an ordinary linear regression (Fielding & Gilbert, 2000). The model in this case is the regression equation, which computes the predicted values of the dependent variable. Similarly, one can measure the fit between the values of a variable that are output from a simulation model and the values observed empirically (this is, in fact, just the correlation coefficient between the two sets of values). However, this simple procedure makes a number of strong assumptions that, although often satisfied for linear regressions, are much less likely to be appropriate for simulation models.

One important characteristic of simulation models is that the values of output variables change as the simulation runs. For example, in a model of consumer behavior, the number of purchasers of a particular brand might be observed growing from zero to a majority during a simulation. The growth trend might then be compared with the growth in sales of an actual product. Because the data to be compared are time series, one must allow for the fact that there is autocorrelation: The value at time \( t + 1 \) is not independent of the value at time \( t \). Statistical procedures called ARIMA can be used to compare such time series (Chatfield, 2004).

3.5 Summary

In this chapter, a number of conceptual issues involved in designing and carrying out agent-based modeling research have been considered. We have shown by example the type of analysis that needs to be done before one begins programming and have mentioned some of the challenges that are raised by wanting one’s model to be verifiable and valid. The next chapter will move on to matters of implementation: how one can code a model; plan its development; and, finally, report its results.

4. DESIGNING AND DEVELOPING AGENT-BASED MODELS

4.1 Modeling Toolkits, Libraries, Languages, Frameworks, and Environments

Although some modelers build their agent-based models using only a conventional programming language (most frequently Java, although any
language could be used), this is a hard way to start. Over the years, it has become clear that many models involve the same or similar building blocks with only small variations. Rather than continually reinventing the wheel, commonly used elements have been assembled into libraries or frameworks that can be linked into an agent-based program. The first of these to be widely used was Swarm (http://www.swarm.org/), and although this is now more or less completely superseded, its design has influenced more modern libraries, such as Repast (http://repast.sourceforge.net/) and Mason (http://cs.gmu.edu/~eclab/projects/mason/). Both of the latter are written in Java and so link most easily to models that are also written in Java, but they can be used with other languages. Repast is available in a version for .NET and can be linked easily to programs in C#, Visual Basic, and Python. Both provide a similar range of features, such as the following:

- A variety of helpful example models
- A sophisticated scheduler for event-driven simulations
- A number of tools for visualizing on screen the models and the spaces in which the agents move
- Tools for collecting results to a file for later statistical analysis
- Ways to specify the parameters of the model and to change them while the model is running
- Support for network models (managing the links between agents)
- Links from the model to geographical information systems (GISs) so that the environment can be modeled on real landscapes.
- A range of debugged algorithms for evolutionary computation (Section 5.2.2), the generation of random numbers, and the implementation of neural networks.

Many person-years of effort have gone into the construction of these libraries, and using them can greatly reduce the time taken to develop a model and the chance of making errors. Both Repast and Mason are open-source software, available free for noncommercial use. Their only disadvantages are, first, their complexity, which means that it can take some months before one can take full advantage of the wide range of features they offer; and second, that the modeler is expected to use a relatively low-level language such as Java to develop his or her model.

More suited to the beginner are modeling environments that provide complete systems in which models can be created and executed, and the results visualized, without leaving the system. Such environments tend to be much easier to learn, and the time taken until one has a working model can be much shorter than it would be if one were using the library approach. However, the simplicity comes at the price of less flexibility and slower
speed of execution. It is worth investing time to learn how to use a library-based framework if you need the greater power and flexibility they provide, but often simulation environments are all that is needed.

Environments primarily intended for other purposes can also be used for simulation, sometimes quite effectively. For example, simple simulations can be created using the spreadsheet package Microsoft Excel, and the free, open source statistics package R (http://www.r-project.org/) can be useful for models that involve processing large amounts of data. Several significant agent-based models have been constructed using the mathematical packages MatLab (e.g., Thorngate, 2000) and Mathematica (e.g., Gaylord & D’Andria, 1998). Nevertheless, an environment designed specifically for agent-based modeling is usually the first choice.

4.1.1 Repast

Repast (North, Collier, & Vos, 2006) is a family of two libraries, one for Java and one for Microsoft’s .NET, and a more visual tool that allows developing simulations using the scripting language Python. The two libraries are functionally identical, and the choice of which one to use should depend only on your choice of programming language. The Python system is closer to a simulation environment and needs less advanced programming skills.

