Overcoming design and development challenges in agent-based modeling using ASCAPE

Mario E. Inchiosa* and Miles T. Parker*

BiosGroup, Inc., 317 Paseo de Peralta, Santa Fe, NM 87501

The ASCAPE agent-based modeling environment greatly eases the task of designing and investigating agent-based models. However, effective design can still require a relatively deep knowledge of programming and agent-based modeling. This issue has almost certainly slowed the adoption of agent-based modeling approaches to social science problems. We will discuss how the ASCAPE software technology can mitigate this requirement and significantly benefit modelers regardless of programming ability.

ASCAPE is a framework for developing and exploring sophisticated agent-based models. The name ASCAPE introduces two fundamental concepts behind the framework: agents and "scapes." The term "scape" comes from Sugarscape (1), where it refers to a territory on which agents live. In ASCAPE (the ASCAPE framework and documentation may be downloaded and used without fee for noncommercial purposes from the Brookings Institution web site, http://www.brook.edu/ES/dynamics/models/ascape), the idea of a scape has been generalized to include any collection of agents. ASCAPE promotes the idea of the scape as something just as relevant as the agents. Clearly it is the behavior of *collections* of agents that is of primary interest in agent-based modeling. This core idea of ASCAPE simplifies execution of behaviors, statistics collection, and composability of models.

ASCAPE enforces the scape idea—every ASCAPE model has a "root" scape. A scape provides a way of organizing agents into a collection or set. A scape is *itself* considered an agent, so hierarchies of scapes can be built, allowing models to be composed of other models. Note that scapes are not in general disjoint sets, so an agent can be a member of more than one scape.

ASCAPE is written in JAVA, and ASCAPE models are, at least for the present, also expressed as JAVA classes. However, the ASCAPE framework can offer benefits to modelers regardless of their programming expertise (or lack thereof). Nonprogrammers can benefit from ASCAPE by making use of the prebuilt example models or models built by others. Variations on models can be investigated intuitively and interactively by turning rules on and off through the Model Settings window (Fig. 1) of the graphical user interface (GUI). Users can also interactively adjust parameter values, assemble charts, and capture movies of model runs. For users who favor spreadsheets, output data can be exported in a format readable by standard spreadsheet packages such as Microsoft EXCEL. Installation on a number of platforms is easy when using the provided, Web-based installer that detects what sort of machine one is using. For example, on machines running Microsoft WINDOWS it downloads and installs the appropriate JAVA and ASCAPE files and adds an ASCAPE item accessible via the Windows Start button. Models can also be run directly from the Web as applets.

ASCAPE benefits programmers by providing "for free" the capabilities of a full-fledged, working application, with a control bar, parameter- and rule-setting dialogs, movie capture, and charts (line, bar, and pie). Furthermore, many commonly used agent rules are provided. Such rules include placing an agent at a random location, executing a random walk, agent fission, and agent death. Built-in "statistic collectors" automatically compute count, sum, average, minimum, maximum, variance, and standard deviation. Neophyte programmers and modelers with a beginner's knowledge of JAVA can make rapid progress by studying the example model code provided.

Although ASCAPE provides many built-in services, ASCAPE does not lock in or limit the creativity of an experienced programmer. Because ASCAPE uses a general-purpose language to describe models, the framework permits an expert to do "almost anything." However, the framework does provide a starting point and a structure that greatly speeds up the development process and saves the developer from reinventing the wheel. This structure includes the previously mentioned hierarchical structure of scapes and the abstraction of scape topology and rule execution from agent specification. The power of ASCAPE's abstractions can lead to greater simplicity in model specification. For example, because ASCAPE has abstracted the idea of spaces and neighborhoods, one usually does not need to write logic to address numbered cells. An informal comparison showed ASCAPE models to be shorter than those written using other popular frameworks (2).

