Proceedings of the 2013 Winter Simulation Conference
R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

THE RELOGO AGENT-BASED MODELING LANGUAGE

Jonathan Ozik
Nicholson T. Collier
John T. Murphy
Michael J. North

Argonne National Laboratory
Decision and Information Sciences
9700 S Cass Ave
Argonne, IL 60439, USA

ABSTRACT

ReLogo is a new agent-based modeling (ABM) domain specific language (DSL) for developing agent-based models in the free and open source Repast Suite of ABM tools; the Java based Repast Simphony ABM toolkit and the C++ high performance computing Repast HPC toolkit both incorporate ReLogo. The language is geared towards a wide range of modeling and programming expertise, combining the sophisticated and powerful ABM infrastructure and capabilities in the Repast Suite with the ease of use of the Logo programming language and its associated programming idioms. This paper will present how ReLogo combines a number of concepts, including object-oriented programming, simple integration of existing code libraries, statically and dynamically typed languages, domain specific languages, and the use of integrated development environments, to create an ABM tool that is easy to learn yet is also capable of creating large scale ABMs of real world complex systems.

1 INTRODUCTION

The free and open source Repast Toolkit was originally developed by Sallach, Collier, and others (Collier et al. 2003) at the University of Chicago in 2000. It was expanded by Argonne National Laboratory as software infrastructure that could support rapid social science discovery based on extensive computational experimentation (Sallach and Macal 2001). The most current version of the Repast Toolkit includes the Java based Repast Simphony (North et al. 2013) and the C++ based high performance computing Repast HPC (Collier and North 2012).

Repast Simphony ReLogo wraps much of the Repast Simphony library's functionality into a semantically simple but powerful package. The goal is to provide a fast track so modelers can quickly develop simulations that can later be scaled up, as needed, using the features of the full Repast Simphony or Repast HPC libraries. ReLogo semantics are drawn from the pedagogically oriented Logo lineage of software (Logo Foundation 2013, Harvey 1997), informed especially by the StarLogo (Resnick 1996) and NetLogo (Wilensky 1999) variants, while maintaining the object-oriented nature of Repast Simphony. ReLogo provides a streamlined approach to creating Repast Simphony models using environmental agents (Patches), networks (Links), coordinating agents (Observers) and mobile agents (Turtles), collectively referred to as PLOT entities. ReLogo models are programmed using the Logo-style ReLogo domain-specific language.

2 BACKGROUND

Logo (Logo Foundation 2013, Harvey 1997) is a widely used educational programming language commonly found in K-12 classes. For many users it is easier to conceive and design a model using the Logo paradigm. Logo provides the basis for several other ABMS platforms, most notably StarLogo (Resnick 1996) and NetLogo (Wilensky 1999).

2.1 StarLogo

StarLogo is a library and environment that uses a Java interpreter and interface. StarLogo is a pedagogically oriented system that leverages the Logo language to make it easier to learn to develop agent models. StarLogo is free and open source for non-commercial use. StarLogo extended Logo by increasing the allowed number of mobile agents (turtles) from tens to thousands, by giving turtles more interactive behaviors, and by transforming the turtles' environment from a rectangular array of pixels into a grid of environmental agents (patches). The organizing principle of StarLogo, according to (Resnick 1996), is the modeling of decentralized complex systems.

2.2 NetLogo

NetLogo is a free and open source (General Public License [GPL]) agent-based simulation environment that further extended the Logo programming paradigm. NetLogo builds on StarLogo (Tisue 2004). In addition to turtles and patches, NetLogo introduces links that connect turtles to form networks. While NetLogo was designed to provide a basic laboratory for teaching complexity concepts, it can be and has been used to develop sophisticated applications. NetLogo is distributed with a large number of example simulations to help beginning and experienced users quickly get up to speed with building models. NetLogo specifically does not try to avoid programming but rather endeavors to make the syntax and semantics as accessible as possible while maintaining the flexibility to create complex models for teaching and research purposes.

