The Swarm Simulation System and Individual-based Modeling*

David Hiebeler

Santa Fe Institute

1399 Hyde Park Road

Santa Fe, NM 87501 USA

and

Applied Math Dept.

Harvard University

`hiebeler@santafe.edu`

September 12, 1994

Abstract

Swarm is a simulation environment which facilitates development and experimentation with simulations involving a large number of agents behaving and interacting within a dynamic environment. Many of the computer simulations performed by researchers in various disciplines rest upon a very similar computational foundation. These researchers often invest significant effort into developing the basic platform to support their simulations. A prototype of the Swarm simulation system was developed as an attempt to provide a general-purpose tool for building these simulations. The Swarm system provides a framework that will allow people to only write code describing the details of their specific problem, and then gain easy access to user-interface, simulation management, and analysis tools. This paper will provide an overview of the Swarm system and discuss some

*Presented at and to appear in the proceedings of *Decision Support 2001: Advanced Technology for Natural Resource Management*, Toronto, Sept 1994.

of the general advantages and disadvantages of using individual-based models.

1 Introduction

Computers are wonderful at turning good scientists into bad programmers.
-Chris Langton

Too many scientists have proven the above statement true. Physicists may spend weeks struggling to understand and implement the nuances of temporary variables for array classes in C++, economists may suffer the frustrations of building a basic user-interface, biologists may invest chunks of their time into writing basic code to manage a list of animals in their simulation, and so on. Many of the simulations that people of different disciplines are writing rest on similar foundations, and share many common tools. Swarm is a system which allows users to develop simulations involving potentially large numbers of agents interacting with each other within a dynamic environment. Swarm provides a framework to manage agents and the environment, as well as analysis tools for interactive experimentation or automated batch jobs. We began developing the Swarm simulation system with the following goals in mind:

- Provide a tool which can be used interactively or via batch jobs.
- Let researchers stop spending their time reinventing similar computer code to manage agents, and instead concentrate on their research.
- Make it easy to write agents and customize the space they exist in.
- Help people in different fields share a common language for describing their models.
- Make people more aware of assumptions underlying their models by documenting assumptions in the simulation environment and providing standard and repeatable options to choose from, such as the scheduling method used to update agents in the simulation.

From the very beginning, the development of Swarm has been driven by user applications. The initial prototype was developed "from the bottom up", by first writing a couple of simple experiments that we wanted our simulator to handle, and then writing a simulation system to support those experiments, rather than first developing the simulator and then trying to fit experiments to that design.

Some of the kinds of simulations to which Swarm is well-suited are:

- Economic models, with economic agents interacting with each other through a market[20, 9].
- Social insects building nests[1], foraging for food[18], or performing other actions [19, 8].
- Molecules interacting in an artificial chemistry[5, 6, 7].
- Traffic simulation[18, 15].
- Ecological simulations[10, 4, 14, 17, 16].
- Simulation games such as SimLife, SimCity, etc.[13].
- Artificial intelligence applications[3].
- General studies of complex systems, artificial life, cellular automata, emergent phenomena, etc.

Simple versions of several of the above kinds of systems have already been implemented within the current Swarm prototype.

2 Swarm

This section describes the current prototype of Swarm. The system is currently being substantially redesigned based on the experiences gained from this prototype and feedback from users of the system. Rather than wait and report on the next version, we felt that others may also benefit from this report of our initial experiences.

First, a brief note on terminology: throughout this paper, the term *agent* will refer to a user-defined entity in the simulation which is an actual part of the model; an *object* refers to any computational object in the system, which may not be part of the actual model being studied (for example, an analysis object gathering data). *Agent* is simply a more specific term meant to clarify the structure of the simulations; the Swarm system itself makes no distinction between agents and objects in general.

In the most general sense, Swarm is intended to provide an environment that facilitates the development of simulations involving a number of agents which exist within some (possibly dynamic) environment. This environment may be a regular spatial environment, a non-spatial environment such as a well-stirred soup, or something more abstract such as a telecommunications network. The agents communicate with each other and with the environment via messages. Swarm supplies a basic system library that manages a dynamic list of objects and handles message passing between objects. Each object contains some local data as well as specifications for how to respond to messages that are sent to it. Swarm also supplies a user interface (currently text- and X11-based UIs are provided), and analysis tools.

