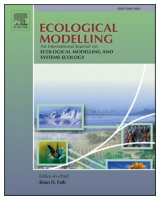
Ecological Modelling 520 (2026) 111669

Contents lists available at ScienceDirect

Ecological Modelling

journal homepage: www.elsevier.com/locate/ecolmodel



Testing NetLogo model code

Steven F. Railsback [a], Jocelyn Bliven [b], Volker Grimm [c,*], Jacob Kelter [d]

[a] Lang Railsback & Associates, 250 California Avenue, Arcata, CA, 95521 USA
[b] QSIDE Institute, qsideinstitute.org, Williamstown, MA, 01267 USA
[c] Department of Ecological Modelling, Helmholtz Centre for Environmental Research – UFZ, Permoserstraße 15, 04318 Leipzig, Germany
[d] NetLogo Center, Northwestern University, Evanston, IL, 60208 USA

ARTICLE INFO

Keywords:
Agent-based modeling
Code testing
Individual-based modeling
NetLogo
Software
Verification

ABSTRACT

In simulation modeling, good code testing practices provide (a) efficiency, by reducing time wasted using erroneous software; (b) credibility, by providing evidence that important errors are unlikely; and (c) understanding, by showing how complex model results arose. However, code testing receives little attention in agent-based modeling, and many modeling projects have been derailed by undetected code errors. We present code testing methods appropriate for models implemented in NetLogo by novice programmers. An efficient cycle for implementing new models includes: drafting a written model description and testing the "submodels" that each represent a separate process, coding the model while updating the written description to correct ambiguities and errors, and then thorough code testing. As the model is then used and revised, the written description, code, and code tests are all updated. We identify eight kinds of errors that are especially common in NetLogo software. Logic errors are the hardest to detect; doing so typically requires producing detailed test output for key submodels and then analyzing that output for unexpected results. We provide methods for producing test output, with example NetLogo code. Test output can be analyzed graphically, statistically, by tracing the fate of individual agents, or by comparing it to an independent implementation of the submodel. When unexpected results are found, a variety of "debugging" methods help determine whether they are caused by software errors and, if so, where the errors are. Documenting code testing methods and results is also key to model credibility and efficiency: documented tests provide evidence that the code is reasonably error-free and makes it easier to repeat tests as models are revised. While NetLogo currently lacks code-testing tools such as stepwise debuggers and unit testers, its powerful commands make it easy to add similar capabilities to models.

1. Introduction

Good code testing practices provide efficiency and credibility to the process of developing and using a simulation model, and therefore are a major concern in professional software development. Testing code throughout the modeling process has multiple benefits (considered more fully at Sect. 3, below), especially avoiding time wasted trying to understand and validate erroneous code and contributing to production of a complete and accurate model description. And, of course, being able to demonstrate that our tools do what we say they do—and that we know exactly what they do—is essential to credible science.

The benefits of code testing are even more important as we increasingly rely on artificial intelligence (AI) to produce model code. When a modeler uses AI to assist with, or entirely take over, code production, the code *may* be less likely to have errors but we cannot assume it is error-free. Just as important, we cannot assume that AI will identify and ask the modeller to resolve the inevitable ambiguities and errors in the model description that the AI is trying to implement, instead of simply making something up. Code testing of the kind we address here is essential for making sure AI-generated code is accurate and documented in sufficient detail.

Code testing seems especially important for mechanistic and agent-based models (ABMs), in part because they typically require larger and more complex programs. For simpler models that are more dependent on parameters fit to observations, minor programming errors may be compensated for and hidden by the calibration process. Mechanistic models, however, often include a variety of submodels for different mechanisms, only some of which might strongly affect results under

* Corresponding author.
E-mail addresses: Steve@LangRailsback.com (S.F. Railsback), jocelyn.bliven@gmail.com (J. Bliven), volker.grimm@ufz.de (V. Grimm), jacobkelter@northwestern.edu (J. Kelter).

https://doi.org/10.1016/j.ecolmodel.2026.111669
Received 10 April 2026; Received in revised form 13 May 2026; Accepted 15 May 2026
Available online 20 June 2026

specific conditions. Many parts of the code may be important only under unusual or extreme conditions, so mistakes in them may not be readily noticed or compensated for by calibration to more common conditions. But this reliance on model mechanisms is exactly why we need mechanistic models for many modern ecological management problems, for which predicting responses to novel and extreme conditions is exactly the model purpose (Railsback et al., 2025). Thorough code testing is therefore increasingly important as we use more complex, mechanistic ABMs to address more challenging problems.

Unfortunately, the importance of code testing does not appear to be fully recognized in the ABM culture. ABMs by nature produce complex, emergent results, and many kinds of results; and contain multiple "submodels" that each represent a mechanism that may be important only under some conditions. Consequently, their computer code is not easily verified by simple methods such as spot-checking a few calculations or making sure results look reasonable. However, new modelers rarely anticipate the costs of inadequate code testing; in our extensive experience supervising graduate students and other beginning ABM developers, many projects are extensively delayed, or even fail completely, because developers did not adequately search for hidden code errors. To confirm that code testing has received inadequate attention, we searched recent papers describing mechanistic models (not just ABMs and not only in NetLogo) in two journals that focus on modeling complex systems in social sciences (*Journal of Artificial Societies and Social Simulation*; 69 papers published in 2024 and 2025) and ecology (*Ecological Modelling*; ~160 papers published in 2025); search methods are described in Appendix A. We found only three papers that described how model code was tested or even explicitly mentioned code testing, and none of those used formal or documented tests. (Modelers that *do* test their code thoroughly are very likely to say so in publications, to enhance credibility.) From that result we can only conclude that many published studies are based on model software likely to contain undiscovered errors.

NetLogo (Wilensky, 1999) is a high-level software platform that has become very popular (Vincenot, 2018) because it makes agent-based modeling accessible to scientists, educators, and others who lack formal software training. Inattention to code testing among NetLogo users no doubt results in part to that lack of training: non-computer scientists are less likely to be aware of code testing methods and their importance. NetLogo users that are familiar with professional testing strategies and technologies may find them excessive or beyond their resources. Another factor discouraging code testing is that NetLogo provides many tools (mentioned below) for finding errors, which can give inexperienced users the illusion that undetected errors are unlikely.

A further explanation for inattention to code testing is the previous lack of emphasis and guidance in the NetLogo user documentation and ABM literature. While there is some general literature on software verification of ABMs (e.g., Gürcan et al., 2013), we found very little literature specifically on testing NetLogo model code. Swannack et al. (2025) provide valuable recommendations for developing and coding NetLogo models, but little specific guidance on testing code. The key resources are the "verification" discussion in Chapter 7 of Wilensky and Rand, 2015, and Chapter 6 of Railsback and Grimm (2019). This article is essentially an update and expansion of those two chapters.

We present software testing methods and tools that have proven practical, effective, and essential for ABMs implemented in NetLogo. This guidance is intended for both scientific and educational users, and in fact for anyone who needs to establish their model's credibility. Our goal is to promote the professional software world's culture of code testing—the understanding that all software contains errors until we look for them, and that modeling is much more efficient when we test code throughout its development—while using methods accessible to any NetLogo user.

Our topic, testing NetLogo model code, refers to finding and fixing unexpected and undesirable behaviors of a model implemented in NetLogo. Our primary focus is on when such behaviors result from programming mistakes—the task of making sure that a model's software faithfully implements the written description, which is often referred to as "software verification". But the objective of finding and fixing programming mistakes cannot be separated from three other objectives that are also important as we implement a model in software. First is model design: we need to find and fix model design errors, algorithms or assumptions that produce unexpected and unuseful results under some conditions. Second, model documentation: we often must produce a complete and accurate written description of a model (e.g., for publication, or for applications to policy or science), and writing and testing the model's software almost always identifies parts of the written description that are incomplete or ambiguous. Third, model analysis: once a model is implemented, we need to understand how it executes and why it produces the results it does—we build models to understand the modeled systems, so they are not useful unless we can understand how they execute. So while we focus primarily on finding programming errors, we also consider how the same methods are useful for finding model design problems, producing a complete written description, and understanding how models work. Testing model code is therefore not just a technical exercise but a key part of model design, understanding, and use. And because code testing is essential to the credibility of a model, it should be documented, ideally using the standard formats we consider here.

We do not address several other classes of activity related to model and software quality. One is "model validation," showing how useful a working model is for its intended purposes. Another is software management practices (e.g., code formatting, version control, and release management; Lemmen and Sommer, 2024). Model validation and software management are too important and broad to also address here.

We begin by briefly reviewing code testing methods used by professional software developers, to give NetLogo users an idea of how important the topic is and some familiarity with terminology. Second, we recommend good general practices for implementing models, suggesting an efficient process for going from a preliminary design to a fully documented and credible working model. Then we identify kinds of software errors that are especially common and important in NetLogo. In the next three sections we provide methods for producing model output specifically to search for errors, analyzing that output to determine if and under what conditions errors occur, and finding where in the code the errors occur. These methods include NetLogo code for producing three kinds of testing output files. Next, we discuss the value of documenting software tests and recommend simple but effective documentation methods. Finally, our conclusions include recommendations for technology development that could make NetLogo code testing more efficient.

2. Professional code testing practices and technologies

In professional software development and computer science, code testing is an important topic of training and research, which address both practices (strategies, methods) and development of technologies to make testing efficient and thorough. Beginner-friendly overviews of practices and technologies include IBM's on-line tutorial (IBM, 2025) and the Wikipedia article on software testing (Wikipedia, 2026). More thorough references include books such as Jorgensen (2008) and Myers et al., 2011.

