Chapter 8
Designing and Building an Agent-Based Model

Mohamed Abdou, Lynne Hamill, and Nigel Gilbert

Abstract This chapter discusses the process of designing and building an agent-based model, and suggests a set of steps to follow when using agent-based modelling as a research method. It starts with defining agent-based modelling and discusses its main concepts, and then it discusses how to design agents using different architectures. The chapter also suggests a standardized process consisting of a sequence of steps to develop agent-based models for social science research, and provides examples to illustrate this process.

8.1 What Is Agent-Based Modelling?

Agent-based modelling is a *computational* method that enables a researcher to create, analyze, and *experiment* with *models* composed of *agents* that interact within an *environment*. Let us shed some light on the core terms italicized in this definition.

A *model* is a simplified representation of a "target" system that expresses as clearly as possible the way in which (one believes) that system operates. This representation can take several forms. For example, in mathematical and statistical modelling, the model is a set of equations (e.g., a regression equation). A graphical network of nodes and edges can model a set of friendships. *Computational* methods, such as agent-based modelling, involve building models that are computer programs. The program (i.e., the model) represents

M. Abdou (✉)
Department of Social Science Computing, Faculty of Economics and Political Science,
Cairo University, Cairo, Egypt
e-mail: moabdou71@yahoo.com

L. Hamill • N. Gilbert
CRESS, Department of Sociology, University of Surrey, Guildford, UK
e-mail: lynne.hamill@surrey.ac.uk; n.gilbert@surrey.ac.uk

the processes that are thought to exist in the social world (Macy and Willer 2002). For example, we might build a model to study how friends ("agents") influence each other's purchasing choices. Such processes are not easy to represent using mathematical equations because what one agent buys will influence the purchasing of a friend, and what a friend buys will influence the first agent. This kind of mutual reinforcement is relatively easy to model using agent-based modelling.

Another advantage of agent-based modelling when doing social research is that it enables a researcher to use the model to do *experiments*. Unlike natural sciences, such as physics and chemistry, conducting experiments on the real system (people for example) is impossible or undesirable. Using a computer model, an experiment can be set up many times using a range of parameters. The idea of experimenting on models rather than on the real system is not novel. For example, it is a better idea to use a model of an aeroplane to test flying under various conditions than to use a real aircraft (where the cost of experimentation is very high).

Agent-based models (ABMs) consist of *agents* that interact within an *environment*. Agents themselves are distinct parts of a program that represent social actors (e.g., persons, organizations such as political parties, or even nation-states). They are programmed to react to the computational environment in which they are located, where this environment is a model of the real environment in which the social actors operate.

In the following, we present two simple examples of ABMs, Sugarscape and Schelling's model of residential segregation, to illustrate the main concepts of agent-based modelling used in the remaining sections of this chapter. A general introduction to agent-based modelling is presented in Crooks and Heppenstall (2012).

8.1.1 Sugarscape

Sugarscape (Epstein and Axtell 1996) is a simple example of an ABM that yields a range of interesting results about the distribution of wealth in a society. The model represents an artificial society in which agents move over a 50×50 cell grid. Each cell has a gradually renewable quantity of 'sugar', which the agent located at that cell can eat. However, the amount of sugar at each location varies. Agents have to consume sugar in order to survive. If they harvest more sugar than they need immediately, they can save it and eat it later (or, in more complex variants of the model, can trade it with other agents). Agents can look to the north, south, east and west of their current locations and can see a distance which varies randomly, so that some agents can see many cells away while others can only see adjacent cells.

Agents move in search of sugar according to the rule: look for the unoccupied cell that has the highest available sugar level within the limits of one's vision, and move there. Agents not only differ in the distance they can see, but also in their 'metabolic rate', the rate at which they use sugar. If their sugar level ever drops to zero, they die. New agents replace the dead ones with a random initial allocation of sugar. Thus there is an element of the 'survival of the fittest' in the model, since those agents that are relatively unsuited to the environment because they have high metabolic rates, poor vision, or are in places where there is little sugar for harvesting, die relatively quickly of starvation. However, even successful agents die after they have achieved their maximum lifespan, set according to a uniform random distribution.

Epstein and Axtell (1996) present a series of elaborations of this basic model in order to illustrate a variety of features of societies. The basic model shows that even if agents start with an approximately symmetrical distribution of wealth (the amount of sugar each agent has stored), a strongly skewed wealth distribution soon develops. This is because a few relatively well-endowed agents are able to accumulate more and more sugar, while the majority only barely survive or die.

8.1.2 Schelling's Model of Residential Segregation

Another simple example is Schelling's model of residential segregation (1971). Schelling was interested in the phenomenon of racial residential segregation in American cities, and he aimed to explain how segregation could happen, and how these segregationist residential structures, such as ghettos, may occur spontaneously, even if people are relatively tolerant towards other ethnic groups, and even when they are happy with being a minority in their neighbourhoods.

A city in Schelling's model is represented by a square grid of cells each representing a dwelling. A cell can be in any of three colours: white, black, or grey according to whether it is occupied by a white agent, a black agent, or is empty. The simulation starts by randomly distributing the agents over the grid. Schelling supposed that people have a 'threshold of tolerance' of other ethnic groups. That means that agents are 'content' to stay in their neighbourhood as long as the proportion of their neighbours (which are the eight cells to the north, north-east, east, south-east, south, south-west, west and north-west) of the same colour as themselves is not less than this threshold. For example, with 50% threshold of tolerance, agents would be happy to stay in place as long as at least four of their eight neighbours are of the same colour; otherwise, they try to move to another neighbourhood satisfying this proportion.

Figure 8.1 shows the result of the simulation with 2,000 agents. The upper-left panel shows the starting random allocation of black and white agents over the grid, and the other three panels show the final configurations after running the simulation with tolerance thresholds of 37.5% (at least three of an agent's eight neighbours must be of the same colour for the agent to be content), 50% (four of eight), and 75% (six of eight). Clustering emerges even when agents are happy to be a minority in their neighbourhood (with 37.5% threshold), and the sizes of these emergent clusters increase with increasing levels of tolerance threshold.

In the following, we discuss the core concepts of "agents" and their "environment" in more detail.

