Natural Resources and Environmental Issues

Volume 8 *SwarmFest 2000* Article 5

1-1-2001

Ascape: Abstracting complexity

Miles T. Parker
BiosGroup, Inc. Washington DC

ASCAPE: ABSTRACTING COMPLEXITY

MILES T. PARKER[*]

BiosGroup, Inc. Washington D.C., 1100 North Glebe Road, Suite 720, Arlington, VA 22201, Email: miles.parker@biosgroup.com

Abstract. Software tools used in science typically take a kitchen-sink approach to design. From statistics to mathematics to engineering to agent modeling, even those tools that have a strong organizing theme tend towards supporting every contingency and methodology. This impulse toward generalization and breadth is laudable and necessary. However, there is a complementary case to be made for the discipline of abstraction, parsimony, and depth, and that is the case I make for Ascape.

I argue in general for the importance of abstraction in agent-based modeling. I then discuss three key abstractions enforced in Ascape, and the opportunities they create for expressibility and simplicity. While these abstractions seem especially suited to the domain of social and economic systems, they are not limited to it. By drawing concrete examples from Ascape and comparing Ascape code to other environments, I show how these apparently constraining abstractions benefit the Ascape user and developer experience.

> "In summary, a primary goal of software design and coding is conquering complexity. The motivation behind many programming practices is to reduce a program's complexity. Reducing complexity is a key to being an effective programmer."
>
> Steve McConnell (1993)

INTRODUCTION

A central argument for agent modeling, elegant in its obviousness, can be expressed in one phrase. "Why don't we model it as it is in the real world?" This argument is a strong tonic following hundreds of years of attempts to model phenomena (and especially social and economic phenomena) by generalizing the "real world" away. Agent modeling is exciting because it allows us to work, not with supposedly ideal (and somehow lifeless) generalizations, but with the ordinary (and somehow lifelike and thus extraordinary) things that occur when discrete objects are allowed to interact with one another.

The argument is so compelling that we could get swept away by it and set as our ultimate goal the creation of real world models of every conceivable system. Imagine as a thought experiment a vast complete model of the universe detailing every scale, connecting every kind of system; an awe-inspiring vision. If we could build such a model we would have an incredible experimental platform and device for prediction. But would we have explained anything, except as a by-product of the discovery of the system components and behaviors?

The real work of explanation still involves some form of generalization. The act of explanation seems to involve describing some interesting essence of a system using less information than the system itself. In agent-based modeling we might discover that a set of simple

[*] Miles Parker was a software engineer at the Brookings Institution and developer of the Ascape framework. Diagrams in this paper were initially presented at a workshop on Agent Simulation at the University of Chicago in October 1999.

rules iterated across a collection of agents produces a particular dynamic, we may observe that a similar phenomenon occurs in the natural world, and we might infer from that that the dynamic is caused by a specific kind of simple interaction. This kind of explanation is different from simply building a model and observing that it seems to have interesting correlation with the real world.

And it is certainly possible to discover these explanations without our "complete" model. In fact, it could be argued that a complete model would not directly contribute much toward their making. On examination, our complete model is different in kind than an explanatory model since it involves no discrimination between factors in the model. In contrast, the act of explanatory modeling is in some sense identifying the objects and behaviors that matter, and ignoring those that don't.

In other words, there is a crucial distinction to be made between modeling reality and 'modeling the model'. In the first case, our aim is to build a mirror world that can be examined. In the latter, we are creating a conceptual model of a system and using modeling tools to realize that conceptualization. But aren't all models in fact based on some set of abstract conceptualizations, no matter how fine-grained? At some level aren't we always modeling an abstraction? Why pretend otherwise?*

In this context, disciplines that we enforce on modeling are not just practical syntactical considerations. They are semantic. Every time you create an abstraction, you reveal a hidden symmetry that allows apparently different semantic structures to be treated in the same way. If we can describe a system using fewer declarations than another description, or in a manner more consistent with other explanations, we have arguably created a better description of that system. Any models that use strong abstractions potentially carry all the explanatory power that they are built upon.

