# THE COMPLEXITY OF COOPERATION

AGENT-BASED MODELS OF COMPETITION AND COLLABORATION

*Robert Axelrod*

PRINCETON UNIVERSITY PRESS PRINCETON, NEW JERSEY

A beginner, however, should definitely pick one of the more modern languages.

3. Pascal was designed to be a first language for serious programmers. It is easy to learn, and is structured to encourage good programming habits. Most of simulations in this volume were programmed in Pascal.

4. C is the most common procedural language among serious programmers. It is designed to allow relatively easy conversion between one type of computer and other. It includes many shortcuts that a beginner need not learn. Unfortunately, the availability of these shortcuts can make understanding someone else's C code difficult. Besides the popularity and compatibility of C, another advantage is that it is the basis of the most popular object-oriented language, C++. An object-oriented language makes really large projects easier to program. It also makes it much easier to use portions of an old program in a new context. For all these reasons, C++ was chosen as the foundation for the Java programming language designed to be used over the World Wide Web.

Where do all these options leave the beginner? If you are a beginner to programming, my advice is to avoid FORTRAN and pick one of the others based upon what help is readily available. What you will need most is someone who can answer questions when you get stuck. If a friend or coworker is available to answer questions about Pascal, for example, pick Pascal. Alternatively, if you want to take a programming course and there is one available on C, then that would be a good choice. If the availability of help or instruction does not lead to a clear choice, then you can make a decision based on how serious you plan to be about programming. If you just want to try it out to get a feel for doing your own programming or because you have a simple idea you want to test, then Visual Basic is a good choice. If you are fairly sure that you will be doing programming for some time and want to start with a language that you can grow with, then C or C++ is the best choice.

## *Goals of Good Agent-Based Programming*

The programming of an agent-based model should achieve three goals: validity, usability, and extendability.

The goal of validity is for the program to correctly implement the model. (Whether or not the model itself is an accurate representation of the real world is a different kind of validity, which is not considered here.) Achieving validity is harder than it might seem. The problem is knowing whether an unexpected result is a reflection of a mistake in the programming or a surprising consequence of the model itself. For

example, in the model of social influence presented in Chapter 7, there were fewer stable regions in very large territories than there were in middle-sized territories. Careful analysis was required to confirm that this result was a consequence of the model, and not due to a bug in the program.[3]

The goal of usability is to allow you and those who follow to run the program, interpret its output, and understand how it works. You may be changing what you want to achieve while you do the programming. This means that you will generate whole series of programs, each version differing from the others in a variety of ways. Versions can differ, for example, in which data are produced, which parameters are adjustable, and even which rules govern agent behavior. Keeping track of all this is not trivial, especially when one tries to compare new results with output of an earlier version of the program to determine exactly what might account for the differences.

The goal of extendability is to allow a future user (including your future self) to adapt the program for new uses. For example, after writing a paper using the model, the researcher might want to respond to a question about what would happen if a new feature were added. In addition, another researcher might someday want to modify the program to try out a new variant of the model. A program is much more likely to be extendable if it is written and documented with this goal in mind.

## *Project Management*

In order to be able to achieve the goals of validation, usability, and extendability, considerable care must be taken in the entire research enterprise. This includes not just the programming, but also the documentation and data analysis. I often find myself wanting to get on with the programming as quickly as possible to see the output of the simulation. Whether I do the programming and documentation myself or use a research assistant, I tend to be too eager to see results. I have learned the hard way that haste does indeed make waste. Quick results are often unreliable. Good habits slow things down at first, but speed things up in the long run. Good habits help by avoiding some costly mistakes and confusion that can take a great deal of effort to unravel.

If you are just beginning to do computer simulation, it pays to build good habits. Your own needs will depend upon your programming experience and the demands of the project. The aim is to develop a set of

[3] Indeed, one of the contributions of the alignment exercise of Appendix A was to confirm the validity of the original program.

habits that are effective, with a minimum of administrative overhead. Here are some recommendations based upon my own experience:

1. Use long names for almost all of the variables rather than short names that will be incomprehensible a month later. It is fine to use *i* and *j*, or *x* and *y* for a few really common variables. Beyond that, however, the clarity of long names is worth the extra typing or cutting and pasting.