3 REPAST SIMPHONY RELOGO

Repast Simphony is the Java based toolkit of the Repast Suite. Repast Simphony ReLogo is based on the Groovy (Groovy 2013) dynamic language. Groovy itself is a widely used, free, and open source Java Virtual Machine (JVM) language that both compiles to Java bytecode and tightly integrates with Java. Thanks to its Groovy foundations, ReLogo freely interoperates with Groovy and Java. ReLogo programmers can use any Java or Groovy library without special syntax. They can also write Groovy and most Java code fluidly at any point in a ReLogo file to gain access to the advanced features of either language.

A ReLogo model is compiled by using the Groovy Eclipse joint compiler (Groovy Eclipse Plugin 2013). Dynamic user methods are created via automated code generation and Abstract Syntax Tree (AST) transformations are used to generate additional Java byte code. The resulting code is run as a regular Repast Simphony model.

ReLogo adds the capabilities of Repast Simphony to the semantic simplicity of Logo, and in doing so contributes a number of concepts to building Logo-like ABMs; object-oriented programming, simple integration of existing code libraries, statically and dynamically typed languages, domain specific languages, and the use of integrated development environments. The result is an ABM tool that is easy to learn yet is also capable of creating large scale ABMs of real world complex systems.

3.1 Object Oriented Programming

From it's origins in the Simula67 computer language (Holmevik 1994) and earlier, object oriented programming (OOP) has evolved into one of the major programming paradigms in computation (Gamma et al. 1994, Booch et al. 2007). ReLogo introduces OOP into the Logo world by separating the patches, links, observers and turtles (PLOT) entities into their own class hierarchies. Figure 1 demonstrates the PLOT class hierarchies in the Zombies_Demo model included in the Repast Simphony distribution. By separating each entity type into its own class hierarchy, the ReLogo modeler can make use of *encapsulation*, *inheritance* and *dynamic dispatch*, three of the main characteristics of OOP. *Encapsulation* isolates the internals of the PLOT classes, separating the implementation of the various capabilities from the interface made available to external classes. *Inheritance* allows for gathering common attributes and behaviors of PLOT entities into generalized classes (or parent classes) from which the specialized classes (or child classes) inherit. This way the code implementing the attributes and behaviors is in one location, reducing the likelihood of introducing bugs in the form of divergent implementations and increasing the maintainability of the developed models. *Dynamic dispatch* enables specializations to override the default behaviors of their parent classes. The overall result of integrating OOP is to endow the modeler with the ability to create more modular, increasingly complex, yet maintainable ABMs.

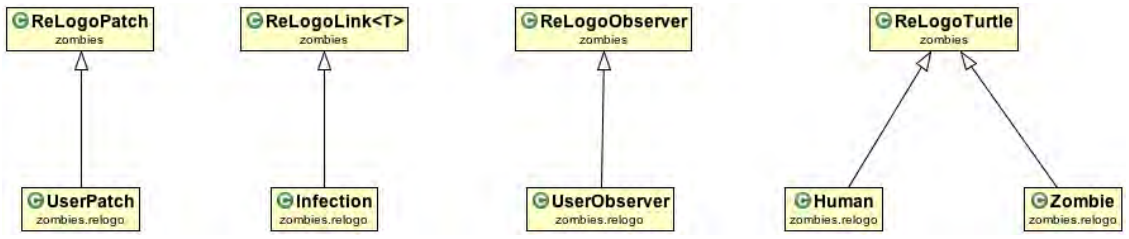


Figure 1: PLOT class hierarchies in the example Zombies_Demo model distributed with Repast Simphony.

3.2 Simple External Library Integration

The widespread use of Java as a general programming language has resulted in large number of libraries and framework that are available for use within any Java based program. The libraries are usually distributed in the form of Java Archives or JAR files. A JAR file will contain all the necessary binary assets, metadata and in some cases even source files, for the library. Simply making the JAR file visible on the compilation and execution Java classpaths of a Java program will make all the functionality of the library available. To include a JAR file in a ReLogo model it's enough to place it in the `lib` folder within the model and add it to the build path (see Figure 2). With this simple step, all the capabilities included in the JAR file are immediately available for use in any part of the model's files.