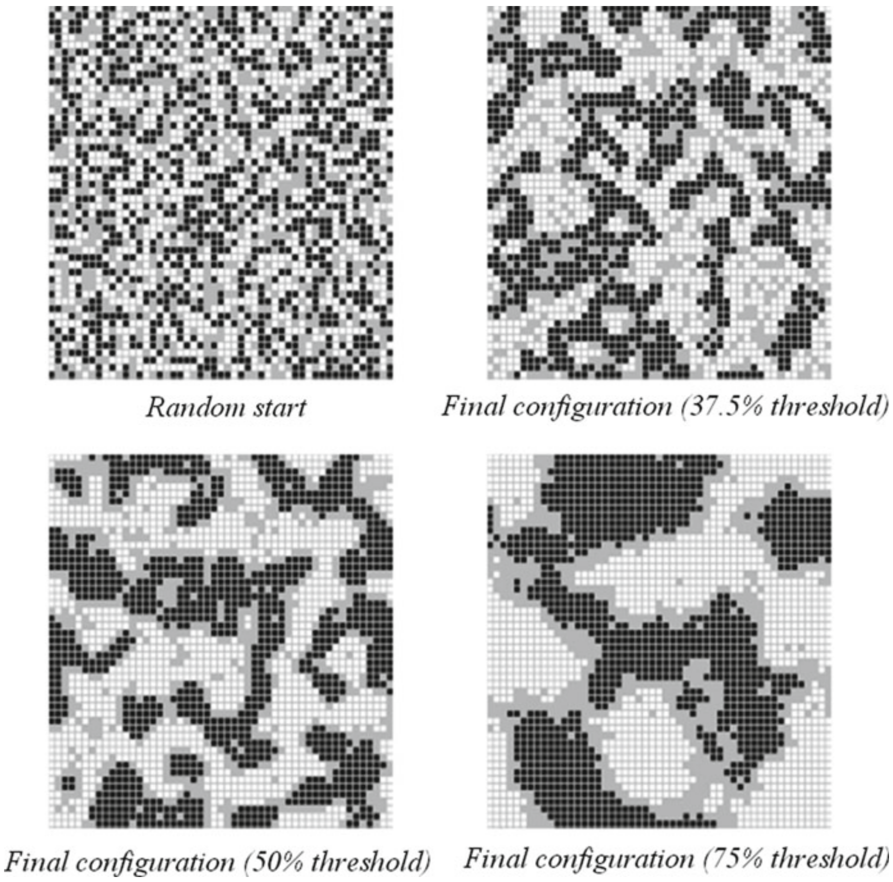


Fig. 8.1 The result of the simulation of the Schelling model

8.2 Agents

Applied to social science research, the concept of agency is usually used to indicate the purposive nature of human activity. It is thus related to concepts such as intentionality, free will, and the power to achieve one's goals. In agent-based modelling, agents are conventionally described as having four important characteristics:

- *Perception*. Agents can perceive their environment, including other agents in their vicinity. In the Sugarscape model, for example, agents can perceive the amount of sugar the current cell has.
- *Performance*: They have a set of behaviours that they are capable of performing such as moving, communicating with other agents, and interacting with the environment. In the Sugarscape model, they move and consume sugar.

- *Memory.* Agents have a memory in which they record their previous states and actions.
- *Policy*. They have a set of rules, heuristics or strategies that determine, given their present situation and their history, what they should do next, e.g. looking for cells with the highest level of sugar.

Agents with these features can be implemented in many different ways. Different architectures (i.e. designs) have merits depending on the purpose of the simulation. Nevertheless, every agent design has to include mechanisms for receiving input from the environment, for storing a history of previous inputs and actions, for devising what to do next, for carrying out actions and for distributing outputs. In the following, we describe three common approaches to agent architecture: using an object-oriented programming language directly, using a production rule system, and using learning approaches.

8.2.1 Object-Oriented Programming

The idea of object-oriented programming (OOP) is crucial to agent-based modelling, which is why almost all ABMs are built using an OOP language, such as Java, C++, or Visual Basic. A program developed in an OOP language typically consists of a collection of *objects*. An object is able to store data in its own *attributes,* and has *methods* that determine how it processes these data and interacts with other objects. As you might have noticed, there is an affinity between the idea of an agent and an object; it is natural to program each agent as an object.

The concept of 'class' is basic to OOP. A class is an abstract specification of an object. For example, a program might include a class called "Customer" to represent a customer of a firm in a model of business. A Customer might have a set of attributes such as name, address, and types of product (s)he likes. In the Sugarscape model, we can create a class named "Agent" with attributes such as *age*, *wealth* (the amount of sugar), *life-expectancy* (the maximum age that can be reached), *metabolism* (how much sugar an agent eats each time period), and *vision* (how many cells ahead an agent can see). A class also usually has some methods to describe its activities (e.g., move, eat sugar, save and die).

As the program runs, classes are *instantiated* to form objects. For example, the Customer class might be instantiated to yield two objects representing two customers: the first with name John Smith and the other with name Sara Jones (along with their other attributes). Although the two customers have the same methods and the same set of attributes, the values of their attributes (e.g., their names and addresses) differ.

When using OOP to design an ABM, one creates a class for each type of agent, provides attributes that retain the agents' past current state (*memory*), and adds suitable methods that observe the agents' environment (*perception*) and carry out agent actions (*performance*) according to some rules (*policy*). In addition, one needs to program a scheduler that instantiates the required number of agents at the beginning of the simulation and gives each of them a turn to act.

8.2.2 Production Systems

One of the simplest, yet effective, designs for an agent is to use a production system. A production system has three components:

1. *A Set of Rules of Behaviour*. These rules determine what an agent will do. Usually, a rule consists of two parts: a condition, which specifies when the rule is to be executed ('fire'), and an action part, which determines what is to be the consequence of the rule firing. Example of rules in Sugarscape include:
   - IF there is any sugar at the current cell, THEN eat it;
   - IF *sugar level* of the current cell exceeds *metabolism,* THEN add the extra sugar to *wealth;* and
   - IF age exceeds *life-expectancy,* THEN die.
2. *A Working Memory*. An agent's memory is represented by variables that store its current or previous states. For example, an agent's memory might store its current location and wealth (the amount of sugar). Rules can include actions that insert facts into the working memory (e.g. I am holding some sugar) or conditions that test the state of the working memory (e.g. IF I am holding sugar, THEN eat it).
3. *A Rule Interpreter*. The rule interpreter considers each rule in turn, fires those for which the condition is true, performs the indicated actions for the rules that have fired, and repeats this cycle indefinitely. Different rules may fire on each cycle either because the immediate environment has changed or because one rule has modified the working memory in such a way that a new rule begins to fire.

Using a production system, it is relatively easy to build reactive agents that respond to each stimulus from the environment with some action. A simple production system can be constructed from a toolkit such as JESS (the Java Expert System Shell, http://www.jessrules.com/) (Friedman-Hill 2003). There are also some much more elaborate systems that are based on psychologically plausible models of human cognition, such as Soar (Laird et al. 1987; Wray and Jones 2006; Ye and Carley 1995), CLARION (Sun 2006), and ACT-R (Taatgen et al. 2006).