2. List all the variables at the start of the program. Some languages, such as Pascal and C, require you to declare all the variables and their type (e.g., integer or floating point) before you use them. Other languages, such as Visual Basic, make explicit declaration of variables optional. Force yourself to be explicit about declaring variables in advance of their use. This will save you many hours of debugging by warning you that a misspelled variable name deep in the code is illegal, rather than letting the compiler make the false assumption that it is meant to be a new variable.

3. Write helpful comments. For example, when you declare a variable, write a comment about what the variable is intended to represent, and how it is distinguished from related variables. Likewise, write comments in the body of the code to describe what each subroutine does and how it fits into the main program. It is a good idea to have as much text in comments as in code. Days or months later, these comments can be very useful.

4. Develop the sequence of programs so that they are upwardly compatible. This means that later versions should include all the useful features of old versions. Suppose, for example, that you start with a model that employs a 10 × 10 array of cells, each with exactly four neighbors. You can do this by having the map "wrap around," thereby making cells on the north and south edges into neighbors, and doing the same for cells on the east and west edges. Suppose that you later wanted to have a flat map with boundaries at the edges. You should not delete the portion of the code that deals with the neighborhoods on the original map. Instead, you should isolate that portion as a subroutine, and write another subroutine dealing with the neighborhoods on the new map. Then add a control parameter to serve as a switch specifying whether a given run uses the wrap-around map or the flat map. Doing it this way gives you the option to return to a wrap-around map at a later time.

5. Fully label the output. The output should include the time and date of the run, a sequential run number, the version number of the program, the settings of all the parameters that specify the nature of the run, and the random number seed if there is one. The output should allow you to replicate a run precisely. If the versions are upwardly compatible, as suggested earlier, then you will be able to replicate a previous run using any later version of the program.

6. Practice defensive programming. Taking a few minutes to be cautious can save hours of searching for a mysterious bug. As already mentioned, it pays to explicitly declare all variables so that some typos can be caught early, and it pays to use long variable names so that a month later you do not get confused about what a mysterious "XKLT" might mean. Another defensive move is to declare the intended limits of your variables so that you can use the automatic debugging features of your compiler. For example, if the Agent_X_Location should be between 1 and Max_X_Coordinate, then be sure that your compiler knows this, or that you confirm it in your code. Finally, using a programming technique called pointers is dangerous. Although pointers can be the most efficient way to program certain relationships, they should be used with great care because a mistake with a pointer can often cause bizarre symptoms that can be hard to isolate and repair.

7. Document each version of the code as you go. The documentation should include the exact specification of the model, a description of the algorithms used in the calculations, details about the inputs needed, and information about how to read the output. Explaining the output is particularly important because a column of data typically has such a brief label that you may not be sure what it means a month later. In many cases, the documentation of a new version can be as simple as the following: "Version 2.3 is the same as version 2.2 except an option is added to have a flat map. This is implemented by setting Flat_Map to True. If Flat_Map is False, then the map is wrap around as in previous versions." The documentation need not be elegant or concise. It does need to be complete and accurate. It pays to put some documentation into the program itself, but usually full documentation is more easily managed as a memo written with your word processor.

8. Use a commercial program to do most of the data analysis. Within your program do only simple data analysis, such as the calculations of averages, to reduce the amount of output that is required. Beyond that, it pays to use reliable software developed by others. For example, a good spreadsheet will allow you to manipulate the variables, do simple statistical analysis, and graph the data in different ways. An alternative to a spreadsheet is a program like Mathematica that includes not only statistical and graphics capability, but also a well-organized notebook structure to keep track of your work.

9. In validating the program, check the microdynamics, not just the aggregate results. Because agent-based models often have surprising results, you typically cannot confirm the code by checking its output against known results, as you could with a prime number generator, for example. Instead you need to confirm that your program is correctly handling the details. For example, you should check how a given agent be-

haves in the various possible circumstances. This will require interim reports that are much more detailed than you will usually need for the main data analysis.

