-C's weak typing, Swarm users can operate on objects (e.g., schedule their actions, collect data from them) without worrying about what class they are. However, strong typing allows the compiler to find more errors, an important advantage.

- **Error checking and garbage collection.** Java provides much more help fixing run-time errors; in Objective-C even basic mistakes, such as having an array index out of bounds, cause cryptic, unhelpful error statements. Java's "garbage collection"—automatic removal of unused objects—is also a major advantage: even simple ABMs can create millions of objects very rapidly, so failure of Objective-C programs to drop unused objects can degrade performance quickly. Java precludes the need to even explain this problem to beginners.

- **Availability of development tools.** Many more development tools such as IDEs and libraries are available for Java. While the Unix-based tools for Objective-C (e.g., the Emacs editor, the gdb debugger) are productive for experienced users, there appear to be no tools for Objective-C as beginner-friendly or powerful as Eclipse (except in the new MacIntosh operating systems, with which the authors are unfamiliar).

4. Conclusions and Recommendations

The evolution of ABM platforms over the past ten or more years has been fascinating: more scientists and students take up more advanced modeling as platforms improve, while platform developers try to adapt to the growing and changing user communities. Our experience as modelers and instructors has been that software development remains a significant barrier to the use of ABMs, especially for the types of modeling and analysis needed to address substantial, real-world scientific problems. Therefore, we begin this section by attempting to provide guidance to potential platform users, and end by recommending priorities for the continued development and improvement of ABM platforms.

4.1 Which Platform is Right?

While there are benefits to standardization of tools, the variety of ABM platforms and their objectives also has its benefits. We cannot say that one platform is best for all modelers. Instead, we offer the following conclusions about each. We remind readers of the following: (i) these platforms continue to evolve, some rapidly; (ii) we could not attempt to explore all the ways our example models could be implemented in each platform and instead followed what seemed simple and intuitive approaches; (iii) there are many other platforms that we did not review.

NetLogo, with its heritage as an educational tool, stands out for its ease of use and excellent documentation. It is easy to recommend NetLogo for models that are (i) compatible with its paradigm of short-term, local interaction of agents and a grid environment and (ii) not extremely complex. We also strongly recommend NetLogo for prototyping models that may later be implemented in lower-level platforms: starting to build a model in NetLogo can be a quick and thorough way to explore design decisions. Its execution speed may not be a significant limitation for many applications, especially compared to the potential reduction in programming time. Experienced programmers, however, could be uncomfortable with NetLogo's simplified programming environment. Restrictions, such as having all code in one "file", enforce less organizational discipline than required in Java or Objective-C and can be cumbersome for large models. NetLogo provides an error checker and makes it easy to develop and try code in small steps, but lacks IDE features such as a stepwise debugger. Reproducibility may be a concern for some scientific users because NetLogo does not provide immediate access to the algorithms implementing its primitives.

Java Swarm seems not to offer a good trade-off of the relative benefits of Java and Objective-C. Java Swarm certainly met its design objective of providing Swarm for Java users; however, now that alternative Java platforms are available, its drawbacks are clear. They include the following: (i) somewhat clumsy work-arounds for Java's incompatibility with key Swarm features (e.g., selectors; the createBegin and createEnd paradigm); (ii) the difficulty of debugging run-time errors that occur in the Objective-C libraries; (iii) the source code being in Objective-C; (iv) slow execution speed. However, many of the positive aspects of Swarm discussed below also apply to Java Swarm.

MASON could be a good choice for experienced programmers working on models that are computationally intensive, for example, having many agents or long run times. MASON currently offers relatively few tools but supports computationally intensive models by allowing jobs to be moved among machines and allowing graphics to be detached and re-attached. Our rough execution speed evaluation found MASON generally to be the fastest platform, but Repast execution times were only 1–54% longer, with the difference generally decreasing as models became more computationally intensive. Several of MASON's innovations are quite clever, especially including all of the drawing methods in the user interface class, having the scheduler skip objects that have been destroyed, and a powerful control panel.

MASON is likely to be particularly challenging for novices or for ABMs with complex logic. Its scheduling concept, in which only methods named "step" can be executed, is designed for speed but complicates programming. Some things we did not like about MASON are the following: (i) its non-standard and sometimes confusing terminology (Table 2); (ii) its incompatible collection classes—the "bag" class could be useful but is incompatible with the scheduler; (iii) its lack of a terminal window to which debugging print statements can be written when working within Eclipse.