A sometimes-overlooked aspect of abstractions is that they involve giving something up to get something else. Typically, you give up some degree of fine-grained control and freedom. What you gain is not just 'free' additional functionality, but power and expressibility. Arguably, this is what distinguishes a framework from a library. A library typically just provides some set of standard functionality that one can draw upon. In contrast, a framework requires one to enter into a kind of contract. The agreement typically reads something like: "I will agree to conform to a particular set of interfaces, and put up with having some rather obvious things hidden from me, in order to gain much more powerful higher level functionality and leverage." That said, in an ideal framework, it should be possible to completely control all aspects of an environment, making it possible to have the best of both worlds. Moving from the realm of theory to practice, we now examine some key abstractions that Ascape uses.

THE ABSTRACTIONS

Ascape is an agent modeling environment developed at The Brookings Institution to support the design, analysis and distribution of agent-based models. Its principal design goals include abstraction and generalization of key agent modeling concepts, ease of use, configurability, and performance. It is written completely in Java in order to provide maximum deployment options, and to take advantage of Java's strong typing and idioms. Taking the term 'agent' broadly, and understanding a 'scape' to be a kind of collection of these agents, they are:

1. All scapes are themselves agents.
2. Scape structure is hidden from agents.
3. Behaviors occur across scapes, as 'rules'.

On initial inspection, these abstractions do not appear particularly novel. Critical aspects of them have appeared as explicit design features pioneered by Swarm. For instance, it is certainly possible in Swarm to build an agent composed of

* This analysis skirts an interesting argument. Might models designed as purely conceptual exercises miss critical model dynamics that either have not or can not be conceptualized? If so, how would explicit work with such dynamics challenge our current approach of the scientific process? These questions merit further study but are beyond the scope of this paper.

other agents. In fact, this is the original vision of Swarm, as immortalized in the swarm logo. So we must be careful to differentiate between foundational ideas and later realizations. There are certainly a number of novel features and techniques in Ascape. But for many of the abstractions I discuss, the real power lies in realizing important aspects of a modeling vision first articulated by Swarm. And, speaking very broadly, the realization of an abstraction rests not in what it allows or envisions, but what it supports, requires and prevents.

All Scapes Are Themselves Agents

Scapes can be seen as glorified collections which seem to correspond quite closely to the conception of swarms. In Ascape, the identifying properties that represent an aggregation of agents and the methods we use for accessing its members are tightly bound. By requiring all scapes to be agents, we are assuring that all structural features of an agent model are defined in the context of a basic agent-modeling paradigm. And because, as I describe below, all behaviors occur upon agents as mediated by scapes, we are assuring that all behavior is also closely defined by this paradigm.

It could be argued that viewing scapes as agents is a stretch. While there are many possible definitions of agent, it is generally agreed that they involve some aspects of autonomy and self-agency. For now, we simply observe that our understanding of how autonomy and agency relate to what we call agent-based models is an open one, and may evolve over time. It is not even clear, for instance, that the primary focus of these models should be independent actors. Perhaps the question of agency *per se* has been overlooked to some extent because agent-based models have their roots in artificial life and have tended to explore models of biological aggregates that by virtue of their 'aliveness' seem to automatically be privileged with agency. But many models that are natural candidates for exploration using agent tools demonstrate that the distinction of autonomy and agency might not be as natural or useful as we assume. For the geomorphologist: does a landform have agency? For the physicist: does a particle? For the economist: does a firm? This question becomes more important as we work with more abstract aspects of models.

