

BREEDING FASTER TURTLES: PROGRESS TOWARDS A NETLOGO COMPILER

F. SONDAHL^{*}, Northwestern University, Evanston, IL
S. TISUE, Northwestern University, Evanston, IL
U. WILENSKY, Northwestern University, Evanston, IL

ABSTRACT

Despite the stereotype concerning their biological counterparts, NetLogo's turtles are fast. NetLogo (Wilensky 1999a) contains a sophisticated interpreter that has been highly optimized. Nevertheless, NetLogo turtles aren't as fast as they could be. Interpretation necessarily incurs a performance penalty. Thus, we are in the process of replacing NetLogo's interpreter with a compiler. This transition is happening in phases. In this paper, we discuss the architecture of NetLogo's interpreter and explain the first phase of the transition to compilation, which uses inlining to generate efficient bytecode from abstract syntax trees. This technique measurably reduces the interpreter overhead, while permitting a gradual transition to a compiled architecture. We approach the task of compiler design from the perspective of a powerful agent based modeling language with “low threshold” design goals. Preliminary benchmark results are presented, in addition to a forecast of further steps towards a full NetLogo compiler.

Keywords: NetLogo, compilers, performance, agent based modeling,
Java Virtual Machine

INTRODUCTION

Despite the stereotype concerning their biological counterparts, NetLogo's turtles are fast. NetLogo (Wilensky 1999a) contains a sophisticated interpreter that has been highly optimized. Nevertheless, NetLogo turtles aren't as fast as they could be. We are working to remedy this. Because the use of even a sophisticated interpreter incurs a necessary performance penalty, we are in the process of replacing NetLogo's interpreter with a compiler. To better understand the current development focus, it is helpful to discuss the historical background and philosophical motivation of NetLogo.

The design of the original Logo language was guided by the slogan “low threshold, high ceiling” (Papert 1980). NetLogo upholds this tradition (Tisue & Wilensky 2004). It should be easy for new users to learn NetLogo and build models, but it should also be possible for advanced modelers to build “research-grade” models. There are inevitable trade-offs between these two design goals. NetLogo's adoption by thousands of modelers, from rank novices to veteran hackers, suggests that a healthy balance between these goals is being achieved.

Largely for reasons of “low threshold”, NetLogo was originally implemented as an interpreted language. Even though a compiler would make models run faster, building a compiler is time-consuming and would not help lower NetLogo's threshold. Early development effort

^{*} *Corresponding author address:* Forrest Sondahl, EECS Tech Institute C359, 2145 Sheridan Rd, Evanston IL 60208-0834; e-mail: forrest@northwestern.edu

was better spent in other directions (e.g., building an integrated development environment and adding features). As NetLogo matured and was more widely adopted by the research community, speed became a ceiling issue for advanced users. In response, the interpreter was substantially restructured and tuned for performance. This resulted in dramatic speed improvements, but eventually we felt that further significant improvements could only be achieved through compilation.

We should note that compilation and interpretation are not mutually exclusive approaches. The Java language is a prime example (Gosling et al. 1996). Java source code is compiled to an intermediate form (bytecode), which is interpreted by the Java Virtual Machine (JVM). Similarly, NetLogo source code is first transformed from text to an intermediate form (arrays of abstract syntax trees), which are then interpreted by the NetLogo interpreter (Tisue & Wilensky 2004). Although this system is quite fast when compared to naive interpreter implementations, it still results in measurable overhead costs when compared to models written in pure Java. To move beyond this performance barrier, we decided to compile the NetLogo language directly into JVM bytecode.