8.2.3 Learning

Production-system-based agents have the potential to learn about their environment and about other agents through adding to the knowledge held in their working memories. The agents' rules themselves, however, always remain unchanged. For some models, it is desirable to create agents that are capable of more fundamental learning: where the internal structure and processing of the agents adapt to changing circumstances. There are two techniques commonly used for this: *artificial neural networks (ANNs)* and evolutionary algorithms such as the *genetic algorithm (GA)*.

ANNs are inspired by analogy to nerve connections in the brain. An ANN consists of three or more layers of neurons, with each neuron connected to all other neurons in the adjacent layers. The first layer accepts input from the environment, processes it and passes it on to the next layer. The signal is transmitted through the layers until it emerges at the output layer. Each neuron accepts inputs from the preceding layer, adjusts the inputs by positive or negative weights, sums them and transmits the signal onward. Using an algorithm called the *back propagation of error*, the network can be tuned so that each pattern of inputs gives rise to a different pattern of outputs. This is done by training the network against known examples and adjusting the weights until it generates the desired outputs (Garson 1998). Using ANNs, it is possible to design agents and train them to identify objects such as letters or words, or recognize voices and pictures.

In contrast to a production system, an ANN can modify its responses to stimuli in the light of its experience. A number of network topologies have been used to model agents so that they are able to learn from their actions and the responses of other agents (e.g. Hutchins and Hazlehurst 1995; Terna 1997).

Another way of enabling an agent to learn is to use an evolutionary algorithm. These are also based on a biological analogy, drawing on the theory of evolution by natural selection. The most common is the genetic algorithm (GA). This works with a population of individuals (agents), each of which has some measurable degree of 'fitness', using a metric defined by the model builder. The fittest individuals are 'reproduced' by breeding them with other fit individuals to produce new offspring that share some features taken from each parent. Breeding continues through many generations, with the result that the average fitness of the population increases as the population adapts to its environment.

Sometimes, it is desirable to use both techniques of learning, GAs and ANNs, in the same ABM. For example, one may need to create a large population of ANNs (each corresponding to one agent). The agents are initialized with a random set of connection weights and are set a task such as gathering "food" from a landscape. An agent's perception of whether there is food in front of it is fed into the ANN inputs, and the outputs are linked to the agent's action, such as move and eat. The agent is given an initial quantity of energy, some of which is used on every time step. If the energy declines to zero, the agent "dies" and it is removed from the simulation. An agent can boost its energy by eating food, which is scattered around the landscape.

Because of the random connection weights with which an agent's ANN is initialized, most agents will not succeed in finding and eating food and will quickly die, although some will succeed. Those more successful agents reproduce, giving their offspring similar connection weights as their own (but with slight mutation). Gradually, the population of agents will learn food harvesting behaviour (Acerbi and Parisi 2006; Gilbert et al. 2006).

8.2.4 The Environment

The environment is the virtual world in which agents operate. In many models, the environment includes passive objects, such as landscape barriers, “roads” down which agents may travel, resources to provide agents with energy or food (as in the Sugarscape model), and so on. These can be programmed in much the same way as agents, but more simply, because they do not need any capacity to react to their surroundings. For example, the environment in the Sugarscape model can be implemented by creating a class, called “Cell”, which has two attributes: *location*, which is the *xy* position of a cell, and *sugar level*, which indicates the amount of sugar the cell has. Then 2,500 (50×50) objects of this class are instantiated at the start of the simulation with their proper locations and random values for their sugar levels.

Environments may represent geographical spaces, for example, in models concerned with residential segregation where the environment simulates some of the physical features of a city, and in models of international relations, where the environment maps states and nations. Models in which the environment represents a geographical space are called *spatially explicit*. In other models, the environment could represent other types of space. For example, scientists can be modelled in “knowledge space” (Gilbert et al. 2001). In spatial models, the agents have coordinates to indicate their location. Another option is to have no spatial representation at all but to link agents together into a network in which the only indication of an agent’s relationship to other agents is the list of agents to which it is connected by network links (Scott 2000). It is also possible to combine both. Think, for example, of a railway network.

8.3 Developing ABMs in Social Science Research

Research in agent-based modelling has developed a more or less standardized research process, consisting of a sequence of steps. In practice, several of these steps occur in parallel and the whole process is often performed iteratively as ideas are refined and developed.

8.3.1 Identifying the Research Question

It is essential to define precisely the research question (or questions) that the model is going to address at an early stage. The typical research questions that ABMs are used to study are those that explain how regularities observed at the societal or macro level can emerge from the interactions of individuals (agents) at the micro level. For example, the Schelling model described earlier starts with the observation that neighbourhoods are ethnically segregated and seeks to explain this through modelling individual household decisions.

8.3.2 Review of Relevant Literature

The model should be embedded in existing theories and make use of whatever data are available. Reviewing existing theories relating to the model's research question is important to illuminate the factors that are likely to be significant in the model. It is also useful to review comparable phenomena. For example, when studying segregation, theories about prejudice and ethnic relations are likely to be relevant.

All ABMs are built based on assumptions (usually about the micro-level). These assumptions need to be clearly articulated, supported by the existing theories and justified by whatever information is available.

8.3.3 Model Design

After the research question, the theoretical approach and the assumptions have been clearly specified, the next step is to specify the agents that are to be involved in the model and the environment in which they will act.

For each type of agent in the model, the attributes and behavioural rules need to be specified. As explained in Sect. 8.2, an attribute is a characteristic or feature of the agent, and it is either something that helps to distinguish the agent from others in the model and does not change, or something that changes as the simulation runs. For example, in Sugarscape, an agent's *life-expectancy* (the maximum age that an agent can reach), *metabolism* (how much sugar an agent eats each time), and *vision* (how many cells ahead an agent can see) are examples of attributes that do not change, while *age* and *wealth* (the amount of sugar an agent has) are changeable attributes.

The agent's behaviour in different circumstances also needs to be specified, often as a set of condition-action rules (as explained in Sect. 8.2). This specification can be done in the form of two lists: one which shows all the different ways in which the environment (including other agents) can affect the agent, and one showing all the ways in which the agent can affect the environment (again, including other agents). Then the conditions under which the agent has to react to environmental changes can be written down, as can the conditions when the agent will need to act on the environment. These lists can then be refined to create agent rules that show how agents should act and react to environmental stimuli.