These issues warrant further exploration; for now, I will rely on the argument that treating scapes as agents seems to work very well on a practical level. For those who are uncomfortable with the usage of 'agent', consider that what we call agents in the existing milieu may more broadly represent a relatively clean, but arguably arbitrary boundary between system components.* The terms of agent-based modeling are made less clear because of the many glancing similarities and obvious differences between the terms of agent-based modeling and those of multi-agent systems in general. Perhaps it is the description 'Agent' itself that is limiting, and, as a radical suggestion, we might invent a new terminology for the constituents of our models. We could directly reflect the viewpoint that agent models are really based on aggregations of objects that are defined more by their useful boundaries than by any agency. New terms (I, only half-seriously, propose 'Agrent') or adoption of other descriptions, such as 'finite models' may serve to clarify this issue, though it is just as likely that they would serve as distractions, given the broad acceptance of existing terminology.

To return to the discussion of the abstraction, if we take the admittedly arguable point that agents are really arbitrary boundaries, our abstraction imposes the requirement that any boundaries between different generalizations must be presented explicitly and must carry the same privilege. In this sense, there can be no structural assumptions about the model that are not reflected in the strict hierarchy itself. At the same time, we must recognize that as we move further up the model hierarchy, the model becomes more abstract, and scapes become more practical conveniences and less natural features of the system we are modeling. Again, this is an area for further study beyond the scope of this paper.

The real practical benefit of all of these requirements is composability. All models become

* As has been observed for millennia (but will not be addressed here,) even the status of a human being as an autonomous well-bounded entity is not completely certain.

defined in the context of a sophisticated hierarchy. Expressed in design pattern language, an Ascape model is essentially a composite. (Gamma et al 1995) This composite pattern takes the form of a sophisticated hierarchy, where each node of the hierarchy can impose any topology on its constituent sub-nodes. This means that a complete tour of the model is always well defined and typically easily understood and expressed.

As an example, consider a simple model of a demographic prisoner's dilemma game (Epstein 1998). In this case, players make up a collection with other players (Figure 1). Players also move upon a lattice made up of specialized host cell agents, or locations, designed to contain other agents. The lattice and the collection of players are members of a root scape.

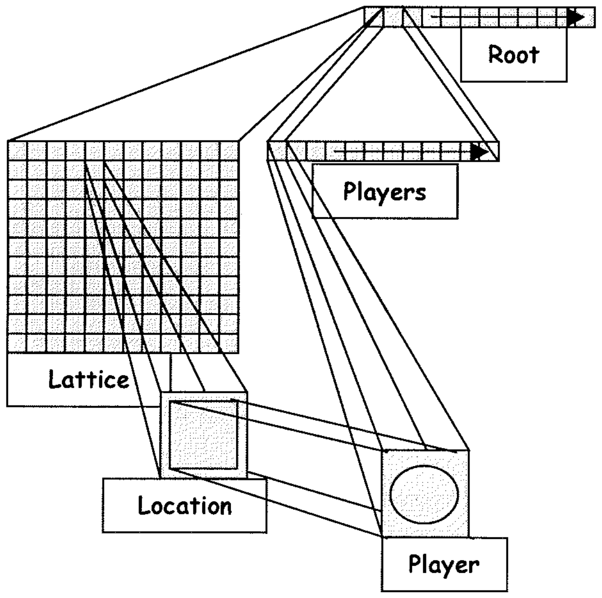


FIGURE 1. A simple Ascape model composition

In the prisoner's dilemma model, the only 'agents' that really have any model behavior are the players. In other models, such as Sugarscape (Axtell and Epstein 1996), the lattice would also have obvious behavior, as for instance map locations replenish their supply of sugar every model period.

A more complex example draws out the composability benefits of the 'scape as agent' approach (Figure 2). This 'Artificial Anasazi' model attempts a reconstruction of the environmental and demographic history of native pueblo people in the Long House Valley of northern Arizona (Dean et al 2000). In this model, map locations are occupied by settlements, which are collections of households. Households are members of a collection composed of households.