Building a new compiler from the ground up was problematic for several reasons. First, NetLogo's code base is now large – there are over 300 built-in language "primitives" in NetLogo. Each primitive is implemented as a Java class. This large body of Java code represents a substantial investment of development time, which we wanted to leverage for use by the compiler. Second, there are features of the NetLogo language that can be smoothly handled by an interpreter, but would frustrate the implementation of a traditional compiler – for example, the frequent context switching as various agents execute their code, to simulate concurrent activity. We are not suggesting that traditional compiler-writing methods are inapplicable to the NetLogo language – in fact, we will be employing them later (see “Future Work” below). However, for the first phase of development we chose an alternative method which achieves significant performance gains, while integrating seamlessly with NetLogo's existing interpreter, and maintaining most of the flexibility of language development that the interpreted system provided. This integration is a strong first step in NetLogo's transition towards a complete compiler system targeting the Java Virtual Machine platform.

IMPLEMENTATION

Bytecode Inlining Overview

Our hybrid solution involves combining the existing interpreter with partial compilation. One important aspect of the new compiler is a technique we call “JVM bytecode inlining”. The NetLogo interpreter itself is running on the JVM platform, which means that each of the primitives accepted by the NetLogo interpreter maps to some sequence of JVM bytecode that gets executed. Our bytecode inliner extracts this sequence and inserts it into the compiled code. Inlining avoids the overhead of calling the sequence as a separate method, which is what the NetLogo interpreter had to do. The combined sequences of bytecode are then dynamically loaded as a single new Java method. The end result is similar to the output that would be given

by a traditional compiler. However, we avoid some of the complexity of a full-blown compiler, because we are able to “steal” the bytecode that was pre-compiled by a traditional Java compiler (e.g., Sun's *javac*). Another simplification is that at present we are only compiling individual NetLogo commands (and their arguments), not yet sequences of commands; the interpreter still moves from command to command and handles procedure calls. For the task of bytecode extraction and generation we use ASM, which is a small and fast Java bytecode manipulation framework (Bruneton et al. 2002).

Bytecode Inlining Example

Conceptually, we can think of the bytecode inlining process as NetLogo dynamically extending the pool of primitives in the interpreter's repertoire, by replacing a command's entire abstract syntax tree with a single combined primitive that we synthesize to do the task more efficiently.

The text of a NetLogo program is first lexically parsed and tokenized. An array of abstract syntax trees is created, variable references are resolved, nested command blocks are linearized, etc. Eventually, the output is a NetLogo `Procedure` object, which consists of an array of `Command` objects, each of which is the root of a tree containing `Reporter` objects. All NetLogo primitives fall into these two categories, reporters (e.g., `+`, `sin`, `patch-ahead`) and commands (e.g., `rt`, `fd`, `print`). Reporters return ("report", in our terminology) values; commands do not – they simply “perform” some action.

For clarification of the bytecode inlining process, we will present a step-by-step example for a simple code fragment: `rt (a + 5)`. This NetLogo code causes a turtle (agent) to turn (change its heading) $(a + 5)$ degrees to the right, as shown in the right-hand side of Figure 1. On the left-hand side of Figure 1 there is a graphical depiction of the abstract syntax tree created by NetLogo's parser. As mentioned above, each node of the tree is a Java object. For instance, “rt” maps to an instance of class “_right”, a subclass of `Command`. Similarly, “+” maps to the class “_plus”, “a” to the class “_turtlevariable”, and “5” to the class “_constdouble”, subclasses of `Reporter`. These classes each define an execution method. Command classes define a “perform()” method (with a `void` return type), and reporter classes define a “report()” method (with an `Object` return type).

