Simulation Model Verification
An Introduction

Alan G. Isaac

American University

Implementation Testing

Verification	verifying that your code accurately implements your model
Prerequisite	an unambiguous model specification (hard to produce)
Rule 1	until the testing is done, the code is not finished
Rule 2	test as you implement; do not wait until you think you have "finished" your implementation
Rule 3	Finding the right trade off between time spent writing tests and time spent implementing is an art.

Syntax

The syntax of a language is the set of rules that determine the permissible structure of source code.
Violating these rules is a syntax error.
Such syntax errors will be detected for you: they prevent your source code from being parsed for compilation or execution.
You may also have access to a separate syntax checker for some languages. (E.g., the NetLogo `Code` tab includes a syntax checker.)

Syntax Errors

The following syntax errors will often but not always produce invalid code. (The first case is the good outcome, for they will fail to be parsed.)

- Typographical error (e.g., a misspelled variable or keyword, including omitted whitespace or, in case senstive languages, improper capitalization).
- Improper statement termination (e.g., no `end` in a NetLogo procedure definition or no semicolon at the end of a Java statement.
- Missing code block delimitation (e.g., missing indentation in Python, missing brackets in NetLogo, missing braces in C).
- Type error (e.g., assigning a string object to a variable that needs to be an integer, or providing an integer input argument to a function that requires a string input.)
- Redefining keywords (keywords are reserved by the language and cannot be redefined)
- Missing comment delimiters

Semantic Errors

The language may be misused if the meaning of its constructs is not properly understood.
Example: the NetLogo primitive `neighbors` returns an agentset of patches -- not a list, and not an agentset of turtles. Also, it does not return the patch of the calling agent.
Example: NetLogo lists are not like Python lists. NetLogo lists are immutable. Python lists are mutable.
Example: some languages use zero-based indexing (e.g., NetLogo, Python, Java). Others use unit-based indexing (e.g., Gauss, Matlab, Mathematica). When you request from a list or array the item with index `1`, you need to know whether you are requesting the second or first item.

Run Time Errors

Your source code may produce a program that runs -- i.e., has a legal syntax -- but also runs into problems.

- divide by zero error
- index error (trying to access an out of bounds index)
- overflow: a computation produces a number larger than the language can represent
- IO error: e.g., trying to write to a file without first opening it.

Logic Errors

Even if your program runs without raising errors, it may not be doing what you think. It may contain errors in logic that prevent it from correctly implementing your model.

Model Feedback

Implementing a model can reveal inadequacies in the specification.
Puzzling results can be exciting or can just be artifacts of

- implementation errors
- incorrect formulation of the model (i.e., the model itself needs more work before implementation)
- an inadequate model specification.

Syntax Checking

Find out if there is a syntax checker for your language.

NetLogo	Syntax checker in the `Code` tab.
Python	There are many options. One option is to ask Python to compile the script: `python -m compileall script.py`. Python's IDLE has a 'Check Module' (Alt-X) facility, which checks the syntax without running the code. Also consider `pylint` or `pyflakes`, although these do much more than syntax checking.

When syntax checking is cheap (i.e., consumes little time), do it often! (Perhaps every line or two.) Continually keep your code free of syntax errors.

Programming Practices: Dummy Procedures

Sometimes you will be developing a procedure `proc01` that calls another procedure `proc02`, which you have not written yet.
Write a temporary “dummy” version of `proc02` that allows you to appropriately test `proc01`.

Consistent Style

Does your language have an explicit style guide?
If not, are there standard style conventions?

Scaffolding

Here the term ’scaffolding’ refers to temporary code, or code that can be suppressed once the model is in use.

Scaffolding: Comments

Use `TODO` comments liberally, to indicate tasks you have not completed. E.g., if you write a dummy version of a procedure in order to test another part of your program, put a `TODO` comment in the dummy procedure that reminds you to implement the procedure body.

Scaffolding: Printing

A classic example of scaffolding is the liberal use of `print` statements (and other methods) to generate output that is useful for analyzing the functioning of the code. This practice is strongly recommended.

Example: if you write a dummy version of a procedure in order to test another part of your program, you might put a `print` statement in the dummy procedure that warns you that the procedure body has not yet been implemented.

Use `print` statements (or other logging methods) liberally. They can always be commented out or removed later. They can be very helpful to

- check intermediate results (e.g., the evolution of key variables)
- indicate tasks you have not completed.
- check that execution order is what you expect
- look for slow-executing code (e.g., by printing the time when you enter and when you exit a procedure)

Scaffolding: Conditional Printing

If code you need to share with users contains scaffolding you are not ready to remove, you may wish to disable it. For example, you could make printing conditional on the value of a global `debug` setting.

NetLogo

- declare a global boolean variable named `debug`
- `if debug print "test"`

Python

- assertions are disabled with the `-O` interpreter flag (or by setting `__debug__` to false)
- the logging module is very flexible: `http://docs.python.org/3/library/logging.html`

Logging

Note that if you are generating a lot of intermediate output from your scaffolding, you will probably find it convenient to direct it to a log file.