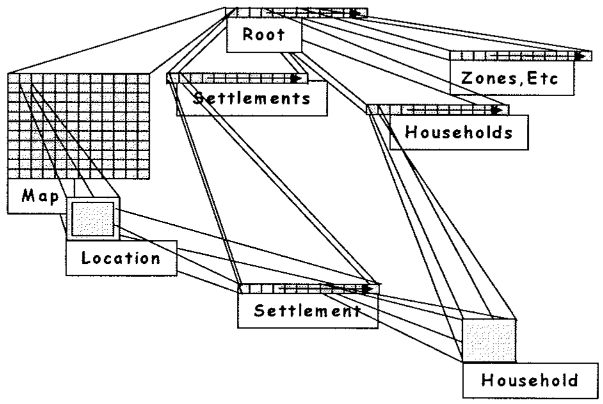


FIGURE 2. Long House Valley model composition

A household in this model thus has two contexts; as a member of a settlement, and as a member of the population of the valley as a whole. As we will see, scapes serve two purposes; as structural contexts, and as contexts for the iteration of behavior across constituents. Note that both contexts are potential sources for household behavior, but the most straightforward context for managing household behavior is the households vector.[*] In recent work, households themselves have been disaggregated into a scape of individual family members.

The base model is defined to be root, and is just another scape within the model. As stated, a completely unified method of touring the model constituents is available throughout. All Ascape system behaviors, such as statistic collection and visualization, are managed by the same rule mechanism. Generic rules that manage these behaviors can be propagated through the levels of model hierarchy. If desired, generic model behavior such as initialization and updating and

[*] Whenever agents exist in more than one scape, it seems natural to push aggregate control up the hierarchy to the most general scape possible.

even model specific behavior can also be easily propagated.

The strict hierarchical structure that the basic abstraction requires is easy to understand, easy to explain, and easy to code. More importantly, the model developer can be easily assured that all behaviors are well defined and completely determined. Finally, it is a natural fit for technologies like XML that rely on a well-defined hierarchy, and may also have significant advantages for strictly declarative approaches. Scape structure is hidden from agents

This abstraction is expressed as a negative. Expressed positively, agents have high-level ways of interacting with the space they exist upon and are free from implementation and topologically specific issues. Ideally, agents should not know anything about the specific structure of the scape they exist within. In practice, agents can potentially access topology specific information, but Ascape's design encourages pushing this down to the framework.

At first glance, this seems an odd burden to impose. Interaction space seems crucial to so many dynamics that abstracting it out of model design seems questionable. However, this abstraction is not about removing scape structure but ensuring that algorithms that relate to scape structure are actually built within the scape, where they belong.

As a quick illustration of how this abstraction works, imagine a model in which an agent searches for the best food source within 2 cells of its present location. A model developer would ordinarily code this search by hand. In Ascape, it is only necessary to make a method call looking something like:

```
Cell mostFoodCell =
findMaximumWithin(FOOD, 2)
```

If we take the filled circle as the target agent, the open circle as the optimal location, and the shaded area as the search region, Figure 3 shows how the search is conducted in each space.

In a basic implementation, it should be possible to change the entire geometry with a single statement change.