Repast has an active user base—from academia, government, and business—and a mailing list that is very helpful if you have questions about how to use it. The Java version will run on almost any platform, including Windows, Mac OS X, and Linux.

4.1.2 Mason

Mason (Luke, Cioffi-Revilla, Panait, Sullivan, & Balan, 2005) is another Java library, also influenced greatly by Swarm. It offers both standard and 3-D visualization libraries, and the user can record movies of the simulation as it runs (both of these features are also available in NetLogo; see below). Mason is free and open source.

4.1.3 NetLogo

Currently, the most popular agent-based simulation environment is NetLogo (Wilensky, 1999). It includes a user interface builder and other tools such as a system dynamics modeler. It is available for use free of charge for educational and research purposes and can be downloaded from http://ccl.northwestern.edu/netlogo/. It will run on all common operating systems: Windows, Mac OS X, and Linux. NetLogo has an active user community that answers users’ questions quickly and thoroughly, and has users
in all levels of education and in the natural as well as the social sciences.
Other environments include StarLogo (http://education.mit.edu/starlogo/) and AgentSheets (http://agentsheets.com/), but these are more suited to creating and demonstrating very simple models for teaching than building simulations for research.

4.1.4 Comparison

Table 4.1 compares Swarm, Repast, Mason, and NetLogo on a number of criteria, using admittedly subjective judgments. None is ideal for all uses. To choose between them, one needs to consider one’s own expertise and experience in programming, the likely complexity of the model, and the aims of the project (e.g., is the project very exploratory and the model likely to be fairly simple, or does the project intend to build a relatively complicated model and exhaustively test its behavior against data?). Repast, Mason, and NetLogo continue to be developed and are evolving quite rapidly, so the information in Table 4.1 needs to be checked against the current state of each of the systems. Other comparisons and reviews can be found in Castle and Crooks (2006), Gilbert and Bankes (2002), Railsback, Lytinen, and Jackson (2006), and Tobias and Hofmann (2004).

NetLogo stands out as the quickest to learn and the easiest to use, but may not be the most suitable for large and complex models. Mason, which is comparable to, but often faster than, Repast, has the advantage of being the newest, drawing on the experience of the older systems, but also has a significantly smaller user base, meaning that there is less of a community that can provide advice and support.

4.2 Using NetLogo to Build Models

In the remainder of this book, we shall use the agent-based simulation environment NetLogo (Wilensky, 1999). NetLogo, like the other environments and libraries mentioned above, is undergoing continuous development, with a major new version appearing more than annually. Hence, the code printed in this book (which was developed using version 3.1) may need modification to run in future versions. However, Wilensky and his team strive to make changes to NetLogo upwards compatible, so any changes needed may be made automatically when you load the code, or may require only minor editing.

The NetLogo system presents the user with three tabs: the Interface tab, the Information tab, and the Procedures tab. The Interface tab is used to visualize the output of the simulation and to control it (see Figure 4.1), the Information tab provides text-based documentation of what the simulation
is for and what should be observed, and the Procedures tab is where one writes the simulation program using a special language specific to this environment (the NetLogo language). NetLogo is based on the programming language Logo (Papert, 1980), which was designed for teaching young children about the concepts of procedures and algorithms and was originally used to control small toy robots called “turtles.” In memory of this, NetLogo’s agents are still called turtles.

The Interface tab includes a black square called the view, which is made up of a grid of patches. This is the spatial environment in which the agents move: A simulation program can instruct agents to move in any direction from patch to patch, and the agents will be visible on the view (see, for

