StarLogo Under the Hood and in the Classroom

Eric Klopfer
Massachusetts Institute of Technology
klopfer@mit.edu

Andrew Begel
University of California, Berkeley
abegel@cs.berkeley.edu

Abstract

StarLogo is a computer modeling tool that empowers students to understand the world through the design and creation of complex systems models. StarLogo enables students to program software creatures to interact with one another and their environment, and study the emergent patterns from these interactions. Building an easy-to-understand, yet powerful tool for students required a great deal of thought about the design of the programming language, environment, and its implementation. The salient features are StarLogo's great degree of transparency (the capability to see how a simulation is built), its support to let students create their own models (not just use models built by others), its efficient implementation (supporting simulations with thousands of independently executing creatures on desktop computers), and its flexible and simple user interface (which enables students to interact dynamically with their simulation during model testing and validation). The resulting platform provides a uniquely accessible tool that enables students to become full-fledged practitioners of modeling. In addition, we describe the powerful insights and deep scientific understanding that students have developed through the use of StarLogo.

1. Background

In the past 20 years a paradigm shift has been taking place in scientific research. A new approach to scientific inquiry has emerged that seeks to understand the intricacies of complex adaptive systems (CAS) by transcending separate disciplines and augmenting traditional experimental methods with the use of sophisticated tools for computer modeling. A growing group of scientists is adopting this approach to understand a range of such systems as varied as the human immune system and the global economy.

Scientists now have more powerful theories and tools for explaining and predicting the behavior of self-organizing, emergent systems, ranging from natural selection and adaptation in local ecologies (e.g., Salthe, 1993) to economic supply chains (e.g., Roy, 1998). These and many other subjects studied by complexity scientists are introduced to students in middle and high schools. Teachers do not typically teach these subjects as

complex systems, but rather as systems that behave more mechanistically. Consequently, students frequently have difficulty understanding the complex dynamics of such systems. For example, most teachers present the topic of ideal gases in physics class as a set of equations to be memorized. Instead, a teacher could present ideal gases as a particular example of a complex system and study it from the point of view of the interactions between gas molecules. Additionally, students tend to hold persistent misconceptions of how complex, adaptive systems work and develop incomplete models of these systems (Wilensky & Resnick, 1999). Moreover, students rarely have the opportunity to understand how to link multiple models together to construct and test alternative representations of situations, something scientists typically do when using models (Frederiksen & White, 2000).

Using simulations in the classroom can have many advantages. Not only do they allow students to explore systems as coherent bodies of knowledge instead of a disjoint collection of facts, but simulations allow them to explore systems at temporal and spatial scales that are not normally accessible to them in the classroom. Students can examine topics from molecular interactions to the evolution of new species which are difficult or impossible to explore experimentally without the aid of computers. Computer simulations in the classroom can be a pervasive glue that brings together experimental experiences to allow students to construct their own understandings of systems.

In this paper, we describe an approach to modeling, implemented using the StarLogo development environment, that makes it possible for students to study, hypothesize, construct, test, and evaluate their own models of complex systems. First, we show how StarLogo's approach to modeling is different than more traditional modeling environments. Then, we propose criteria that we believe to be important in designing such a modeling environment for students. Next, we introduce the StarLogo platform by way of an example model, and describe StarLogo's major actors: the turtles, the patches, and the observer. Viewed from a purely technological perspective, StarLogo has evolved over the past ten years from its origins running on a massively parallel supercomputer to one that runs on desktop computers of the kind found commonly in schools today. We describe, in a slightly more technical fashion, how StarLogo is structured internally as

well as discuss key design decisions that distinguish StarLogo from a professional modeling environment to one that is especially suited for students learning to model. We then talk about our approach to bring StarLogo to schools through workshops utilizing our new book, *Adventures in Modeling*. Finally, we tell some anecdotal stories about our students and their models from past workshops we have given, and then conclude.

2. Approaches to Modeling

Science has long relied on the use of scientific models based on ordinary differential equations (ODE) that can be solved (in simple cases) without computers. These models describe how aggregate quantities change in a system, where a variable in the model might be the size of a population or the proportion of individuals infected by a disease. The mathematics required for these models is advanced, but several commonly used modeling programs like Model-It (Soloway et al., 1997), Stella (Roberts, Anderson, Deal, Garet, & Shaffer, 1983), and MatLab (The MathWorks, 1994) have graphical interfaces that make them easy to construct. These programs have become very popular in the classroom as well as the laboratory. The user places a block on the screen for each quantity, and draws arrows between the blocks to represent changes in those quantities. While this interface does not remove the need for the user to learn math, it lowers the barrier for entry. However, the abstraction required to model these systems at an aggregate level is a difficult process for many people, which often limits the utility of this modeling approach.

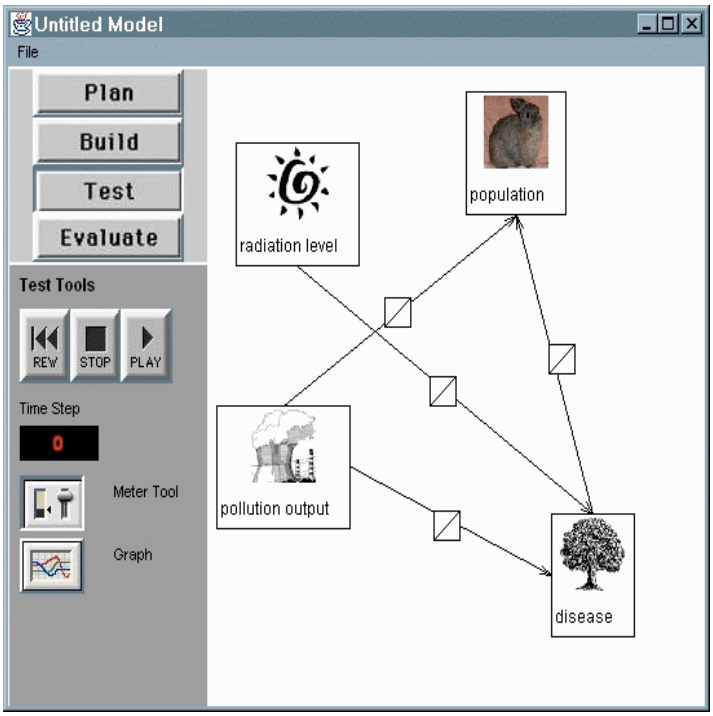


Figure 1 *A model built in Model-It that shows the impact of disease, pollution and radiation on a the numbers of rabbits in a population (courtesy of Model-It).*

Additionally, many of the systems that are studied in the classroom are more amenable to simulation using agent-based modeling. Rather than tracking aggregate properties like population size, agent-based models track individual organisms, each of which can have its own traits. For instance, to simulate how birds flock, one might make a number of birds and have each bird modify its flight behavior based on its position relative to the other birds. A simple rule for the individual's behavior (stay a certain distance away from the nearest neighbor) might lead to a complex aggregate behavior (flocking) without the aggregate behavior being explicitly specified anywhere in the model. These emergent phenomena, where complex macro-behaviors arise from the interactions of simple micro-behaviors, are prevalent in many systems and are often difficult to understand without special tools.