Many other people have also written simulation programs which fit this description. However, most of these systems have several key assumptions built into them, such as requiring that the "world" the experiments take place in is a two-dimensional lattice, that there are a limited number of types of agents or state-variables within each agent, and so on. Most of these limitations either do not exist in Swarm, or are contained within replaceable modules so that they may be changed if desired.

The main assumption within the current Swarm prototype is the concept of time. The current system uses a discrete time-stepped scheduling algorithm — on each time step, every object in the system receives a "step" message, directing the agent to perform some unit of computation. This scheduling mechanism could be easily modified in some ways, for example to perform asynchronous random updating of objects; the next version of Swarm will provide even more flexible scheduling algorithms.

Swarm is itself a swarm of objects with a very small kernel that handles initialization and invokes the user interface. Almost everything else is handled by objects themselves, some of which perform what are usually regarded as system operations. The current version is implemented in the C language using object-oriented methodology for many of the components such as method-lookup for messages.

System overhead in Swarm is fairly small. Performance measurements on a single-processor Sparc 10 workstation indicated that we can obtain approximately 15,000 object updates per second for fairly simple objects. Of course objects performing complex computations will make the simulation run more slowly, as will large spaces containing dynamic variables which must be updated on every time step.

An object in Swarm has three main characteristics: *Name*, *Data*, and *Rules*. An object's Name consists of a unique ID tag that is used to send messages to the object, a type (in the current implementation this is a character string), and a module name (which, together with the type, may be thought of as a class name). The Data are whatever local data the user wants to have in the agent (e.g. internal state variables). The Rules are a set of functions that handle any messages that are sent to the object, including the "step" message.

There are several special types of object within a Swarm simulation. With one exception, these objects all have uniform representation within the system; their special meaning is only apparent to the user, not to Swarm itself. The main types of special objects are regular *agents*, *space* objects, *analysis* objects, and the *object list manager*.

2.1 Agents

Agents are the objects written by the user. They represent entities that exist within the model, such as ants, plants, stock brokers, molecules, etc. When users write code for a new type of agent, there are several things that they must supply:

- A data structure containing the internal state-variables for an agent.
- A *step* function, invoked on every time step (this is optional; users can write agents that have no internal dynamics, but only respond to messages from other agents).
- Action functions that handle messages sent to the agent by other objects.

There are some conventions to be followed when writing agents. For example, for every type of agent a user writes, the class object should understand a "create" message, which requests that some agents of the given type be created. Agents should recognize a “destroy” message, which gets sent to agents to remove them from the simulation. There are also two special messages (“probe” and “set”) used for analysis and interaction, described below in section 2.3.

2.2 Space

Rather than building assumptions into Swarm about the type of environment agents would move around in, the notion of space was encapsulated within objects. Currently, almost all of our sample experiments use a two-dimensional lattice space, implemented within an object we call GridSpace2, although other types of spaces are possible, such as:

- GridSpaceN, a general N-dimensional lattice.
- SoupSpace, in which agents meet/collide randomly. This would correspond to non-spatial models which assume thorough mixing.
- GraphSpace, an arbitrarily-connected graph describing which spatial sites are “neighbors”.

Other spaces can be developed as necessary.

The current GridSpace2 has two main responsibilities: managing agents and managing spatial variables. One may think of the space as defining a geometry, with variables layered on top conforming to that geometry (those who have written data-parallel programs using e.g. the C* language will find this familiar), and with agents moving around in the space.

Managing agents The space keeps track of the locations of any agents that are located in the space. When an agent wishes to move to a new location, it sends a request to space indicating where it wants to move; space replies, telling the agent where it actually ended up. This allows the space to enforce different boundary types, such as a wrap-around torus or an impassable fixed boundary. Agents may also send queries to the space to find out what other agents are nearby within a specified distance. These queries may be restricted to only find agents of a specified type, if desired.