Software testing practices are strategies for making testing more thorough and efficient. One of the most established practices is "unit testing," thoroughly exploring and testing individual code units (e.g., NetLogo procedures or multiple procedures that encode one submodel) by comparing their results to values known to be correct. For large projects, unit testing is followed by "integration testing," verifying that the units work together as desired. Another widespread practice, "test-driven development" (reviewed by Agha et al., 2023), prioritizes software quality by writing the test specifications and methods before writing the code, with the code designed to pass the tests. "Spectrum

based fault localization" estimates the probability of each program unit being faulty from the results of test cases and information on how the code executes (Sarhan and Beszédes, 2022).

Code testing technology (reviewed by Wong et al., 2016) has been developed since the earliest days of programming. "Debuggers," available for most programming languages, allow users to pause execution at "break points" in the code and then observe key variables as the code is executed one statement at a time. Unit testers have long been used to partially automate the unit testing practice; examples include JUnit (Gulati and Sharma, 2017) for Java and testthat for R packages (Wickham, 2011). "Omniscient debuggers" (Pothier and Tanter, 2009) record the events that happen during execution so users can trace execution to see when and why a particular unexpected result was obtained. This ability to trace execution could be especially valuable for ABM users trying to understand why models produced particular results; in fact, tools such as that of Ahlbrecht (2024) are designed to tell users of ABMs with complex decision-making behaviors why specific results arose. Finally, AI technologies are now thoroughly incorporated into software development tools, including some trained specifically to assist with finding and fixing code errors (e.g., Bajpai et al., 2024).

3. Good practices for implementing NetLogo models

Railsback and Grimm (2019; Chapter 1) present a "modeling cycle": an iterative process of designing, building, and applying models. Here we focus on one phase of that cycle: implementing the model in software. However, we also link model implementation to the previous phase—designing the model's structure and algorithms—and the subsequent phase of analyzing and learning from the model. This "model implementation cycle" (Fig. 1) is what we do once we have formulated the question or problem that the model addresses and assembled hypotheses for what we need in the model.

In this section we recommend good practices for the five steps in the model implementation cycle. These recommendations are based on our experience with many new models and modelers, and many correspond with those of Swannack et al. (2025). They have several critical benefits:

Efficiency. The cycle of model development, testing, and analysis we recommend helps avoid many problems that often cause modeling projects to take excess time and money. Those problems include (1) trying to develop a model by writing its code before thinking through its design (or even purpose) thoroughly; (2) conversely, spending too much time on model design and documentation before software implementation reveals problems and ambiguities; (3) frustration because a model appears not to work when the problems are due only to programming errors; (4) expending time, money, and reputation communicating model results that are later found invalid due to software errors; and (5) inefficient implementation and testing of later versions of the model.

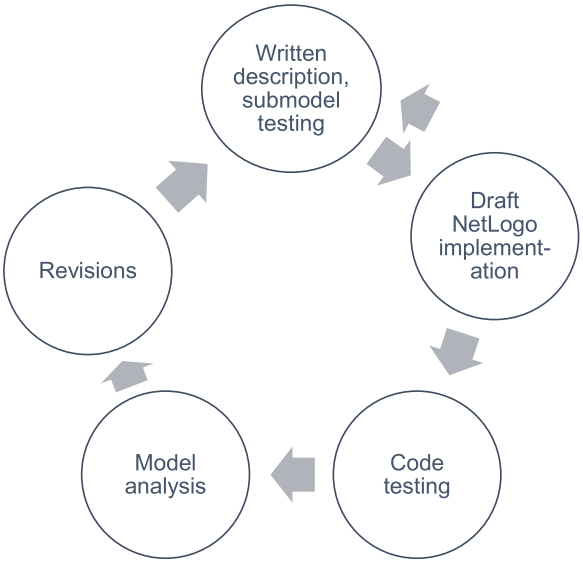


Fig. 1. The model implementation cycle.

Understanding. The methods and tools we recommend for testing software can also provide valuable understanding of how the model works, including under conditions that could not have been anticipated during model design. Understanding and explaining why an ABM produced the results it did is often a modeler's most important challenge, and impossible without understanding what goes on *inside* a model—how its different submodels behave. The approaches presented here can provide essential windows into a model's interior workings.

Credibility. Providing evidence that a model is useful for its purpose is a major challenge in simulation, especially for ABMs. Convincing evidence that (a) the software is free of errors, and (b) the model's behavior is well-understood, are critical steps toward credibility that we can always attain. For many models, further "validation," such as by comparing results to empirical data, is beyond our capabilities; in such cases, documentation of software testing may be the only evidence we can provide that we take credibility seriously.

3.1. Step 1: draft model description

The first step of this model implementation cycle is to write a draft model description in text and equations, not computer code. This step makes model development more efficient by encouraging the modeler to think carefully about the model before starting to develop software; once we start writing code it is natural to focus on programming problems instead of on a design that best meets the model's purpose. Further, written descriptions are almost always needed (e.g., for scientific credibility and publication, or professional applications), so drafting a description from the start does not add work but instead makes it less work to develop a complete model with both software and written description.

The "ODD" protocol for describing agent-based models (Grimm et al., 2020; see its supplement S1) provides a comprehensive checklist of model design decisions that should be made before starting software development. Those decisions include: What is the model's specific purpose? What kinds of entities need to be in the model, and what variables do they need? What are the model's spatial and temporal scales? What actions and behaviors do those entities need?

Drafting a model description includes designing the "submodels" that each represent a key behavior or other action of the agents or their environment. Each submodel that is not extremely simple or already well-understood in the literature is a substantial project by itself; it should be tested and explored thoroughly, by programming it by itself and seeing what results it produces under all conditions that could occur in the full model. These test implementations can be in spreadsheet software, or in platforms such as MatLab or R. Implementing each submodel independently this way allows the modeler to identify mistakes and ambiguities as soon as possible, calibrate and test submodels one at a time, and demonstrate that the submodel is useful. It is far easier to find and solve problems one submodel at a time than by analyzing the full model. Another reason to implement submodels independently in separate software is that those implementations can be used in software testing, as we discuss below (Sect. 6.4).

3.2. Step 2: initial NetLogo implementation

Step 2 is to develop the first version of the model software in NetLogo, by translating the written description into code. Another advantage of using ODD for the written description is that its elements translate easily and naturally into NetLogo code (Grimm et al., 2025).

This translation into code is never a one-way process. Step 1 is called "draft model description" because modelers rarely (and perhaps should

not) attempt to write a complete and final model description before starting to develop code. Writing the first version of the software here in Step 2 inevitably uncovers ambiguities, mistakes, and improvements in the model description: we forgot to say how variables are initialized in setting up, we did not foresee a condition under which an algorithm does not work, we find a NetLogo primitive that makes it very easy to program a submodel if we modify it in a trivial way, etc.

Two practices concurrent with writing code make subsequent model development steps more efficient. One is carefully updating the written model description as problems are identified and resolved, so the description remains up to date and complete, and exactly matches the software. The second practice is to test the code for each nontrivial submodel as it is written. Writing testing code (Sect. 5) along with the rest of a procedure makes it less likely the testing code introduces mistakes, and allows the procedure to be tested immediately. Testing each submodel as soon as possible, before the whole code is complete, lets any problems that affect other parts of the model be resolved before they cause additional effort.

At the end of this step, the modeler has produced a complete written description and working draft code for the entire model.

3.3. Step 3: code testing

As soon as a complete implementation of the model in NetLogo is complete, it should be tested thoroughly and the tests documented. Experienced programmers have the attitude that all new code contains mistakes and finding them as soon as possible makes development more efficient. (This step is also a good time to investigate improving execution speed; Railsback et al., 2017.) After completing this step, the modeler should have a complete and accurate model description, working and reliable software, and an archive of documented software tests.

3.4. Step 4: model analysis

Now the model is ready for its intended purposes, which typically require running many simulations, exploring and evaluating results, and communicating the model and conclusions drawn from it. Communication may include scientific publication, which typically requires publication of the software and written description; but communication may also be making a model available for others to use, e.g., in classrooms. However, model analysis may also reveal that revisions are needed.

3.5. Step 5: revisions

Models are often revised to try alternative assumptions or submodels, to address new questions or model different systems, to incorporate new knowledge, and even because increased computing power makes it possible to add detail. Model revisions can be made more efficient and less error-prone by producing new versions ("releases") with carefully updated written descriptions, documentation of where changes were made in the software, and updated tests of the revised code.

4. Typical errors in NetLogo software

Here we describe the kinds of software errors that seem most common or important in NetLogo models. We compiled these from the experience of the authors and their colleagues, who have taught and supervised hundreds of beginning modelers and also comprehensively tested large and complex models; Swannack et al. (2025) provide a similar list.

4.1. Syntax mistakes

Syntax errors are violation of NetLogo's rules for combining commands and symbols into code statements. Common examples are misspelling primitives or variable names, omitting or misplacing the brackets that many NetLogo primitives use (e.g., "`[...] of turtles with [...]`"), and failing to put spaces between variable names and mathematical symbols (e.g., using "`set age+1`" instead of "`set age + 1`").

NetLogo's syntax checker (the "Check" button) is extremely good at finding these mistakes, so they are relatively rare in code that passes the syntax check. However, at least one kind of syntax error cannot be caught by the checker: misplaced brackets and parentheses. NetLogo, like many languages, allows the use of parentheses to control the order in which primitives are executed in the complex statements that make its language powerful; and whether a primitive or variable is inside or outside of brackets has strong effects. For example, these two statements produce different results:

```
let near-turtles other [turtles in-radius 3] of one-of turtles with [color = red]
let near-turtles [other turtles in-radius 3] of one-of turtles with [color = red]
```

because "`other`" applies to different agentsets.

4.2. Cut-and-paste errors

Undetected errors often arise when we copy and paste statements to re-use them. For example, if we want to initialize turtle locations by placing them at random locations mostly near the center of the space, we could use a normal distribution with a mean of 0 and standard deviation of 1/3 the world's extent. This statement sets each turtle's X coordinate:

```
set xcor random-normal 0 (world-width / 3)
```

Then, to set the Y coordinate, we simply copy that statement, paste it, and modify it:

```
set ycor random-normal 0 (world-width / 3)
```

These statements will produce the desired result as long as the world is square, but because we neglected to change "`world-width`" to "`world-height`" in the second statement, it will not produce the desired result when the world is not square.