10. If the modeler and the programmer are two different people, make sure the two of you understand each other at each step. It is surprisingly easy for two people to be confident they understand each other, and then find out much later that there was a subtle difference in what each thought they had agreed upon. Because it is not easy to validate an agent-based model, it is especially important that this sort of programming be based on thorough and accurate communication between the modeler and the programmer.

There is more to project management than good programming. You also need to develop systematic methods for archiving the output, analyzing the data, and interpreting the results. For example, it is a good idea to write memos to yourself as you go about your interpretation of particular runs. Like the documentation, these memos need not be concise or elegant, but they do need to be accurate. For example, suppose you are doing sets of runs to see what difference it makes whether the map is flat or wrap-around. When you get the results you can write a brief memo comparing a set of runs done one way with a set of runs done the other way. The memo should include the identifying information from the output, such as the version number of the program, the parameter settings, and the run numbers. This can easily be done by opening the output file from your word processor, selecting the required information, and pasting it into the memo you are writing. Then you can write your interpretation of the data, including a graph or two copied from your analysis of the comparative data. A series of memos such as these can serve as a "lab notebook" recording your progress in understanding the implications of your modeling decisions. Eventually, the memos can provide the basis for the data analysis section of your final report.

## Exercises

The best way to learn about agent-based modeling is to build and run some model. Here is a set of exercises that can help you get started. Each one can be done in a few days once you have mastered a programming language. These exercises are suitable for classroom use or independent study. The first three exercises can be done by relatively simple modifications of source code available on the Internet.[4]

[4] See the first footnote of this Appendix.

### *Exercise 1. Schelling's Tipping Model*

Thomas Schelling, who is best known for his work on deterrence theory, was also one of the pioneers in the field of agent-based modeling. He emphasized the value of starting with rules of behavior for individuals and using simulation to discover the implications for large-scale outcomes. He called this "micromotives and macrobehavior" (Schelling 1978).

One of his models demonstrates how even fairly tolerant people can behave in ways that can lead to quite segregated neighborhoods. This is his famous tipping model (Schelling 1978, 137–55). It is quite simple. The space is a checkerboard with sixty-four squares representing places where people can live. There are two types of actors, represented by pennies and nickels. One can imagine the actors as Whites and Blacks. The coins are placed at random among the squares, with no more than one per square. The basic idea is that an actor will be content if more than one-third of its immediate neighbors are of the same type as itself. The immediate neighbors are the occupants of the adjacent squares. For example, if all the eight adjacent squares were occupied, then the actor is content if at least three of them are the same type as itself. If an actor is content, it stays put. If it is not content, it moves. In Schelling's original model, it would move to one of the nearest squares where it would be content. For the purposes of this exercise, it is easier to use a variant of the model in which a discontented actor moves to one of the empty squares selected at random.

The exercise is to implement this model and explore its behavior. In particular, test one of Schelling's speculations about his tipping model. His speculation was, "Perhaps . . . if surfers mind the presence of swimmers less than swimmers mind the presence of surfers . . . the surfers will enjoy a greater expanse of water" (Schelling 1978, 153).

To make things concrete, assume there are twenty surfers and twenty swimmers.[5] Let the surfers be content if at least a third of their neighbors are surfers, and let the swimmers be content if at least half of *their* neighbors are swimmers. The actors can be numbered 1 to 40, and activated in the same order each cycle. Run the model for fifty cycles. Then for each of the twenty-four empty cells, write an *A* in the cell if more of its neighbors are surfers than swimmers, and write a *B* in the cell if more of its neighbors are swimmers than surfers. (It will probably be easier to do this step by hand than to automate it.) Then see if the surfers enjoy a greater expanse of water by seeing if there are more *A*'s than *B*'s. Do ten runs to get some idea of the distribution of results.

[5] Schelling made his speculation in the context of unequal numbers for the two types, but equal numbers are a good place to start.