3. Additional Design Criteria

In designing an appropriate modeling tool for use in the K-12 classroom we needed to consider several design criteria. One criterion is the foundation on agent-based as opposed to aggregate-based modeling. This approach is not only more amenable to the kinds of models that we would like to study in the classroom, but it is also readily adopted by novice modelers.

The next criterion is to create a modeling environment that is a "transparent box." Many of the simulations that have been used in classrooms to date are purchased for the purpose of exploring a specific topic such as Mendelian genetics or ideal gases. Modeling software has been shown to be particularly successful in supporting learning around sophisticated concepts often thought to be too difficult for students to grasp (Roschelle & Kaput, 1996; White, 1993). This software allows students to explore systems, but they are "black box" models that do not allow the students to see the underlying models. The process of *creating* models—as opposed to simply using models built by someone else—not only fosters model-building skills but also helps develop a greater understanding of the concepts embedded in the model (Resnick, Bruckman, & Martin, 1996; Roschelle, 1996; White, 1993). When learners build their own models, they can decide what topic they want to study and how they want to study it. As learners'

investigations proceed, they can determine the aspects of the system on which they want to focus, and refine their models as their understanding of the system grows. Perhaps most importantly, building models helps learners develop a sound understanding of both how a system works and why it works that way. For example, to build a model of a cart rolling down an inclined plane in the population Interactive Physics program, a student could drag a cart and a board onto the screen and indicate the forces that act about each object. In doing so, the student assumes the existence of an unseen model that incorporates mass, friction, gravity, etc. that calculates acceleration of the cart. It would be a much different experience to allow the student to construct the underlying model herself and have that act on the objects that she created.

We also considered the level of detail that we felt would be appropriate in student-built models. All too often, students want to create extremely intricate models that exhaustively describe systems. But it is difficult to learn from these "systems models" (Roughgarden 1996). It is more valuable for students to design and create more generalized "idea models" that abstract away as much about a system as possible and boil it down to the most salient element. The ability to make these abstractions and generalize scientific principles is central to the idea of modeling. For example, a group of students might want to create a model of a stream behind their school, showing each species of insect, fish and plant in the stream. Building such a model is not only an extremely large and intricate task, but the resulting model would be extremely sensitive to the vast number of parameters. Instead, the students should be encouraged to build a model of a more generalized system that includes perhaps one animal and one plant species. The right software should support the building of such "idea models."

Based on these criteria and our approach to modeling, we created a computer modeling environment, named StarLogo, that we describe in the next section.

4. The StarLogo Platform

To enable students to build their own CAS models, we developed StarLogo, a programming language and environment specifically designed to support simulation design, construction and testing (Resnick, 1994). While there have been several versions

of StarLogo on different platforms through the years (detailed below), they have each strived to meet the design goals described above and provide a program language and development platform that is accessible to students of a broad age range. Each version of StarLogo has shared many common language features and an underlying metaphor that describes the StarLogo world in terms of three entities – turtles, patches and an observer. "Turtles" is our term for all entities that move. On other platforms these might be called "agents." But StarLogo's lineage brings with it the turtles that defined Logo, along with much of that language as well. While we call them "turtles" generically, they might be rabbits, atoms, or cars in any particular model.

The turtles walk around on top of a grid that is composed of patches. If you think of the grid as a large checkerboard, each square on the checkerboard is a patch (and the turtles would be the checkers moving over the board). The turtles can interact with the patches by responding to their features, or even modifying their features. One way for turtles to modify the patches is by using the pen that each turtle carries to draw on the patches. The patches are also able to run their own instructions, through which they can modify themselves or the turtles that are standing on them.

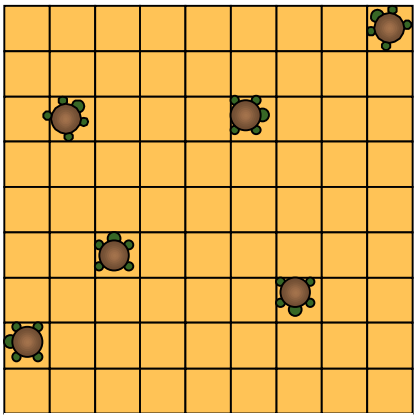


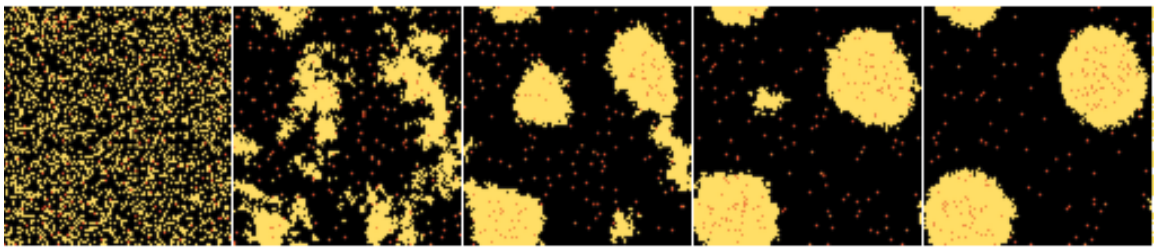
Figure 2 *Turtles move around on a grid of patches.*

Finally, there is a single observer that watches over the entire grid. Continuing with the checkers metaphor, the observer might be thought of as a person playing (solitaire) checkers on this checkerboard filled with the turtles and patches. The observer takes care of certain operations like clearing the whole board, creating new turtles, and keeping track of time that either cannot be done by individual turtles, or are easier to conduct through the observer.

4.1. Termites Example

To familiarize readers with StarLogo, we present a small StarLogo project about termites. This project is inspired by the behavior of termites gathering wood chips into

piles. The termites follow a set of simple rules. Each termite wanders randomly. If it bumps into a wood chip, it picks the chip up, and continues to wander around. When it bumps into another wood chip, it finds a nearby empty space and puts its wood chip down. We show a run of this simulation in Figure 3.