4.3. Context errors

By "context errors" we refer to when code is executed by the wrong agents. One kind of context error is when one type of agent (observer, patch, turtle, or link) is told to execute a NetLogo primitive that can only be executed by a different agent type. (The NetLogo dictionary makes it clear, for example, that "`forward`" can only be executed by turtles: only turtles can move.) The NetLogo syntax checker finds many of these errors, though it does not always make it clear exactly where the mistake is. The checker often produces an error where a procedure is called, indicating that the procedure cannot be executed by a turtle (or patch, etc.) because the procedure is patch-only (or turtle-only, etc.); the actual mistake is that somewhere within that procedure is code that only patches can use.

Other kinds of context errors are not reliably found by the syntax checker. One very common beginner mistake is using "`ask`" when we should not. When code is in observer context and we want it to cause all turtles to execute a block of code, we use "`ask turtles [...]`" to do so. However, when already in turtle context, we do not need or want to use "`ask turtles`" to cause the current turtle to do something; doing so would cause all the turtles to ask all the turtles to do something, instead of each turtle doing it just once. To avoid that problem, NetLogo does not allow agents other than the observer to use "`ask turtles`" or "`ask patches`". However, if a model uses breeds of turtle, such as the rabbits in the NetLogo library's Rabbits Grass Weeds model, an agent is allowed by the syntax checker to use "`ask rabbits`". A new user might

mistakenly use "ask rabbits" in the "eat-grass" and "eat-weeds" procedures (shown below, Sect. 5.3), thinking that we must use "ask rabbits" any time we want a rabbit to do something. However, these procedures are already in rabbit context because they are called from an "ask rabbits" statement in the go procedure. If we used "ask rabbits" in the "eat-grass" procedure, all rabbits would each be asking all rabbits to eat. One sign of this error is that the model executes very slowly.

Another kind of context error is when the wrong agent of the correct type executes code. As a simple example, a beginner may think that "[color] of myself" refers to the color of the turtle executing that code, but "myself" instead refers to another turtle—the one that caused the turtle to run the statement. NetLogo's primitives that change the context (cause some other agent to execute something), such as "of", "with", "ask", and "create-turtles", are powerful and essential but they require us to be careful about who executes what. (The difference between the two "let near-turtles" statements at the end of Sect. 4.1 is due to context: in the first, "other" is executed by the agent executing the whole statement, while in the second statement, "other" is executed by the one red turtle identified by "one-of".)

4.4. Misused primitives

NetLogo's hundreds of primitives are invaluable, but using the wrong one or misunderstanding what they do is common (Swannack et al., 2025; an example is below at Sect. 6.4). In almost all cases, the NetLogo dictionary provides a complete description of a primitive's behavior—if we take the time to read it carefully.

4.5. Incorrect world settings

Some important characteristics of a NetLogo model are defined in the dialog opened by the "Settings" button on the Interface tab. Incorrect settings, such as the location of the origin or whether "world wrapping" is turned on, can have very strong effects on model results yet be difficult to notice because they are not in the code.

4.6. Run-time errors

Run-time errors occur when a model is executing and the code tells the computer to do something it cannot. Common examples are dividing by a variable that sometimes has a value of zero, producing a number larger or smaller than the computer can store (for an example, enter "show exp 1000" in the Command Center), and trying to print to a file that has not been opened.

The example code above to set turtle locations using "random-normal" produces a run-time error when the world settings disable world-wrapping: "random-normal" can produce values far from the mean, so this code occasionally tries to place a turtle at a location outside the world boundaries. And this modification of the statements used above to illustrate the importance of brackets and parentheses:

```
let near-turtles [other (turtles in-radius 3)] of (one-of turtles) with [color = red]
```

causes a run-time error because "with" requires, to its left, an agentset but "(one-of turtles)" produces one turtle.

One of NetLogo's major advantages over standard programming languages is that it is extremely good at stopping execution when a run-time error happens and showing which statement caused it.

4.7. Mis-conversion of spatial and temporal scales

NetLogo's built-in spatial units (patches) and time units (ticks) and the primitives that use them are extremely convenient, but models that use specific spatial or temporal scales—e.g., one patch width represents 10 m and one tick represents one day—require the programmer to carefully convert between NetLogo's built-in units and the model's units. Failing to make these conversions consistently and correctly is a common mistake.

4.8. Logic errors

Logic errors are the ones that NetLogo's syntax checker and run-time error checking cannot identify: the model executes and produces results, but those results are incorrect because either the code does not accurately implement the model description or the model design has errors. The only way to find logic errors is to test the code, and the testing methods we address next are intended primarily to detect and find logic errors.

5. Code testing methods: producing test output

The process of testing software systematically includes: (1) looking for unexpected results; (2) determining whether such results are caused by programming errors, or result from problems in the model's design, or are instead unanticipated but valid model outcomes; and then (3) finding and fixing any errors. The most reliable way to find errors (and to understand how a model works) is to look at many different kinds of results. In this section we present ways to produce different kinds of results. Except for the first (using the View), these methods produce file output intended for systematic analysis using the methods presented in the following section. However, when it becomes obvious at any point that a model is not working as expected, it is best to immediately use the methods presented below (Sect. 7) to find and fix problems. (Those "debugging methods" include other temporary, informal ways of producing useful output.)

We emphasize producing comprehensive test output because many code errors are not widespread and common but instead affect only some agents on some time steps. It is very common for an error to affect results only in one instance (one agent on one tick) in tens of thousands. We cannot know whether such errors strongly affect the model's ability to meet its purpose unless we find them.

5.1. The View

NetLogo's View provides many ways to observe model behavior and potentially detect unexpected results. Kornhauser et al. (2009) provide guidance for designing the View to present as much useful information as possible; modelers can make it more likely that errors (and interesting valid results) are observed by using turtle and patch colors, shapes, and plots to provide information.

In our experience, new models always contain errors that are rapidly detected from the View, and often those errors are unlikely to be detected by any other method. Therefore, this testing approach is a critical prelude to the more quantitative methods we present below.

The main strategy in using the View to test software is to look for unexpected model states. For example, a student used red patches to represent permanent predation refuges for simulated fish, and noticed that the red patches gradually disappeared over long simulations. That observation quickly led to discovery of an error: patches were randomly chosen, at random times, to contain food, which was indicated by turning them green. When the food was consumed, the patch color was turned back to black. Occasionally food was placed in a red refuge patch, which was then turned from green to black, not back to red, when the food was gone. This error would not have been detected and resolved quickly without the View.

This kind of "visual debugging" (Grimm, 2002) has its pitfalls: mistakes in setting up visualizations in the View or plots, or in update statements, can make it look like the model is behaving strangely when only the visualization itself is wrong. When a plot indicates unexpected model behavior, the first step should be to make sure the error is not in the View settings or plotting code. NetLogo's histograms are especially useful, but challenging to set up so that all results are displayed properly;

the X axis range must be set manually (even if the "Auto scale x-axis?" option is selected) and if that range is too narrow, some values will not be displayed.

5.2. Defensive programming

Defensive programming is the practice of anticipating where errors could occur and then adding code to stop the model and provide information when an error, or other unexpected results, occurs. For example, consider a model in which the turtle agents have a feeding submodel that determines how much energy they acquire each tick. If the model is designed so that energy acquisition should never be negative, but the turtles seem to be losing weight, the modeler could add code like this to the end of feeding procedure:

```
if energy-intake < 0
[
inspect self
error "Energy intake is negative"
]
```

The statement "`inspect self`" opens one of NetLogo's agent monitors that will display all the turtle's variable values. The "`error`" statement then halts model execution and tells the user that an incidence of negative energy intake occurred. This information will at least provide clues about what causes the unexpected result. (The "`error`" statement takes NetLogo to the code tab to show the user where the statement is, which hides the agent monitor. Simply click back to the Interface tab to see the newly opened agent monitor.)

Defensive code such as this can be used generously throughout a model—professional programmers very commonly do so—and left permanently. It usually has negligible effect on execution speed.

5.3. Events output

The previous two methods for producing output for identifying unexpected results rely on the user watching the Interface for evidence of problems, but the rest of the methods produce file output that can be systematically analyzed (using methods presented below) for evidence of errors. We call the first method "events output" because it produces file output describing the conditions under which specific model events occur. Events output can also be very useful for analyzing routine model results.

The idea of events output is to make it very easy to record various events within a model by adding one short statement. For example, if we want to know how frequently rabbits eat grass and weeds (in NetLogo's Rabbits Grass Weeds library model) we can add two statements (in bold type) that call a procedure called save-turtle-events:

```
to eat-grass  ;; rabbit procedure
  ;; gain "grass-energy" by eating grass
  if pcolor = green
  [ set pcolor black
    set energy energy + grass-energy
    save-turtle-event "ate grass"
  ]
end

to eat-weeds  ;; rabbit procedure
  ;; gain "weed-energy" by eating weeds
  if pcolor = violet
  [ set pcolor black
    set energy energy + weed-energy
    save-turtle-event "ate weeds"
  ]
end
```

The procedure "`save-turtle-event`" produces a line of file output each time it is called; that line identifies the turtle that executed the procedure, reports selected variables of the turtle, and reports the event name (here, "`ate grass`" or "`ate weeds`"). Similar procedures can be used to record events of patches and links. (Even though "`save-turtle-event`" can be called by any breed of turtle and report the turtle's breed, it does not appear possible for one procedure to save both turtle and patch events.)