For this example, we will denote the instances of classes `_right`, `_plus`, `_turtlevariable`, and `_constdouble`, as R, P, T, and C respectively (see Listing 1). The NetLogo interpreter would evaluate our example tree by

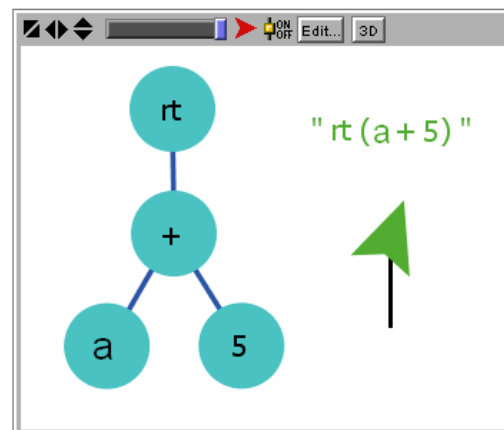


FIGURE 1: Abstract syntax tree

calling the `R.perform()`. This method in turn would call `P.report()`, which would call the `report()` methods for each of P's children nodes (T and C), add the results together, and return the result to R's `perform()` method, which would then change the executing agent's heading by the appropriate number of degrees. Pseudo-code is shown in Listing 2.

LISTING 1: Textual representation of the tree

```
_right (object R): "rt"
  _plus (object P): "+"
    _turtlevariable (object T): "a"
    _constdouble (object C): "5.0"
```

Underscored words correspond to Java classes representing NetLogo language primitives.

Instead of stopping with the abstract syntax tree, the new NetLogo compiler processes the tree to create JVM bytecode. First it performs a post-order traversal of the tree (e.g. order: a 5 + rt). For each node it visits, we use ASM's `ClassReader` to extract the bytecode from the `perform()` or `report()` method that would have been called by the interpreter (e.g. `_constdouble.report()`, `_turtlevariable.report()`, etc). We perform some minor transformations on the extracted bytecode before passing it to ASM's `ClassWriter`, to generate the `perform()` method of our new class. Instead of transferring "return" statements from the extracted method to the generated method, it leaves the result that would have been returned on the JVM operand stack. When the traversal is finished, the resulting class is written to a byte array, and dynamically loaded into the JVM using a custom `ClassLoader`. We create a new object G from the newly loaded class.

The pseudo-code that is representative of the transformation is shown in Listing 3. A textual representation of the JVM bytecode that is produced by the new bytecode compiler is shown in Listing 4. Note that the tree which originally consisted of four objects (R, P, T, and C) was replaced by a flattened version with just one object (G).

LISTING 2: Pseudo-code for interpreted system

```
R.perform() :
  context.agent.turnRight( P.report() ) ;

P.report() :
  return C.report() + T.report() ;

C.report() :
  // C has a member field that holds the constant value, 5
  return C.storedValue ;

T.report() :
  // The symbol "a" corresponds to an index into a variable array
  // For this example, assume the index, T.variable_number, is 7.
  return context.agent.getTurtleVariable( T.variable_number ) ;
```

LISTING 3: Pseudo-code for the bytecode

G.perform() :

```
context.agent.turnRight( 5 + context.agent.getTurtleVariable( 7 ) ) ;
```

LISTING 4: Simplified* JVM bytecode that results from compiling "rt (a + 5)"

G.perform() :

```
ALOAD 1 // push the "context" onto stack
GETFIELD Context.agent : Lagent; // get the current context's agent
BIPUSH 7 // push 7 onto stack
INVOKEVIRTUAL Agent.getTurtleVariable // get Object stored in var 7
CHECKCAST Double // check that var 7 held a number
INVOKEVIRTUAL Double.doubleValue ()D // convert Double Object -> double
LDC 5.0 // push 5 onto stack
DADD // now "a + 5" is on stack
DSTORE 2 // store "a + 5" in JVM local #2
ALOAD 1 // push "context" onto stack
GETFIELD Context.agent : Lagent; // get the current context's agent
CHECKCAST Turtle // make sure agent is a Turtle
DLOAD 2 // load "a + 5" back onto stack
INVOKEVIRTUAL Turtle.turnRight (D)V // cause turtle to "rt (a + 5)"
```

**Package names have been omitted for brevity. Actual compiler output contains additional bytecode for runtime type-checking and error handling, which has been omitted for clarity.*

Bytecode Inlining Advantages

Several aspects of this process increase performance:

1. Constant values. In the old system, constant values that are known at compile time -- such as 5 and 7 (the turtle-variable index) -- were stored in member fields. In the new system, they are hard coded as more efficient PUSH or LDC bytecode instructions.
2. Casting. The old interpreter's `report()` methods only return generic Objects, and the calling method must check the return type and cast it to the appropriate type. The new compiler is often able to perform this type checking at compile-time, and generate the appropriate bytecode, omitting the unnecessary casting.
3. Primitive type checking. Similarly, the new compiler is able to deal more efficiently with Java's primitive types -- e.g. `booleans` and `doubles` -- avoiding many cases where the interpreter was forced to "box" the results as `Boolean` or `Double` objects.
4. Method invocations. The old system required four `perform/report` method invocations, whereas the new system only requires one.

We conducted several tests to approximately measure the comparative influence of these aspects on increasing performance. Optimization of constant values (#1) accounted for around 4% of the performance improvement, whereas the type conversion aspects (#2 and #3) accounted for roughly 50%. Decreasing the number of method invocations (#4) is credited with the remaining 46% of the speedup.

Another aspect that could be contributing to the performance increase is synergy with JIT (just in time) compilers. Inlining method bytecode creates larger contiguous sections of bytecode in a single method, which can improve opportunities for standard intraprocedural compiler optimizations, particularly when the inlined method bodies are simple (e.g., Scott 2000; Bellotti et al. 2004). It is our hope that JIT compilers can better optimize our generated bytecode. As of yet, we have not measured the influence of bytecode generation on JIT compilers. Since Sun Microsystems' HotSpot compiler performs its own form of “class-hierarchy aware” method inlining (Paleezny et al., 2001), it is unclear whether synergistic interaction is occurring. Further benchmarking is required to examine this issue.

PERFORMANCE RESULTS

Performance Benchmarks

In our graphs, “Cur” denotes the current development build of NetLogo as of July 21, 2006, with compilation disabled, and “Cur+” denotes the same build with compilation enabled. All benchmarking was done on a 3.2 Ghz Pentium 4 with 2 GB of RAM, running Windows XP Professional. The results shown in Figure 2 used Sun's Java 2 Runtime Environment version 1.5.0_06, with the HotSpot(TM) Client VM. The results shown in Figures 3 and 4 used Sun's Java 2 Runtime Environment version 1.4.2_10, with the HotSpot(TM) Server VM. All benchmarks were run with the graphical display disabled, to better measure engine speed.

Figure 2 presents a view of the history of performance in NetLogo on one particular benchmark, the so-called “GasLab” benchmark. Our benchmarks are not synthetic micro-benchmarks; they are real models from NetLogo's models library. The GasLab benchmark is based on a model called “GasLab Gas in a Box” that demonstrates the Maxwell-Boltzmann distribution in an ideal gas (Wilensky 1997, 1999b). Figure 2 shows that performance improvements came quickly in NetLogo's early days. Between versions 1.1 and 1.2, the interpreter was restructured from a stack-based to tree-based (Tisue & Wilensky, 2004). Since that time NetLogo hit a

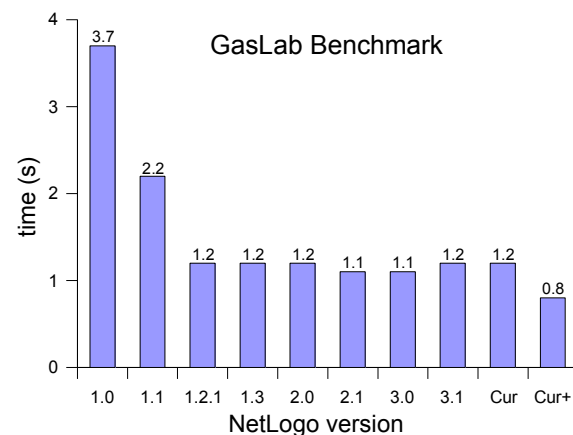


FIGURE 2: NetLogo performance history

performance barrier that persisted up until the creation of the new compiler. This breakthrough cut execution time on the GasLab benchmark to 66% of what it had previously been.