Because ASCAPE has a tight design, it reduces one's chances of getting off track in the modeling process. For example, it allows the developer to concentrate on implementing agent-based models instead of simulation infrastructure. Infrastructure is already provided and incorporates best practices such as a model-view-controller design that separates the code implementing a model from the code that provides views of the model (i.e., collects data from the model for display or other output) and the code that controls the model (starting, pausing, stopping, or sweeping over parameter values). After each iteration of the model, views and controllers are reported to and given a chance to respond. This design makes possible intelligent controllers implementing, for example, sophisticated, automated parameter exploration strategies.

As with modern rapid application development environments supporting frameworks such as Microsoft Foundation Classes (MFC), JAVA Foundation Classes (JFC), or MacApp, ASCAPE provides the developer with a feature-rich skeleton application right from the start. A line or two of code (Listing 1) provides a runnable application featuring a control bar (Fig. 2) with a window list, an iteration counter, a Running/Paused/Stopped indicator, (Re-)Start, Pause, Single-Step, Stop, Open Model, and Quit buttons, a speed control slider, as well as buttons to adjust model settings, generate a QUICKTIME movie, display application

This paper results from the Arthur M. Sackler Colloquium of the National Academy of Sciences, "Adaptive Agents, Intelligence, and Emergent Human Organization: Capturing Complexity through Agent-Based Modeling," held October 4–6, 2001, at the Arnold and Mabel Beckman Center of the National Academies of Science and Engineering in Irvine, CA.

*To whom reprint requests may be addressed. E-mail: mario.inchiosa@biosgroup.com or miles.parker@biosgroup.com.

www.pnas.org/cgi/doi/10.1073/pnas.082081199

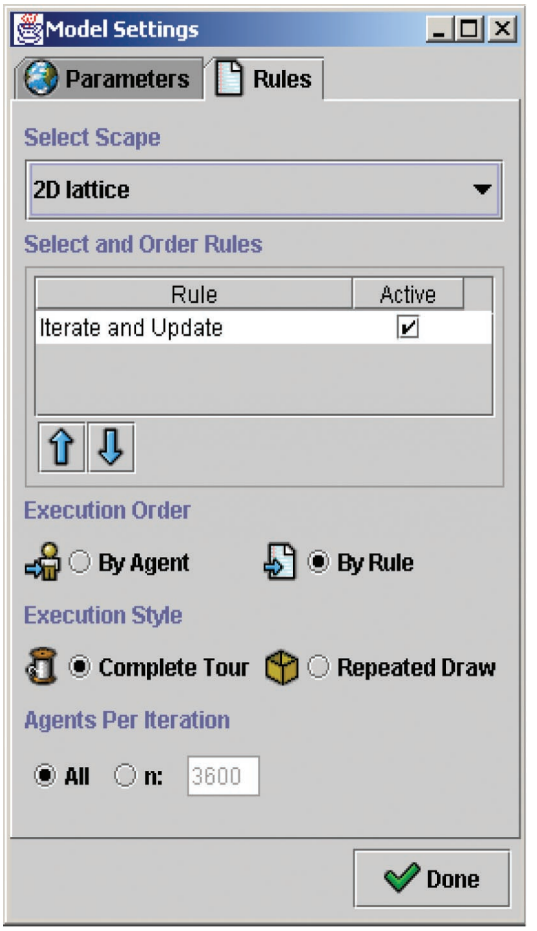

Fig. 1. Rules tab of the Model Settings window.

information, and produce line, bar, and pie charts. To create a useful model, one adds parameter variable, model state variable, and constant declarations, createScape and createViews methods, and "get" and "set" methods for the parameters. The createScape method populates the "root" scape with agents, which might themselves be scapes. These agents are instances of classes, typically declared in separate JAVA files, that the modeler derives (subclasses) by extending suitable ASCAPE agent classes with model-specific variables and methods.

Demonstration

As a concrete demonstration of what is actually required to create a working model in ASCAPE, we will discuss the code required to implement Conway's Game of Life (3).