(a) (b) (c) (d) (e)

Figure 3 *This figure shows the time evolution of the termites project. At the beginning (a), all termites and wood chips are randomly scattered over the patches. As the termites pick up and drop the wood chips (b), the number of piles begins to decrease (c). Small piles shrink (d) and eventually disappear (e). If we run this further, all of the wood chips will end up in one pile.*

We look at the StarLogo source code to get a feel for what the language feels like. The following is the setup procedure for termites.

```
to setup
      clearall
      if (random 100) > 80 [setpatchcolor yellow]
      create-turtles 200
      ask-turtles [
        setcolor red
        setxy random (screen-edge * 2)
              random (screen-edge * 2)
      ]
end
```

To begin, we kill all of the turtles and set the patch colors to black. Then, on each patch, we throw a random 100-sided die. If it exceeds a threshold, we set the patch's color to yellow (i.e. we give it a wood chip). We then create 200 turtles (termites), ask them to color themselves red, and scatter them around the screen.

```
to go
      search-for-chip
      find-new-pile
      find-empty-spot
```

```
end
```

The go procedure is the main loop. First, a termite looks for a wood chip and picks it up. Then it wanders until it finds another wood chip in a pile and finds a place to put it down.

```
to search-for-chip
      if patchcolor = yellow
        [stamp black jump 20 stop]
      wiggle
      search-for-chip
end
```

In search-for-chip, a termite wanders around, wiggling, until it is standing on a yellow patch. That means there is a wood chip there. It picks up the wood chip and jumps 20 turtle steps away.

```
to wiggle
      forward 1
      right random 50
      left random 50
end
```

A termite wiggles by moving forward one turtle step, then turning right and left a random number of degrees.

```
to find-new-pile
      if patchcolor = yellow [stop]
      wiggle
      find-new-pile
end
```

The termite then wiggles around until it finds a pile to put down the wood chip.

```
to find-empty-spot
      if patchcolor = black
         [stamp yellow get-away stop]
      setheading random 360
      forward 1
      find-empty-spot
end
```

A termite does not want to put a wood chip down on top of another one, so it moves forward in a random direction until it finds an empty patch. Once it finds that spot,

it stamps the patch to make it yellow (giving it the wood chip), and then jumps away to look for new wood chips.

```
to get-away
      setheading random 360
      jump 20
      if patchcolor = black [stop]
      get-away
end
```

To get away, a termite keeps jumping 20 steps in random directions until it lands on a spot without any wood chips. Then it starts the cycle over again, looking for another wood chip to pick up.

One interesting thing to notice about this model – over time, the number of piles of woodchips decreases. Why? There's certainly nothing obviously programmed into the model to make this happen. But, inevitably, when the simulation is finished, the termites will be left with one pile. How does this happen? First, ask yourself how a pile can disappear. A pile disappears when termites carry away all of the chips. Can a new pile be formed? No, there is no way to start a new pile by a deliberate termite action, since termites only put their wood chips next to other wood chips. This behavior will only increase an existing pile's size; it will not create a new separate pile. In fact, the only way to create a "new" pile is to take away enough woodchips to split an existing pile into two. However, this is quite rare except when a pile gets very small. So, overall, the number of piles must decrease until there is only one left. This is known as an emergent property of the model – an aggregate behavior that is unexpected given the simple rules programmed into the model.

Next, we will begin the discussion of how StarLogo works internally. This discussion will be more technical than the rest of this article, but will reveal some interesting design tradeoffs that are visible directly to the user and affect the fidelity of the models that can be built.

5. StarLogo Design Through the Ages

Building an agent-based modeling environment like StarLogo is not a trivial task. It involves balancing the pedagogical needs of students with the efficiency requirements of running thousands of agents at the same time. In this section, we will give a little flavor of what actually runs under the hood to make StarLogo go.

There are three main pieces of work: the design of the StarLogo virtual machine (which has changed over time as the technology has improved), the process scheduler which determines the order in which turtle, patch and observer operations run, and the turtle, patch and observer data structures (which form the core data of the StarLogo runtime system).

5.1. The StarLogo Virtual Machine

StarLogo has more going on in parallel than most other computer environments. Several thousand turtles move about on a grid of over ten thousand patches, each one independently executing code. Our first implementation of this system was on an actual parallel computer called the Connection Machine. With 16,384 physical processors, each running at 8 MHz, we could devote one processor to each turtle and have plenty to spare. This gave us plenty of parallelism, but the machine's bulk and extreme cost made it all but inaccessible to school children.

To solve this problem, we brought StarLogo to the Apple Macintosh in 1994. The challenge now was to bring our parallel environment to the moderately under-powered, single processor, desktop computers that were common in schools at the time (our target platform at the time was a Mac IIfx, with only a single 25 MHz processor and 16 MB of RAM). A single processor would have to be made to emulate thousands of "virtual processors," something that we recognized would be a challenge.

How does a computer run more than one thing at a time? Instead of trying to run all programs at the same time (which is impossible on a computer with only one processor), we run each one by itself for a short amount of time, and then switch to the next one. Each program is deceived into thinking it is the only program running on the processor. Most operating systems such as Windows or MacOS are designed to run tens or hundreds

of these processes at a time, each one getting one little slice of the processor's time (usually around 16 milliseconds per slice). Since they are switching so fast, the user perceives everything on the computer as running in parallel. StarLogo's needs require that a processor be able to switch between tens of *thousands* of processes at a time, which is beyond the capabilities supported by any operating system.

To achieve the desired performance, we created a *virtual machine* – a simulation of one computer within another – to run the StarLogo programming language. To this very basic machine (written in extremely low-level machine language for speed), we added a large number of turtle, patch, and observer commands, and created an extremely lightweight multi-processing system to run it all.

The StarLogo virtual machine is very similar to the one developed for the Cricket (Mikhak, Berg, Martin, Resnick & Silverman, 2000), another project hosted in our research group at the MIT Media Laboratory. A Cricket is a tiny computer, powered by a 9 volt battery, that can control two motors and receive information from two sensors. Crickets are equipped with an infrared communication system that allows them to communicate with each other. Powered by a Microchip PIC processor, the virtual machine in Cricket Logo holds about 1000 words of instruction memory and only a few dozen words of RAM. Even though a modern desktop computer is much more powerful than a PIC microcontroller, there is some good justification to use a similar virtual machine for both the Cricket and StarLogo. Even though the desktop computer is much faster than the PIC, the PIC only has to run one process, while the desktop computer has to run thousands. Similar memory constraints hold as well.