The events output can look like this:

Tick	Breed	Who	X-cor	Y-cor	Energy	Event
0	rabbits	38	1.116889	−15.6463	6.5	ate grass
0	rabbits	10	16.41889	6.792568	13.5	ate grass
0	rabbits	72	−19.5481	−9.79781	2.5	ate weeds
0	rabbits	95	−19.1844	−15.519	5.5	ate grass
1	rabbits	36	−5.78991	5.245402	5	ate grass
1	rabbits	38	0.977715	−16.6366	11	ate grass
1	rabbits	66	−14.1871	15.58063	1	ate weeds
1	rabbits	128	0.892494	11.13374	13	ate grass
1	rabbits	123	1.409061	4.025228	8	ate grass

Example code for producing events output, readily adapted to any model, is provided in Appendix B. It is very easy to change which events in a model are reported, to test and understand different submodels.

5.4. Procedure monitor

A procedure monitor is code that monitors and reports the values of selected variables at the start and end of a selected procedure, each time that procedure is called during model execution. It is a form of "model instrumentation" (Banks, 1998) that lets us see what is going on inside the model.

Procedure monitors have the advantage of showing what happens during real model runs as well as producing output for testing. Their main disadvantage, especially for procedures called many times each tick (which are especially important to test), is that they can produce very large output files very rapidly. The example below, in a typical simulation, produces ~20,000 lines of output per tick, so runs of even 50 ticks produce files too large to open in standard spreadsheet software. Consequently, test runs must be short and therefore unlikely to include wide ranges of the variables driving a procedure, unless we set up special test model runs designed to include wide ranges of conditions within a few ticks. This method therefore may not be reliable for finding errors that occur only under unusual conditions.

At Appendix C, we provide code implementing a procedure monitor for one procedure. The example monitor code uses:

- A global Boolean variable "`PM-on?`" that determines whether the procedure monitor is active, implemented as a switch widget on the Interface;
- Global variables "`PM-row`" and "`PM-output`", both lists of values to be written to the output file;
- Procedure "`PM-setup`", which must be customized to provide the name and column headings of the output file;
- Procedures "`PM-add`" and "`PM-save-row`", which assemble and then save one row of output; and
- Procedure "`PM-write-output`", which is called from the go procedure to write to the output file once per tick.

Multiple monitors for different procedures can be used by duplicating the code in Appendix C, with separate copies of the PM- procedures (with different procedure names) for each monitored procedure. However, when monitoring multiple procedures it can be tidier to put the functions of "`PM-setup`" in the setup procedure, the functions of "`PM-add`" and "`PM-save-row`" in the monitored procedures, and of "`PM-write-output`" in "`go`" or another procedure executed once per tick.

The following code illustrates use of the procedure monitor. The

monitor is used to instrument a procedure in a trout model (Railsback et al., 2021) that calculates a trout's daily respiration (energy consumption) rate. Respiration has two components: "standard" respiration depends on the trout's weight and the water temperature, and is multiplied by an "activity function" that depends on the trout's length and swimming speed. In this example, several "PM-add" statements are used to create a list of output that contains all the variables that affect respiration and the resulting values of standard respiration and the activity function. (Procedure monitor code is in bold type.) The "PM-write-output" procedure (not shown here) then uses NetLogo's CSV extension to turn each list of output into a formatted output file row. This code produces an output file that looks like this:

5.5. Procedure explorer

The procedure explorer is designed to conduct a one-time, thorough exploration of how a NetLogo reporter, procedure, or other unit of code behaves over wide ranges of the variables that affect results. It is conceptually similar to BehaviorSpace in that the user identifies variables to vary and defines ranges of values for each, and the procedure explorer then executes the code for all combinations of those values. The main advantage of a procedure explorer, compared to the previous methods, it that it can produce test output over all possible conditions in the smallest possible file. Therefore, procedure explorers are especially useful for finding errors that arise only under some conditions. Producing and analyzing procedure explorer output is essentially a form of unit testing.

Unlike the procedure monitor code, the procedure explorer is programmed in a separate new NetLogo procedure that the user can execute via the Command Center or a button on the interface. The procedure explorer code must be customized for each application, but easily accommodates complexities such as the need to vary variables that belong to agents other than the one executing the procedure being explored.

We illustrate a procedure explorer using one designed to test two procedures in the NetLogo library's Flocking model: "average-flockmate-heading" is a turtle procedure that calculates the mean heading of "flockmates," an agentset of other nearby turtles; and "average-heading-towards-flockmates" calculates a heading toward the flockmates (the mean of the headings toward each flockmate). The flocking model with the full procedure explorer code is in Appendix D.

The heart of the procedure explorer code is a set of nested "foreach" statements that set the values of the variables that are varied. The "foreach" primitive executes a block of code once for each value of a list, with that value held in a local variable. The "range" primitive can be used to generate a list of values that are evenly spaced over a selected range.

The number of output lines is the number of combinations of all the variable values, calculated as the product of the number of values used over all the variables that are varied. The time needed to execute a

```
to-report respiration-for [ a-fish a-cell a-swim-speed ]
  ; Observer procedure to calculate total respiration for a fish
  ; Some equation terms are updated in update-habitat and update-trout

  ; *** PM
  ; Code to monitor variables affecting respiration
  PM-add (list
    ticks
    ([trout-species] of a-fish)
    ([who] of a-fish)
    ([trout-weight] of a-fish)
    ([trout-length] of a-fish)
    ([[reach-temperature] of patches-reach] of a-cell)
    a-swim-speed
    )

  ; Standard respiration
  let temp-term [reach-resp-temp-term] of ([patches-reach] of a-cell)
  let resp-standard ([trout-resp-std-wt-term] of a-fish) *
   (item ([trout-spp-index] of a-fish) temp-term)

  ; Activity resp. depends on ratio of swim speed to max swim speed
  ; If ratio is large (> ~25) then "exp" is too large to calculate
  ; so instead report an arbitrary large number.
  let the-swim-speed-ratio a-swim-speed /
      (max-swim-speed-for a-fish a-cell)
  if the-swim-speed-ratio > 20 [ report 999999 ]
  let resp-activity-func
    exp ((item ([trout-spp-index] of a-fish) trout-resp-D) *
     (the-swim-speed-ratio ^ 2))

  ; *** PM
  ; Code to monitor respiration results
  PM-add resp-standard
  PM-add resp-activity-func
  PM-save-row

  report resp-standard * resp-activity-func

end
```

procedure explorer is rarely a concern, but the size of the output file can be. If output is to be analyzed in Excel, for example, the explorer should be designed to avoid producing files too large to open in Excel, perhaps by running several experiments that each cover only some of desired ranges of variable values.

The following excerpt of the flocking model procedure monitor includes nested "`foreach`" statements executed after previous code sets up conditions under which the explorer's output is predictable. That code (a) identifies one turtle ("`test-turtle`") to execute the experiment, and (b) sets the flockmates of "`test-turtle`" to two other turtles. One flockmate ("`stationary-flockmate`") is set to a known location; in the experiment coded below, "`stationary-flockmate`" does not move but does change its heading. The other flockmate ("`moving-flockmate`") moves (in the "`ask moving-flockmate`" block below) in a circle around the center of the world.

The above code produces output like this:

Stationary turtle heading	Moving turtle xcor	Moving turtle ycor	Moving turtle heading	average-flockmate-heading	average-heading-towards-flockmates
0	0.871557	9.961947	5	2.5	153.2827
0	2.58819	9.659258	15	7.5	151.2086
0	4.226183	9.063078	25	12.5	149.5402
0	5.735764	8.19152	35	17.5	148.323
0	7.071068	7.071068	45	22.5	147.5454
0	8.19152	5.735764	55	27.5	147.164
0	9.063078	4.226183	65	32.5	147.1236

6. Code testing methods: analyzing test output

Now we turn to the question of how to analyze the file output produced by the above methods. The objective of this analysis remains to determine whether there are any unexpected model results that indicate the likely presence of code errors; in the following section we discuss ways to find and fix such errors. We examine four approaches of (in general, with many exceptions) increasing difficulty but increasing ability to identify subtle or rare errors. The first three of these methods are also valuable for understanding how a model works once it appears error-free.

All of these analysis methods require software other than NetLogo. We illustrate the methods using spreadsheet software, for reasons explained below, but the methods can also readily be implemented in other data analysis software such as MatLab and R. The "PivotTable" tool in Excel spreadsheet software (and similar tools in other spreadsheets) is very efficient for many of these analyses, although Excel's graphing capabilities can be frustrating.

6.1. Graphical analysis

Sometimes it is possible to test all or part of a submodel by graphing test output. Graphical analysis can be either general and qualitative or rigorous and quantitative.

Qualitative graphical analysis can show whether output of a submodel does or does not reproduce expected trends and ranges in results. Contour plots are especially useful because they show how results respond to two variables. For example, Fig. 2 was generated from the procedure monitor results illustrated above (Sect. 5.4). It was produced by plotting how total respiration (standard respiration × activity function) varies with both trout length and swimming speed, at one temperature. The graph shows respiration increasing as expected with both size and swimming speed, with swimming speed becoming increasingly important as it increases. However, the vacant triangle in the upper left corner of the plot, and the sharp hooks at the upper end of some contour lines are exactly the strange result that modelers should investigate to see if they reveal a programming error, a problem with the submodel's design, or mistake or artifact in the test itself. (In this case, the strange results arise because small fish never use high swim speeds, so there were no results to plot in the upper left corner. The hooks are artifacts of the contour plot algorithm.)