Two public classes will be defined. Every model in ASCAPE has a root scape, and in the case of Life we will be extending a ScapeArray2D-type scape class to serve this function. Recall that a scape defines a grouping of agents. A ScapeArray2D scape places those agents on a two-dimensional grid. Specifically, we define a class named Life that extends the ScapeArray2DMoore class (Listing 2). ScapeArray2DMoore augments ScapeArray2D with the concept that nearest neighbors of a cell are the eight cells that share a side or a corner with that cell.

Listing 1. Minimal ASCAPE model: MinimalModel.java

```
public class MinimalModel extends
  com.biosgroup.ascape.model.ScapeVector {}
```

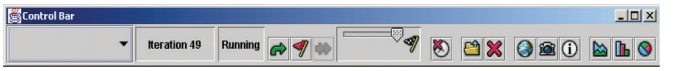

Fig. 2. ASCAPE Control Bar.

The Life class declares one instance variable, "initialAliveDensity," which serves as a model parameter. Provided the developer defines standard JAVA "get" and "set" methods (see the end of Listing 2), the framework automatically makes the model parameter accessible to the user via the graphical and command line interfaces.

When starting the execution of a model, the ASCAPE framework calls the root scape's createScape and createViews methods. In the createScape method we set the array's horizontal and vertical extent and the prototype agent that gets cloned to fill the array. Having set the root scape's properties, we call the createScape method of the superclass, ScapeArray2DMoore, which results in the array being instantiated and filled with clones of the prototype agent. We next set the scape's execution order to "RULE_ORDER," which specifies that all agents in the scape execute the first rule associated with the scape, then all agents execute the second rule (if any), and so on. The rules tab of the GUI (Fig. 1) allows the user to investigate the effects of execution order by changing it "on the fly." In createScape we also clear any preexisting rules and add any desired rule(s), in this case the "ITERATE_AND_UPDATE_RULE." This rule provides independent computing and then "parallel" updating of states (all agents compute their new state based on their and

Listing 2. Life.java

```
package com.biosgroup.life;

import com.biosgroup.ascape.model.*;
import com.biosgroup.ascape.util.*;
import com.biosgroup.ascape.view.*;

public class Life extends ScapeArray2DMoore {
  private double initialAliveDensity = 0.1;

  public void createScape() {
    setExtent(60, 60);
    setPrototypeAgent(new LifeCell());
    super.createScape();
    setExecutionOrder(RULE_ORDER);
    getRules().clear();
    addRule(ITERATE_AND_UPDATE_RULE);
  }

  public void createViews() {
    super.createViews();
    StatCollectorCond aliveStat = new StatCollectorCond("Alive") {
      public boolean meetsCondition(Object o) {
        return ((LifeCell) o).isAlive();
      }
    };
    addStatCollector(aliveStat);
    addView(new Overhead2DView());
  }

  public double getInitialAliveDensity() {return initialAliveDensity; }
  public void setInitialAliveDensity(double initialAliveDensity) {
    this.initialAliveDensity = initialAliveDensity;
  }
}
```