For some years after the initial Macintosh version of StarLogo, we received requests by an increasing number of users who wished to see StarLogo on a Windows PC. Around this time (1995), the Java programming language was introduced by Sun Microsystems, Inc., so we decided to take the opportunity and rewrite StarLogo in this new cross-platform language. Designing it in the same way as our Macintosh version, we were able to directly port over all of the code for StarLogo's virtual machine into Java fairly quickly, and within three days of programming, had a first implementation of a turtle world running. Gradually, over a period of a few years, we recreated the rest of the

StarLogo experience in the Java realm. This is the version now available at our website: *http://www.media.mit.edu/starlogo*.

5.2. The Anatomy of a Virtual Machine

What exactly is found within a virtual machine? A virtual machine is a computer program that simulates the behavior of a physical processor. An example of a commercial virtual machine is Virtual PC for MacOS, which simulates an Intel Pentium processor on a Macintosh computer. Simulations of commercial processors make up a small number of the kinds of virtual machines available, however. Designers take advantage of the inherent flexibility of software programming to create virtual machines that simulate processors never before found in nature. The StarLogo virtual machine is this kind of virtual machine; it simulates a Logo processor.

Logo is a programming language invented in the 1960s by Wallace Feurzig and Seymour Papert. A variant of the Lisp programming language, Logo was designed to be easy to learn and use by children. It also tends to be easy to implement using a virtual Logo processor (a piece of software that our group at MIT has created numerous times for almost every project we undertake). StarLogo's Logo processor contains two types of commands: Logo language operations, and StarLogo primitive commands. There are ten commands used for implementing Logo; these handle procedure invocation and returns and manage data and instruction lists. The other 300 primitives provide support for moving the turtle around on the screen, communicating between the turtles, patches, and observer, observing and modifying the turtle's environment, reading and writing turtle, patch and observer state and user variables, and executing mathematics and control functions.

All modern machines (including simulated processors, like the one we built) execute in similar fashion. A user begins by starting a program (often by double-clicking on an icon). The operating system loads the program's code into memory, creates a new *process* to store the program's execution state, and jumps to the program's starting point. The processor loads the program's instructions, one by one, into its execution unit. After each instruction runs, the processor increments its instruction pointer (which points to the

next instruction to be executed) and continues. After some amount of time, a timer goes off and signals to the processor that the current process has been running for too long. The *process scheduler* captures the runtime state of the current process, stores it, finds another process that is ready to execute and swaps it in. This swap is known as a *context-switch*. The process scheduler is responsible for performing context-switches, as well as determining the order of the processes that get to execute.

The StarLogo virtual machine adds three more pieces to this generic processor: the turtles, the patches, and the observer. Each of these entities is an object in memory. Turtles are made up of turtle state (the coordinates of the turtle on the screen, its id number, its color, heading, breed, shape, pen state (up or down), visibility (shown or hidden), and a timer), bookkeeping data (a pointer to the patch the turtle is currently standing on, a set of "underme" and "overme" pointers to keep track of turtles stack on top of one another, a pointer to the partner turtle used when this turtle is communicating with another one, and a true-false variable that indicates when this turtle is alive (useful for generating proper error conditions)), and a collection of user-defined variables (in StarLogo terminology, *turtles-own* variables).

A patch is like a turtle in structure, but contains less information. Patch state consists of the patch's coordinates on the screen, a patch color, and a pointer to the first turtle standing on the patch. Patches contain less bookkeeping data as well, only requiring a pointer to a partner turtle used when the patch is communicating with a turtle. Finally, patches also contain a collection of user-defined variables called *patches-own* variables.

The observer contains only a collection user-defined variables called *globals*.

5.3. The Process Scheduler and Its Processes

An important part of the design of StarLogo is the process scheduler. As we discussed above, the scheduler controls the order of execution of the processes that are running. It also controls how long a process gets to run before being swapped out for another. Both of these tasks are controlled by a carefully chosen policy. We will discuss the rationale for our particular choices below.

Recall that StarLogo is supposed to run the turtle and patch processes fast enough to appear to be running in parallel. In order to maintain the fiction of parallelism with a single physical processor, we must context-switch rapidly among all the processes that all turtles, patches and the observer are executing. There are two forms of context-switching that we could choose from. We could support *preemptive* multi-processing, in which a timer goes off every few milliseconds and causes the virtual machine to context-switch, or we could choose *cooperative* multi-processing, and only context-switch at carefully chosen points in the program.

We chose the latter for several reasons, but will only explain the most important one here. Under preemptive multi-processing, synchronization issues would become exposed to the user. For instance, the following common StarLogo idiom would not work as expected:

```
if count-turtles-here > 1
   [mate-with one-of-turtles-here]
```

A turtle is looking for another to mate with. It looks on the current patch to see if there is any other turtle there. If there is, it mates with it by asking for its turtle id and calling a user-defined procedure mate-with. Under preemptive multi-processing, it is possible to context switch between the condition `count-turtles-here > 1` and the consequent of the if statement, `mate-with one-of-turtles-here`; if this happens, in one glance, the first turtle might see the other one on its patch, but in the next, the other turtle may have moved before the first turtle has had to a chance to mate with it!

To avoid this kind of problem, we only allow context switches at "safe" times such as the end of each *command*. A *command*, in Logo, is what we call a primitive operation or user function that does not return a value (e.g., `setcolor blue`, `forward 10`). In contrast, a *reporter* is a primitive or user function that does return a value (e.g., `3`, `5 + 7`, `color-of 5`). In our cooperatively multi-processing virtual machine, we elect to context-switch between commands, but not after reporters. This gives the *if* statement above a guarantee of *atomicity* (meaning that the two statements must execute together without any context-switching in between). The reporter in the predicate is guaranteed to still be true when the first command in the consequent executes. This policy also enables

the fairly common idiom of fetch and update (e.g. `setfoo foo + 1`) to work without user-supplied synchronization, as well.

There are several primitives that interact with context-switches. One in particular is the forward command. When a turtle wants to move in the direction of its current heading, it calls forward with a number of turtle steps. If we implemented this by context switching after each turtle went forward the full number of turtle steps, we would see (if we slowed the computer down) individual turtles scooting, one by one, to their final locations. We wish to maintain an illusion of realistic-looking parallelism, so we want to see all of the turtles move forward one step at a time. To do this, we context switch in the middle of the forward primitive after the turtle has taken one step. We jump back to the middle of the forward primitive when the turtle is rescheduled. Context switching on this granularity gives us the nice-looking parallel behavior we want.