It will also be necessary to consider what form the environment should take (for instance, does it need to be spatial, with agents having a definite location, or should the agents be linked in a network) and what outputs of the model need to be displayed in order to show that it is reproducing the macro-level regularities as hoped (for example, the wealth distribution in the Sugarscape model, and the size of clusters of dwellings of the same colour in Schelling's model).

Once all this has been thought through, one can start to develop the program code that will form the simulation.

8.3.4 Model Implementation

After the model has been designed, and when the agents and environment are fully specified, the next step is to convert the design into a computer program. Most ABMs involve two main parts or *procedures*:

- ***Setup Procedure***. The Setup procedure initializes the simulation (and is therefore sometimes called the initialization procedure). It specifies the model's state at the start of the simulation, and it is executed once at the beginning. This part of the program might, for example, lay out the environment and specify the initial attributes of the agents (e.g., their position, wealth and life expectancy in the Sugarscape model).
- ***Dynamics Procedure***. This procedure is repeatedly executed in order to run the simulation. It asks agents in turn to interact with the environment and other agents according to their behavioural rules. This will make changes in the environment and invoke a series of action-reaction effects. For example, in Schelling's model of segregation, the dynamics procedure may ask all 'unhappy' agents to move from their neighbourhood. When an unhappy agent moves to a new place (where it feels happy), this may make some other agents (that were happy in the previous step) unhappy and want to move, and so on. The dynamics procedure may contain a condition to stop the program (e.g., if all agents are happy in Schelling's model).

An important decision is whether to write a special computer program (using a programming language such as Java, C++, C#, or Visual Basic) or use one of the packages or toolkits that have been created to help in the development of simulations. It is usually easier to use a package than to write a program from scratch. This is because many of the issues which take time when writing a program have already been dealt with in developing the package. For example, writing code to show plots and charts is a skilled and very time-consuming task, but most packages provide some kind of graphics facility for the display of output variables. On the other hand, packages are, inevitably, limited in what they can offer, and they are usually run more slowly than specially written code.

Many simulation models are constructed from similar building blocks. These 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 generally 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 RePast and Mason provide a similar range of features, including:

- 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 in 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 between the model and a Geographic Information System (GIS) so that the environment can be modeled on real landscapes (see Crooks and Castle 2012).
- A range of debugged algorithms for evolutionary computation (Sect. 8.2.3), the generation of random numbers and the implementation of ANNs.

Modelling environments provide complete systems in which models can be created, executed, and the results visualized without leaving the system. Such environments tend to be much easier to learn, and the time taken to produce a working model can be much shorter than using the library approach, and so they are more suited to beginners. 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.

NetLogo (Wilensky 1999) is currently the best of the agent-based simulation environments. (NetLogo will be briefly introduced in Sect. 8.4). This is available free of charge for educational and research use and can be downloaded from http://ccl.northwestern.edu/netlogo/. It will run on all common operating systems: Windows, Mac OS X and Linux. Other simulation environments include StarLogo (http://education.mit.edu/starlogo/) and AgentSheets (http://agentsheets.com), which are more suited to creating very simple models for teaching than for building simulations for research.

Table 8.1 provides a comparison between Swarm, RePast, Mason, and NetLogo on a number of criteria. The choice of the implementation tool depends on several factors, especially one's own expertise in programming and the complexity and the scale of the model. NetLogo is the quickest to learn and the easiest to use, but may not be the most suitable for large and complex models. Mason is faster than RePast, but has a significantly smaller user base, meaning that there is less of a community that can provide advice and support. A full discussion of the environments is presented in Crooks and Castle (2012).

8.3.5 Verification and Validation

Once we have a 'working' simulation model, it has to be verified and validated before using it to answer the research questions or to build theories about the real social world (model verification and validation are discussed in detail by Ngo and See (2012)). As Balci (1994) explains, *"model validation deals with building the right model … [while] model verification deals with building the model right"* (pp. 121–123).

It is very common to make errors when writing computer programs, especially complicated ones. The process of checking that a program does what it was planned to do is known as 'verification'. In the case of simulation, the difficulties of verification are compounded by the fact that many simulations include random number generators, which means that every run is different and that it is only the

Table 8.1 A Comparison of Swarm, RePast, Mason and NetLogo

	Swarm	RePast	Mason	NetLogo
License[a]	GPL	GPL	GPL	Free, but not open source
Documentation	Patchy	Limited	Improving, but limited	Good
User base	Diminishing	Large	Increasing	Large
Modelling language(s)	Objective-C, Java	Java, Python	Java	NetLogo
Speed of execution	Moderate	Fast	Fastest	Moderate
Support for graphical user interface development	Limited	Good	Good	Very easy to create using "point and click"
Built-in ability to create movies and animations	No	Yes	Yes	Yes
Support for systematic experimentations	Some	Yes	Yes	Yes
Ease of Learning and Programming	Poor	Moderate	Moderate	Good
Ease of Installation	Poor	Moderate	Moderate	Good
Link to geographical Information System	No	Yes	Yes	Yes

Source: Gilbert (2008)

[a]*GPL* General Public License, http://www.gnu.org/copyleft/gpl.html

distribution of results which can be anticipated by the theory. It is therefore essential to 'debug' the simulation carefully, preferably using a set of test cases, perhaps of extreme situations where the outcomes are easily predictable.

While verification concerns whether the program is working as the researcher expects, validation concerns whether the simulation is a good model of the real system, the 'target'. A model which can be relied on to reflect the behaviour of the target is 'valid'. A common way of validating a model is to compare the output of the simulation with real data collected about the target. However, there are several caveats which must be borne in mind when making this comparison. For example, exact correspondence between the real and simulated data should not be expected. So, the researcher has to decide what difference between the two kinds of data is acceptable for the model to be considered valid. This is usually done using some statistical measures to test the significance of the difference. While goodness-of-fit can always be improved by adding more explanatory factors, there is a trade-off between goodness-of-fit and simplicity. Too much fine-tuning can result in reduction of explanatory power because the model becomes difficult to interpret. At the extreme, if a model becomes as complicated as the real world, it will be just as difficult to interpret and offer no explanatory power. There is, therefore, a paradox here to which there is no obvious solution. Despite its apparently scientific nature, modelling is a matter of judgement.

8.3.6 Some Practicalities

Two important practical issues to consider are how big the model should be and how many runs should be done.

8.3.6.1 How Big?

How many agents should be used? Over how big a space? There is little guidance on this question, because it depends on the model. The model must be sufficiently large to permit enough heterogeneity and opportunities for interaction. But more agents mean longer run times.