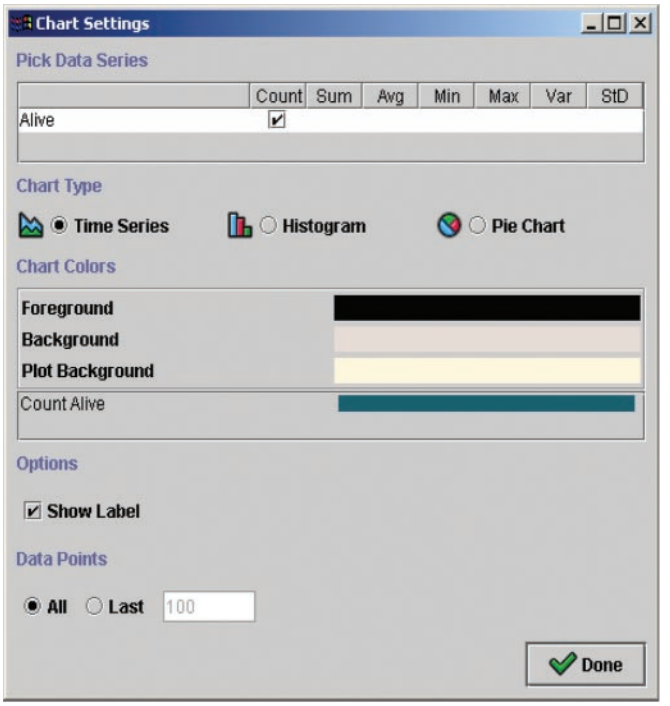

Fig. 3. Chart Settings.

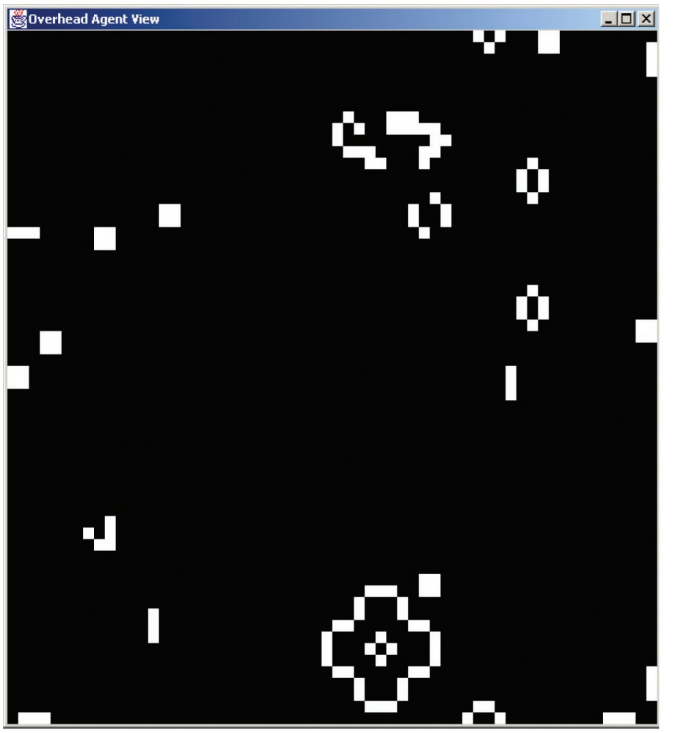

Fig. 4. Overhead Agent View.

their neighbors' current states, and then all agents' states are updated to their new states simultaneously).

Note that the need for parallel update in this model arises because Life is a traditional cellular automata; in contrast, most agent-based models typically use various styles of serial update. The ASCAPE GUI allows the user to experiment interactively with different rule execution styles: all agents can execute their rules each iteration, or the user can specify that only a given number of agents execute their rules in one iteration (Fig. 1). The user can also interactively specify agent activation regimes, such as whether agents are selected for behavior execution by repeated, independent random draw, or by making a complete execution *tour* through the agent population, thus guaranteeing that agents execute equally many times.

Returning to our discussion of the Life class, we next declare the createViews method. In createViews we call ScapeArray2DMoore's createViews method, which automatically generates an application control bar (Fig. 2) because the scape is functioning as the model's root scape. This method uses a local variable called "aliveStat" to count the number of alive cells in the root scape. We initialize "aliveStat" with an instance of an anonymous subclass [anonymous classes, introduced in JAVA 1.1, conveniently create a single object subclassing an existing class, but with some method(s) overridden by new methods] of the StatCollectorCond class; these objects collect statistics conditionally, based on their "meetsCondition" method, so we provide a meetsCondition method that returns true if the cell passed to it is alive. By adding aliveStat to the root scape, we cause it to collect statistics (conditionally) over all of the cells in the scape. The GUI automatically makes it available to the user for inclusion in line, bar, and pie charts (Fig. 3). Our createViews method also creates a two-dimensional array "overhead" view (Fig. 4) and connects it to the root scape via the addView method.