Objective-C Swarm is the father of the framework and library platforms and it still has advantages. Swarm is stable (new versions and even bug discoveries are rare),

is relatively small and well organized while providing a fairly complete set of tools, has a clear conceptual basis and a clever design, and allows clear separation of graphical interfaces and the model. Its concept of a "swarm" helps organize models: models can be designed and implemented as a hierarchy of separate swarms, each with its own collections of objects and schedule of their actions. This capability provides a level of organization useful for simple models (e.g., separate swarms for the model and its graphical interface) and especially for complex ABMs. (To represent the diverse life stages of a salmon population, two of the authors, Jackson and Railsback, have implemented a model with five *kinds* of swarm, with a variable number of swarms—each with its own schedule—instantiated as the model runs.) The disadvantages of Objective-C Swarm are primarily those of the Objective-C language (Section 3.5): a lack of novice-friendly development tools, weak error handling, and the lack of "garbage collection". These disadvantages are aggravated by an even lower availability of documentation and tutorial materials than for other platforms. Unfortunately, our tests do not allow clear execution speed conclusions for Swarm: for some types of model (e.g., version 1 of StupidModel) Swarm is much faster than the Java platforms and for other types of model Swarm is much slower. Objective-C should be faster for pure computation, but Swarm's use of its own code for low-level functions such as defining objects and storing their variables—the design that allows unique capabilities such as Swarm's probes—may have a speed cost.

Repast is certainly the most complete Java platform. In addition to implementing most of Swarm's functions, Repast has added such capabilities as the ability to reset and restart models from the graphical interface and the "Multi-run" experiment manager. We found its execution speed to be good compared to the other platforms. Repast also includes many classes for geographical and network functions.

While Repast is likely a good choice among the framework and library platforms for many scientists, we found it disappointing in several ways. While several major attempts to make Repast more accessible to beginners have been made, some of its basic elements seem incomplete or not very carefully designed. To start, the software distribution could be better organized, for example, by labeling packages primarily by their function and separating demonstration models and domain-specific tools from the core software. Examples of questionable design decisions include the following. (i) The schedule executes top-level actions in randomized order (in our experience, unlikely to be desirable) yet there is no built-in method to randomize the order in which one action is executed over a list of agents (commonly used to represent concurrent actions). (ii) Several incompatible types of collections are used (Java's ArrayList and Vector; Repast's DefaultGroup) while the scheduler can only use ArrayList. (iii) The DataRecorder is handy but does not provide a variety of commonly used statistics. As with MASON and Swarm, basic documentation is largely incomplete. The Repast development program is tremendously productive and helpful to the scientific community and none of these problems is severe, but tidying up and fully documenting Repast's core might benefit its users more than new development.

4.2 Development Priorities

If we could direct the work on the framework and library platforms, what would we do? Our recommended development priorities are listed from near- to long-term.

1. Fulfill the most critical, immediate need: complete documentation of classes and methods, with examples. Documentation has been a persistent problem, probably because developing and maintaining it is difficult and unglamorous, and good programmers are not always good at writing documentation. Yet nothing would do more to improve the usability of these platforms.

2. Continue developing—and maintaining—how-to documents and template models. Starting a new project by modifying a template model and copying examples can greatly reduce startup time and teach users how to do tasks that are common yet not so simple.

3. Integrate the software library with an IDE such as Eclipse. We found Eclipse more productive, even for beginners, than menu-driven (e.g., Repast Py) or graphical systems for building models.

4. Revive the "framework" part of the platform: establish the software library as one part of an overall process leading modelers through the model design, analysis, and publication cycle. One of the most important purposes of a platform is to provide a common language for thinking about and describing ABMs (see Chapter 8 of Grimm and Railsback [2]). This task is likely to require more thinking and writing (see, for example, Minar et al. [4]) than software development. Following standard terminology (Table 2) makes users' lives easier.

5. Make sure the framework accommodates complex, multilevel models. The growing use of ABMs to address real-world problems means such models will become more common.