Managing spatial variables Spatial variables are variables that exist throughout the space. For example, in a simulation of ants moving within an environment, there may be a pheromone in the environment. The pheromone is represented by a lattice containing a concentration at every site in the array. This pheromone variable may have its own dynamics; for example it may have a diffusion operator applied to it on every time step. Other examples of spatial variables within a physical environment would be attributes such as temperature, humidity, light, soil nutrients, and so on. An arbitrary number of spatial variables may be created. Each one may be of type byte, integer, or double-precision floating point.

After the user creates a space object with a specified geometry, she may then request that it create a spatial variable within that geometry. The space object takes care of managing memory for the spatial variable. Agents may query space to obtain the value of a spatial variable at a given location (or within an entire region), and may send messages to set the value of the variable at a desired location. Spatial variables may be static or dynamic. A static variable only changes its value when an agent explicitly requests it. For a dynamic variable, on the other hand, the space regularly sends messages to another object requesting that it update the variable, e.g. by diffusing it. A dynamic spatial variable may be thought of as a cellular automaton, although somewhat more general.

Functional spatial variables are also allowed, where value lookup is performed via a function rather than by maintaining an array of values. An example where this would be useful is an ecological simulation with rainfall; the user can write a function *rainfall(x,y,time)* which is called for a specific position at a given time to determine the rainfall at that location at that time. The function may do some computations to determine what value to return, or it may for example be consulting an external database of rainfall measurements.

There may also be analysis objects attached to spatial variables, performing analysis of the variable on each time step.

Finally, space also talks to the image object if one exists, and tells it how to display agents and variables. The space object lets the user filter the variables to display, or display only certain types of objects, to focus on specific elements of the simulation or simply make the display less cluttered.

2.3 Analysis objects

Analysis objects are a crucial part of any simulation, as they fetch data from within the system and present it to the user or store it to disk. Several basic analysis objects have been implemented within the current Swarm prototype.

The most commonly used analysis object is the *Image* object. This object presents a two-dimensional image on the screen. Typically the image represents the values of spatial variable(s) as well as the positions of agents within a two-dimensional space. However, other aspects of the simulation may also be represented by sending the appropriate messages to the Image object. For most typical simulations on a two-dimensional lattice, GridSpace2 takes care of sending the necessary messages to Image in order to display a representation of the state of the simulation.

Probe and Set Several other analysis objects exist, which rely upon two standard messages that all agents in Swarm are required to recognize. These two messages are "probe" and "set". A "probe" message is used to retrieve some internal state variable(s) from an agent. A parameter string is also supplied with the message, indicating which state variable the sender wants, or the special string "all" to retrieve all variables. The "set" message is used to set an internal state variable in an object; parameters to the message are the name of the variable to set, and the new value to set it to.

There is a *Probe* object that uses these two special messages. When the user clicks on the representation of an agent in the Image window, a Probe object is created that probes all of the internal state variables of the desired agent, and pops up a window displaying the state of the agent. The Probe object has its own dynamics; on each time step it again probes the agent and updates the displayed variables. Making the Probe object be just another object has the advantage of making the window be self-updating, as well as making the design of Swarm simpler and more elegant. The user may also interactively change the values of variables in the Probe window, which causes the Probe object to send a "set" message back to the agent to update its internal state variables to the values the user specified.

Another analysis object that is built on top of the "probe" message is the *Graph* object. Graph simply probes numerical values from an object, and plots these values over time. Again, the Graph object is another regular object within Swarm, and updates itself on every time step.

Some analysis objects need to interact with more than just a single object. For example, in an ecological simulation, the user may want to know how many individuals of a particular species exist at each time step, or the average age of individuals in the population. Swarm supports sending messages to all objects of a given type, to facilitate gathering this kind of data. The current system contains an *Average* object that probes all objects of a given type and computes the average, minimum, and maximum of a numerical value over the entire population[1], as well as how many objects there are in the population.