As mentioned above, methods beginning with "get" ("is" for Boolean variables) or "set" act as a signal to the ASCAPE framework to provide user access to the named parameter.

In a separate file, we define a "LifeCell" class that extends the Cell agent class to endow it with the behaviors of a Game of Life cellular automaton (Listing 3). A LifeCell's state is just a Boolean variable indicating "alive" or "dead." The Boolean variable "nextAlive" records whether the cell will be alive in the next iteration and helps us implement the parallel state update.

The initialize method calls Cell's initialize and randomly initializes the agent's state to alive or dead according to the initialAliveDensity parameter.

The iterate method computes the agent's next state. To find out how many of the agent's neighbors are alive, we use Cell's countNeighbors method and pass it a Conditional object whose meetsCondition method returns true if the agent passed to meetsCondition is alive (this is a more succinct use of the anonymous class syntax used to create the aliveStat object in Life.java). The countNeighbors method demonstrates an important ASCAPE abstraction, because it allows the LifeCell agent's code to be independent of the scape's geometry. Thus, it correctly separates the agent's details from details of the space in which the agent lives. We could change the scape's geometry by replacing ScapeArray2DMoore with ScapeArray2DVon-Neumann or ScapeArray2DSmallWorld without necessitating any changes in LifeCell. Having calculated the number of neighbors alive, we implement the birth and death rule of the Game of Life: an agent will be alive in the next iteration if it either has three neighbors currently alive or if it has two neighbors currently alive and it itself is currently alive.

After all agents have executed their iterate method, the update method is called, thus effecting a parallel or "synchronous" state update. The getColor method (used for the overhead view) returns white for living agents and black otherwise. The isAlive method, because it begins with "is," acts as an automatic signal to ASCAPE to make its Boolean value visible to the user. The user simply metaclicks (while clicking, hold down Alt on the PC or Option on the Mac) on any cell in the overhead view to bring up a cell inspector window (Fig. 5) that lists the values of any cell parameters made available by the developer via "get" or "is"

Listing 3. LifeCell.java

```
package com.biosgroup.life;

import java.awt.Color;

import com.biosgroup.ascape.model.*;
import com.biosgroup.ascape.util.*;

public class LifeCell extends Cell {

  private boolean alive;
  private boolean nextAlive;

  public void initialize() {
    super.initialize();
    alive = getRandom().nextDouble() <
      ((Life) getRoot()).getInitialAliveDensity();
  }

  public void iterate() {
    int neighborsAlive = countNeighbors(new Conditional() {
      public boolean meetsCondition(Object o) {
        return ((LifeCell) o).isAlive();
      }
    });
    nextAlive = neighborsAlive == 3 ||
      (neighborsAlive == 2 && alive);
  }

  public void update() {alive = nextAlive;}

  public Color getColor() {
    return alive ? Color.white : Color.black;
  }

  public boolean isAlive() {return alive;}
}
```

methods. If the cell is a host cell, the inspector also displays the parameters of any agent currently being hosted by the cell.

The Game of Life model just described was chosen for its simplicity. However, typical ASCAPE models usually incorporate some hierarchical structure and often have agents moving in a space. As an example, consider a demographic Prisoners' Dilemma model (4) implemented in ASCAPE (5). In this model, two agents belong to the root scape. Both of these agents are themselves scapes. One of these scapes is a collection of "prisoner" agents. The other is a ScapeArray2D lattice. Its members are agents called "host cells" that are capable of "hosting" a prisoner agent. The prisoner agents play the Prisoner's Dilemma game with any prisoners that happen to be "visiting" neighboring host cells.