It is often best to start programming with just a few agents in a small environment. Then, when the program is working satisfactorily, increase the scale until one feels there is a satisfactory balance between the size and the stability of the output. Some ABMs use millions of agents (see Parry and Bithnell 2012), but for most purposes, this is unnecessary and impractical. One should probably aim for at least 1,000 agents unless there is good reason to use fewer.

8.3.6.2 How Many Runs?

Because of the stochastic nature of agent-based modelling, each run produces a different output. It is therefore essential to undertake more than one run. The question is, how many runs? The more runs, the more confidence one can have in the results, but undertaking too many runs wastes time and there is more data to analyze. Basic statistical theory suggests 30 is sufficient and frequently, 30 or 50 runs are undertaken, e.g. Epstein (2006). Again, there is no clear guidance on this topic. However many runs are done, it is worth quoting the standard deviation to provide some indication of the variability.

8.4 Examples

This section presents two simple models based on models in NetLogo's library: Traffic Basic and Segregation (Wilensky 1997a, b). The models are taken from NetLogo version 4.0.2.

8.4.1 A Basic Traffic Model

This is a very simple model developed from Wilensky's basic NetLogo traffic model (1997a). It is not possible to give a full introduction to NetLogo here: there are tutorials on the NetLogo website and books such as Gilbert (2008). However, for those unfamiliar with NetLogo, an explanation of what the program is doing is provided alongside the code (see Box A).

Sect. 8.3 identified five stages to developing a model: identifying the research question, reviewing the literature, designing and implementing the model and finally verifying and validating it.

Stage 1: Identifying the research question
The research question to be addressed is the relationship between the level of congestion and the speed and smoothness of traffic flow.

Stage 2: Reviewing the literature
Because the main purpose of this model is to demonstrate agent-based modelling, it is sufficient to note here that it is a well recognised fact that traffic jams can arise without any obvious cause. In general, a good literature review is essential to support the model.

Stage 3: Model design
The environment is a road and the agents are drivers represented by cars. The drivers change their speed according to whether there are other cars in front so as to remain within set speed limits. The program records the speed of the vehicles and the number of vehicles queuing at any one time.

Stage 4: Model implementation
The set-up procedure involves setting the parameters and creating the agents and their environment. The environment – the road – is built and the cars are created, distributed randomly along the road and randomly allocated a speed, determined by three parameters, set by sliders on the interface:

- the number of cars (nOfCars): minimum 2, maximum, 30
- the minimum speed (minSpeedLimit): 0–0.5
- the maximum speed (maxSpeedLimit): 0.5–1.

The details are shown in Box A, and a sample of the result is illustrated in Fig. 8.2.

Box A: Setting up the model

Explanation	Code
Agents are cars	`to setup`
Agents' attributes	`breed [cars car ]`
	`cars-own [speed queuing]`
Set everything to zero	`to setup`
Make the road.	`  clear-all`
	`  ask patches`
	`    [ if ( pycor < 1 )`
	`    and ( pycor > -1)`
	`      [ set pcolor white ] ]`

(continued)

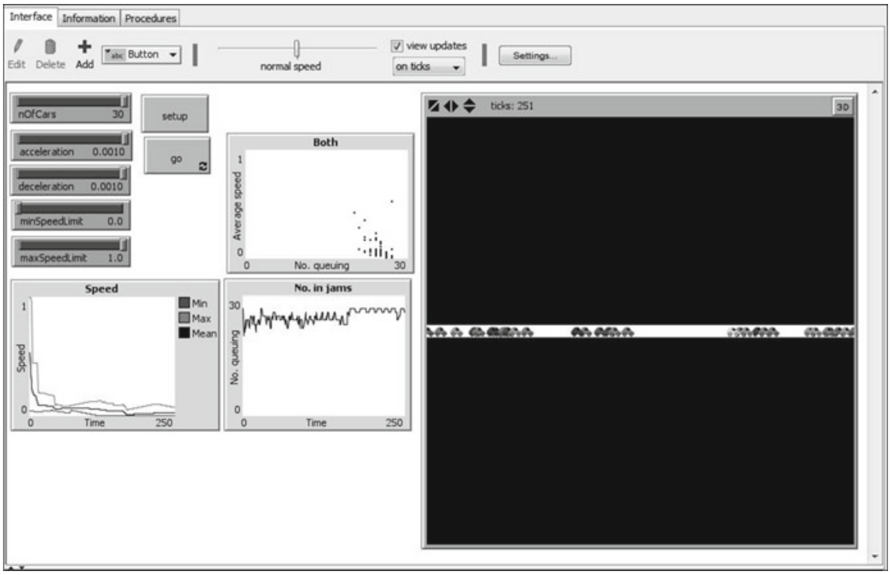
Box A: (continued)

Explanation	Code
Generate required number of cars	`create-cars nOfCars`
Set the cars' shape.	`ask cars`

Distribute them randomly along road. Set direction of movement.	`[ set shape "car"`
	`setxy random-xcor 0`
	`set heading 90`

Set speed randomly within the speed limits. Call up procedure.	`set speed`
	`minSpeedLimit +`
	`random-float`
	`( maxSpeedLimit )`
	`separate-cars`
	`]`
	`end`

Procedure to ensure only one car occupies the same patch of road.	`to separate-cars`
	`if any? other cars-here`
	`[ fd 1`
	`separate-cars ]`



nOfCars 30
acceleration 0.0010
deceleration 0.0010
minSpeedLimit 0.0
maxSpeedLimit 1.0

Fig. 8.2 Road with cars distributed randomly along it

Next the dynamic processes must be defined. All the cars move forward in the same direction. If the drivers see another car not far in front, they decelerate, at a rate set by the slider on the interface (decelerate), and if they catch up with the vehicle in front, slow to its speed, which may require rather abrupt deceleration! If they see no car within a specified distance, they accelerate again, set by a slider on the interface (accelerate). The rate of acceleration is small but sufficient to allow the cars to speed up to the maximum speed limit if the road is clear. Both deceleration and acceleration are allowed to vary between 0 and 0.001 in increments of 0.0001. The simulation is halted after 250 steps. The details are shown in Box B.