Finally, since the analysis tools are simply regular Swarm objects, they may be layered on top of each other, using the same standard "probe" interface. For example, the user may have an Average object computing the average age of a population, and then attach a Graph object to the Average object, to display the resulting average on the screen. See figure 1 for a representation of this layering, and figure 2 for a picture of the current Swarm prototype as it appears on the screen.

Other simple analysis objects that currently exist are a state counter for counting how may sites in a spatial variable are in a particular state, and an object for measuring spatial entropy in a spatial variable. Obviously, many more analysis objects can be written.

2.4 Object List Manager

Swarm needs to keep a list of all objects in the simulation — regular agents, space object, analysis objects, and so on. All of the objects need to have a "step" message sent to them on every time step. Rather than putting code for maintaining this list into the kernel, it is encapsulated within an object called the *Object List Manager (OLM)*. The OLM is really the heart of Swarm; it keeps lists of all objects and valid object-types in the system, and handles registration of new objects and allocation of system wrappers for the objects, sending messages to populations, destruction of objects, and other system tasks.

The OLM could be implemented in many different ways. The current implementation uses linked lists organized by type. For example, all objects

[1]Actually, the user may provide an arbitrary function to be invoked on the vector of data, to compute things other than the average.

Figure 1: Layered analysis tools: An Average object probes the population of ants, while Probe, Graph, and File objects probe the Average object.

Figure 2: The Swarm screen, showing the control panel, an Image object, a Probe object, and some Graph objects.

of type “ant” are kept in one list, all objects of type “graph” are kept in another, and so on. This improves the efficiency of performing operations on all objects of a given type.

The OLM has also been implemented using dynamic arrays, which is more efficient for some types of experiments. Packaging the OLM into an object makes such a reimplementation much easier, and again makes the system more elegant.

The OLM is the only truly special object within Swarm. Because of its central role, it must be the first object created when the system is started up, and registers itself as the “root object” in Swarm. Several library functions in Swarm, such as the routine to register a new object, are actually just wrappers that send a message to the root object, which should be an OLM.

2.5 An example

As an example of the type of experiment which Swarm is well suited for, a set of simulations were performed to investigate extinction probabilities of competing predators for various world sizes. This model has three interacting species: one plant (prey) and two herbivores (predators). The herbivores compete for space and for food.

The plants reproduce and die according to life-history tables, where there is a given probability of reproducing and of dying at each age. When a plant reproduces, it drops the offspring at a random location within a specified dispersal radius of the parent. If the offspring is dropped on an occupied site, it does not survive due to crowding. Plants are sessile, but populations spread through the environment via dispersal of offspring. If the world is seeded with plants but no herbivores, the population follows the familiar pattern of logistic growth, eventually stabilizing at equilibrium (with a bit of noise).

The herbivores also reproduce and die according to life-history tables, but with the added complexity of motion and energy budgets. There is a metabolic cost for simply surviving, and energy is gained by feeding on plants (only plants within the herbivore's sensing radius may be consumed). If the energy level of an herbivore reaches zero the individual dies, regardless of its mortality probability in the life-history table. If the probability table suggests it is time for reproduction, the animal must also possess more than a minimum threshold of energy before it is allowed to reproduce. Offspring are again distributed randomly within a specified radius. Upon reproducing, the parent distributes a specified amount of its energy to the offspring. The offspring does not survive if it is dropped on a site occupied by another herbivore of either species, but will survive if dropped on a plant. Herbivores move in random walks, with a specified velocity and mean free path. For simplicity, their motion is not affected by the presence of plants nearby.

On each time step, the behavior of each plant is as follows:

1. Consult the probability table: is it time to reproduce? If so, drop an offspring randomly within my dispersal radius (note that the offspring will not survive if dropped on an occupied site).

2. Consult the probability table: is it time to die? If so, remove this individual from the simulation.

3. Increment the age of this plant. If it has reached the maximum age allowed for the species, remove the individual from the simulation.

The behavior of each herbivore is roughly as follows:

1. If it is time to turn (if I have walked farther than my mean free path since turning), then turn to face in a new (random) direction.

2. Move forward along the current heading.

3. If any plants are within the sensing radius of this herbivore, and the herbivore is not currently satiated, pick a random plant within range and eat it (destroy the plant and increment the herbivore's energy level).