StarLogo has two scheduling policies that can be selected by the user. The scheduling policies influence the order of process execution during each *once-through*. A *once-through* is one complete iteration of all of the processes – a unit of execution in our scheduler, where we are sure that we have given each process one unit of time to execute. Between consecutive once-through's, the particular scheduling policy chosen may have affect the order of process execution without accidentally starving any particular processes of a chance to run. The first scheduling policy executes each process *in-order* as it appears in the scheduler. This policy gives reasonable user-visible behavior in many cases, however it sometimes introduces artifacts into a user's program.

For example, consider a rope made up of individual turtles spread across the screen. If we jiggle one of the turtles, it should exert a spring force on its two neighbors. If we force the leftmost turtle to move up and down in a sine wave, it will send a sine wave down the right side of the rope. To make this happen, each turtle computes its change in velocity as a function of its distance from its two neighbors, and then moves. If we run this spring force process for each turtle, and the turtles execute it from left to right across the screen, the first turtle to move is the one directly to the right of the sine wave turtle. This turtle's motion propagates to the right and the sine wave appears to travel to the right across the turtles. Consider what would happen if the spring force process instead

executed from right to left. The rightmost turtle will not move, because its left neighbor has not moved. Its left neighbor will not move either. In fact, no turtle will move until we execute the spring force function for the turtle directly to the right of the sine wave turtle. The next iteration repeats this non-motion except where the wave has propagated to the right by one turtle. This kind of artifact can be completely unexpected but occurs because the turtle processes *are* executing in series, rather than in parallel.

To eliminate this artifact, we support a *randomized* scheduling policy. Before each once-through, we randomize the order of all processes in the scheduler. This produces somewhat more disconcerting appearing behavior (a more jaggy motion when you see a line of turtles do something), but it removes order-dependent artifacts, and more accurately simulates parallelism in most models.

5.4. The StarLogo Interface

The implementation of the StarLogo virtual machine is critical to its operation. But to most users, these details are transparent, allowing them to take on the modeling challenge at hand. Instead, what most users experience is a user-friendly graphical environment that simultaneously facilitates the creation of models and an accompanying user interface. A user of a model is presented with a screen that displays the running model along with the interface elements that control it (Figure 4).

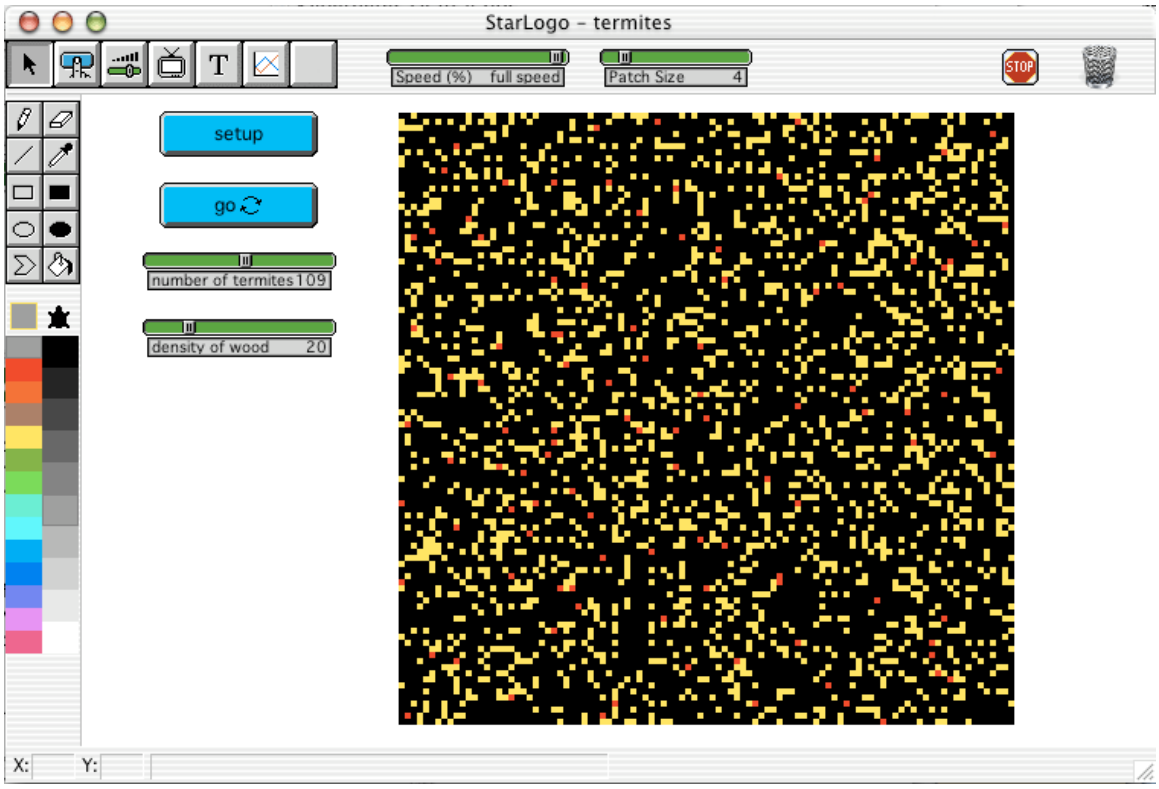


Figure 4 *The StarLogo 2.0 user interface for the Termites project. There are two user-created buttons:* setup *runs the user's setup procedure, and* go *is a forever button that execute the main loop of the model. There are two sliders which enable a user of the model to control the initial number of termites created as well as the density of the woodchips in the environment.*

All user interface elements are placed on the screen merely by clicking on the appropriate tool and then on a blank space in the StarLogo window. The creator of the model can then specify the instructions or values associated with that user interface element. There are several types of user interface elements in StarLogo including tools for input, output, and help. The primary user interface elements in StarLogo are buttons and sliders.

Buttons control the execution of procedures. For example, in the termites model there is a button that controls the *setup* procedures, which creates the initial distribution of termites and wood. (A button behaves similarly to an icon in Windows or MacOS. When it is pushed, StarLogo creates one process for every turtle that exists and starts to execute the instructions in the virtual machine.) The termites model also contains a *go* button that executes the instructions for the termites to move around and pick up wood.

This button is different than the *setup* button in that once it is pressed down it stays down until it is pressed again. This kind of button, known as a forever button and represented by the two looping arrows, causes the turtles to continuously follow the prescribed instructions, until the user pops the button using the mouse. (When the user pops a button, it causes the StarLogo process scheduler to find the processes associated with the button, and remove them from the running process list.)