It is also sometimes possible to test a submodel rigorously and

```
ask test-turtle
    [
      ; Loop over stationary flockmate's heading
      foreach (range 0 360 10) [sta-heading ->
        ask stationary-flockmate [set heading sta-heading]

    ; Loop over moving flockmate's location and heading
        foreach (range 5 365 10) [move-heading ->
          ask moving-flockmate
          [
            setxy 0 0
            set heading move-heading
            fd 10
          ] ; end of ask moving-flockmate

          ; Now all the inputs have been set,
          ; so execute the procedures and write output
          file-print csv:to-row (list
                [heading] of stationary-flockmate
                [xcor] of moving-flockmate
                [ycor] of moving-flockmate
                [heading] of moving-flockmate
                average-flockmate-heading
                average-heading-towards-flockmates
          ) ; end of file-print list
        ] ; end of foreach move-heading
      ] ; end of foreach sta-heading

    ] ; end of ask test-turtle
```

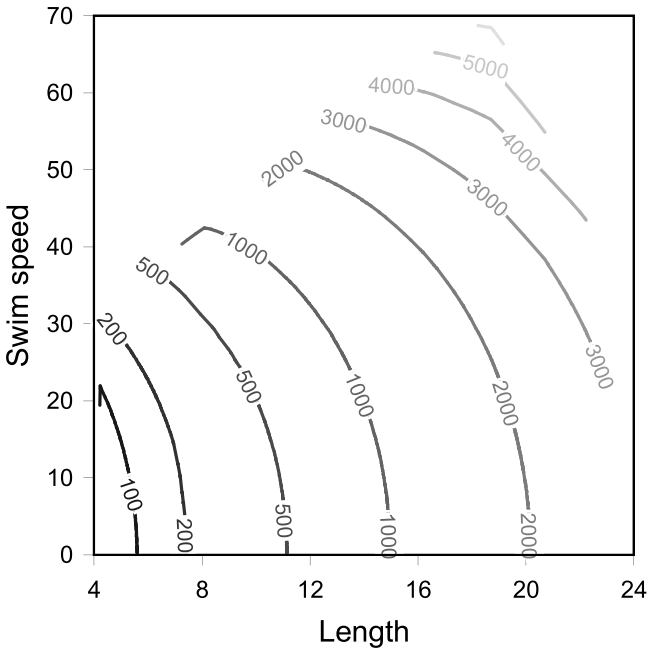


Fig. 2. Example contour plot for qualitative analysis of test output. The plot shows how total respiration rate varied with swim speed and length of a simulated trout, at a temperature of 14.9 °C, in procedure monitor output from the trout model of Railsback et al. (2021).

quantitatively using graphical analysis. We can use our eyes or regression analysis to determine whether test output from a submodel exactly reproduces the equations or relations specified in its written description. For example, the respiration submodel to which we applied a procedure monitor uses the equation: $R = aW^b \exp(cT^2)$ where R is standard respiration, W is trout weight, T is temperature, and a, b, and c are parameters with values of 36.0, 0.783, and 0.0020. If we select the procedure monitor output for trout over wide ranges of weight but all from a temperature of 14.6, we can graph standard respiration vs. weight (Fig. 3).

The regression curve shows that standard respiration output exactly fits the model description's equation for when T is 14.9 (because $a \exp(c14.9^2)$ equals 56.123). This analysis indicates that the NetLogo software correctly implements the relation between standard respiration and trout weight, though it does not test the submodel over a wide range of temperatures.

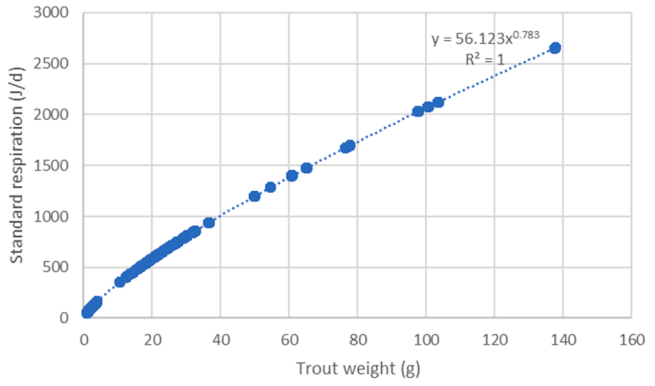


Fig. 3. Example quantitative graphical analysis of test output. The graph was produced in Excel as an XY plot with a power trendline.

6.2. Statistical analysis

In some cases, simple statistical analyses can indicate the presence of unexpected results; statistical analyses generally cannot show that model code is correct but can sometimes reveal errors. The general approach is to identify statistical relations that we expect the model to produce, and then see whether it does.

The relations tested by statistical analysis can be extremely simple. For example, in a population model in which turtles reproduce and die, we expect the number of turtles alive at the end of a simulation to equal the number created at setup plus the number born during the simulation minus the number that die. If we track how many births and deaths there are (using global variables or output like that illustrated above at Sect. 5.3) and find this expectation not met, then we know that something is wrong.

A slightly more complex example is provided by the events output. Because the Rabbits Grass Weeds model includes no rabbit behavior to prefer patches with grass instead of weeds, we can expect that the frequency with which rabbits eat weeds is linearly proportional to the frequency with which patches grow weeds. We can run a BehaviorSpace experiment varying the parameter **weeds-grow-rate** over the range (0 to 40) at which the rabbit population does not go extinct, and test this expectation. We can use an events output file or simple code that uses global variables to count the number of times weeds are grown and eaten and then reports their values at the end of simulations lasting 10,000 ticks.

The results of this experiment (Fig. 4) are promising, but not *exactly* as expected: the relation between the number of weed patches grown and number of times weeds were eaten was close to but not exactly 1:1, and in fact the number of weeds eaten was slightly but consistently less than the number produced. Is this small but unexpected discrepancy due to a programming error, or a legitimate model outcome?

One potential explanation of the discrepancy is that the weeds left at the end of a simulation are not considered in this experiment. We can simply add a count of those remaining weeds to our BehaviorSpace experiment (by also reporting `"count patches with [pcolor = violet]"` at the end of each simulation). Now we can test the expectation that, for each model run, the total number of weeds grown must exactly equal the number eaten by rabbits plus the number left at the end. When we re-run the experiment and test this explanation, we find

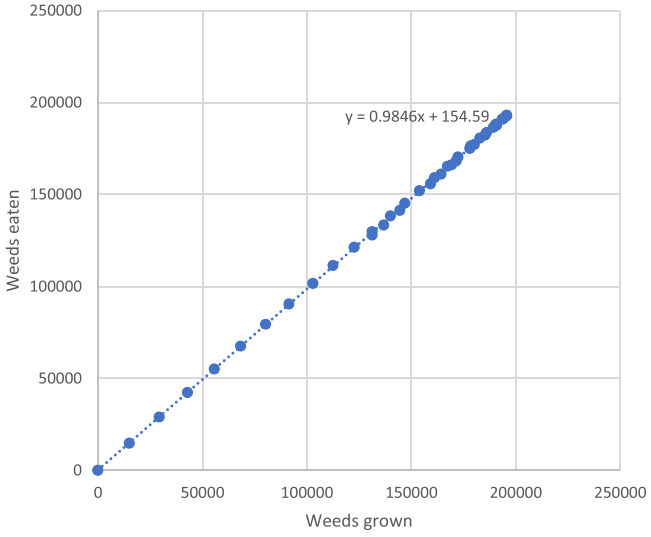


Fig. 4. Example statistical analysis of test output, from the Rabbits Grass Weeds model executed over a wide range of **weed-growth-rate** values. Over a long simulation, we expect the number of weeds eaten to very closely approach the number grown, but the regression statistics indicate that slightly fewer weeds were eaten than grown.

that it does not explain the missing weeds: the number of weeds grown always exceeds the number eaten plus the number remaining at the end.

When we look at the code for creating weed and grass patches, we see another potential explanation: patches that grow weeds can then replace the weeds with grass on the same time step. We cannot say that this is a programming error because the model description on the Info tab is not specific about exactly how grass and weed growth is represented (except for showing the code). But if this model design explains the discrepancy we are investigating, then we expect the number of "missing" weeds to increase linearly with the probability of growing grass instead of weeds. A further BehaviorSpace experiment confirms that the number of missing weeds appears proportional to the grass growth probability.

6.3. Execution tracing

By execution tracing we refer to analyzing what happened over simulated time in a simulation, often by following individual agents. For example, events output (Sect. 5.3) can be opened in a spreadsheet, then sorted by agent ("who" identifier for turtles; pxcor and pycor for patches) and tick. Then we can examine what events happened to one turtle or patch over time and how its characteristics changed (Fig. 5). Output from procedure monitors can also be analyzed in this way, to see how an agent changed among times it executed the monitored procedure.

Event tracing using events or procedure monitor output constitutes a simple form of "omniscient debugging" (Sect. 2), providing the ability to see what happened to specific agents over model execution and explain their fate. We can follow selected agents over time to see (for example) how they entered unexpected or erroneous states: did problems occur suddenly in one procedure, or did they accumulate over time? We can also search the output for specific events, for example to determine the time steps and procedures in which agents entered a specific state.

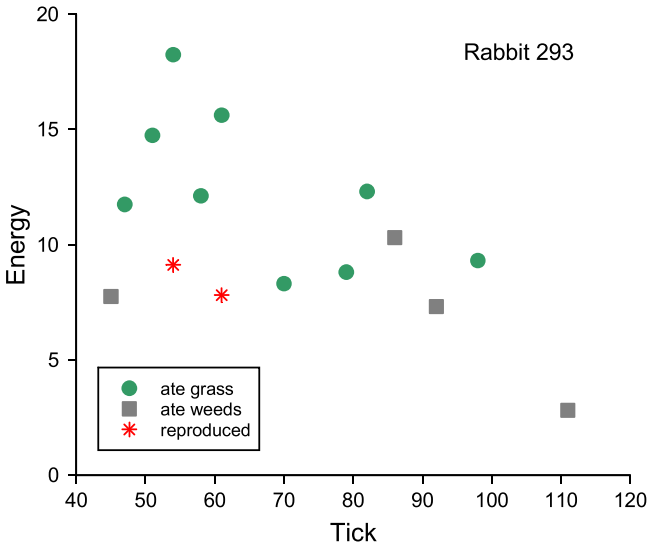


Fig. 5. Example execution tracing analysis of the rabbits grass weeds model events output. The events output file from one model run was sorted by rabbit "who" number and tick; here, the energy of rabbit 293 is graphed against tick, each time a recorded event occurred. From this execution trace, we can see that after being born at or before tick 45, this rabbit ate weeds once and then was lucky enough to eat grass three times, which provided the energy needed to reproduce at tick 54. It then ate grass two more times and reproduced again at tick 61. After that, it was less lucky and eventually died sometime after eating weeds at tick 111. The analysis provides evidence that the birth threshold of 15 energy units and birth cost of 50% of energy are implemented correctly.