4. Consult the probability table and energy level: is it time to reproduce? If so, produce an offspring.

5. Consult the probability table and energy level: is it time to die? If so, remove the individual from the simulation.

6. Increment the age of this herbivore. If it has reached the maximum age allowed for the species, remove the individual from the simulation.

All three species read in their parameters from files when being initialized. For the simulations described here, the two species of herbivores were functionally identical, using the same parameter file.

For these experiments, the world was initially populated as follows: $\frac{1}{64}$ of the sites had plants placed on them, $\frac{1}{128}$ of the sites had herbivore species 1 placed on them, and $\frac{1}{128}$ initially had herbivore species 2. The actual sites which were seeded were chosen randomly. The simulation was run for 2000 time steps on an $N \times N$ lattice for several values of N. For each value of N, the system was run 150 times with different random initial conditions. Averaging objects recorded how many individuals of each species there were throughout each run, and dumped the results to files.

When the simulations were done, results were gathered from the files and extinction probabilities for the two species of herbivores versus world size were measured. These results are shown in figure 3.

This is an example of the type of simulation for which Swarm is useful. The simulation was initially explored interactively through the graphical user interface. When the system was ready, the text interface of Swarm was used to perform large batch runs for gathering data. Only code for the plants and herbivores needed to be written; code for managing the agents was provided by Swarm.

In this example, all individuals of a given species had the same parameters, although each individual had a local copy these parameters. Another version of the simulation was written in which the herbivores were allowed to evolve their life-history tables through crossover and mutation. Selection of mating partners is done by querying the space to find another individual of the same species within a given radius; the tables of both parents are then combined to produce the offspring. Modifying the code to allow this evolution was not at all difficult or time-consuming.

2.6 The Future of Swarm

Swarm is currently being redesigned, based on our experiences with the current prototype. One of the most visible changes will be that the new version will be implemented using the Objective-C language. Objective-C is an object-oriented extension of the C language that is widely available as part of the GNU C compiler. This will allow inheritance in user objects, a feature sorely missing in the current prototype. For example, a generic "ant" object

Figure 3: Results from several runs of a three-species ecological simulation. Extinction probabilities were measured from 150 runs each for various lattice sizes.

may be written, and then more specific agents such as "worker ants" may inherit the generic ant code and specialize from there. Objective-C was chosen over C++ because it includes a comprehensive run-time system needed for dynamic interfaces within Swarm.

The internal organization of Swarm will become more flexible as well. A simulation will be able to contain multiple hierarchical swarms, which may interact with each other. Each swarm may contain its own scheduler for updating objects within that swarm. Data querying and collection will become more flexible and powerful as well. Details of the new Swarm will be written up later.

3 Individual-based modeling

The use of individual-based modeling in ecology has been growing rapidly in recent years (see [10, 11] for reviews). This growth is motivated by dissatisfaction primarily with two assumptions of traditional ecological models: that individuals are identical, and that every individual may affect every other. Individual-based models (IBMs) allow each agent to have its own set of internal state variables affected by its own history, and also allow spatial locality in the dynamics. The continual advance of computer technology has made implementing IBMs continually more feasible, as computers simultaneously become faster and cheaper. Usage of these models is also increasing in other fields (such as economics), where they are often referred to as agent-based models. In some economic models, spatial locality may not be relevant for example if agents are communicating through telephones; but the ability for different economic agents to use different trading strategies is crucial[12].

The two reasons stated above, individuality and spatial locality, are those most often given for turning to IBMs. The usual criticism of these models is the weakness of conclusions based on simulation as compared to analytical results of other models[2].

There are several other arguments for and against IBMs. Independent of a specific simulation framework such as Swarm, it is useful to discuss the advantages and disadvantages of individual-based modeling. Different researchers have their own reasons for pursuing (or criticizing) these models, but it may be helpful to present many of these reasons together in one place. Although the discussion centers primarily around ecological models, much of it should be relevant to other fields as well.