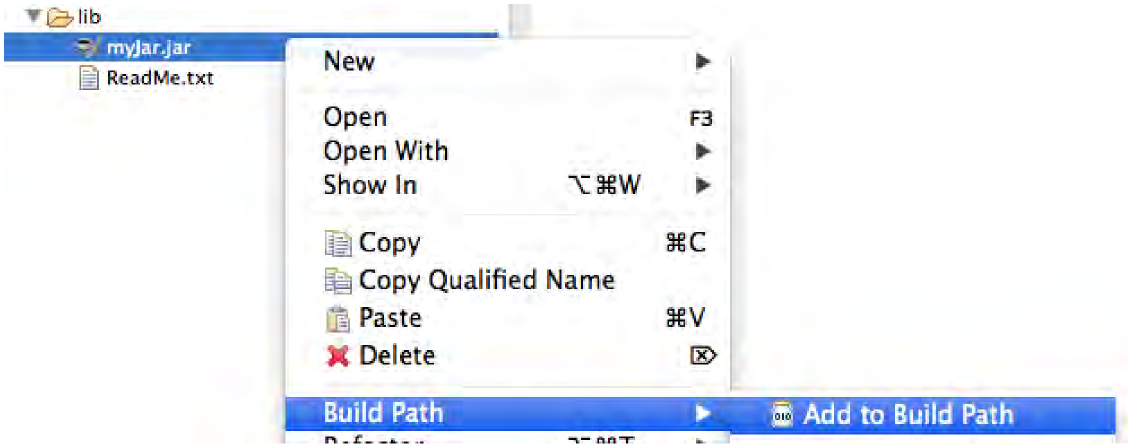
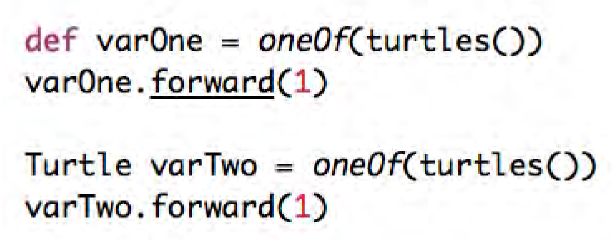


Figure 2: Adding a library (myJar.jar) to a ReLogo model. Place the JAR file in the `lib` folder and add it to the build path (in this case by right-clicking to reveal the contextual menu while the file is selected).

3.3 Statically and Dynamically Typed Languages

Dynamic languages (e.g., Ruby, Python, JavaScript, or Groovy) have gained popularity in recent years. Features such as dynamic typing enable individuals and small teams of programmers to rapidly develop applications and engage in prototype exploration, while test-driven development mitigates the loss of type checking when moving away from a statically typed environment (e.g., Java, C, or C++). Although not a characteristic of every dynamic language, in many cases there is a great reduction of boilerplate code, making the code much more expressive and readable by humans. As a practical reality, some computation tasks are better done in a dynamic environment and others in a static one. Moreover, when two languages share object orientation, integration issues are more easily localized. Groovy steps into this "sweet spot," bringing advanced dynamic language features to one of the most widely used, robust, and well-supported (statically typed) language, Java, via seamless integration. (As of Groovy 2.0, Groovy can also be used as a statically typed language itself.)

The dynamically typed nature of ReLogo affords the modeler the ability to pay attention to or ignore type information as desired. As a simple example, in Figure 3 the local variable `varOne` has the dynamic `def` type. The modeler expects `varOne` to be a turtle so they can move the turtle by issuing the turtle's `forward` command. The compiler allows this and (optionally) underlines `forward` as an indication that the method may not exist on `varOne`. For `varTwo` the type is specified so the `forward` method is known to exist on the variable. In addition to it's convenience for modeling, dynamic typing can be very helpful for pedagogical purposes when conveying to a lay audience the essence of a model without allowing them to get distracted by the potentially extraneous type information.



```
def varOne = oneOf(turtles())
varOne.forward(1)

Turtle varTwo = oneOf(turtles())
varTwo.forward(1)
```

Figure 3: Dynamically typed expressions with `def` type (`varOne`) and explicit type (`varTwo`) variables. The `forward` method call is underlined following `varOne` indicating that the method is a statically unknown reference (which may get resolved at runtime). This *semantic highlighting* is an optional setting.