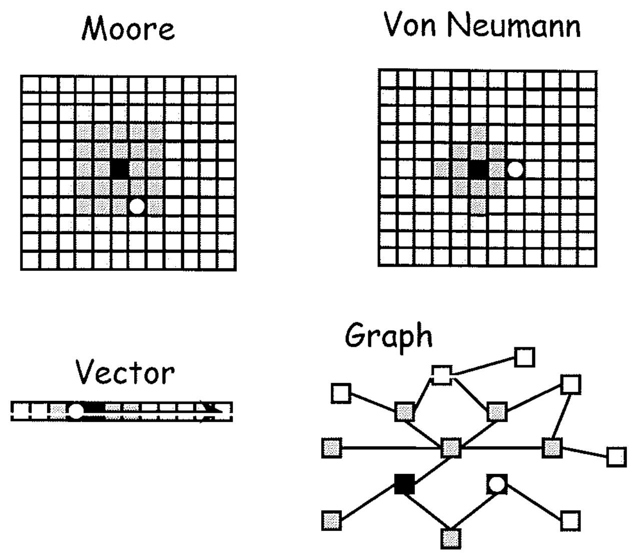


FIGURE 3. Polymorphic searching

This simple abstraction is responsible for a lot of the productivity advantages of Ascape. By pushing all searching and movement algorithms into the framework, we eliminate masses of redundant code. More importantly, the code for a particular algorithm is managed in one place. To see the practical advantage of this approach I show a sample of code realizing a common example model in three modeling frameworks (Table 1).* This model, first realized in Swarm's 'Heatbugs' code, causes an agent to seek, with some random component, the nearby cell that has the optimal heat level based on the agent's preferences.

This is an example rather obviously designed to show Ascape in the best possible light, but the important issues is that the same basic functionality used here is used throughout a modeling environment. In fact, it is probably the case that a large percentage of modeling code is wrapped up in agent searching and movement throughout an environment.

Now, a set of libraries could be created that would infer a number of the benefits described above. For example, one could create a two-dimensional library that provided a common routine for exploring a two-dimensional space. But such a library would in some sense miss the point. First, abstraction means more than

* To facilitate direct comparison I have removed white space and comments and made bracket usage consistent.

TABLE 1. Example of code-size benefit achieved principally through abstraction.

Swarm

```
NewX = x;
  newY = y; {
    HeatCell heatCell = new HeatCell (newX, newY);

    heat.findExtremeTypeSX$Y (((heatHere < idealTemperature)
                             ? HeatSpace.hot
                             : HeatSpace.cold),
                            heatCell);
    newX = heatCell.x;
    newY = heatCell.y;
  }
  if ((Globals.env.uniformDblRand.getDoubleWithMin$withMax (0.0, 1.0))
     < randomMoveProbability) v {
     // pick a random spot
     newX =
       x + Globals.env.uniformIntRand.getIntegerWithMin$withMax (-1, 1);
     newY =
       y + Globals.env.uniformIntRand.getIntegerWithMin$withMax (-1, 1);

     // normalize coords
     newX = (newX + worldXSize) % worldXSize;
     newY = (newY + worldYSize) % worldYSize;
    }

  if (unhappiness == 0)  {
     // only update heat - don't move at all if no unhappiness
     heat.addHeat$X$Y (outputHeat, x, y);
    }
  else {
    tries = 0;
    // only search if the current cell is neither the optimum
    // or randomly chosen location - else don't bother
    if ( (newX != x || newY != y) )  {
       while ( (world.getObjectAtX$Y (newX, newY) != null)
             && (tries < 10) )  {
          int location, xm1, xp1, ym1, yp1;
          // choose randomly from the nine possible
          // random locations to move to
          location =
            Globals.env.uniformIntRand.getIntegerWithMin$withMax (1,8);
          xm1 = (x + worldXSize - 1) % worldXSize;
          xp1 = (x + 1) % worldXSize;
          ym1 = (y + worldYSize - 1) % worldYSize;
          yp1 = (y + 1) % worldYSize;
          switch (location)  {
            case 1:
              newX = xm1; newY = ym1;   // NW
              break;
            case 2:
              newX = x ; newY = ym1;    // N
              break;
            case 3:
              newX = xp1 ; newY = ym1;  // NE
              break;
            case 4:
              newX = xm1 ; newY = y;    // W
              break;
            case 5:
              newX = xp1 ; newY = y;    // E
              break;
            case 6:
              newX = xm1 ; newY = yp1;  // SW
              break;
            case 7:
              newX = x ; newY = yp1;    // S
              break;
            case 8:
              newX = xp1 ; newY = yp1;  // SE
            default:
              break;
            }
          tries++;                          // don't try too hard.
        }
      if (tries == 10) {
         // no nearby clear spot, so just don't move.
         newX = x;
         newY = y;
        }
    }
  .....
  world.putObjectSatX$Y (null, x, y);
  x = newX;
  y = newY;
  world.putObjectSatX$Y (this, newX, newY);
```