Sliders allow the user to control global variables in the model. The sliders might affect the model at setup time (e.g. by controlling the initial distribution of termites and wood) or dynamically at run time (e.g. by changing the probability that a termite picks up wood when it sees it). The slider values are changed by clicking and dragging on the slider to manipulate the designated variable. For example, in the termites model the user can specify the number of initial termites or the density of wood.

The ability to design and implement a simple user interface makes the StarLogo modeling experience highly interactive. In addition to buttons and sliders modelers can place monitors that provide continuously updated numeric output, labels and legends that assist in the operation and interpretation of the model, and graphs that give visual feedback from the models, Together these tools allow developers to rapidly create useful models.

6. Learning to Model Through *Adventures in Modeling*

Using computer simulations of complex adaptive systems as a platform, we have crafted an introduction to scientific modeling (Klopfer & Colella, 1999; Klopfer & Colella, 2000). We have found that these tools can enable students to become full-fledged practitioners of modeling (Colella, 2001). Students design scientific models and then go on to investigate and explore those same models. The use of these tools allows non-experts to act as scientists, creating and exploring models of phenomena in the world around them, evaluating and critiquing those models, refining and validating their own mental models, and improving their understandings (Colella, 2001).

The StarLogo Workshops are designed to introduce participants to the computational and cognitive aspects of modeling complex, dynamic systems. During

these Workshops, participants work together to design, build, and analyze agent-based computer models. Participants engage in an iterative process of model creation and scientific investigation as they explore important scientific principles and processes. We design the Workshops to foster a playful, cooperative, creative spirit, while at the same time providing adequate structure for learning how to build models. To accomplish this balance between structure and exploration, we organize the Workshops around a set of open-ended StarLogo design Challenges on the computer and a series of off-computer Activities in which participants enact and analyze a simulation.

Each Challenge is a problem statement that is meant to guide participant's explorations and get their creative juices flowing. For example, one Challenge asks participants to build a model in which creatures change their environment and subsequently react to those changes. In response to this Challenge one might create a model of a beaver altering its environment by cutting down trees to build a dam, or termites chewing on a log to create passageways. Every Challenge includes sample projects, which teachers are encouraged to explore. The Challenges and accompanying sample projects facilitate model design and construction, build familiarity with the StarLogo environment, and introduce the principles of complex systems.

Though "on-screen" computer modeling is one focus of our workshops, "off-screen" Activities provide another way to connect abstract notions of scientific systems to personal experience (Colella 2001). These Activities allow participants to think about concepts like exponential growth, local versus global information, and group decision-making from a personal perspective. For instance, in one Activity, participants "fly" around a parking lot trying to form cohesive "bird flocks" without the assistance of a leader.

Recently, we have captured the essentials of our workshops in a book entitled, "*Adventures in Modeling: Exploring Complex, Dynamic Systems with StarLogo*" (Colella et al. 2001) published by Teachers College Press. This book brings the Design Challenges and supporting Activities to students and educators everywhere. *Adventures in Modeling* includes a series of ten StarLogo Design Challenges, and a complementary set of ten participatory Activities. Additionally, we provide guidelines for educators in

facilitating the Challenges and Activities, for integrating them into a variety of classes, and for mapping them to state curriculum standards. As a package, it provides a flexible but well defined pathway for teachers to follow.

8. Lessons Learned

It is difficult to appreciate the many ways that people use and learn from StarLogo without seeing it in action. In the following section we relate several stories of how we have observed people use and learn from StarLogo. Each of the stories highlights aspects of our software and our approach to modeling that we have found to be unique enlightening.

8.1. Fire

Often science classes dictate the explorations that students undertake during their laboratories. Typically these "experiments" follow a prescribed set of instructions outlined in a cookbook fashion. These experiences leave little room for the students to inject any of their own personal interests or fascinations into the experiments. Not surprisingly, students often lose interest in these experiments and lose sight of the real scientific method as they search for the "right" answer to the experiment.

One of the powers of StarLogo is that it allows students to conduct explorations of systems that are interesting to them. Students can explore systems that are personally interesting. This brings the flexibility of a science fair to the classroom, but removes the constraints of the size, location, timescale or cost of the system that the student wants to explore. A student could easily explore the interactions of protons and electrons in an atom, or the evolution of a trait over hundreds of generations— systems that could not normally be explored in a classroom.

During the summer of 2000 forest fires raged across New Mexico, causing extensive damage across thousands of acres and making national news headlines. At that time a student summer school was taking place in Santa Fe, New Mexico. One group of three ninth-grade girls was intrigued by the forest fires that summer. There were often small forest fires in that area at that time of year, but what made them so bad this time? They decided to create a model of forest fires to explore this phenomenon.

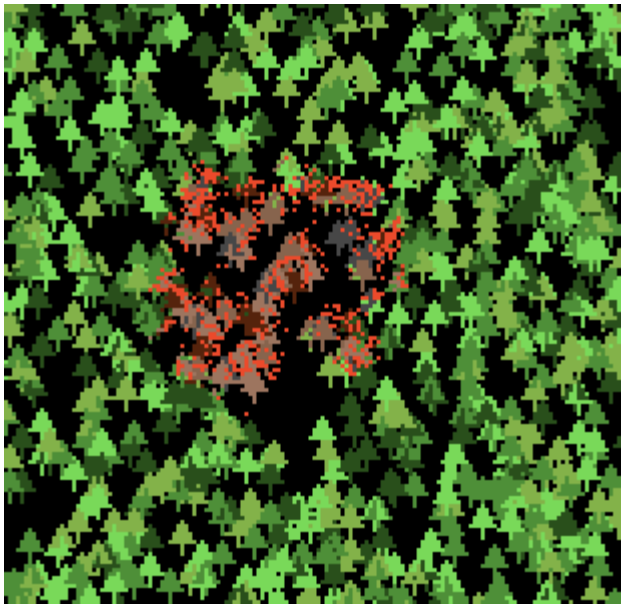


Figure 5 *A student-generated model of a forest fire. The different color trees have variable flammability. The red areas are burning, and the brown areas are already burned.*

The model that they created initially included many factors. After some exploration they narrowed down the factors of interest to wind, rain, and density of trees. They made sliders that controlled each of these factors and recorded the number of trees burned, number of trees alive and time to extinguishing of fire for several combinations of these parameters. In the end they were able to develop an understanding of each of the factors in isolation, as well as in combination such as high density of trees, low rain and high winds.