Figures 3 and 4 present a broader perspective on the performance gains attributable to the compiler. Figure 3 shows the results for each of the 13 benchmarks in NetLogo's benchmark suite. Note that the performance increase varies considerably between models. For instance, the bytecode compiler only shaved 5% off of the execution time of the Flocking benchmark (#7), while the time for the 1-D Cellular Automata benchmark (#13) was nearly cut in half. Figure 4 shows the performance gain across the board; on average, execution time was cut by 23%.

Comments on Performance

For the purposes of this paper, we have limited ourselves to comparing NetLogo against its past performance. Although it would be interesting to do so, we have not compared NetLogo's performance against that of other popular agent based modeling platforms, or against models written in “raw” Java code without the aid of a specialized toolkit. It is often difficult to make such comparisons fairly, since various modeling platforms suggest different natural implementations of a given model, as well as different techniques for tuning and optimization. For further discussion on this topic, and a general review of several popular agent based modeling toolkits, see Railsback et al. (2006).

The new NetLogo compiler is still very much a work in progress. Not all of NetLogo's language primitives are yet taking advantage of the new system, and we expect continued performance increases. Some preliminary tests give us hope that the continuation of this project, in addition to further forays into bytecode generation (see “Future Work” below), may eventually lead to as much as 3x speed improvement over NetLogo 3.1 (i.e. a reduction of execution time to 33% of its previous value).

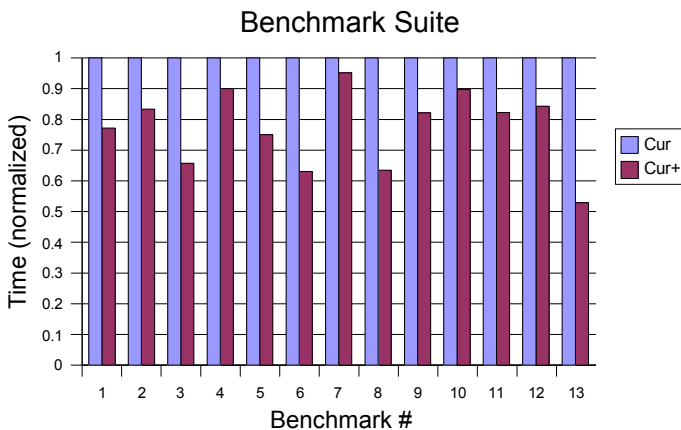


FIGURE 3: Bytecode inlining improvement

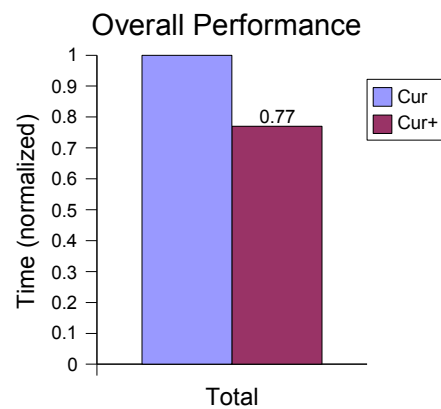


FIGURE 4: Benchmark average

Will NetLogo performance ever match/surpass the performance of raw Java code? If both Java and NetLogo are being compiled to bytecode, can't NetLogo language be just as fast? We expect not, because of differences in the source languages. For example, Java is statically typed (you must declare variable types – “int n”, “double d”), whereas NetLogo is dynamically typed (any variable can hold any data type). The conciseness and flexibility of dynamic typing contributes to NetLogo's low threshold. Static typing allows more type-checking to be done at compile-time, and thus more efficient bytecode can be produced. Dynamic typing is one example of a trade-off between “low threshold” and “high performance.”