RePast

```
Int type = (heatHere < idealTemp) ? space.HOT : space.COLD;
Point p = space.findExtreme(type, x, y);
If (Uniform.staticNextFloatFromTo(0.0f, 1.0f) < randomMoveProb) {
  p.x = x + Uniform.staticNextIntFromTo(-1, 1);
  p.y = y + Uniform.staticNextIntFromTo(-1, 1);
}
if (unhappiness == 0) {
  space.addHeat(x, y, outputHeat);
} else {
  int tries = 0;
  if (p.x != x || p.y != y) {
    while ((world.getObjectAt(p.x, p.y) != null) && tries < 10) {
      int location = Uniform.staticNextIntFromTo(1, 8);
      // get the neighbors
      int prevX = (x + xSize - 1) % xSize;
      int nextX = (x + 1) % xSize;
      int prevY = (y + ySize - 1) % ySize;
      int nextY = (y + 1) % ySize;
      switch (location) {
        case 1:
          p.x = prevX;
          p.y = prevY;
          break;
        case 2:
          p.x = x;
          p.y = prevY;
          break;
        case 3:
          p.x = nextX;
          p.y = prevY;
          break;
        case 4:
          p.x = nextX;
          p.y = y;
          break;
        case 5:
          p.x = prevX;
          p.y = y;
          break;
        case 6:
          p.x = prevX;
          p.y = nextY;
          break;
        case 7:
          p.x = x;
          p.y = nextY;
          break;
        case 8:
          p.x = nextX;
          p.y = nextY;
        default:
          break;
      }
      tries++;
    }
    if (tries == 10) {
      p.x = x;
      p.y = y;
    }
  }
  space.addHeat(x, y, outputHeat);
  world.putObjectAt(x, y, null);
  x = p.x;
  y = p.y;
  world.putObjectAt(x, y, this);
}
```

Ascape

```
DataPoint maximizeFor = (((HeatCell) getHostCell()).getHeat() < idealTemperature)
                            ? HeatCell.MAXIMUM_HEAT_POINT :
HeatCell.MINIMUM_HEAT_POINT;
        Cell bestLocation = getHostCell().findMaximumWithin(maximizeFor, 1, true);
        if (!bestLocation.isAvailable() && bestLocation != oldHost) {
            randomWalkAvailable();
        }
        else if (bestLocation != oldHost) {
            moveTo((HostCell) bestLocation);
        }
        if (getRandom().nextFloat() < ((HeatbugModel)
getRoot()).getRandomMoveProbability()){
            randomWalk();
        }
```

simply wrapping an elegant, high-level shell around an existing design - in fact, it is often not practical to create meaningful abstractions on top of pre-existing systems. Abstraction involves collapsing apparent differences away, and this typically involves deep support for the abstraction throughout the environment. Second, such a library, unless it generalizes space, becoming in itself a framework, would still lock a model into a specific topology.[*]

[*] Of course, I do not argue that such a library would not have great value.

There is an important science reason for not doing this. Space, at least as we relate to it in finite models, is essentially also an arbitrary idealized concept. To test our models effectively we should be willing to subject them to many different spaces. Hopefully, a tool that enforces the capability to explore these spaces transparently will encourage further exploration.