NetLogo	`netlogo-intro.xhtml#open-a-file-for-writing`
Python	`http://docs.python.org/3/library/logging.html`

Scaffolding: Assertions

If your chosen language supports assertions, use them liberally to check your program logic.
Not all languages give good support to assertion.

NetLogo `if (not asserted-condition) [error "error msg"]`
`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#error`

Python `assert asserted_condition, "describe failure conditions"`
`http://docs.python.org/3/reference/simple_stmts.html#grammar-token-assert_stmt`

Smaller Examples

Run your model for the smallest size that can reveal its behavior. Test your procedures/functions the same way.

Stress Tests

Test the behavior of your procedures for corner cases (e.g., 0 or 1 agents).
Use extreme values, outside the normal range of values you expect.
In addition to determining whether you code remains well behaved, you may have strong predictions for the model behavior when your parameters (or input data) take on extreme values. These predictions can be easier to test.

Modularity and DRY

Use modularity in your programming. Avoid intricate interdependencies between different parts of your code.
Modularity is connected to the DRY principle: "don’t repeat yourself". If you find yourself implementing the same idea multiple times, write a function or procedure that you call each time you need it. Make sure you have a single implementation. That way you will not debug your implementation in one location and accidentally forget to debug another implementation elsewhere in your code.

Visual Testing

Visual checking of model output is often very helpful for identifying anomalous results.
Humans are very good at noticing visual patterns. Are any patterns produced by your output unexpected? If unexpected, do they indicate problems?
Visual checking may involve time-series plots, histograms, or summary statistics. It may involve the visual display of attributes of model objects (e.g., coloring, resizing, or labeling patches or agents to represent key attributes). In spatial model, displaying the spatial location of mobile agents can be helpful.

Visual Testing: NetLogo

NetLogo hint: the `scale-color` primitive can be useful for attribute visualization
labels displaying attribute values can be easily attached to patches and turtles

If you are displaying information for many objects (e.g., many patches or many agents), it can be helpful to zoom the display to consider only a few of them.

NetLogo hint: use `inspect`

If you display dynamic visual information (e.g., the movement of agents, or the evolution of a histogram), it can be helpful to slow down the speed at which it displays.

Example

This example was provided by a student.
In a Gambler's Ruin simulation with random pairing, patches had a wealth attribute. He scaled patch color by wealth and noticed that diagonal patches were changing colors faster than the off diagonal patches. This led him back to his code, where he discovered that due to a subtle coding error, diagonal patches were more likely to be paired each round.

Debuggers

Debuggers are available for many languages. These allow you to step through the execution of you code, examining the change in variable values as you go.

NetLogo:	As of version 5, NetLogo does not yet have an integrated debugger.
Python	`http://docs.python.org/2/library/pdb.html`

Profilers

Profilers are available for many languages. These allow you to identify the processor time and memory use by different blocks of you code.

NetLogo: As of version 5, NetLogo provides modest profiling capabilities. `http://ccl.northwestern.edu/netlogo/docs/profiler.html`

Python `http://docs.python.org/2/library/profile.html`

Test Procedures and Programs

A test procedure is written to check model logic. It can be like a more complex assertion.

As long as aspects of your code remain untested, you should continue to write test procedures. Write as many as you have time for.

Sometimes testing your model will require you to produce output that you would not bother to produce without the motivation of the test. That is fine, as long as it does not cost you too much in terms of model complexity.

Sometimes you may find it helpful to debug part of a program by implementing it as a separate module that can be run an tested independently.

Document your tests. Add comments explaining precisely what problem with the model will be revealed if your test fails.

Note that if you write your tests as you implement your model, as you should, then existing test will sometimes catch new errors when you return to earlier code an attempt to improve it. So run *all* of your tests as you develop your code.

Code Review

Get someone to review your code. A famous maxim, known as Linus's Law, is often summarized by saying that, given enough eyeballs, all bugs are shallow. While this formulation is a bit simplistic, it makes an important point. Code review is a crucial part of software development.

Ask your reviewer to check whether it conforms to your model specification. They should also check for organization, readability, and comment adequacy. In short, is your code easy for others to read and understand?

Reabability is crucial and often undervalued. Make sure any names you introduce in your program are descriptive. It can often pay to sacrifice compactness (or even some execution speed) for clarity in the source code. Note that when you learn whether others can easily read and understand your code, you are probably learning whether *you* will easily read and understand your own code if you return to it after a month or a year. So even if you do not have access to someone to review your code, you should write it as if it will undergo code review.

Docking

It can be very helpful to independently implement parts of your model in another language or application. In the extreme case, you may do this for an entire model. This can even be part of the development process: you may pick one language for rapid prototyping of your model, and another another for production code implementing the model.
Example: Suppose you have written a complicated procedure to modify the state of an agent. Modify the procedure to write to a file the agent state when you call the procedure, and the agent state when the procedure completes. Read this data into another application (e.g., a spreadsheet), and confirm that the inputs and outputs match as they should.
See the example in RG 6.3.10