<table>
<thead>
<tr>
<th>TABLE 4.1</th>
<th>A Comparison of Swarm, Repast, Mason, and NetLogo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Swarm</td>
</tr>
<tr>
<td>License*</td>
<td>GPL</td>
</tr>
<tr>
<td>Documentation</td>
<td>Patchy</td>
</tr>
<tr>
<td>User Base</td>
<td>Diminishing</td>
</tr>
<tr>
<td>Modeling Language(s)</td>
<td>Objective-C, Java</td>
</tr>
<tr>
<td>Speed of Execution</td>
<td>Moderate</td>
</tr>
<tr>
<td>Support for Graphical User Interface Development</td>
<td>Limited</td>
</tr>
<tr>
<td>Built-in Ability to Create Movies and Animations</td>
<td>No</td>
</tr>
<tr>
<td>Support for Systematic Experimentation</td>
<td>Some</td>
</tr>
<tr>
<td>Ease of Learning and Programming</td>
<td>Poor</td>
</tr>
<tr>
<td>Ease of Installation</td>
<td>Poor</td>
</tr>
<tr>
<td>Link to Geographical Information System</td>
<td>No</td>
</tr>
</tbody>
</table>

example, Figure 4.3; the small triangles on the view are agents). Usually, the NetLogo environment is configured so that the left-hand edge joins on to the right-hand one and the top edge to the bottom, so that if an agent moves off the left-hand side of the view, it immediately reappears at the right-hand side (the environment is topologically equivalent to the surface of a toroid, a donut-shaped solid). Patches start colored black but can easily be recolored, so that, for example, one could create a contour map. The number of patches in the view can also be configured: When NetLogo starts, the view consists of 35 by 35 patches, but the number can be increased to many thousands.

A NetLogo program has three parts. First, there is a section that says what kinds of agents there will be and names the variables that will be available to all agents (the global variables). Second, there is a setup procedure that initializes the simulation. Third, there is a go procedure, which is repeatedly executed by the system in order to run the simulation. Figure 4.2 shows a very simple example to give a flavor of a NetLogo program in which 10 agents are created and move randomly around indefinitely.

In this program, there are no global variables, so the program starts with the setup procedure. Any turtles left from a preceding run are cleared away, and
10 new turtles are created (these are placed in the center of the NetLogo view). The go procedure tells each turtle (agent) to carry out the commands within square brackets: first, turn to the right (i.e., clockwise) by a random number of degrees, and then move one unit forward, where the unit is the length of the side of a patch. Each turtle moves independently of the others, all at the same time (because NetLogo runs on an ordinary computer, the agents cannot all operate at precisely the same time, but NetLogo makes it look as though they do by using an asynchronous random update; see Section 2.3).

To run this program on your own computer, you would need to download and start up NetLogo. Then click on the Procedures tab and type in the lines of code shown in Figure 4.2. Move back to the Interface tab. Click on the Button icon at the top and then on the white area next to the view. NetLogo will draw a button that you should label “Setup.” Then do the same for a Go button, also setting the check box Forever on (this will cause the go procedure to be executed in a loop when the Go button is clicked, continuing until the button is clicked a second time to stop the run). Clicking on your Setup button will create 10 turtles, shown piled on top of each other in the center of the view. Clicking on the Go button will send the agent turtles darting around the view, following a random trajectory. Click on the Go button again to stop the program. The NetLogo interface should then look similar to Figure 4.3.

Although this is a very simple example, it does give an idea of how quickly one can develop agent-based simulations in an environment like NetLogo. The graphics for buttons (and sliders, switches, etc.) to control a simulation are available through “drag and drop.” The view offers many possibilities for the visualization of agents and their environment without doing any programming. NetLogo will also show dynamically changing plots of output variables on the Interface tab. Although the NetLogo programming language is somewhat different from the usual procedural
languages, it is both powerful and (mostly) elegant, with the result that complex simulations can be programmed in surprisingly few lines of code.

There is no space in this book to provide a detailed tutorial on NetLogo, but the system includes a good built-in tutorial (find it under the Help menu) and comes with a large number of demonstration and example models, some of which are relevant to social science.

4.3 Building the Collectivities Model Step by Step

In Chapter 3, the “collectivities” model was introduced. In brief, this is a model that simulates the dynamic creation and maintenance of knowledge-based formations such as communities of scientists, fashion movements, and subcultures. The model’s environment is a spatial one, representing not geographical space, but a “knowledge space” in which each point is a different collection of knowledge elements. Agents moving through this space represent people’s differing and changing knowledge and beliefs. The agents have only very simple behaviors: If they are “lonely,” that is, far from a
local concentration of agents, they move toward the crowd; if they are crowded, they move away.