Models developed with ReLogo can seamlessly use statically compiled components (Groovy or Java) for computationally intensive parts if computational bottlenecks are identified. In addition, since Java 7 the `invokedynamic` instruction (Rose 2009) was introduced to the JVM in an effort to improve compilers and runtime systems for dynamic languages. Groovy has begun utilizing `invokedynamic` which, among other benefits, results in faster execution of dynamically typed code. Looking ahead, the performance of dynamically typed code is expected to continue to improve as the JVM does a better job optimizing `invokedynamic`, potentially allowing the modeler to more often remain focused on the modeling domain rather than have to consider the statically or dynamically typed nature of their code.

3.4 Domain Specific Languages

Domain specific languages (DSLs) are increasingly used to enable subject matter experts in a variety of fields to take advantage of the power and convenience afforded by advancements in computation. A DSL is a language and associated idioms and concepts developed for a specific domain. A DSL can simplify programming in a given domain by focusing on the concepts and constructs relevant to that domain. DSLs can also introduce paradigm changing research methodologies into areas that have not traditionally relied on computation.

The ReLogo DSL is an agent-based modeling DSL that combines the Logo world of PLOT entities and their associated primitive operations with a simplified syntax, resulting in a focused and semantically simple yet powerful package. A summary of the PLOT primitive categories are show in Table 1. For further details see (Ozik 2013b).

Table 1: Summary of ReLogo primitive categories for PLOT entities and Utility.

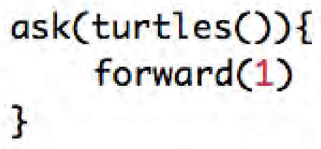
Turtle	Patch	Link	Observer	Utility
Motion				Color
Spatial	Spatial			String
Turtle creation	Turtle creation	Turtle creation	Turtle creation	Collection
Rotation				AgentSet
Turtle property	Patch property	Link property		Math
Ask	Ask	Ask	Ask	Random number
Turtle-centric	Turtle-centric	Turtle-centric	Turtle-centric	Time and ticks
Patch-centric	Patch-centric	Patch-centric	Patch-centric	Input/Output
Link-centric	Link-centric	Link-centric	Link-centric	
AgentSet	AgentSet	AgentSet	AgentSet	
World	World	World	World	
Query	Query			Query
Pen		Tie	Diffusible	
Visibility		Visibility	Clear	
Miscellaneous	Miscellaneous	Miscellaneous	Miscellaneous	Miscellaneous

In addition, whenever a new turtle type is defined a number of methods become automatically available to the PLOT entities. These methods are dynamically generated as source code, with accompanying auto-generated documentation, into the parent classes of the model PLOT entities and are made available to the compiler. As an illustrative example, the specific methods generated when a Zombie turtle type is defined are listed in Table 2. These are similar in functionality to existing ReLogo primitives with similar names but specialized for the particular turtle type (in this case the Zombie type). Analogously, when a new link type is defined a number of link methods become available to PLOT methods. See (Ozik 2013a) for further details on the generated link methods.

Table 2: Methods generated when a Zombie turtle type is defined.

Generated for each			
Turtle	Patch	Link	Observer
hatchZombies	sproutZombies		
zombiesHere	zombiesHere		
zombiesAt	zombiesAt		
zombiesOn	zombiesOn	zombiesOn	zombiesOn
isZombieQ	isZombieQ	isZombieQ	isZombieQ
zombies	zombies	zombies	zombies
zombie	zombie	zombie	zombie
			createZombies
			createOrderedZombies

Some of the simplified syntax in ReLogo is achieved with the use of code blocks, or closures, in combination with the `ask` primitive. Within an `ask` block it is understood that the commands should be executed on the entity or entities being `ask`-ed, alleviating the need to specify this (see Figure 4).



```
ask(turtles()){
    forward(1)
}
```

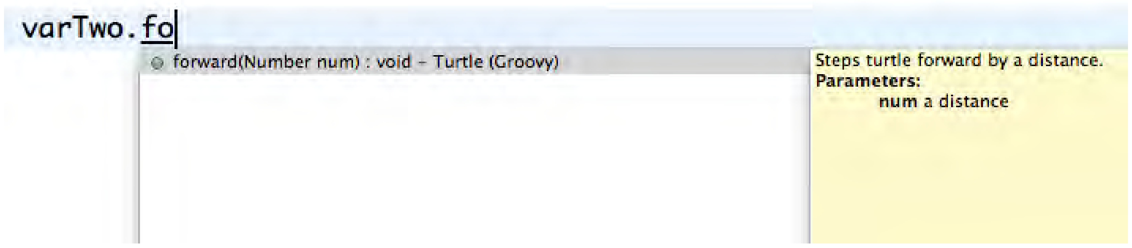
Figure 4: The `ask` primitive in ReLogo. The `ask` command switches context to the entity or entities being `ask`-ed.