8.2. But This One Goes to 1000

One of the most important design features of StarLogo is that all aspects of the models and modeling environment are open to user inspection and manipulation. While the designer of a model might set parameters or specify particular behaviors, this information is always accessible to the user and can be changed.

Recently, a fifth grade class that was beginning to learn StarLogo by playing with a sample project from the *Adventures in Modeling* book. The model included many buttons and sliders that controlled the movement and creation of turtles. One boy was fiddling with the slider that controlled the numbers of turtles that were on the screen, which by default, ranged from 1 to 50. At some point, he double-clicked on the slider to see what it would do. He was then presented with a dialog box that controlled the minimum and maximum values for that slider. Being a fifth grade boy, he immediately replaced the seemingly small value of 50 turtles with a new maximum of 1000 turtles. He tried out the new value and quickly proclaimed his finding as "cool" since the new patterns were much different than the old ones with 50 turtles.

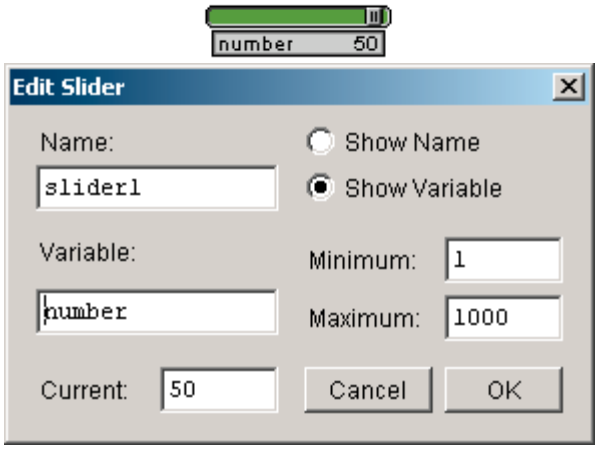


Figure 6 *A StarLogo slider (top) is clicked on to change the maximum value from 50 to 1000 (bottom).*

Figure 7 *Two students working exploring a StarLogo model. A student makes a discovery (left) and shares it with his neighbor (right).*

While this innovation was interesting, it might have taken quite some time for others in the room to make similar changes if each one of them had to independently discover this same mechanism. But the accessibility of StarLogo, and the social

atmosphere that it facilitates in the classroom permits and encourages the sharing of information. Within minutes of the boy's discovery of the way to change the slider, nearly half the class had changed their sliders in a similar way. Of course, this being a fifth-grade class the idea never jumped the boy-girl divide, and until they were forced to share, the girls were left out of the loop.

8.3. The Evolution of Rabbits and Grass

StarLogo models have an advantage over off-the-shelf simulations when it comes to integrating them into the curriculum. Unlike typical purchased simulations that are closed off from being changed by the user, StarLogo models can be customized to fit the unique needs of each classroom. One good example of this customization started with our *Rabbits and Grass* project that has long been a part of StarLogo. In *Rabbits and Grass* the rabbits (represented by red turtles) move around randomly on the screen, using up small amounts of energy as they move, but gaining energy as they eat grass (green patches) when they encounter it. If the rabbits gain enough energy they can reproduce (by binary fission), but if they lose all of their energy they will die. These rules lead to the classic predator-prey oscillations that are characteristic of the Lotka-Volterra equations often studied in beginning calculus.

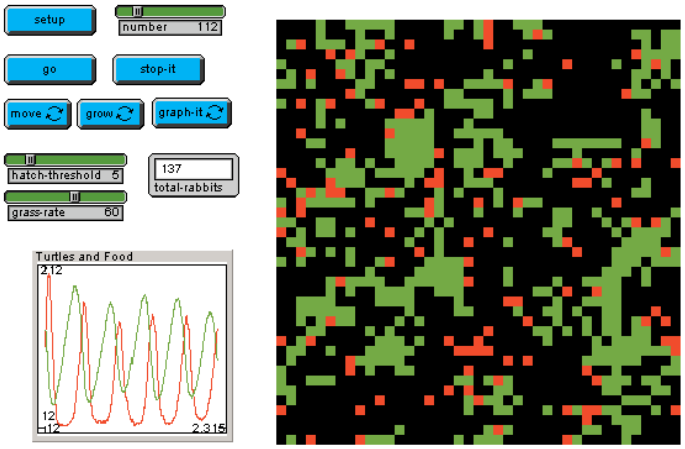


Figure 8 *The Rabbits and Grass model showing the oscillations in predator-prey populations over time. The rabbits (red) increase as grass (green is plentiful). The rabbits soon deplete their resources, and die off, allowing the grass to return. The return in food is followed closely by a return of rabbits.*

Some time ago, Noah, a biology teacher who was in one of the *Adventures in Modeling* workshops, was tinkering around with the *Rabbits and Grass* model and decided to make a small change. Instead of the rabbits all being red when they were created, they were randomly assigned a color. When the modified model ran for a couple

of oscillations, quickly there were only one or two colors of rabbits left. After searching through the code for a possible bug that would lead to this behavior Noah realized that this was indeed not a bug, but a feature of his new model. The predator-prey model was now a model of genetic drift. As the rabbit population got very small on the downside of the population oscillations, there would only be a few rabbits left. Those remaining rabbits would found the next generation of rabbits that would reproduce rapidly when the food became more plentiful. The offspring of these rabbits would be the same colors as they were, leading to a population in the next generation with limited colors.

Noah said that the students in his class rarely understand this bottleneck effect after reading it in their texts. But there are no traditional laboratories that can help the students explore this and related phenomena first hand. So, Noah set out to build an entire suite of *Rabbits and Grass* models for teaching ecology and evolution to his students. He built models with sexual and asexual reproduction, selection, mutation and genetic drift, and his model that put it all together was a model of speciation. In the speciation model, students can apply different selection pressures to a population of rabbits that is split due to an earthquake that isolates the two populations. These principles are difficult to teach to high school students, perhaps because the time scales are so long. As students experience evolutionary phenomena in real time through these models, Noah says that his students are developing a much deeper understanding of the concepts, and enjoying themselves in the process. In fact a neighboring biology class complained that Noah's class was having too much fun.