Thus, formally, there are two agent behavior rules:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The agent is lonely</td>
<td>Move toward the crowd</td>
</tr>
<tr>
<td>The agent is crowded</td>
<td>Move away from the crowd</td>
</tr>
</tbody>
</table>

The first step in building the model is to make some basic decisions about the agents and the environment. The model specification implies that there will be only one type of agent and that the agents will move about in a space. We need to decide the dimensionality of this space: For simplicity, we shall use a two-dimensional grid that can be mapped directly onto NetLogo’s view. To avoid special effects that might occur at the edges of the grid, we shall use a toroid, which has no edges. This is the default arrangement for NetLogo, so nothing extra is required.

Next, it is helpful to lay out the logic of the model, either graphically or in “pseudo-code.” To show the logic graphically, it is convenient to use the Unified Modeling Language (UML), a means of representing programs that has been developed as a way of communicating software independently of the details of programming languages (Miles, 2006). UML provides a range of standardized diagrams that can be used to show the class hierarchy of the objects in the program; a sequence diagram that shows how one thing leads to the next; and an activity diagram, which is similar to a flowchart. UML is very good for describing a model in, for example, a published paper, but the collectivities model is so simple that UML is hardly necessary. For instance, there is only one class (for the agents) and only two agent actions (move forward and turn around).

With such a simple model, an alternative approach is more helpful: to use “pseudo-code,” an informal mixture of natural language and programming conventions that makes the structure and flow of a program clear without requiring the reader to be familiar with any particular programming language. Figure 4.4 shows the collectivities program in pseudo-code.

The program is in two parts: the initialization (called “Setup” in NetLogo) and the execution (“Go” in NetLogo). The indentation of the pseudo-code helps to clarify which lines go with which. For example, the program loops repeatedly carrying out the lines between Loop forever and End loop. Constant parameters of the model are shown in italics.

Once one has a pseudo-code version of the program, it is relatively easy to translate it into a programming language such as NetLogo, and the results of doing so are shown in Figure 4.5. Figure 4.6 is a screen shot of the
**Initialization**

Create agents and distribute them at random in knowledge space

**Execution**

Loop forever

Each agent:

Counts the number of other agents within its local-radius

Each agent:

Comparer the number of other agents within its local-radius with the threshold

If the number is greater than the threshold

Then (the agent is crowded)

The agent locates that agent within visible-radius with the most agents surrounding it

The agent moves a distance proportional to speed away from this agent

Else (the agent is lonely)

The agent locates the agent within visible-radius with the most agents surrounding it

The agent moves a distance proportional to speed towards this agent

End loop

**Figure 4.4** The Collectivities Program in Pseudo-Code

program code in the NetLogo environment, and Figure 4.7 shows the user interface with buttons to set up the simulation (pressing this button executes the initialization code, from setup to the end statement 10 lines further on) and to run the simulation (Go, which executes the go procedure repeatedly until the button is pressed again).

**4.3.1 Commentary on the Program**

In this section, each line of the program in Figure 4.5 is explained.

**breed [agents agent]**

This line names the class that will be used in the model, in both its plural and singular forms.

**agents-own [ around visible ]**
breed[agents agent]
agents-own[ around visible ]
to setup
  clear-all
  ask patches[ set pcolor white ]
  create-custom-agents 500 [ 
    set color green
    set size 2
  ]
  ; distribute agents randomly
  setxy random-pxcor random-pycor
  set heading random 360
  ; ensure that each is on its own patch
  while[ any? other-agents-here ] [ fd 1 ]
end
to go
  ask agents[ count-those-around ]
  ask agents[ move ]
end

; store the number of agents surrounding me within 
; local-radius units 
; and the agents that I can see within visible-radius 
to count-those-around
  set around count agents with[ self != myself ] in-radius local-radius
  set visible agents with[ self != myself ] in-radius visible-radius
end
to move
  if any? visible [
    ifelse around > threshold [ ; if more than threshold agents surround me, 
      I’m crowded: 
      ; face away from the most popular patch and 
      become red
      face-away set color red ]
    [ ; else I’m lonely: face towards the most popular spot and 
      become green
      face-towards
      set color green ]
  ]
  ; and move in my (new) direction
fd random speed
; ensuring that I stop on an empty patch
   while[any? other-agents-here] [ fd 1 ]
end