Box B: Running the model

Explanation	Code

Stop the program after 250 steps.	`to go`
	`  if ticks > 250 [ stop ]`

Reset queuing indicator at start of each step	`ask cars [ set queuing “No” ]`
	`ask cars`
	`    [ if any? cars-at 1 0`
	`      [ set speed`

If a car catches up with the one in front it slows to match its speed.	`       ( [speed] of one-of cars-at 1 0 )`
	`       set queuing “Yes” ]`
	`  ]`
	`ask cars with [queuing=“No” ]`
	`  [ ifelse`

If there is no car immediately in front but there is one a little further ahead, the car decelerates. Otherwise, it accelerates.	`  any? cars-at 5 0`
	`    [ set speed speed - deceleration ]`
	`    [ set speed speed + acceleration ]`
	`  ]`
	`    ]`
	`ask cars`
	`    [ if speed < minSpeedLimit`
	`      [ set speed minSpeedLimit ]`

To keep the cars within speed limits.	`    if speed > maxSpeedLimit`
	`      [ set speed maxSpeedLimit ]`

Cars move forward at the speed determined.	`fd speed`
	`  ]`

Time moves forward.	`tick`

Stage 5: Verifying and validating
To verify and validate the model requires outputs to be produced. Here three graphs are drawn:

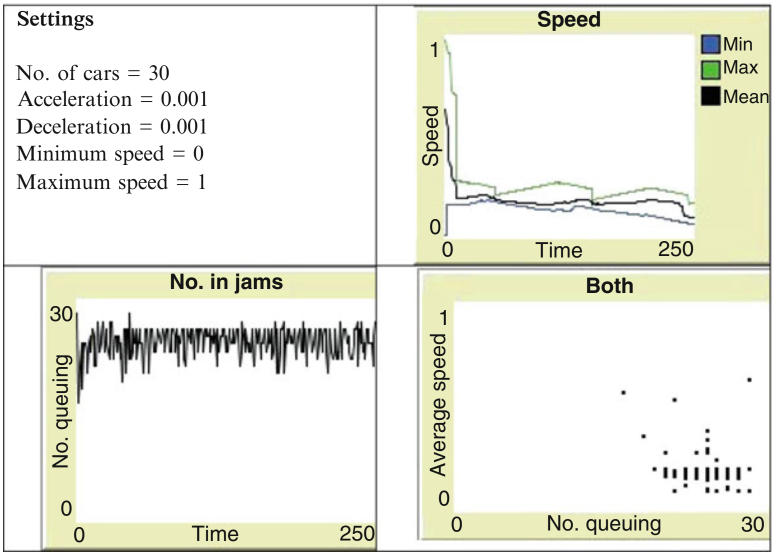
- to show the minimum, average and maximum speeds
- to show the number of queuing cars, and
- to plot the number queuing against the average speed.

The details are in Box C.

Verification and validation are discussed in Sect. 8.3.5 above and in detail in Ngo and See (2012). In this example, one simple method of verification is setting the minimum and maximum speeds to the same value and checking that all the

Box C: Generating the output (continuing the 'go' procedure)

Explanation	Code
Name the plots	`  plot-speed`
	`  plot-jams`
	`  plot-both`
End the "to go"	`end`
Plots minimum,	`to plot-speed`
mean and maximum	`  set-current-plot "Speed"`
speeds.	`  set-current-plot-pen "Min"`
	`    plot min [speed] of cars`
	`  set-current-plot-pen "Mean"`
	`    plot mean [speed] of cars`
	`  set-current-plot-pen "Max"`
	`    plot max [speed] of cars`
	`end`
Plots the number queuing.	`to plot-jams`
	`  set-current-plot "No. in jams"`
	`    plot count cars with`
	`    [ queuing = "Yes" ]`
	`end`
Plots the mean speed	`to plot-both`
against the number	`  set-current-plot "Both"`
queuing.	`    plotxy`
	`      count cars with`
	`      [ queuing = "Yes" ]`
	`      mean [speed] of cars`
	`end`


Settings

No. of cars = 30
Acceleration = 0.001
Deceleration = 0.001
Minimum speed = 0
Maximum speed = 1

Fig. 8.3 Sample of results

drivers do adopt the same speed. By watching the movement of the cars on the screen, it can be seen that, for example, there is no overtaking, as there should not be. Also the queuing status of individual cars can be checked: if there is no car immediately in front, it should not be "queuing".

Even a simple model like this can produce a wide range of scenarios and reproduce observed characteristics of traffic flows. For example, Fig. 8.3 shows what can happen if the road is near full-capacity with 30 cars, speeds are allowed to vary from 0 to 1, and drivers accelerate and decelerate at the maximum rates. The top right plot shows that the maximum speed drops quickly, but maximum, average and minimum speeds fluctuate. As a result, the number queuing constantly changes, albeit within a small range, as shown in the bottom left hand panel. However, a reduction in the number queuing does not necessarily increase the average speed of the traffic: the bottom right hand panel shows that there is no clear relationship between the average speed and the number queuing.

8.4.2 Segregation Model

The segregation model can be found in the Social Science section of NetLogo's library (Wilensky 1997b).

Stage 1: Identifying the research question
As explained in Sect. 8.1.2, Schelling tried to explain the emergence of racial residential segregation in American cities. The main research question of Schelling's

models can be formulated as: *can segregation be eliminated (or reduced) if people become more tolerant towards others from different ethnic/racial groups?*

Stage 2: Reviewing the literature
Theories of intergroup relations (Sherif 1966) are relevant when discussing the emergence of residential segregation. Some of these theories are Social Identity and Social Categorization Theories (Tajfel 1981), Social Dominance Theory (Sidanius et al. 2004), and System Justification Theory SJT (Jost et al. 2004). The Contact Hypothesis (Allport 1954), which implies that inter-group relations decrease stereotyping, prejudice and discrimination, is also relevant. Reviewing literature on how to measure segregation is clearly essential (Massey and Denton 1988).

Stage 3: Model design
As explained in Sect. 1.2.2, the environment is a city that is modelled by a square grid of cells each representing a dwelling. A household (agent) would be 'happy' to stay at its place as long as the proportion of its neighbours of the same colour as itself is not less than its threshold of tolerance. Agents keep changing their places as long as they are not happy. Box D presents the complete code of the segregation model.[1]

Stage 4: Model implementation
Lines 1–30 of Box D initialize the model. The first line creates an agent type (breed in NetLogo's language) called 'household' to represent the main agent of the model. The attributes of agents (households) include the following (lines 2–7):

- *happy?:* indicates whether an agent is happy or not
- *similar-nearby:* how many neighbours with the same colour as the agent
- *other-nearby:* how many neighbours with a different colour
- *total-nearby*: total number of neighbours.

There are two global[2] variables (lines 8–12): the first is *percent-similar,* which is the average percent of an agent's neighbours of its own colour. This variable gives a measure of clustering or segregation. The second variable, *percent-unhappy*, reports the number of unhappy agents in the model. There are another two variables determined by sliders (so that the model user can change their values on each run as desired): the number of agents, *number*; and agent's threshold, *%-similar-wanted* (which is the same for all agents).