3.1 Why use individual-based modeling — the advantages

Probably the largest advantage of individual-based modeling for many people is the conceptual shift from mathematical descriptions of an entire ecosystem to rule-based specifications of the behavior of individual agents. This is beneficial both because the rule-based specifications are often probably closer to the mental model people have of the systems, and because the data from field studies may be more directly mapped onto agents' behavioral rules than onto system-level equations. Bringing theoretical ecologists and field ecologists closer together would be a great benefit of individual-based models.

Second, complex boundary conditions are typically much easier to implement in individual-based models than in other models such as partial differential equations. If one wishes to study an isolated ecosystem with a very irregular boundary geometry, one may simply "draw" the boundary in the environment. An irregular shape requires no more computational effort than a simple rectangular or circular boundary, in most cases.

Third, these models may be very suitable for implementation on massively-parallel machines, where agents are distributed across processors. In most cases, researchers wouldn't even need to modify their code or know anything about programming parallel machines in order to take advantage of this feature. Since an agent typically communicates with a few other agents and does some local computation, as long as the message-handler can dispatch the messages to the appropriate destination processor, the user's code doesn't need to know whether the agent will communicate with other agents on the same processor or another one. Some objects, such as the environment object, should be rewritten to take advantage of the parallel architecture, but the typical end user should not need to worry about this.

Fourth, individual-based models have the virtue of simply being different from traditional models, and therefore giving us another perspective on the systems being modeled. They may give us new insights into how local interactions produce large-scale phenomena, how significantly "historical accidents" affect system dynamics, etc. More specifically, they give us the ability to let a system develop multiple times from (perhaps only very slightly) differing initial conditions, to see which features of the dynamics are typical phenomena that do not strongly depend on the precise initial conditions.

Fifth, individual-based models may be more appropriate for studying small population models, where it's not feasible to approximate the population with a continuous distribution.

Finally, it may be argued that individual-based models are simply more realistic. Rather than assuming that all individuals are the same, the agents in the simulation may each have their own different histories, which have affected their internal state as well as having different conditions in their local environment.

3.2 Disadvantages of individual-based models

Individual-based models have their own set of problems as well. One problem that may often be encountered, depending on the complexity of the agents and the environment, is that it may only be possible to use relatively small population sizes or short time-scales, given the speed or memory constraints of current computers. If only small populations may be simulated, there may be pronounced "drift" effects; indeed, the phenomena to be studied may only occur with very large population sizes or over long periods of time. It should be noted that massively parallel processing may offer some relief from these constraints.

Second, it is easy to construct individual-based models with so many parameters that a reasonably thorough exploration of even a small fraction of the parameter space becomes infeasible.

Third, it may be difficult to consolidate the time-scales in the model, ranging from the time-scale of individual behaviors such as foraging to the time-scale of a single generation. It is too impractical to simulate time with a resolution of 5 minutes when one wants to examine population dynamics over several decades or even centuries. In this case, one may need to make the agents in the simulation represent entire groups or populations, rather than single individuals.

Fourth, these models may still leave us with very little understanding of how local mechanisms give rise to global dynamics. The models may have captured too much of the complexity of the real world, and be nearly as difficult to understand (see e.g. [2]). However, we do have the comfort of knowing that at least there are no "unknown" influences on the simulation —

e.g. no migration, environmental fluctuations, etc., unless they are put into the model (or unless these features are emergent properties from lower-level primitives).

Finally, many of the global dynamics observed may be artifacts from the model or its implementation, rather than true features of the real system being studied. They may also be very sensitive to implementation details such as the scheduling mechanism used to update the agents (see [14] for a very good discussion of this).

4 Conclusions

The current prototype has already been used by a handful of scientists from different disciplines for performing computer simulations. From discussions with these users and many others, we feel that there is a common architecture shared by many models and simulations; we designed Swarm to provide that architecture. The existence of a common environment for developing these experiments will facilitate interdisciplinary communication, allow researchers to avoid tedious computer work, and encourage researchers to carefully consider issues that they otherwise might have missed while developing their simulations.