; face towards the most popular local spot
to face-towards
   face max-one-of visible [around]
end

; face away from the most popular local spot
to face-away
   set heading towards max-one-of visible[around] - 180
end

Figure 4.5 The Collectivities Program

Figure 4.6 The NetLogo Procedures Tab Showing Part of the Collectivities Program
The agent class has two attributes, around and visible. Every agent has its own values for these variables, which store the number of other agents surrounding the agent and the set of agents that it can see, respectively.

```
to setup
  clear-all
  ask patches [set pcolor white]
```

The setup procedure is executed when the user presses the Setup button on the interface. The procedure first deletes any agents left over from a previous run and then colors all the patches on the grid white. The code `ask patches[...]` tells all the patches (recall that a patch is a cell on the grid) to execute the code inside the square brackets. The term `pcolor` is a NetLogo variable that sets the color of a patch.

```
create-custom-agents 500 [
  set color green
  set size 2
]
Then 500 agents are created. The commands inside the square brackets, are executed by each of the new agents independently. The image of each agent on the view (see Figure 4.7) is colored green, and its size is set to twice the minimum.

```netlogo
setxy random-pxcor random-pycor
set heading random 360
```

Then each agent is moved to a random spot on the grid (random-pxcor yields a number corresponding to a random position between the left and right edges of the grid, and random-pycor does the same for a position between the top and bottom edges), and the heading (the direction that the agent faces) is also set to a random direction.

```netlogo
while [any? other-agents-here] [ fd 1 ]
```

It is possible that the result of the random distribution of agents will be that two or more agents end up on the same patch, one standing on the head of another, so this line gets each agent to check whether there are any other agents in the same patch. If there are, the agent advances by one unit (fd 1) in the direction of its current heading and checks again, continuing to advance until it has found an unoccupied patch.

```netlogo
end
```

This completes the commands that are executed by each agent immediately after it is created, and also brings the setup procedure to an end.

```netlogo
to go
    ask agents [ count-those-around ]
    ask [ move ]
end
```

The go procedure gets executed repeatedly when the Go button on the interface is pressed. Each time through the procedure, the simulation moves on one time step. At each step, all the agents are asked to count the number of agents that are in their local area on the grid, and then all the agents are asked to move. It is important that all agents complete counting the agents around them before any agents move; otherwise, the counts may be confused by agents moving before they have been counted. NetLogo guarantees that all agents complete the commands in a construction such as ask agents [...] before the next command (in this case, the second ask agents [...] is started. This would not have been the case if the program had been written ask agents [count-those-around move] because then each agent would have started to move as soon as it had finished counting, and for some agents, this may have occurred before
other agents had added them to their counts. This is an example of the kind of timing problem that one needs to guard against when programming agents that appear to act simultaneously.

The count-those-around and move procedures are both methods of the agent class, that is, procedures that are defined in the program code to show what it means for agents to count and move. The next lines specify what count-those-around consists of.

```plaintext
to count-those-around
    set around count agents with [self != myself]
    in-radius local-radius
    set visible agents with [self != myself] in-radius visible-radius
end
```

In the first line of this procedure, the variable around, owned by each agent, is set to the number of agents that are located within a radius of local-radius. The user sets the value of local-radius before the simulation starts by moving a slider on the interface (see Figure 4.7). It is easiest to see what this line of code does by working backwards from the end. The code in-radius local-radius yields those agents that are located at or within a distance of local-radius away from the agent executing this procedure. This set of agents includes the agent itself (it is zero units away from itself, and therefore closer than local-radius). The code with [self != myself] excludes that agents, and count returns the number of agents remaining. The agent’s attribute around is set to this number. One can think of the agent looking around, counting the number of agents it sees within the local-radius and remembering this in the around variable. In the terminology of production systems (see Section 2.1.2), around is part of the working memory.

The next line of code is very similar. It finds which other agents can be seen by the agent, that is, which agents are within the visible-radius, a parameter whose value is set by another slider on the interface. The second line is slightly different from the first because the variable, visible, stores not a count of the visible agents but the names of the agents themselves. This is necessary so that they can later be identified (in procedures face-towards and face-away).

The move procedure both decides the direction in which the agent is to move and performs the action. In terms of production systems, the move procedure is a very simple rule interpreter.