The *setup* procedure (lines 14–30) (which is triggered when the user presses the *setup* button, see Fig. 8.4) creates a number of agents (households), half black and half white, at random positions. The *setup* procedure also calls another two procedures: *update-variables* that updates the agents' variables, and *do-plots* that updates the model's graphs (both procedures will be explained later).

[1]There are minor differences between the code of the original model in NetLogo's library and the code presented here.

[2]*Global* variables are defined (or declared) outside any procedure, and they can be accessed or refer red to from any place in the program. In contrast, *local* variables are defined inside a procedure, and can be accessed only within this procedure. The variables *similar-neighbors* and *total-neighbors* (lines 75–76) are local variables.

Box D: Code of Segregation Model

```
Breed [households household]
households-own[
  happy?       ;indicates whether the house-
   hold is happy or not
  similar-nearby ;how many neighbours with
   the same colour as mine?
  other-nearby  ;how many neighbours with
   different colour?
  total-nearby  ;sum of previous two variables
]
globals[
  percent-similar ;average percent of a
   household's neighbours
       ;of the same colour as that household?
  percent-unhappy ;percent of the households
   are 'unhappy'
]

to setup
  clear-all ;clear any variables or plots from
   previous runs
  if number>count patches
    [ user-message (word "This pond only has
   room for " count patches " households.")
      stop ]
  ;; create households on random patches.
  ask patches [set pcolor 7] ;; patches are
   initialized in grey
set-default-shape households "square"
  ask n-of number patches
    [sprout-households 1
      [ set color black ] ]
  ask n-of (number / 2) households
    [ set color white ]
  update-variables
  do-plots
end

to go
  if all? households [happy?] [ stop ] ;keep
   running as long as
   there                       ;are unhappy
   agents
```

(continued)

Box D: (continued)

```
  move-unhappy-households
  update-variables
  tick
  do-plots
end

to move-unhappy-households
  ask households with [ not happy? ];; only
   moves unhappy agents
    [ find-new-spot ] ; find new patch to move to
end

to find-new-spot
  rt random-float 360
  fd random-float 10
  if any? other households-here
    [ find-new-spot ] ;; keep going until we
   find an unoccupied patch
  move-to patch-here ;; move to center of patch
end

to update-variables
  update-households
  update-globals
end

to update-households
  ask households [
    ;; in next two lines, we use "neighbors"
    to test the eight patches
    ;; surrounding the current patch
    set similar-nearby count (households-on
    neighbors)
      with [color=[color] of myself]
    set other-nearby count (households-on
    neighbors)
    with [color !=[color] of myself]
    set total-nearby similar-nearby+other-
    nearby
    set happy? similar-nearby>= ( %-similar-
    wanted * total-nearby / 100 )
   ]
end
```

(continued)

Box D: (continued)

```
72  to update-globals
73    let similar-neighbors sum [similar-nearby]
      of households
74    let total-neighbors sum [total-nearby] of
      households
75    set percent-similar (similar-neighbors /
      total-neighbors) * 100
76    set percent-unhappy (count households with
      [not happy?]) / (count households) * 100
77  end
78
79  to do-plots
80    set-current-plot "Percent Similar"
81    plot percent-similar
82    set-current-plot "Percent Unhappy"
83    plot percent-unhappy
84  end
85
86
87
```

The dynamic process (which starts when the user presses the *go* button, see Fig. 8.4) is implemented using a simple behavioural rule for an agent in this model: *IF I'm not happy THEN I move to another place.* As the *go* procedure (lines 32–38) shows, the simulation will continue to run until all agents became happy with their neighbourhood (or the user forces it to stop).

The model provides two plots to present the two global variables *percent-similar* and *percent-unhappy* visually. Figure 8.4 shows the user interface and plots of the segregation model.

Stage 5: Verifying and validating
Like the previous traffic example, a simple verification method is to use extreme values for the model's parameters. For example, when setting the agents' threshold, *%-similar-wanted,* to zero and running the model, no agents move as they are all happy regardless of the percentage of neighbours of the same colour. On the other hand, setting this parameter to 100 makes most of the agents unhappy and they keep moving from their places.

Regarding validation, the main objective of the basic Schelling model is to explain an existing phenomenon rather than to replicate an existing segregation pattern in a specific city, and the model was successful in this regard. It provides a plausible answer to a puzzling question: *why these segregation patterns are so*

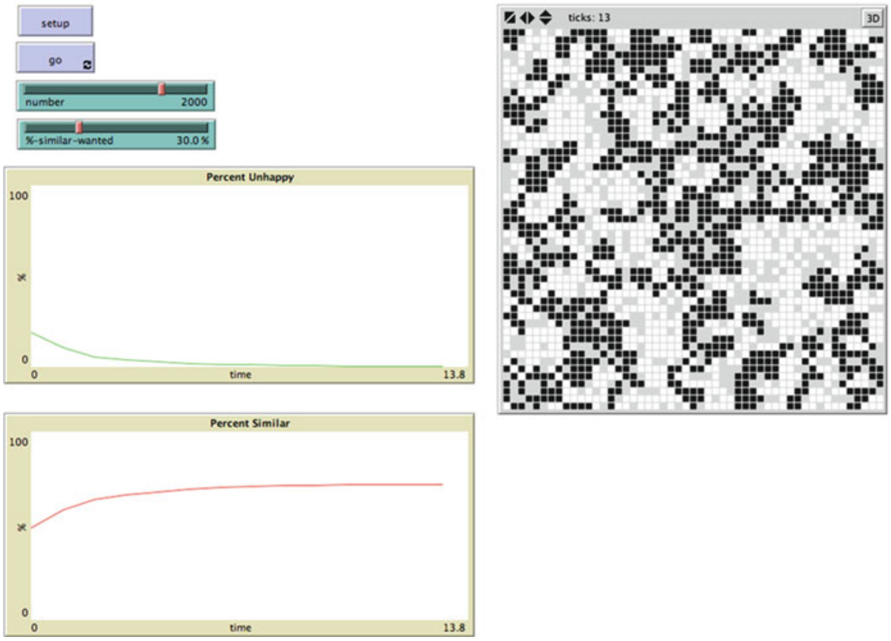


Fig. 8.4 User interface and plots of the segregation model

persistent regardless of the observed decline in ethnic prejudice. However, some attempts have been successful in extending the basic segregation model to replicate existing city segregation structures.

8.5 Conclusions

In this chapter, we discussed the process of designing and building an ABM. We recommended a set of standard steps to be used when building ABMs for social science research. The first, and the most important, step in the modelling process is to identify the purpose of the model and the question(s) to be addressed. The importance of using existing theories to justify a model's assumptions and to validate its results was stressed.