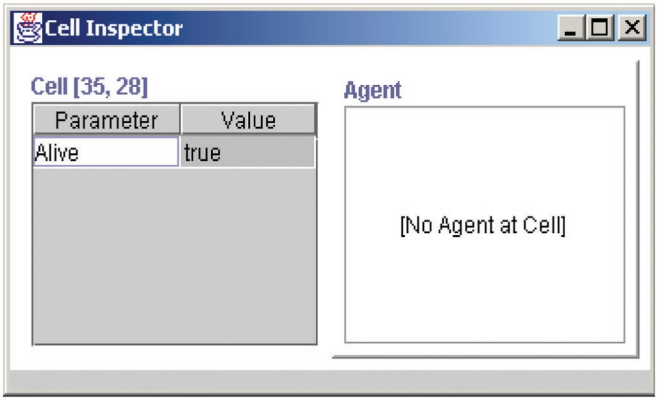

Fig. 5. Cell Inspector.

The idea of host cells can be a source of confusion and therefore bears some more explanation. We have seen that specialized scapes (e.g., ScapeArray2DMoore) exist in ASCAPE to represent various discrete geometries. Such scapes are derived from the superclass ScapeGraph. If a ScapeGraph just has a static agent residing at each node (as in the Game of Life), then those agents are typically of the class "Cell" or a subclass of "Cell." However, if the ScapeGraph will have agents moving on it, then the ScapeGraph would typically have a fixed HostCell agent at each node, and CellOccupant agents (for example, the prisoner agents in the demographic Prisoners' Dilemma model) belonging to another scape would move among the HostCells.

Roadblocks and Helpful Principles

Here we address some common roadblocks and helpful principles applicable to agent-based modeling in general and ASCAPE model development in particular.

Perhaps the most common typical modeling mistake is developing a model that is not well disciplined and that has too many pieces. Macroscopic equation-based modeling uses the readily available tools of mathematics to reduce models by means of exact simplifications or carefully characterized approximations. The goal in this form of modeling is typically to find the smallest number of irreducible top-down equations that adequately describe a system's dynamics.

The goal of agent-based modeling seems different, and this may lead researchers to wield Occam's razor with less zeal. One may think adding more detail yields a more accurate picture, but the modeler's job is to discover the system's essential core dynamics, and not necessarily to construct a complete description. Resist the temptation to add detail solely because you think it would make the model look more like the real world. Do not presume that having more complex descriptions will add anything to the model. In agent-based modeling, one is trying to generate non-obvious outcomes and one does not know *a priori* which of the interactions give rise to the outcomes. Therefore, one is often not sure on which dimension to pare away detail, and much experimentation is usually needed. It is best to take a disciplined, incremental approach to adding detail, beginning with the simplest model possible and developing an intuition about its core dynamics. Try to develop a thorough understanding of the dynamics of the system before adding to it. Recognize that further model development may require some refactoring of existing code rather than simple addition of new code. Be ready to throw things away!

Invest time in learning about object-oriented programming theory and techniques as well as basic development techniques and tools. Because one expresses ASCAPE models using JAVA, which is an object-oriented programming language, an insufficient grasp of object-oriented programming concepts and principles can be a problem. Developers building ASCAPE models should familiarize themselves with the basics of JAVA, such as classes and inheritance, as well as the somewhat more advanced anonymous class syntax. Understanding basic JAVA tools such as Javadoc and their common usage is also extremely helpful.

Armed with this knowledge, then study the ASCAPE example models. Because a good deal of context is required (e.g., how to compile a JAVA program on a particular platform), it is often much more efficient to have an experienced JAVA and ASCAPE "mentor" available to answer questions and help novices through the compilation of their first model. Alternatively, a development team can be composed of software developers who know JAVA and ASCAPE well and who set up the model, while scientists

who are less proficient in JAVA and ASCAPE can still add to and tweak the model at the level of individual agent methods.

Invest time in learning about the tool. Although it is possible to accomplish quite a bit simply by tweaking existing models or making brief excursions from them, as you work with ASCAPE you will almost certainly find yourself exploring further and further afield. Good resources for deepening one's understanding of ASCAPE include refs. 2 and 5 and the Javadoc documentation of the Scape, Rule, and StatCollector classes. Working with any software framework and exercising it well does require a deeper and deeper knowledge of the framework. You are often (although not always) rewarded in proportion to the energy you put in. Users often experience easy initial adoption, but there is more to explore the deeper you get into ASCAPE, and this can cause some frustration.