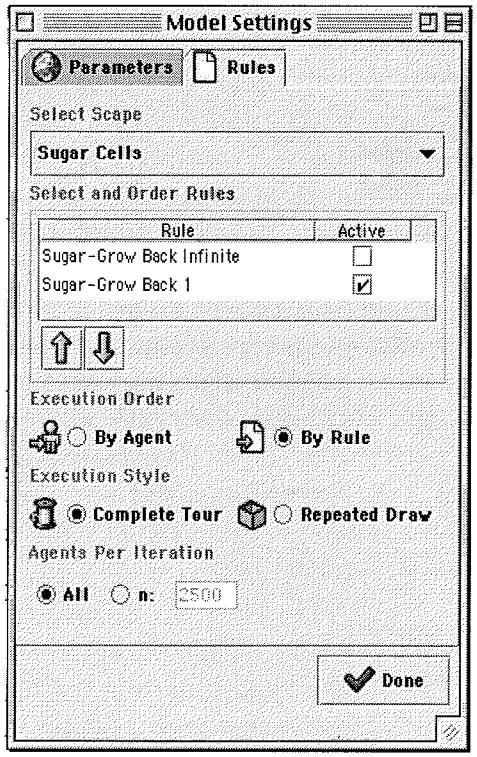
Behaviors Occur Across Scapes, As 'Rules'

The final abstraction is quite straightforward, and has been pointed to earlier. Behaviors can only be executed collectively across a scape as a whole. Rather than support more complex action scheduling, Ascape supports only the iteration of rules across members of a scape. The primary justification for this is that it makes organization and comprehension of scape models more straightforward, and it greatly facilitates user-level control of models.

There is some theoretical justification for treating behavior this way. First, we should recognize that Ascape 'rules' are not completely analogous to Swarm 'actions', though their usage is typically indistinguishable. Actions appear intended to be more fine-grained and dynamic or dispatchable. Rules are somehow more high-level, though it is difficult to draw any solid distinctions. While rules can conceivably be fired at individual agents, they typically refer to a class of behavior, or a variation on a class of behavior, that apply to an entire collection of agents. But these distinctions are really nitpicking; in actual usage Ascape simply treats the very common case of iterating actions for each agent as an explicit and assumed part of the model design.

In an Ascape model, while scape hierarchy is organized along structural lines, agents are also organized into scapes because they share the same class of behaviors, or behavioral context. Again, this only makes explicit something that is already a clear feature of most agent-based models. Because rules have a one to many relationships with scapes, they become a more clear and definite part of the model structure. Because scapes are already organized as a composite, this means that behaviors also always have a clear place in a potentially declarative hierarchy. Rules carry important information of their own; for instance a rule knows if it needs to be executed randomly, or if might potentially cause elements to be deleted, and this information can be used by the scape engine for optimization of execution.

The user and developer experience benefits most from requiring behavior to fit into a strict rule approach. Because all rules must belong to scapes, the developer has one clear target for managing and maintaining behavior. For the user, the abstraction also allows the creation of very clear run-time tools for experimenting with model behavior. For example, the following window allows a user to easily explore a model's scapes, selecting rules and changing model execution order and style (Figure 4).



Select Scape: Sugar Cells

Select and Order Rules

Rule	Active
Sugar-Grow Back Infinite	☐
Sugar-Grow Back 1	✓

Execution Order: By Agent [✓ By Rule]

Execution Style: [✓ Complete Tour] Repeated Draw

Agents Per Iteration: [✓ All] n: 2500

FIGURE 4. A model settings window

Of all the Ascape abstractions, this is one that appears most likely to be unworkable for some classes of models, and that might benefit from more sophistication. There are many features of the Swarm scheduling system that would no doubt be valuable in many contexts, and RePast appears to take a nice higher level approach to scheduling actions. However, for Ascape, given its design goals, it would be better to retain the current scheme's simplicity. There are many opportunities to build additional sophistication into the existing Ascape infrastructure, while retaining a scape-centric approach. That said, I have not encountered models that do not work well with the Ascape methodology -- though our experience with models is necessarily limited and I welcome counter examples.

CONCLUSION

The following table summarizes the abstractions discussed, and their [Gamma et.al.] design pattern (Table 2).