Note that “low threshold” here isn't only relevant to novice programmers. Expert users, too, would pay a cost of slower authoring if type declarations were required. One reason NetLogo is popular among researchers and other “high-end” users is their ability to rapidly develop prototype models in NetLogo. In the end, the question is not which language is the fastest; nobody wants to write agent-based models in assembly language. The pertinent question is whether the language you want to model in is high level enough to ease development and maintenance, yet fast enough for your needs.

FUTURE WORK

Towards a NetLogo Compiler

As mentioned earlier, bytecode inlining is just the first phase in implementing a complete NetLogo compiler. The details we have discussed only involve creating the bytecode to deal with a single NetLogo command. Compiling the abstract syntax tree for each single command is effective at boosting performance if expressions are long (e.g., see Listing 5). However, many NetLogo models have a low command-to-expression-length ratio (e.g., see Listing 6), and in

LISTING 5: A procedure from the NetLogo “CA 1D Elementary” model (Wilensky, 1998a)

```
to do-rule ;; patch procedure
  let left-on? on?-of patch-at -1 0
  let right-on? on?-of patch-at 1 0

  ;; each of these lines checks the local area and (possibly)
  ;; sets the lower cell according to the corresponding switch
  let on?-of patch-at 0 -1
    (iii and left-on?      and on?      and right-on?)      or
    (iio and left-on?      and on?      and (not right-on?)) or
    (ioi and left-on?      and (not on?) and right-on?)      or
    (ioo and left-on?      and (not on?) and (not right-on?)) or
    (oii and (not left-on?) and on?      and right-on?)      or
    (oio and (not left-on?) and on?      and (not right-on?)) or
    (ooi and (not left-on?) and (not on?) and right-on?)      or
    (ooo and (not left-on?) and (not on?) and (not right-on?))
end
```


LISTING 6: A procedure from the NetLogo “Flocking” model (Wilensky, 1998b)

```
to turn-at-most [turn max-turn] ;; turtle procedure
  ifelse abs turn > max-turn
    [ ifelse turn > 0
      [ rt max-turn ]
      [ lt max-turn ] ]
    [ rt turn ]
end
```

such cases this technique is not as effective. These models should see a greater performance increase as we extend the compiler to generate bytecode for more than a single command at a time, which will be the next the phase of compiler work.

The first step in this direction will be to compile basic (that is, non-branching) blocks of adjacent commands. In the next step we will extend the compiler to process control structures — branches, loops, and procedure calls. Finally, whole procedures and entire NetLogo models will be compiled.

Additional Optimizations

In addition to the core compiler plan outlined above, there are several other promising areas of optimization work:

- NetLogo language procedures could be inlined. This is a separate issue from the Java method inlining discussed in this paper, which occurs at the JVM level. The motivation, however, is much the same. When modelers write short NetLogo procedures that are called frequently, speed could be increased by inlining that procedure into the calling NetLogo procedure.
- A type inferencing system could be designed for local variables. Even though NetLogo is dynamically typed, there are situations where we could detect the type of a variable at compile time and optimize the code accordingly.
- We have already designed a peephole bytecode optimizer, which removes some inefficient code that is created during the bytecode generation process. More peephole optimizations could be introduced.
- Higher-level optimizations. NetLogo currently has a variety of sophisticated optimizations in place. For example, the code snippet “turtles with [color = red]” reports an agentset of all the red turtles in the world. The primitive “any?” tests whether or not an agentset is empty. A naive interpreter running the code “if any? turtles with [color = red]” would first find all the red turtles, and then see if that set is empty. NetLogo internally rewrites this code to stop looking for red turtles as soon as it has found one. There are a fair number of such optimizations already in place, but more could be designed.