For example, users often want to make the tool behave in a particular way: they have found one approach that works for them and then try to use that for everything, in the process using mechanisms awkwardly—in a way other than for which they were designed. Another very common error is to drill down myopically, writing one's own code (for example a search routine) without noticing that that functionality already exists in the framework. Therefore, take time to understand ASCAPE, particularly the underlying metaphors of ASCAPE: collections of collections of agents and rules executed across collections. Take a survey of the classes to get acquainted with the framework, widen your focus, and understand the context in which things are occurring.

Spend some time thinking about how you develop software and commit to testing. Although it is certainly not necessary to have a formal software development methodology, it might be useful to examine how others work to create quality software. Often, there are very simple principles involved, such as having a simple design in mind before coding, defining programming tasks in simple human readable terms, or clearly separating model development from model exploration. Object-oriented software and agent-based models are particularly amenable to unit-based testing. We use the JUnit testing framework (http://www.junit.org) in our modeling and in ASCAPE itself. Becoming familiar with unit-based testing principles and applying them to all of your work will vastly improve the rigor and efficiency of your research and provide much greater confidence in your results.

At BiosGroup, we are engaged in bringing ASCAPE from a solid and useful but more narrowly focused tool to creating a common platform for the creation of many kinds of agent-based models. Recent work has focused on developing specific models in ASCAPE of interest to government and corporate clients, allowing many people at BiosGroup to gain familiarity with ASCAPE and its approach. This work has exposed us to a number of unique models and created opportunities to expand the role and scope of the ASCAPE toolset. At the same time, we have focused on quality assurance and efforts to make ASCAPE more widely accessible. A number of new features have also been added.

A great deal of future work is contemplated. We anticipate that future tools will allow the composition of models by dragging and dropping model pieces together and building model hierarchies graphically. One should be able to define basic attributes and rules without programming and hook these rules together by using some iconic facility. Another possibility would be to allow the specification of models by using XML-based tools. Along the way, there are many other technical challenges, such as seamlessly allowing the modeling of discrete and continuous time, as well as modeling agents within continuous space. And there are many incremental changes that may also occur. As just one fairly minor example, elevating attributes, now typically JAVA primitive types, to some kind of higher-level facility would carry advantages if done in a way so as to minimize performance penalties. Of course, an ever-present challenge is accomplishing significant changes while retaining backward compatibility wherever possible. As ASCAPE grows from the work of one developer to the combined efforts of a number of developers sharing a common architectural vision, much more is possible.

ASCAPE is a living, evolving research tool, and often people think of improvements or additions to ASCAPE that we either have not thought of or have not had a chance to implement. (Occasionally, people even communicate with us about things that we thought worked properly, but do not, or about documentation that needs clarification.) We appreciate any and all input provided: user involvement is a critical component in our mission to provide a great tool for building and exploring agent-based models.

1. Axtell, R. L. & Epstein, J. M. (1996) *Growing Artificial Societies: Social Science from the Bottom Up* (Brookings Institution Press/MIT Press, Cambridge, MA).
2. Parker, M. T. (2001) in *SwarmFest 2000: Proceedings of the 4th Annual Swarm User Group Conference*, ed. Pitt, W. C. (S. J. and Jessie E. Quinney Natural Resources Research Library, Logan, UT), Natural Resources and Environmental Issues, Vol. XIII, pp. 22–30.
3. Gardner, M. (1970) *Sci. Am.* **223,** 120–123.
4. Epstein, J. M. (1998) *Complexity* **4,** 36–48.
5. Parker, M. T. (2001) *J. Artificial Societies Social Simulation* **4,** http://www.soc.surrey.ac.uk/JASSS/4/1/5.