6.4. Independent implementation

In professional software develoment, critical programs are tested by having separate teams code them two or more times and resolving the differences among the independent implementations. Often, the different implementations are in different programming languages to reduce the risk of making the same mistake in each. We could test NetLogo software by re-implementing the whole model in another software platform, but doing so would exceed the resources and skills of many NetLogo users. Another limitation of complete re-implementation is that differences in results among implementations can be due to differences in languages that are difficult to find and avoid yet unimportant; examples include differences in how random number generators are programmed and how floating point numbers are represented.

We can, however, get most of the benefit of independently re-implementing a NetLogo model, with relatively little effort, by re-implementing the model's key submodels. If we thoroughly test each submodel's code this way, we develop substantial credibility because the other important code, especially "scheduling" when each agent executes each submodel, is quite straightforward in NetLogo.

Testing submodels against independent implementations is important because most submodels are too complex to test thoroughly using only graphical or statistical analysis. Even very simple submodels can have mistakes revealed only by an independent implementation, as exemplified by the case presented by Railsback and Grimm (2019; section 6.4): a submodel with only four lines of code had an important error because it used a primitive ("`uphill`") that does not do exactly what the model design called for. This error, like many, was very unlikely to be revealed by other analysis methods.

Completely re-implementing each of a model's non-trivial submodels need not significantly add to the effort of developing a model. One key way to make this kind of test efficient is to use one independent code first to develop, test, and calibrate the submodel during model design (the "Draft model description" step; Sect. 3), and then to test the final NetLogo code. This approach is essentially "test-driven code development," a software development technique in which the test for a piece of code is written before the code itself is.

The independent submodel implementations can be in any suitable software that users are familiar with, e.g., MatLab, R, or a spreadsheet. Spreadsheet software has several advantages:

- It is common and many people are experienced users;
- Its format is quite different from NetLogo, making it less likely that the same mistakes are made in both implementations;
- While it can be clumsy to program complex algorithms in a spreadsheet, it is rarely impossible to reproduce a submodel;
- Spreadsheets display all variable values and calculations instead hiding them in arrays, etc., which makes mistakes easier to find;
- Spreadsheets make it easy to put the test output from NetLogo side-by-side next to the second implementation, then calculate the difference in results for each row of output, and then use sorting and searching to find important differences; and
- Spreadsheets are easy to format and annotate to document the tests (considered at Sect. 8).

An example independent implementation analysis, for a trout feeding and growth submodel (Railsback et al., 2021), is at Appendix E. It is in an Excel spreadsheet file. The first 22 columns of the file (columns A-V) are imported NetLogo output of a procedure monitor applied to the feeding and growth submodel, for 10,000 times the submodel was executed during a simulation. This output includes the values of variables affecting how a trout feeds and the growth it obtains (columns A-L, shaded grey); intermediate results of the submodel, including the rates at which the trout would acquire and expend energy using each of three feeding modes (columns M-U); and its final result, the feeding mode that provides highest growth (column V).

The second set of columns (X-AT, shaded green) implement the entire feeding and growth submodel in Excel. Each row contains the calculations of intermediate and final results for the input variables from columns A-L. Parameter values—coefficients used in the feeding and growth equations—are specified in the rows above these calculations (the yellow-shaded cells).

The final set of columns (AV-BF, shaded blue) compare the Excel implementation to the NetLogo implementation of the submodel, by reporting the difference between them for nine intermediate results and the final result. The differences in intermediate results, all numerical, are reported as the percent by which the Excel result differs from the NetLogo result. A set of (red-shaded) cells at the top of these columns reports the minimum and maximum differences over the 10,000 rows. The final result has discrete values: the best feeding mode is either "drift", "search", or "hide". Therefore, column BE reports a value of 0 or 1 if the two implementations do or do not produce the same result, and cell BE4 counts the number of times they produced different results. As explained in a text box in the spreadsheet, the very small number of discrepancies between NetLogo and Excel implementations can be explained by conditions under which both "drift" and "search" produce the same growth so the choice of best feeding mode is arbitrary.

When comparing results calculated by NetLogo to those calculated in a second implementation, the question arises: how closely should the results match? For non-numerical results, e.g., procedures that produce TRUE or FALSE or choices among discrete options (such as the feeding modes discussed in the previous paragraph), the results should always match unless there are clear reasons why results might be arbitrary in some situations (as in the previous paragraph's example). For numerical results, our experience has been that procedures using simple mathematics should produce (using 64-bit software) percent differences between implementations $<10^{-10}$% (<1E-10). However, larger discrepancies (e.g., up to 10^{-4}%) can arise when algorithms use the differences between numbers that can be very large or very small, such as $e^{40} - e^{38}$, because the result depends more on the roundoff that is unavoidable in computing.

7. Finding the errors

The previous sections address how to determine whether a NetLogo model produces unexpected and presumably erroneous results. Now we address the next question: if we think there are code errors, how do we find and fix them? Experienced programmers develop intuitive strategies for finding errors ("debugging"); this is essentially detective work, so no approach is always most efficient or reliable. However, we can offer a set of general methods that each may be successful in some cases, and a list of tips and tricks that experienced NetLogo programmers use.

First, we discuss two issues that are very likely to arise when trying to explain and resolve unexpected model results. One is that, very often, it is not clear whether the NetLogo code exactly implements the written model description because the written description is incomplete or ambiguous. When this happens, modelers need to decide exactly what they want the model to do and update the written description. Similarly, the testing process may reveal discrepancies between code and written description that are best resolved by changing the written description to match the code, such as when the change makes the code simpler and is not expected to negatively affect results. The analysis may also reveal that the model description's assumptions produce undesirable results and need to be revised.

The second issue is that when graphical, statistical, or re-implementation analyses indicate errors, the errors are often in the analysis, not the NetLogo code. Graphs can be formatted incorrectly and contour plots can have misleading artifacts of the data density and contouring algorithm (as in Fig. 2). The expectations we test with statistical analyses can be wrong (Sect. 6.3.9 of Railsback and Grimm, 2019 discusses an example). Especially, independent implementations such as the one at Appendix E require thorough checking. Therefore, explaining discrepancies is a matter not of looking for errors in the NetLogo code but of reconciling differences that can be caused by errors in either the code or in our analysis and independent implementations.

7.1. Debugging methods

Debugging method 1: Execute the code in your head. This is the approach that programmers typically try first. It consists of reading the NetLogo code very carefully and trying to understand exactly what the computer does. It is especially important to think carefully about which agent is executing each statement: many errors arise because the agent actually executing some piece of code is not the one we think it is. Primitives that change the context (including "`ask`", "`with`", "`of`") are especially important to examine: the agent executing code within the brackets of these statements is different from the agent executing the surrounding code.

Method 2: Collect more information. Learning more about when and where problems arise is often the next debugging tactic. For example, examining how the variables of individual agents change from procedure to procedure, or from tick to tick, may reveal what part of the code or what conditions produce unexpected results. In NetLogo, ways to observe more about what is going on include:

- Using agent monitors to observe the variables of selected patches, turtles, or links. This method (and others) is facilitated by adding a "step" button to the interface that executes the go procedure just once. In fact, users can execute one procedure or submodel at a time using NetLogo's Command Center.
- Adding more defensive programming statements.
- Adding print statements, either to provide information (e.g., "`show count turtles-on neighbors`") or to verify that a statement works as intended (e.g., "`show breed`" to see what breed of turtles executes a block of code).
- Adding procedure monitors or procedure explorers to more procedures, or adding variables to the output produced by existing ones.
- Analyzing test output to identify the conditions under which errors occur. At Sect. 6.4, we discuss spreadsheet reimplementation of submodels to test code; one advantage of this approach is that spreadsheets can be sorted by the difference in submodel results between the NetLogo and spreadsheet codes to identify the instances in which errors were greatest.

Method 3: Develop and test hypotheses. When trying to solve any problem, we intuitively develop and test ideas about what might be wrong. When we have a programming error that does not quickly reveal itself, we can do this more consciously and systematically. We can think about all the possible causes, or all the possible parts of the code (or, in NetLogo, interface), where mistakes could be. Then we can design and conduct experiments to test whether each possible cause is the real one.

As a simple but real example, a student's model code produced a run-time error with the statement "No heading is defined from a point (23,−35) to that same point. Error while turtle 6 running TOWARDS." The NetLogo dictionary entry for "`towards`" says that it is a run-time error to use "`towards`" to get the heading from an agent to an agent at the same location. We can then hypothesize that a turtle tried to get the heading from itself to a patch when it had moved to that patch. (The turtle primitive "`move-to`", when used to move a turtle to a patch, causes the turtle to have the same coordinates as the patch.) The dictionary also tells us to see the primitive "`face`", which also sets a turtle's heading toward another agent. The dictionary entry for "`face`" tells us that if a turtle tries to face another agent at the same location, "`face`" leaves the heading unchanged instead of causing a run-time error. So we can test our hypothesis by searching the code for "`set heading towards`" and replacing it with "`face`"; if we do this replacement one instance at a time, we can determine exactly which statement(s) produced the conditions that lead to the run-time error, if our hypothesis

was correct.

Debugging usually uses all three of these methods: executing code in our head suggests hypotheses about what the location or cause of a problem is, and then we must collect more information to test the hypotheses.