6. Provide powerful tools for generating statistical output. (For example, two of the authors, Jackson and Railsback, have written a Swarm tool that creates output files providing a variety of statistics on an agent collection, broken out by categorical variables such as species, age, etc.) Existing statistical libraries could be used, but this capability needs to be built-in and easy to use because it is needed by even the most novice modelers.

7. Provide powerful tools for setting up and executing simulation experiments. (One of the authors, Jackson, has written an experiment manager in Objective-C Swarm that can manipulate input file names and parameters for any class, and can send different parameter values to different instances of the same class.)

8. Look for ways to improve the trade-off between ease of use and generality of platforms. One way to do so is by making common tasks (e.g., displaying spaces and agents; adding and removing agents from the scheduled list; looping through lists) easier to program, for example by using higher-level (NetLogo-like) code. High-level tools that hide some functions are very helpful and still allow models to be reproducible and flexible—if the tools are thoroughly documented (so users know how they work in full detail, which they do not with NetLogo) and can be overridden. Another technique is better use of standardized design patterns: making as many methods as possible as familiar as possible to users. Graphical or menu-driven tools for adding observer capabilities can be powerful, as illustrated by NetLogo and platforms such as Borland Delphi.

9. Research technologies for testing, analyzing, and understanding ABMs. Pervasive testing of ABMs (both verifying a code's accurate implementation of its model and validating the model's usefulness) and understanding causality—how results arose—are difficult and frustrating problems [5]. Developing tools for these problems should be a long-term goal.

5. Acknowledgments

We are very grateful to the platform developers and users who patiently answered many questions. Especially, we thank Marcus Daniels (Swarm), Tom Howe and Michael North (Repast), Sean Luke (MASON), and Uri Wilensky, Seth Tisue, and Bill Rand (NetLogo). Marcus Daniels kindly performed the profiling discussed for Swarm in Section 3.4. This work was funded in part by the National Center for Environmental Research (NCER) STAR Program, U.S. Environmental Protection Agency, under agreement RD-83088601-0 with Humboldt State University.

6. References

[1] DeAngelis, D. L., and W. M. Mooij. 2005. Individual-based modeling of ecological and evolutionary processes. *Annual Review of Ecology, Evolution, and Systematics* 36:147–68.

[2] Grimm, V., and S. F. Railsback. 2005. *Individual-based Modeling and Ecology*. Princeton, NJ: Princeton University Press.

[3] Grimm, V., E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, et al. 2005. Pattern-oriented modeling of agent-based complex systems: Lessons from ecology. *Science* 310:987–91.

[4] Minar, N., R. Burkhart, C. Langton, and M. Askenazi. June 1996. The Swarm simulation system: A toolkit for building multi-agent simulations. Report No. 96-06-042. Santa Fe, NM: Santa Fe Institute.

[5] Ropella, G. E. P., S. F. Railsback, and S. K. Jackson. 2002. Software engineering considerations for individual-based models. *Natural Resource Modeling* 15:5–22.

[6] Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. MASON: A multiagent simulation environment. *Simulation* 81:517–27.

[7] Tobias, R., and C. Hofmann. 2004. Evaluation of free Java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation* 7(1). (Available online at http://jasss.soc.surrey.ac.uk/7/1/6.html.)

[8] An, G. 2001. Agent-based computer simulation and SIRS: Building a bridge between basic science and clinical trials. *Shock* 16:266–73.

[9] Matsumoto, M., and T. Nishimura. 1998. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* 8:3–30.

[10] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA: Addison-Wesley Professional Computing Series.

[11] Ginot, V., C. Le Page, and S. Souissi. 2002. A multi-agents architecture to enhance end-user individual-based modelling. *Ecological Modelling* 157:23–41.

[12] Gilbert, N., and S. Bankes. 2002. Platforms and methods for agent-based modeling. *Proceedings of the National Academy of Sciences* 99(Suppl. 3):7197–8.

Steven F. Railsback *is a consulting environmental scientist and adjunct professor in the environmental modeling program, Department of Mathematics, Humboldt State University (www.humboldt.edu/∼ecomodel).*

Steven L. Lytinen *is a professor in the School of Computer Science, Telecommunications, and Information Systems, DePaul University.*

Stephen K. Jackson *is a consulting software developer and instructor in the Department of Mathematics, Humboldt State University.*