Individual-based models have great potential for advancing our understanding of how behaviors at the level of the individual affect global dynamics, and perhaps for bringing theorists and field ecologists closer together. However, great care must be taken with these models, just as with any other class of models.

For more information about the availability of Swarm, send e-mail to "`swarm-request@santafe.edu`".

5 Acknowledgements

This research was partially supported by grants from Carol O'Donnell Grantham and Michael Grantham, by the National Science Foundation under grant PHY-9021437 "A Broad Research Program on the Sciences of Complexity", and by the Advanced Research Projects Agency under grant N00014-94-1-G014 "Simulation of Adaptive Complex Systems."

Chris Langton originally conceived of Swarm and has been involved during the entire process of development. Howard Gutowitz contributed to the design, as our primary user of the system. Roger Burkhart contributed greatly to the design of the next version of Swarm, as did Nelson Minar, who will be taking over primary development of the system. I am especially thankful to the staff of the Santa Fe Institute for their support, their hard work makes the Institute a wonderful place to work.

References

[1] E. Bonabeau, G. Theraulaz, E. Arpin, and E. Sardet. The building behavior of lattice swarms. In R. Brooks and P. Maes, editors, *Artificial Life IV*. MIT Press, 1994.

[2] H. Caswell and A. M. John. From the individual to the population in demographic models. In D. L. DeAngelis and L. J. Gross, editors, *Individual-Based Models and Approaches in Ecology*, pages 36–61. Chapman & Hall, 1992.

[3] *Communications of the ACM*: Special issue on 'Intelligent Agents', July 1994.

[4] D. L. DeAngelis and L. J. Gross, editors. *Individual-Based Models and Approaches in Ecology: Populations, Communities and Ecosystems.* Chapman & Hall, 1992.

[5] W. Fontana and L. W. Buss. Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 56(1), 1994.

[6] W. Fontana and L. W. Buss. What would be conserved if 'the tape were played twice'? *Proceedings of the National Academy of Science USA*, 91, January 1994.

[7] W. Fontana, G. Wagner, and L. W. Buss. Beyond digital naturalism. *Artificial Life*, 1, 1994.

[8] H. A. Gutowitz. Complexity-seeking ants. In Deneubourg and Goss, editors, *Proceedings of the Third European Conference on Artificial Life*, 1993.

[9] J. H. Holland and J. H. Miller. Artificial adaptive agents in economic theory. *American Economic Review, Papers and Proceedings*, 81:365–370, May 1991.

[10] M. Huston, D. DeAngelis, and W. Post. New computer models unify ecological theory. *BioScience*, 38(10):682–691, November 1988.

[11] O. P. Judson. The rise of the individual-based model in ecology. *TREE*, 9(1):9–14, January 1994.

[12] B. LeBaron. *personal communication.*

[13] Maxis. SimLife, SimCity, SimAnt, and other software products.

[14] E. McCauley, W. Wilson, and A. deRoos. Dynamics of age-structured and spatially structured predator-prey interactions: Individual-based models and population-level formulations. *American Naturalist*, 142(3):412–442, 1993.

[15] K. Nagel and S. Rasmussen. Traffic at the edge of chaos. In R. Brooks and P. Maes, editors, *Artificial Life IV.* MIT Press, 1994.

[16] S. Pacala, C. Canham, and J. Silander, Jr. Forest models defined by field measurements: I. the design of a northeastern forest simulator. *Canadian Journal of Forest Research*, 23:1980–1988, 1993.

[17] S. Pacala and J. Silander, Jr. Neighborhood models of plant population dynamics. i. single-species models of annuals. *American Naturalist*, 125(3):385–411, 1985.

[18] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* MIT Press, 1994.

[19] G. Theraulaz and J. Deneubourg. Swarm intelligence in social insects and the emergence of cultural swarm patterns. Technical Report 92-09-046, Santa Fe Institute, 1992.

[20] G. Weisbuch, H. A. Gutowitz, and G. Nguyen. Dynamics of economic choices involving pollution. Submitted to *Journal of Economic Behavior and Organization*, and published as Santa Fe Institute working paper 94-04-018, 1994.