7.2. Tips and tricks

The following are often-useful ideas for finding code errors.

Always fix syntax errors immediately. NetLogo checks for syntax errors when users click the "Check" button, and automatically when leaving the Code tab. We tell users to use the syntax checker after writing or revising each statement. Whenever NetLogo indicates a syntax error, you should fix it immediately, before trying to write more code or fix other problems, for several reasons. First, if you fix errors as soon as they are detected, you will know that the error was caused by the last change you made, so it will be easier to find. Second, the syntax checker cannot tell you when you make a new error if you have not fixed the previous one. Third, the unfixed error could affect the code you are working on while ignoring the error: the problem you are trying to solve often goes away when you fix the syntax error.

Set the random seed. The primitive "`random-seed`" sets NetLogo's random number generator so that each model run produces exactly the same results unless an input or algorithm changes. Setting the seed can be essential for distinguishing the effects of a change to the code (e.g., reprogramming a procedure or using a different primitive) from the effects of randomness, and for finding errors that occur only sometimes because of randomness.

Use "`show`" to observe the values of key variables during execution. You can see how a variable changes within a procedure, from statement to statement, in several ways. The most time-honored debugging method is to add print statements between program steps. In NetLogo it is very easy to add statements showing who has what value of selected variables at known places in the code, with statements such as "`show (word "turtle " who " has an energy of " turtle-energy " at line 257")`". Similar statements after each step of an algorithm will show how turtle energy changes from step to step.

Use "`user-message`" to pause execution and observe variable values. Debugging tools for other programming languages include "break points": the programmer can specify points in the code where execution is paused while variable values can be observed. NetLogo's "`user-message`" primitive provides similar capability, by pausing execution. Replacing "`show`" with "`user-message`" in the above statement causes NetLogo to not only show the value of a variable but to stop and let the user investigate what is going on. Preceding the "`user-message`" statement with "`inspect self`" (Sect. 5.2) makes it easy to see the state of the agent executing the code when it is paused.

Use the Command Center. NetLogo's Command Center on the Interface is a unique and powerful tool for debugging. At any point, users can invent and execute NetLogo statements that help understand and find problems, e.g., "`show count turtles with [turtle-energy < 0]`". The Command Center can also be used to execute the model one submodel at a time by manually entering the statements that are in the go procedure. By default, statements entered in the Command Center are executed by the observer, but it lets users instead send commands to turtles, patches, or links (entering a command sent to turtles is equivalent to "`ask turtles []`").

Use agent monitor mini-command centers. Agent monitors also provide a command center where you can enter statements executed only by the agent being inspected (Fig. 6). You can tell one selected turtle or patch to execute one procedure at a time or do other things that help debug. Agent monitors can be opened from the Interface or by using "`inspect`" statements in the code as illustrated above.

Use NetLogo's "Show Usage" and "Jump to Declaration" tools. In the code tab, when the cursor is placed in the name of a variable or procedure, a right mouse click opens a menu of options. (You must first click on the variable or procedure name, not just place the mouse over it.) This menu includes "Show Usage," which shows all the places in the code where the variable or procedure name appears; and "Jump to Declaration," which shows where the variable is initially defined or the procedure is coded. These tools are especially useful for the first debugging method, executing the code in your head.

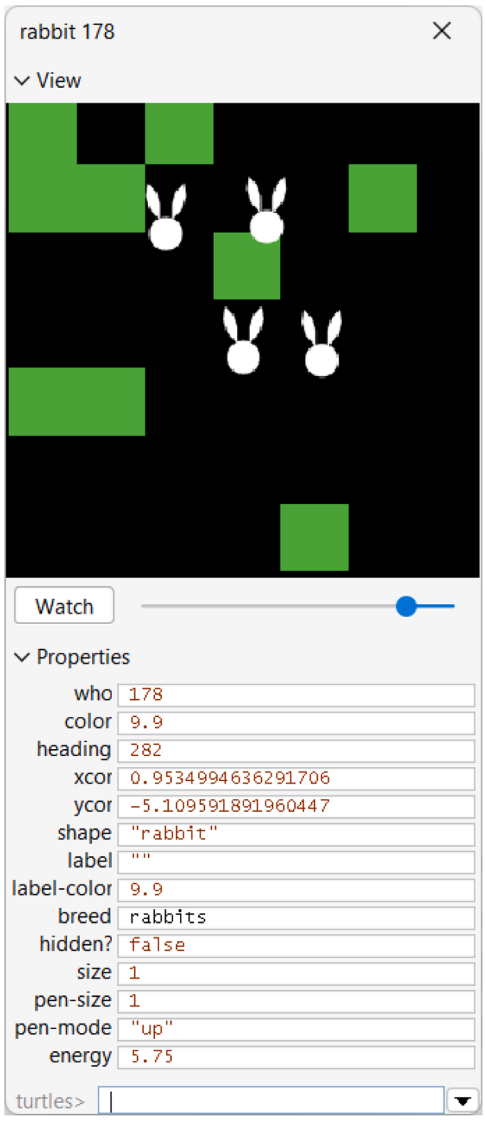


Fig. 6. At the bottom of an agent monitor is a mini-command center. NetLogo statements entered there are executed only by the monitored agent.

Use display characteristics to identify and track problem agents. You can use the display size, color, or labels of agents to identify and follow agents of special interest. For example, when some turtle variable is updated, you can have the code increase the turtle's size or change its color if the variable has an unexpected value.

Use local variables to break complex statements into smaller ones. Breaking long, complex NetLogo statements into smaller ones can help prevent and identify errors. We can do so by using "`let`" to create local variables for intermediate results; for example, the long statement used above (Sect. 4.1) can be broken into:

```
let a-red-turtle one-of turtles with [color = red]
let near-turtles other [turtles in-radius 3] of a-red-turtle
```

Using these two shorter statements could make it easier to understand exactly what the code does, and allows observation of its execution: we could add a statement "`ask a-red-turtle [show count`

`turtles in-radius 3]"` between the two lines to see which red turtle was selected and how many turtles are within a radius of 3 of it.

Understand exactly what NetLogo primitives do. If you have any doubt about how a primitive works, read its entry in the NetLogo dictionary carefully. If any doubt remains, you can write a test code (or use "`show`" statements, etc.) to check its behavior.

Consider that the error may be elsewhere. If you are sure that an error must be within a few lines of code in one procedure, but cannot see anything wrong with those lines, then it is very likely that the problem is actually elsewhere. For example, if a model produces unreasonable values of some turtle variable, and that variable is updated in one procedure, it is natural to focus debugging on that procedure. However, the problem could be somewhere else: some other procedure changes the variable (e.g., a turtle procedure might change a patch variable because turtles can easily change their patch), a parameter used in the procedure could have an incorrect value, some previous code could be missing a bracket or end statement, the procedure is executed at the wrong time or by the wrong agent, etc.

Explain the error to someone else. Experienced programmers ask each other for help when struggling to find an error. A "fresh set of eyes" often can quickly see a problem that someone else has stared at for too long. But even more often, the act of explaining an error causes us to see it. If you do not have a colleague nearby who knows NetLogo, posting questions to NetLogo's forum or Stack Overflow serves the same purposes: you may solve the problem while trying to explain it, or get a quick solution from someone else.

8. Documenting code tests

For many modeling projects, the small extra effort to document software testing is well worthwhile. "Documenting" here refers to annotating and archiving evidence of how model code was tested and that it passed the tests.

8.1. Documenting the testing process

Keeping records of what parts of a model's software have been tested, how they were tested, what problems and errors were found, and how they were fixed, helps modelers make sure the testing process is thorough yet efficient (e.g., by making it easier to re-test models after revisions), helps learn from the testing process, and provides evidence of the model's credibility. TRACE (Ayllón et al., 2021) is a standard format for organizing and documenting the entire modeling process, including code testing. The primary tool of TRACE is the modeling notebook, in which modelers (like laboratory and field scientists) keep daily records of what they did and learned.

A notebook on model development and software testing is likely to be the only documentation of less-formal testing methods such as looking for strange behavior via the View or using defensive programming. These methods often find the most important problems but do not produce the kind of files suitable for archiving that we discuss in the following subsection.

For some models, a modeling notebook will primarily be a convenience for the modeler, a way to keep track of what has been done and why, what remains to be done, and what the modeler chose not to do or put into the model and why. The notebook then supports final products such as published models and research based on them. However, for models intended for serious purposes such as supporting major research programs or making regulatory decisions, modeling notebooks and TRACE documents summarizing them can also be an important element of the model's record of credibility.

8.2. Documenting final code tests

As code testing and debugging is completed, the final tests can be annotated and archived. Archiving final code tests provides evidence to "clients" of a model—reviewers, potential users, graduate student committee members—that the software was tested. Just being able to say that an archive of tests is available for review provides substantial credibility to a model. Appendix E is an example of such documentation: an Excel file containing the code test plus notes, color coding, etc., explaining how the test was conducted and highlighting the final results that show a lack of discrepancies between NetLogo and Excel implementations of a submodel. The annotations should be written for two purposes: to help any reviewers or other readers understand the testing methods and results, and to facilitate future re-use of the testing methods.

9. Conclusions and development recommendations

Among the main reasons that NetLogo is widely used in science as well as education are its software quality capabilities: its large library of primitives greatly reduce the amount of code that users need to write and test, and its Interface, syntax checker, and run-time error handling greatly reduce the initial code testing effort. In NetLogo we can very quickly produce a model that executes with results that seem reasonable.

However, producing a model that executes with reasonable-seeming results only brings us to the most interesting phase of code testing: we still need to determine whether those results are actually correct. The only productive attitude at this phase is that of professional software developers: that new code always has hidden errors and our job is to find them. This job goes hand in hand with two other crucial modeling tasks: precisely defining what "correct" is by preparing a thorough written model description, and understanding why the finished model produces the results it does. The implementation cycle (Fig. 1) provides a structure for efficiently linking model design and description, code development and testing, and model analysis.