```plaintext
to move
    if any? visible[
```
The procedure begins by checking whether there are any agents visible to the agent executing this method. If there are, the agent needs to decide whether it is “lonely” or “crowded,” that is, it needs to decide whether either of the condition-action rules apply.

```netlogo
ifelse around > threshold
  [ face-away
    set color red ]
```

The condition consists of checking whether the number of agents around this agent is greater than the `threshold` parameter set with a slider on the user interface. If it is, it is “crowded” and the agent must turn away from the center of the crowd. The procedure `face-away` does this. In addition, to show these agents on the view, their color is changed to red.

```netlogo
[ face-towards
  set color green ]
```

The alternative is that the agents are “lonely,” and they must turn toward the crowd and become green.

```netlogo
fd random speed
while [any? other-agents-here] [ fd 1 ]
end
```

Once the agent has set its direction, it can move, at a rate determined by the value of the `speed` parameter (`random speed` returns a random number less than the value of `speed`). As before, if the agent lands on a patch that is already occupied by another agent, it continues to move forward in the same direction until it finds an empty patch.

That concludes the `move` procedure. We still have to define `face-towards` and `face-away`.

```netlogo
to face-towards
  face max-one-of visible [around]
end
```

The `face-towards` command retrieves the set of agents that this agent can see (the agents that have been remembered in the `visible` variable) and finds how many agents surround each of those agents. One of the agents with the most agents around it (`max-one-of visible [around]`) is considered to be at the center of the crowd toward which this agent wants to move. The NetLogo command `face` sets the heading of the agent to the direction of that central agent.
The to face-away command is almost the same, but the direction is 180 degrees opposite to the direction of the most central agent.

That concludes the program code for the collectivities model.

Running the model shows that the initial uniform random distribution of agents separates into “clumps,” in which some agents are central and others are distributed around them. The central agents are crowded, and so move. In doing so, they shift the centroid of the clump slightly and may make other agents either crowded or lonely, and they too will move. Thus, the clump of agents, although remaining together for long durations (as measured in time steps), drifts across the view. Lonely agents move toward the clump, sometimes joining it and sometimes continuing to trail behind it. The clumps never merge.

Figure 4.8 illustrates a typical snapshot. In this figure, agents that are crowded are a dark grey and those that are lonely are a lighter grey.

Comparing the behavior of this model with the features of collectivities described in Section 3.1, we see the following:

1. When we run the model, we see “clumps,” but drawing a boundary around clumps involves some arbitrary definition, perhaps in terms of local densities of agents.

2. Although a definition of which agents are in and which are out of a clump is possible (e.g., in terms of the distance to the nearest neighbor), again this seems arbitrary.

3. Agents in the same clump are close together and so could be thought of as sharing some aspects of their knowledge.

4. The location of the clump, as indicated by the position of its centroid, is constantly changing as some agents move more closely into the clump and others seek new, less crowded locations.

5. Some agents consider themselves to be crowded, and these behave differently from the other agents in the clump (by “innovating” or trying to find less crowded positions by moving through the knowledge space). These agents are located more centrally in the clumps and are influential in setting the direction of travel of the other agents.

The features of collectivities that we observe in society thus emerge in the model as a result of the behavior of the agents. Although other micro-level...
actions could produce the same or similar macro-level patterns (Gilbert, 2002), it is useful to know that these do yield the macro-level behavior that we observe. Specifically, we can conclude from the model that if

1. agents change their ideas in knowledge-space in response to “over-crowding” (Mulkay & Turner, 1971),

2. some ideas and some agents are considered to be high status or important, and

3. agents are motivated to copy and adopt those ideas or a variation on them,

then the phenomenon we have described as a “collectivity” will emerge from the agents’ behavior.

A more detailed comparison of the model output with empirical data is not appropriate for this abstract model (see Section 3.3.1). The value of abstract models is twofold: They can account for the generation of

Figure 4.8  Snapshot of the Simulation
particular phenomena, following Epstein’s maxim that “to explain macroscopic social patterns, we generate—or ‘grow’—them in agent models” (Epstein, 2007, p. 50; see also Epstein, 1999); and they can help to highlight commonalities and differences between phenomena that otherwise might be considered incomparable. A better criterion for this model’s success is therefore the degree to which it generates further theoretical questions or informs middle range theory that can be empirically validated. For example, the model suggests the question, What are the significant similarities and differences between “punks” and “scientists,” given that their social formations can be re-created using the same generic model? It does seem that there are many areas of social life where similar micro-behaviors may be found and correspondingly many emergent collectivities, and so this model may be applied to account for a wide range of social phenomena.