Perhaps most importantly, the simplicity and flexibility of the ReLogo DSL can itself be used as a basis for developing further DSLs. See (Ozik et al. 2012) for an example of a DSL that was developed from the ReLogo DSL to represent geopolitical scenarios.

3.5 Integrated Development Environment

Integrated development environments (IDEs) have become an increasingly common way for developing computer code. With features that help the developer with common tasks (e.g., compilation/building, collaborating in teams, template code generation) and the ability to create specification of specialized components (e.g., graphical user interfaces, model logic), great efficiency is gained when compared to more simple and ad hoc text editor based development approaches. Eclipse (Eclipse Foundation 2013) is a widely used, free, and open source IDE. Eclipse can be used to develop code in many languages, including Java and C++. Repast Simphony uses Eclipse as its primary development environment, leveraging Eclipses plug-in architecture to provide a rich set of development options.

ReLogo leverages many of the Eclipse IDE's capabilities to help model builders create robust models efficiently. One of the most helpful features offered by IDEs is code-completion, a way to suggest possible completions for the code being typed. This directly connects the modeler with the ReLogo application programming interface (API), without having to look for documentation elsewhere. Figure 5 shows an example of code-completion at work, including the specification of the suggested completion.



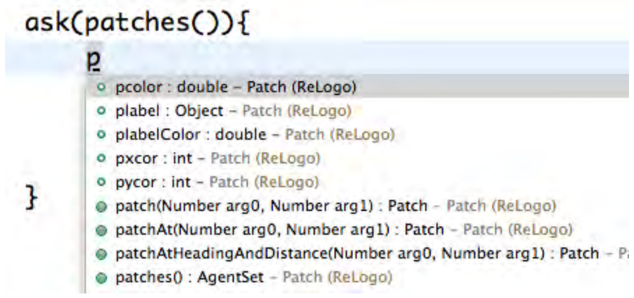
```
varTwo.fo
```

forward(Number num) : void - Turtle (Groovy)

Steps turtle forward by a distance.
Parameters:
num a distance

Figure 5: Code completion in Repast Simphony ReLogo. As code is typed possible completions and related information is displayed.

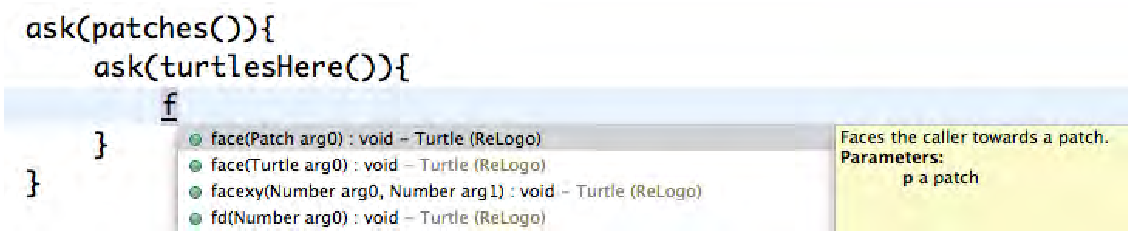
The Groovy Eclipse plugin (Groovy Eclipse Plugin 2013) offers the ability to specify DSL descriptors (DSLDs) that ”describe domain-specific extensions to Groovy-Eclipse's inferencing engine and content assist” (Groovy Eclipse DSLD 2013). ReLogo defines a custom DSLD that allows the inference engine to determine the appropriate context for context changing methods such as the `ask` method. In Figure 6 we see that within an `ask` command block where patches are being `ask`-ed the inference engine correctly understands that patches are the relevant entities and suggests patch appropriate completions. In Figure 7, on the other hand, there is a nested sequence of `ask` method calls, and the inference engine knows to suggest turtle appropriate completions within the innermost command block.



```
ask(patches()){
    p
}
```

```
pcolor : double – Patch (ReLogo)
plabel : Object – Patch (ReLogo)
plabelColor : double – Patch (ReLogo)
pxcor : int – Patch (ReLogo)
pycor : int – Patch (ReLogo)
patch(Number arg0, Number arg1) : Patch – Patch (ReLogo)
patchAt(Number arg0, Number arg1) : Patch – Patch (ReLogo)
patchAtHeadingAndDistance(Number arg0, Number arg1) : Patch – P
patches() : AgentSet – Patch (ReLogo)
```