Recommended Reading

Edmonds, B., & Scott, M. (2005). From KISS to KIDS – An 'anti-simplistic' modelling approach. In P. Davidson, B. Logan, & K. Takadema (Eds.), *Multi-agent and multi-agent-based simulation. MABS, 2004*. New York/Berlin: Springer.

Epstein, J. M. (2006a). *Generative social science*. Princeton: Princeton University Press.

Epstein, J. M. (2008). Why model? *Journal of Artificial Societies and Social Simulation*, *11*(4), 12. Available at: http://jasss.soc.surrey.ac.uk/11/4/12.html

Gilbert, N., & Troitzsch, K. G. (2005). *Simulation for the social scientist*. Milton Keynes: Open University Press.

Gilbert, N. (2008a). *Agent-based models*. London: Sage.

Moss, S. (2008). Alternative approaches to the empirical validation of agent-based models. *Journal of Artificial Societies and Social Simulation*, *11*(1), 5. Available at: http://jasss.soc.surrey.ac.uk/11/1/5.html

References

Acerbi, A., & Parisi, D. (2006). Cultural transmission between and within generations. *Journal of Artificial Societies and Social Simulation 9*(1). Available at: http://jasss.soc.surrey.ac.uk/9/1/9.html

Allport, G. W. (1954). *The nature of prejudice*. Cambridge, MA: Addison-Wesley.

Balci, O. (1994). Validation, verification, and testing techniques throughout the life cycle of a simulation study. *Annals of Operations Research, 53*, 121–173.

Crooks, A. T., & Castle, C. (2012). The integration of agent-based modelling and geographical information for geospatial simulation. In A. J. Heppenstall, A. T. Crooks, L. M. See & M. Batty (Eds.), *Agent-based models of geographical systems* (pp. 219–252). Dordrecht: Springer.

Crooks, A. T., & Heppenstall, A. J. (2012). Introduction to agent-based modelling. In A. J. Heppenstall, A. T. Crooks, L. M. See & M. Batty (Eds.), *Agent-based models of geographical systems* (pp. 85–105). Dordrecht: Springer.

Epstein, J. M. (2006b). *Generative social science*. Princeton: Princeton University Press.

Epstein, J. M., & Axtell, R. (1996). *Growing artificial societies: social science from the bottom up*. Cambridge, MA: MIT Press.

Friedman-Hill, E. (2003). *Jesss in action: Rule-based systems in Java*. Greenwich: Maning.

Garson, D. G. (1998). *Neural networks: An introductory guide for social scientists*. London: Sage Publications.

Gilbert, N. (2008b). *Agent-based models*. London: Sage.

Gilbert, N., Pyka, A., Ahrweiler, P. (2001). Innovation networks: A simulation approach. *Journal of Artificial Societies and Social Simulation, 4*(3). Available at: http://jasss.soc.surrey.ac.uk/4/3/8.html

Gilbert, N. et al. (2006). Emerging artificial societies through learning. *Journal of Artificial Societies and Social Simulation, 9*(2). Available at: http://jasss.soc.surrey.ac.uk/9/2/9.html

Hutchins, E., & Hazlehurst, B. (1995). How to invent a lexicon: The development of sharedsymbols in interaction. In N. Gilbert & R. Conte (Eds.), *Artificial societies: The computer simulation of social life* (pp. 157–189). London: UCL Press.

Jost, J. T., Banaji, M. R., & Nosek, B. A. (2004). A decade of system justification theory: Accumulated evidence of conscious and unconscious bolstering of the status quo. *Political Psychology, 25*(6), 881–919.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33*(1), 1–64.

Macy, M., & Willer, R. (2002). From factors to actors: Computational sociology and agent-based modeling. *Annual Review of Sociology, 28*, 143–166.

Massey, D. S., & Denton, N. A. (1988). The dimensions of residential segregation. *Social Forces, 67*(2), 281–315.

Ngo, T. A., & See, L. M. (2012). Calibration and validation of agent-based models of land cover change. In A. J. Heppenstall, A. T. Crooks, L. M. See & M. Batty (Eds.), *Agent-based models of geographical systems* (pp. 181–196). Dordrecht: Springer.

Parry, H. R., & Bithnell, M. (2012). Large scale agent-based modelling: A review and guidelines for model scaling. In A. J. Heppenstall, A. T. Crooks, L. M. See & M. Batty (Eds.), *Agent-based models of geographical systems* (pp. 525–542). Dordrecht: Springer.

Schelling, T. C. (1971). Dynamic models of segregation. *Journal of Mathematical Sociology, 1*, 143–186.

Scott, J. (2000). *Social network analysis* (2nd ed.). London: Sage.

Sherif, M. (1966). *In common predicament; social psychology of intergroup conflict and cooperation*. Boston: Houghton Mifflin.

Sidanius, J., Pratto, F., van Laar, C., & Levin, S. (2004). Social dominance theory: Its agenda and method. *Political Psychology, 25*(6), 845–880.

Sun, R. (2006). The CLARION cognitive architecture: Extending cognitive modeling to social simulation. In R. Sun (Ed.), *Cognition and multi-agent interaction: From cognitive modeling to social simulation* (pp. 79–99). New York: Cambridge University Press.

Tajfel, H. (1981). *Human groups and social categories: Studies in social psychology*. Cambridge: Cambridge University Press.

Taatgen, N., Lbiere, C., & Anderson, J. (2006). Modeling paradigms in ACT-R. In R. Sun (Ed.), *Cognition and multi-agent interaction: From cognitive modeling to social simulation* (pp. 28–51). Cambridge: Cambridge University Press.

Terna, P. (1997). A laboratory for agent based computational economics. In R. Conte, R. Hegselmann, & P. Terna (Eds.), *Simulating social phenomena* (pp. 77–88). Berlin: Springer.

Wilensky, U. (1997a). *NetLogo traffic basic model*. Evanston: Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available at: http://ccl.northwestern.edu/netlogo/models/TrafficBasic

Wilensky, U. (1997b). *NetLogo segregation model*. Evanston: Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available at: http://ccl.northwestern.edu/netlogo/models/Segregation

Wilensky, U. (1999). *NetLogo*. Evanston: Center for Connected Learning and Computer- based Modeling, Northwestern University.

Wray, R. E., & Jones, R. M. (2006). Considering Soar as an agent architecture. In R. Sun (Ed.), *Cognition and multi-agent interaction: From cognitive modeling to social simulation* (pp. 53–78). New York: Cambridge University Press.

Ye, M., & Carley, K. M. (1995). Radar-Soar: Towards an artificial organization composed of intelligent agents. *Journal of Mathematical Sociology, 20*(2–3), 219–246.