Caveats

So far, we have only discussed the benefits of inlining. There is also a drawback associated with inlining that becomes more salient as the amount of bytecode generation increases – namely “code bloat.” As reported by Bellotti et al (2004), excessive method inlining in Java can result in decreased performance. Because our bytecode generation technique does not require the use of any extra JVM local variables, we have reason to hope that we avoid this negative effect of inlining. But performance issues aside, the JVM imposes a limit of 64 kilobytes for the bytecode of a method body, and so completely inlining the contents of a long NetLogo procedure into a single method will not be possible. We will need to find a balance between inlining and method invocation.

A second issue that arises is not particular to inlining, but is a consequence of generating bytecode. NetLogo allows models to be saved as Java applets, which can then be run in a web browser. Currently, the applet embeds our interpreter. With the compiler, the model would need to be compiled before it could be run; however, for security reasons unsigned applets may not load dynamically generated bytecode. We will resolve this issue by generating a custom JAR file for the applet, which will contain the compiled bytecode for the given model.

CONCLUSION

Bytecode inlining provides greater flexibility than a more traditional compilation process. New primitives can still be added to the NetLogo language with the same ease as before – bytecode inlining extracts the compiled bytecode behind the scenes. This is particularly useful for NetLogo, which remains a rapidly evolving language. Our hybrid approach also allows some code to remain interpreted while other code is compiled. This intermingling of interpreted and compiled code provides the foundations for a gradual transition towards a full NetLogo compiler. Using the techniques described in this paper, we have already experienced a significant performance increase, and we expect future work on bytecode generation to result in further speedups. NetLogo's turtles are faster now than ever before, and they are still picking up speed.

REFERENCES

- Bellotti, F., Berta, R., & De Gloria, A., 2004, "Evaluation and optimization of method calls in Java," in *Software: Practice and Experience*, Vol. 34, No. 4, pp. 395-431.
- Bruneton, E., Lenglet, R., & Coupaye, T., 2002, "ASM: A code manipulation tool to implement adaptable systems," in *Adaptable and extensible component systems*, Grenoble, France; available at <http://asm.objectweb.org/current/asm-eng.pdf>.
- Gosling, J., Joy, B., & Steele, G., 1996, *The Java Language Specification*, Reading, MA: Addison-Wesley.
- Paleezny, M., Viek, C. & Click, C., 2001, "The Java HotSpot Server Compiler," in *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 1-12.
- Papert, S., 1980, *Mindstorms: Children, Computers, and Powerful Ideas*, New York: Basic Books.
- Railsback, S., Lytinen, S., & Jackson, S., 2006, "Agent-based Simulation Platforms: Review and Development Recommendations," *Simulation*, (manuscript in review).
- Scott, M., 2000, *Programming Language Pragmatics*, San Francisco, CA: Morgan Kaufmann Publishers Inc.
- Tisue, S., & Wilensky, U., 2004, *NetLogo: Design and implementation of a multi-agent modeling environment*, Paper presented at the Agent 2004 conference, Chicago, IL, October 2004.
- Wilensky, U., 1997, *NetLogo GasLab Gas in a Box model*, Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University, available at <http://ccl.northwestern.edu/netlogo/models/GasLabGasinaBox>.
- Wilensky, U., 1998a, *NetLogo CA 1D Elementary model*, Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University, available at <http://ccl.northwestern.edu/netlogo/models/CA1DElementary>.
- Wilensky, U., 1998b, *NetLogo Flocking model*, Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University, available at <http://ccl.northwestern.edu/netlogo/models/Flocking>.

Wilensky, U., 1999a (updated 2006), NetLogo [Computer software] (Version 3.1), Evanston, IL: Center for Connected Learning and Computer-Based Modeling; available at <http://ccl.northwestern.edu/netlogo>.

Wilensky, U., 1999b, "GasLab: An extensible modeling toolkit for exploring micro-and-macro-views of gases," in N. Roberts, W. Feurzeig & B. Hunter (Eds.), *Computer Modeling and Simulation in Science Education*, pp. 151-178, Berlin: Springer Verlag.