Figure 6: Code completion within an `ask` command where patches are being `ask`-ed.



```
ask(patches()){
    ask(turtlesHere()){
        f
    }
}
```

```
face(Patch arg0) : void – Turtle (ReLogo)
face(Turtle arg0) : void – Turtle (ReLogo)
facexy(Number arg0, Number arg1) : void – Turtle (ReLogo)
fd(Number arg0) : void – Turtle (ReLogo)
```

Faces the caller towards a patch.
Parameters:
p a patch

Figure 7: Code completion within two nested `ask` commands, where `ask`-ed patches are in turn `ask`-ing turtles.

4 REPAST HPC RELOGO

Repast HPC is a C++ based high performance computing ABM toolkit that is part of the Repast Suite (Collier and North 2012). Simulations can be written directly in C++ or using the constructs in Repast HPC ReLogo (or HPC ReLogo). HPC ReLogo aims to promote ease of use and hides many of the complexities of implementing a parallel simulation. Like its Java counterpart, HPC ReLogo uses the core PLOT entities and many of the associated ReLogo primitives. The need for the simulations to run on large distributed architectures does introduce some semantic differences, including the fact that the ReLogo world is distributed across computational processes and that observers are defined *per process* rather than per ReLogo world. There are also a few differences due to the fact that Java (and Groovy) do not map directly to all C++ constructs. For example, while HPC ReLogo does use the `ask` construct, rather than taking blocks of code as arguments, the `ask` accepts functors-type objects (Collier and North 2012). Additionally, the models themselves are only run as *headless* processes (i.e., without a graphical user interface).

Perhaps the largest differences between the two ReLogo variants are that, due to the inherent complications of parallel programming, HPC ReLogo requires a lot more computational expertise. Nevertheless, one of the goals of HPC ReLogo is to provide a pathway for agent-based modelers to dramatically increase the scale of their simulations when necessary. Thus, a moderately large Repast Simphony ReLogo (RS ReLogo) model, taking advantage of all the associated conveniences of developing within Repast Simphony, can be ported to HPC ReLogo to run at dramatically larger scales without having to re-conceptualize the fundamental logic of the model.

As an example of this process, an epidemiology model (Macal et al. 2012) was ported from RS ReLogo to HPC ReLogo. In this case, a single developer was able to port the entire model written in RS ReLogo to HPC ReLogo in approximately a week. The resulting model could be run more efficiently with millions of agents. The developer was already familiar with the API of both ReLogo variants, but the similarity of the ReLogo constructs and their expression in the two APIs did provide a smooth pathway for scaling up the simulation.

5 CONCLUSION

This article has presented an overview of the the ReLogo agent-based modeling domain specific language. Combining the semantic simplicity of Logo with the capabilities of Repast Simphony, Repast Simphony ReLogo introduces a number of computational concepts to building Logo-like ABMs in the Java ecosystem, including object-oriented programming, simple integration of existing code libraries, statically and dynamically typed languages, domain specific languages, and the use of integrated development environments. Repast HPC ReLogo extends Logo constructs into distributed high-performance computing. Together, the two ReLogo variants seek to create ABM tools that are easy to learn yet are also capable of creating large scale ABMs of real world complex systems.

ACKNOWLEDGMENTS

This work is supported by the U.S. Department of Energy, Basic Energy Sciences, Office of Science, under contract #DE-AC02-06CH11357.

REFERENCES

Booch, G., R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston. 2007, April. *Object-Oriented Analysis and Design with Applications*. 3 ed. Addison-Wesley Professional.

Collier, N., T. R. Howe, and M. J. North. 2003. "Onward and Upward: The Transition to RePast 2.0". *First Annual North American Association for Computational Social and Organizational Science Conference*.