The code testing objective we focus on, finding logic errors, is underemphasized in many scientific fields where ABMs are now widely used. However, as these fields mature and as code written by AI becomes more prevalent, code testing will receive more attention. Because ABMs address difficult problems of complex systems, establishing their credibility is especially challenging. But with modest effort we can always establish one key aspect of credibility, software quality: we should always be able to show that our model code has been systematically tested with a reasonable degree of thoroughness and rigor.

NetLogo currently lacks debugging and software testing tools such as stepwise debuggers, unit testers, and omniscient debuggers. However, its powerful language makes it easy to provide similar capabilities via the methods we illustrate for producing and analyzing test output; using these methods may not require more effort than it would take to use such tools.

Several code testing technologies do appear worth consideration for application to NetLogo. Unit testers are an obvious candidate; a unit testing tool could at least partially standardize and automate testing via comparison to independent implementations. However, standardized unit testing may be more challenging for NetLogo because its code is not always in discrete, stand-alone units. Use of global variables and the ability of turtles to use variables of their patches and other turtles, for example, complicate the need in unit testing to control all the variables affecting the results of some procedure or submodel. The trout respiration submodel (Sect. 5.4) is one example: the effect of temperature on respiration is calculated outside this procedure (so that it is only updated once per tick, instead of the thousands of times per tick that the respiration procedure is executed). That intentional lack of modularity makes model execution much faster but makes unit testing more complicated. The flocking model test (Sect. 5.5) is another example: the procedures we test are driven by variables (location, heading) of an agentset of flockmates; it is not easy to control those variables for sets of unknown agents. Unit testing software that can accommodate such non-modularity may not be much easier to use than the methods we present.

Execution tracing tools seem potentially valuable both for finding errors and for understanding model results. It may be feasible to standardize and automate production of events output, such as by providing a simple command that, whenever called in the code, logs the agent's complete state and the event it is experiencing. More sophisticated tools, perhaps using database queries, could facilitate tracing, searching, and learning from these logged events.

Finally, the potential for AI to assist in code testing can be neither ignored nor predicted confidently. As long as AI is considered an unreliable partner, we must be concerned about using an unreliable tool to test code reliability. However, AI assistance in rigorous and human-verifiable testing methods clearly has promise.

CRediT authorship contribution statement

Steven F. Railsback: Writing – review & editing, Writing – original draft, Investigation, Conceptualization. **Jocelyn Bliven:** Writing – original draft, Investigation. **Volker Grimm:** Writing – review & editing, Writing – original draft, Conceptualization. **Jacob Kelter:** Writing – original draft, Investigation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.ecolmodel.2026.111669.

Data availability

No data was used for the research described in the article.

References

Agha, D., Sohail, R., Meghji, A., Qaboolio, R., Bhatti, S., 2023. Test driven development and its impact on program design and software quality: a systematic literature review. VAWKUM Trans. Comput. Sci. 11, 268–280. https://doi.org/10.21015/vtcs.v11i1.1494.

Ahlbrecht, T., 2024. An algorithmic debugging approach for belief-desire-intention agents. Ann. Math. Artif. Intell. 92, 797–814. https://doi.org/10.1007/s10472-023-09843-4.

Ayllón, D., Railsback, S.F., Gallagher, C., Augusiak, J., Baveco, H., Berger, U., Charles, S., Martin, R., Focks, A., Galic, N., Liu, C., van Loon, E.E., Nabe-Nielsen, J., Piou, C., Polhill, J.G., Preuss, T.G., Radchuk, V., Schmolke, A., Stadnicka-Michalak, J., Thorbek, P., Grimm, V., 2021. Keeping modelling notebooks with TRACE: good for you and good for environmental research and management support. Environ. model. Softw. 136, 104932. https://doi.org/10.1016/j.envsoft.2020.104932.

Bajpai, Y., Chopra, B., Biyani, P., Aslan, C., Coleman, D., Gulwani, S., Parnin, C., Radhakrishna, A., Soares, G., 2024. Let's fix this together: conversational debugging with github copilot. 2024 IEEE symposium on visual languages and human-centric computing (VL/HCC) 1–12. https://doi.org/10.1109/VL/HCC60511.2024.00011.

Banks, J., 1998. Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice. Wiley, p. 864. ISBN 9780471134039.

Grimm, V., 2002. Visual debugging: a way of analyzing, understanding and communicating bottom-up simulation models in ecology. Nat. Resour. Model. 15, 23–38.

Grimm, V., Railsback, S.F., Vincenot, C.E., Berger, U., Gallagher, C., DeAngelis, D.L., Edmonds, B., Ge, J., Giske, J., Groeneveld, J., Johnston, A.S.A., Milles, A., Nabe-Nielsen, J., Polhill, J.G., Radchuk, V., Rohwäder, M.-S., Stillman, R.A., Thiele, J.C., Ayllón, D., 2020. The ODD protocol for describing agent-based and other simulation models: a second update to improve clarity, replication, and structural realism. J. Artif. Soc. Soc. Simul. 23 (2), 7. https://doi.org/10.18564/jasss.4259.

Grimm, V., Berger, U., Calabrese, J.M., Cortés-Avizanda, A., Ferrer, J., Franz, M., Groeneveld, J., Hartig, F., Jakoby, O., Jovani, R., Kramer-Schadt, S., Münkemüller, T., Piou, C., Premo, L.S., Pütz, S., Quintaine, T., Rademacher, C., Rüger, N., Schmolke, A., Thiele, J.C., Touza, J., Railsback, S.F., 2025. Using the ODD protocol and NetLogo to replicate agent-based models. Ecol. Modell. 501, 110967. https://doi.org/10.1016/j.ecolmodel.2024.110967.

Gulati, S., Sharma, R., 2017. Java unit testing with JUnit 5. Apress, Berkeley, CA, p. 151. https://doi.org/10.1007/978-1-4842-3015-2.

Gürcan, Ö., Dikenelli, O., Bernon, C., 2013. A generic testing framework for agent-based simulation models. J. Simul. 7, 183–201. https://doi.org/10.1057/jos.2012.26.

IBM, 2025. Software testing. ibm.com/think/topics/software-testing. Archived at. http://archive.today/2025.08.06-162908/https://www.ibm.com/think/topics/software-testing..

Jorgensen, P.C., 2008. Software testing: a craftsman's approach, third edition. Auerbach Publications, New York, p. 440. https://doi.org/10.1201/9781439889503.

Kornhauser, D., Wilensky, U., Rand, W., 2009. Design guidelines for agent based model visualization. J. Artif. Soc. Soc. Simul. 12, 1. https://jasss.soc.surrey.ac.uk/12/2/1.html.

Lemmen, C., Sommer, P.S., 2024. Good modelling software practices. Ecol. Modell. 498, 110890. https://doi.org/10.1016/j.ecolmodel.2024.110890.

Myers, G.J., Badgett, T., Thomas, T.M., Sandler, C., 2011. The art of software testing, 3rd edition. John Wiley & Sons, Chichester, p. 256. ISBN: 978-1118031964.

Pothier, G., Tanter, E., 2009. Back to the future: omniscient debugging. IEEe Softw. 26, 78–85. https://doi.org/10.1109/MS.2009.169.

Railsback, S., Ayllón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C., Thiele, J., 2017. Improving execution speed of models implemented in NetLogo. J. Artif. Soc. Soc. Simul. 20, 3. https://doi.org/10.18564/jasss.3282.

Railsback, S.F., Grimm, V., 2019. Agent-based and individual-based modeling: a practical introduction, 2nd edition. Princeton University Press, Princeton, NJ, p. 329.

Railsback, S.F., Ayllón, D., 2021. InSTREAM 7: Instream flow assessment and management model for stream trout. River Res. Appl. 37, 1294–1302. https://doi.org/10.1002/rra.3845.

Railsback, S.F., Gallagher, C.A., Grimm, V., McCary, M.A., Harvey, B.C., 2025. Empirical ecology to support mechanistic modelling: different objectives, better approaches and unique benefits. Methods Ecol. Evol. 16, 1564–1573. https://doi.org/10.1111/2041-210X.70083.

Sarhan, Q.I., Beszédes, Á., 2022. A survey of challenges in spectrum-based software fault localization. IEEE Access. 10, 10618–10639. https://doi.org/10.1109/ACCESS.2022.3144079.

Swannack, T.M., Cushway, K.C., Carrillo, C.C., Calvo, C., Determan, K.R., Mierzejewski, C.M., Quintana, V.M., Riggins, C.L., Sams, M.D., Wadsworth, W.E., 2025. Cracking the code: linking good modeling and coding practices for new ecological modelers. Ecol. Modell. 499, 110926. https://doi.org/10.1016/j.ecolmodel.2024.110926.

Vincenot, C.E., 2018. How new concepts become universal scientific approaches: insights from citation network analysis of agent-based complex systems science. Proc. R. Soc. B: Biol. Sci. 285, 20172360. https://doi.org/10.1098/rspb.2017.2360.

Wickham, H., 2011. testthat: get started with testing. R. J. 3, 5. https://doi.org/10.32614/RJ-2011-002.

Wikipedia contributors. (2026). Software testing. In *Wikipedia, The Free Encyclopedia.* Retrieved 14:58, May 18, 2026, from https://en.wikipedia.org/w/index.php?title=Software_testing&oldid=1354782901.

Wilensky, U., 1999. NetLogo. Center for Connected Learning and Computer-based Modeling. Northwestern University, Evanston, IL, USA. http://ccl.northwestern.edu/netlogo/.

Wilensky, U., Rand, W., 2015. An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo. MIT Press, Cambridge, Massachusetts, p. 482.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42, 707–740. https://doi.org/10.1109/TSE.2016.2521368.