Any theoretical reduction in complexity should carry with it a practical advantage in coding complexity. And an unscientific review of code does demonstrate real advantages in code size. While code line counts are at best a rough and somewhat discredited method of measuring code complexity, especially across languages, it can certainly be a strong measure of potential productivity (Table 3). In fact, Java and Objective C seem very similar in their expressiblity (arguably, Java is slightly denser), so relative differences in code size is a strong measure of cognitive load. To avoid Obj C's inherent disadvantage, I do not count the size of header files (shown in parentheses) in the comparison, though a programmer cannot so easily avoid the necessity of them!

As many agent modelers have discovered, it is all too easy to create models that seem to have fascinating dynamics, but for which it is difficult to tease out any real understanding or even a consistent description. Careful block by block exploration of basic agent interactions seems necessary. This kind of exploration requires common languages and ideas that we can use to describe model structure and interaction. It can also be helped by the judicious use of strong abstractions. When we use abstractions, we are working within a high-level expressible semantic structure.

TABLE 2. Summary of key abstractions.

Abstraction	All scapes are themselves agents.	Scape structure is hidden from agents.	Behaviors occur across scapes, as 'rules'.
Theoretical justification	'Agents' are essentially arbitrary boundaries between different systems. Any boundaries between different scales and generalizations must be presented explicitly and must carry the same privilege.	Space is essentially an arbitrary construct. We need to be able to understand important mechanisms in the contexts of many different spaces.	Rules provide a clear logic for collective behavior. A scape is a context for behavior. Agents are organized into scapes not simply because they share the same place in a structure, but because they share the same class of behaviors.
Practical Benefits	Composability Consistency	Flexibility Consistency Robustness Expressiveness Experimentation	Comprehensibility Repeatability
GOF Pattern	Sophisticated Composite	Strategy-like	Iterator

TABLE 3. Code size compared.

	Swarm	Repast		Ascape	
Heatbugs	989 (1157)	622	63%	395	40%
Heatbugs (No Comments)	683	577	84%	304	45%
Sugarscape	1050 (1246)	815	78%	585	56%

The difference between a hodgepodge of description and a truly useful explanation is structure. A measure of growth in the agent community will be our adoption of strong common patterns and abstractions. Many efforts, including continuing theoretical work, adoption of standard structures such as XML, and experimentation with approaches such as strictly declarative models will be helpful. I hope that Ascape will make a strong contribution.

ACKNOWLEDGMENTS

For valuable comments he thanks Robert Axtell, Joshua M. Epstein and David Hines of the Brookings Institution. For valuable discussions he also thanks Marcus Daniels, Alex Lancaster, Irene Lee, Sander van der Leeuw, Nelson Minar, Glen Ropella and the many other members of the agent-based modeling and complexity community. Of course, he is solely responsible for any omissions or shortcomings.

LITERATURE CITED

Axtell, R. A. and J. M. Epstein. 1996. Growing Artificial Societies: social science from the bottom up. Brookings Institution Press/MIT Press.

Collier, N. 2000. RePast. http://repast.sourceforge.net. University of Chicago.

Dean, J. S., G. J. Gumerman, J. M. Epstein, R. A. Axtell, A. C. Swedlund, M. T. Parker, S. McCarroll. 2000. Understanding Anasazi Culture Change through Agent-Based Modeling The Dynamics of Human Primate Society Oxford University Press.

Epstein, J. M. 1998. Zones of Cooperation in Demographic Prisoner's Dilemma. Complexity 4(2):36-48.

Gamma, E., R. Helm, R. Johnson, J. Vlissides. 1995. Design Patterns: elements of reusable object-oriented software. Addison-Wesley.

Langton, C., R. Burkhart, N. Minar, M. Askenazi, G. Ropella, M. Daniels, A. Lancaster, I. Lee, and V. Jojic 1995. Swarm. http://www.swarm.org. Swarm Development Group.

McConell, S. 1993. Code Complete Microsoft Press. pp. 776.