4.4 Planning an Agent-Based Modeling Project

As with any research project, it is helpful to plan a simulation project step by step in advance. Then you can be more confident that what you plan is likely to be achievable, and remedial action can be taken if it becomes clear that you are falling behind schedule. Although most simulation projects are not different in their essentials from projects using other styles of research, there are some special features that need attention.

- **Do Not Underestimate How Long It Takes.** It is tempting to think only about the time spent writing the code, but it often takes as long to design a model as to code it, and frequently longer to debug a program than to write it. Therefore, it is not being too pessimistic if one estimates the time needed to write a program and then multiplies this by at least three to give the total time for model development. Unless one is using a modeling environment such as NetLogo, most of the programming will be taken up with the development of the user interface and output display routines, not with coding the model itself. This is one reason why modeling environments are so valuable—they can save a great deal of work.

- **Keep a Diary.** Ideas will occur to you at all stages of the project, and you risk forgetting them unless you jot them down in a diary or lab book. Pay special attention to problems that you encountered while building the model: Difficulties that you initially assume are just technical programming problems may turn out to have a wider significance. For example, if it seems that the results of the simulation are very sensitive to the particular value of a parameter, this may just be an issue in building the model, but it may also
suggest some substantive conclusions about the role of this parameter in the real world.

There are some additional points that need consideration in larger projects, where there are several researchers working in a team.

- **Find People With Appropriate Skills.** If you are a lone researcher, you will know the extent to which you are already skilled in programming models. If the project is a larger one in which there is some division of labor, you will probably need to recruit people with expertise in modeling. Because agent-based simulation is still a new approach, researchers with significant experience are hard to find, and you may need to be content with hiring people with other skills and training them in agent-based modeling. Particularly useful skills are a familiarity with programming in Java (even if you are not intending to use one of the Java-based libraries—see Section 4.1—the grounding in programming that a Java course gives is very useful); some experience in researching in the domain to be modeled; and the ability to write clearly, which is vital for preparing reports and papers.

- **Attend to Intraproject Communication.** If more than one person is working on a project, attention needs to be paid to making sure that everyone understands each other and is aware of what the others are doing. Although this is true in all team tasks, in modeling projects, it is usual to have some people who are domain experts but know little about modeling, and others who are modeling experts but know relatively little about the domain. Both sides may feel inhibited about asking questions and exposing their ignorance to the others. In larger projects, it may be worth scheduling specific training sessions, where those with greater knowledge of particular aspects of the project teach the others in order to bring everyone up to a common level of knowledge and skill.

- **Pay Attention to Scheduling.** Most modeling projects involve some data collection and some model development. This can be tricky to schedule if the specification of the model awaits the collection and analysis of empirical data, but the collection of data rests on the prior definition of precisely what is to be measured. Unless care is taken, one can get into a Catch-22 situation, in which neither modelers nor data collectors can make a move.

### 4.5 Reporting Agent-Based Model Research

The best way of learning how to report the results of agent-based modeling is to study how others have done it. Take a sample of papers that you have
found helpful or interesting and look closely at how the authors constructed them and what makes them persuasive. Although agent-based modeling is too young an area to have a well-developed set of conventions about how papers should be written, there are some common elements (Axelrod, 1997a). A helpful discussion can be found in Richiardi, Leombruni, Saam, and Sonnessa (2006).

The main sections of an agent-based modeling report or journal article are usually as follows:

1. An abstract. This should indicate (in roughly this order)
   a. The main research question considered in the paper
   b. The findings and conclusions of the paper
   c. The methods used (e.g., agent-based modeling; survey analysis)
      and, for empirical data, the sample from which data were collected

2. An introduction that sets out the background to the issue addressed in the paper and explains why it is of interest.

3. A literature review that discusses previous work and shows why the research reported in this paper is a worthwhile addition or improvement to the prior work. Both literature on the research problem or domain, and on related models, even if these have not previously been applied to the domain, should be reviewed. This section should make clear in which respects the reported research is an advance and how it is using previous work.