Collier, N., and M. North. 2012, November. "Parallel agent-based simulation with Repast for High Performance Computing". *SIMULATION*.

Eclipse Foundation 2013. "Eclipse". http://eclipse.org/.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994, November. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 ed. Addison-Wesley Professional.

Groovy 2013. "Groovy - Home". http://groovy.codehaus.org/.

Groovy Eclipse DSLD 2013. "Groovy Eclipse DSLD". http://groovy.codehaus.org/DSL+Descriptors+for+Groovy-Eclipse.

Groovy Eclipse Plugin 2013. "Groovy Eclipse Plugin". http://groovy.codehaus.org/Eclipse+Plugin.

Harvey, B. 1997. *Computer science logo style: Symbolic computing. Vol. 1*. MIT Press.

Holmevik, J. R. 1994, December. "Compiling SIMULA: A Historical Study of Technological Genesis". *IEEE Ann. Hist. Comput.* 16 (4): 2537.

Logo Foundation 2013. "Logo Foundation". http://el.media.mit.edu/logo-foundation/.

Macal, C. M., M. J. North, N. Collier, V. M. Dukic, D. S. Lauderdale, M. Z. David, R. S. Daum, P. Shumm, R. S. Daum, J. A. Evans, J. R. Wilder, and D. T. Wegener. 2012. "Modeling the spread of community-associated MRSA". In *Proceedings of the Winter Simulation Conference*, WSC '12, 73:173:12. Berlin, Germany: Winter Simulation Conference.

North, M. J., N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. 2013, March. "Complex adaptive systems modeling with Repast Simphony". *Complex Adaptive Systems Modeling* 1 (1): 3.

Ozik, J. 2013a. "ReLogo Getting Started Guide". http://repast.sourceforge.net/docs/ReLogoGettingStarted.pdf.

Ozik, J. 2013b. "ReLogo Primitives". http://repast.sourceforge.net/docs/api/repast_simphony/ReLogoPrimitives.html.

Ozik, J., N. Collier, M. North, W. Rivera, E. Palomaa, and D. Sallach. 2012, July. “The vmStrat Domain Specific Language”. In *Advances in Applied Human Modeling and Simulation, Advances in Human Factors and Ergonomics Series*, 447–459. CRC Press.

Resnick, M. 1996. “StarLogo: an environment for decentralized modeling and decentralized thinking”. In *Conference Companion on Human Factors in Computing Systems*, CHI ’96, 1112. New York, NY, USA: ACM.

Rose, J. R. 2009. “Bytecodes meet combinators: invokedynamic on the JVM”. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL ’09, 2:12:11. New York, NY, USA: ACM.

Sallach, D. L., and C. M. Macal. 2001, August. “Introduction The Simulation of Social Agents”. *Social Science Computer Review* 19 (3): 245–248.

Tisue, S. 2004. “NetLogo: Design and implementation of a multi-agent modeling environment”. In *Proceedings of Agent 2004*. Chicago, USA: Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence.

Wilensky, Uri 1999. “NetLogo”. http://ccl.northwestern.edu/netlogo/.

AUTHOR BIOGRAPHIES

JONATHAN OZIK, Ph.D., is computational scientist at the Center for Complex Adaptive Agent Systems Simulation within the Decision and Information Sciences Division of Argonne National Laboratory and a Fellow at the Computation Institute at the University of Chicago. His email address is jozik@anl.gov.

NICHOLSON COLLIER, Ph.D., is a software engineer at the Center for Complex Adaptive Agent Systems Simulation within the Decision and Information Sciences Division of Argonne National Laboratory and a staff member at the Computation Institute at the University of Chicago. His email address is ncollier@anl.gov.

JOHN T MURPHY, Ph.D., is a post-doctoral appointee at the Center for Complex Adaptive Agent Systems Simulation within the Decision and Information Sciences Division of Argonne National Laboratory and a staff member at the Computation Institute at the University of Chicago. His email address is jtmurphy@anl.gov.

MICHAEL J NORTH, MBA, Ph.D., is the deputy director of the Center for Complex Adaptive Agent Systems Simulation within the Decision and Information Sciences Division of Argonne National Laboratory and a Senior Fellow at the Computation Institute at the University of Chicago. His email address is north@anl.gov.