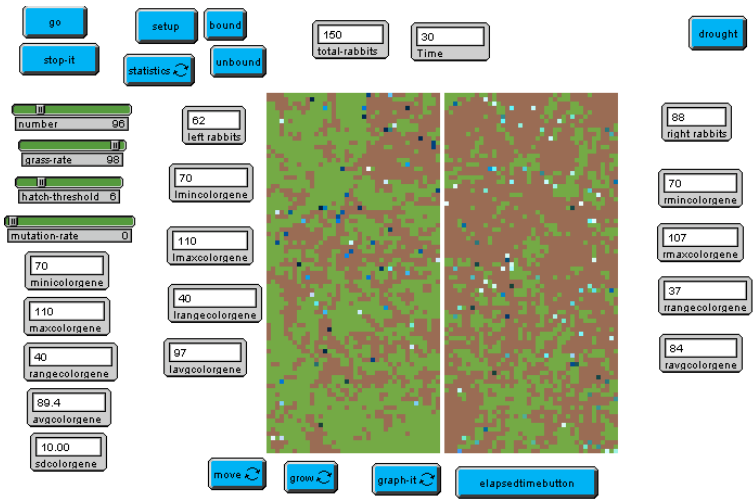


Figure 9 *The speciation model, showing the two populations of rabbits and grass divided by an impassable white barrier. The two populations can be given the same, or different selection pressures, and students can observe how long it takes to create a new "species."*

8.4. The Tides are Turning

In another of our recent *Adventures in Modeling* workshops, two of the participants were interested in exploring the patterns in formation of tidal sandbars. This phenomenon is difficult to study in nature because of the large temporal and spatial scales required. Most modeling tools are not applicable to this purpose either, because this process is intensely visual. So these participants were excited to be able to have an opportunity to explore this phenomenon.

The modelers started their project by borrowing a landscape generation procedure from another project. This gave them the capability to create underwater terrains with some existing variation in sand height. From there they set out to implement wave action. They represented waves by a line that swept from left to right on the screen, moving some of the underlying sand to adjacent patches. After exploring this version for some time, they weren't satisfied with the scale at which they saw the sandbars changing. So, they decided to add another feature that they thought might be important – tides. Tides were implemented by rising and falling water tables that caused underlying larger scale changes in the sand, and also interacted with the waves as they moved across the water. The resulting model produced some striking visualizations of tidal and wave movement of sandbars.

9. Conclusion

StarLogo provides many unique benefits to students when used in the classroom. It can change the way that kids view both science and technology as well as the relationship between the two. By empowering kids to "take over the technology," StarLogo allows students to become creators, not just consumers of technology. It also helps students develop deep understanding of scientific concepts as they design, build and explore models of systems in which they are personally invested. Further, StarLogo provides an opportunity to engage students in what we deem "the real scientific method," the messy process of iteratively designing experiments, learning about systems, and subsequently modifying experiments.

While StarLogo has met with much success in many classrooms, there are still a lot of students left to reach. But we, too, are engaged in an iterative process of model building, where the StarLogo world is our model. As we learn from our own experiments in the classroom use of StarLogo, we modify the tool to better meet the needs of our audience. This means providing greater accessibility and applicability through the development of new tools and techniques. This spring, we once again enter the design phase for "StarLogo: The Next Generation," and will begin constructing the software and developing new workshops later on this year.

Acknowledgements

Mitchel Resnick, Vanessa Colella, Brian Silverman, and countless MIT undergraduate student researchers have provided inspiration and ideas for the StarLogo project. We are grateful to C. Andrew Frank for helpful comments on an earlier draft of this paper. The LEGO Company and the National Science Foundation provided financial support.

References

Colella, V. (2001). Participatory simulations: Building collaborative understanding through immersive dynamic modeling. *Journal of the Learning Sciences*, *9* (4), 471-500.

Colella, V., Klopfer, E., & Resnick, M. (2001). *Adventures in modeling: Exploring complex, dynamic systems with StarLogo*. New York: Teachers College Press.

Klopfer, E., & Colella, V. (1999). *Structuring collaboration in workshops and classrooms: The StarLogo community of learners.* Paper presented at the Computer Supported Collaborative Learning Conference (CSCL), Palo Alto, CA.

Klopfer, E., & Colella, V. (2000). *Modeling for understanding*. Paper presented at the Society for Information Technology and Teacher Education (SITE) Conference, San Diego, CA.

The MathWorks [Computer software]. (1994). Natick, MA: MatLab.

Mikhak, B., Berg, R., Martin, F., Resnick, M., & Silverman, B. To Mindstorms and Beyond: Evolution of a Construction Kit for Magical Machines. *Robots for Kids: Exploring New Technologies for Learning Experiences.* (Edited by Allison Druin, published by Morgan Kaufman / Academic Press, San Francisco, March, 2000).

Resnick, M. (1994). *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* Cambridge, MA: MIT Press.

Resnick, M., Bruckman, A., & Martin, F. (1996, Sept./Oct). Pianos not stereos: Creating computational construction kits. *Interactions*, *3* (6).

Roberts, N., Anderson, D., Deal, R., Garet, M., & Shaffer, W. (1983). *Introduction to computer simulation: A system dynamics modeling approach.* Reading, MA: Addison-Wesley.

Roschelle, J. (1996). Learning by collaborating: Convergent conceptual change. In T. Koschmann (Ed.), *CSCL:Theory and Practice of an Emerging Paradigm* (pp. 209-248). Mahwah, NJ: Lawrence Erlbaum.

Roschelle, J., & Kaput, J. (1996). Educational software architecture and systemic impact: The promise of component software. *Journal of Educational Computing Research, 14* (3), 217-228.

Roughgarden J., Bergman A., Shafir S. and Taylor C. (1996) Adaptive Computation in Ecology and Evolution: A Guide for Future Research. pp 25 - 30, in *Adaptive Individuals in Evolving Populations: Models and Algorithms*, Belew R. K. and Mitchell M. eds., Proceedings Volume XXVI Santa Fe Institute Studies in the Science of Complexity, Addison-Wesley, Reading MA

Roy, B. (1998). Using agents to make and manage markets across a supply web. *Complexity*, *3* (4), 31-35.

Salthe, S. N. (1993). *Development and evolution: Complexity and change in biology.* Boston, MA: MIT Press.

Soloway, E., Pryor, A., Krajik, J., Jackson, S., Stratford, S. J., Wisnudel, M., & Klein, J.T. (1997). ScienceWare Model-It: Technology to support authentic science inquiry. *T.H.E. Journal*, *25* (3), 54-56.

White, B. (1993). ThinkerTools: Causal models, conceptual change, and science education. *Cognition and Instruction, 10,* 1-100.

Wilensky, U., & Resnick, M. (1999). Thinking in levels: A dynamic systems approach to making sense of the world. *Journal of Science Education and Technology*, *8*, (1), 3-19.