4. A statement of the regularities that you want to explain (this will usually be a summary of the material in the introduction and review). These may be stated as a set of formal hypotheses that you aim to (dis)prove, or they may be presented less formally.

5. A description of the model. The description needs to be sufficiently detailed that a reader could, in principle, reimplement your model and obtain the same results, but it should not include program code (some readers will not know or understand the programming language you have used). Instead, use diagrams (e.g., UML) or pseudo-code to describe your model (see Section 4.3). Pay particular attention to the sequence in which events occur in the model: This is the most frequent source of problems in reimplementing a model accurately. Do not be afraid of including equations relating variables if these will help to specify your model precisely.

6. A description of the parameters. The values you have chosen for each of the parameters need to be explained and justified. Some may be
based on observations of the social world (e.g., the employment rate in a model of the labor market); some may be plausible guesses, and you have investigated the effect of varying their values using a sensitivity analysis; and some may have been inferred “backwards,” because it is only these values that give the patterns of output that you want to demonstrate with the simulation. All this needs to be explained.

7. A description of the results. This will almost certainly involve presenting and commenting on graphs that show how variables that you have observed from the simulation runs are related. Be careful to be clear about the conditions under which these simulations were carried out. For example, do the plots show averages of several runs, and if so, how many runs and how much variation was there between runs (you might consider using error bars to show the degree of variation)? If you are showing the trend in a variable over time from step zero onward, make sure that you have plotted a run long enough that it is clear that the trend has become established and is unlikely to change drastically just off the graph (Galan & Izquierdo, 2005). If you are relating the values of variables as they are at a particular time step, make sure that you state the time step at which the measurements were made.

8. A discussion of what steps you took to verify (see Section 3.2) and validate (see Section 3.3) the model, and what confidence the reader should therefore place in your results.

9. A conclusion. This should take the hypotheses listed in (4) and clearly state whether the model suggests that they are true, false, or not proven. This section can then develop the ideas from the introduction, proposing a general conclusion and perhaps speculating about the implications (e.g., if the paper is about the labor market, what policies might or might not be successful in reducing rates of unemployment).

10. Acknowledgments. Brief thanks to sponsors, funders, and those who have helped you to do the research.

11. A list of references containing only those works cited in the paper and no others. As always, you need to be sure to provide full bibliographic details in the format required by the journal in which you hope to publish.

12. Optionally, an appendix in which large tables and possibly the pseudo-code version of your model is placed.
4.6 Summary

This chapter has described the process of implementing an agent-based model, from choosing a toolkit to reporting on the results. The most important step is to plan your work before you get too engaged with it. This will be helpful in ensuring that you have a well-defined research question to answer, and that you have allowed sufficient time and resources to be able to answer it.

5. ADVANCES IN AGENT-BASED MODELING

Agent-based modeling is a rapidly advancing field, and new approaches are being introduced all the time. In this chapter, we shall briefly describe some of the directions of current research.

5.1 Geographical Information Systems

Most agent-based models involve agents moving over a landscape, but usually this terrain is a rectangular plane or a toroid. In place of these abstract surfaces, newer simulation environments are offering the possibility of creating complex artificial surfaces, or incorporating terrains mapped from real landscapes. They do this by integrating a geographical information system (GIS) into the model. A GIS is a software system for managing, storing, and displaying spatial data such as maps (Chang, 2004; Heywood, Cornelius, & Carver, 2002). GIS study is a research specialty in its own right, and this is one of the reasons why using a GIS for agent-based modeling has been slow to develop: GISs have their own software technologies and techniques that have been somewhat difficult to merge with the tools of agent-based modeling.

GISs store spatial data in specially built or adapted “spatially aware” database systems. These are designed to be efficient at answering the kind of queries that one needs for managing geographical data, such as “tell me about all the other objects that are no more than 10 units away from this object.” GIS data are often arranged in layers, containing data on one or a few variables. When displaying or manipulating a map, one can turn some layers on or off, to see just the variables in the visible layers. For example, a map might include roads and lakes in separate layers. If one wanted to see