

ROAD MAP: TRANSFORMING AND EXTENDING REPAST WITH GROOVY

J. OZIK,* Argonne National Laboratory, Argonne, IL
and The University of Chicago, Chicago, IL
M.J. NORTH, Argonne National Laboratory, Argonne, IL
and The University of Chicago, Chicago, IL
D.L. SALLACH, Argonne National Laboratory, Argonne, IL
and The University of Chicago, Chicago, IL
J.W. PANICI, Northern Illinois University, DeKalb, IL

ABSTRACT

This paper discusses the integration of the dynamic object-oriented programming language Groovy into the Repast Symphony platform. It shows how the integration of Groovy benefits agent-based modeling in a number of ways, including its ability to simplify agent behavior specifications, create expressive and human-readable code, generate complex and adaptive agent behavior, and move between agent-modeling environments.

Keywords: Repast Symphony, Groovy, agent-based modeling, dynamic languages

INTRODUCTION

Dynamic languages have gained popularity in recent years. Groovy (König et al. 2007) is such a dynamic language with the added benefit of tight integration with Java and, hence, the ability to integrate into the Repast (ROAD 2005) agent-modeling platform. Repast Symphony (Repast S) is the latest extension to the Repast portfolio, a widely used, free, and open-source agent-based modeling and simulation (ABMS) toolkit (North et al. 2005a,b). Repast S offers a variety of approaches for developing and executing simulations, and it includes many advanced features for agent storage, display, and behavioral activation and new facilities for data analysis and presentation. This paper explores a few aspects of Groovy integration in Repast S.

A dynamic language such as Groovy is vital for implementing the next generation of ABMS models, such as interpretive agent (IA) models. Although in most agent models, meaning is stipulated by the model designer, interpretive agents are designed to discern and attribute meaning from within the model and shape their communications and actions accordingly.

Groovy capabilities can help achieve such a design goal in multiple ways. Tasks can be defined as closures and passed among agents as needed. Examples of such uses are commands or directives, skills acquired through training, and the imitation of rituals. Each of these can convey a closure to achieve its effect, and furthermore, the closure can be customized by context, arguments, agent state, composition with other closures, or additional embedded closures. In whatever way the customization is achieved, the effect is to calibrate the way in which the

* Corresponding author address: Jonathan Ozik, Argonne National Laboratory, Bldg. 900/DIS, 9700 S. Cass Ave., Argonne, IL 60437; e-mail: jozik@anl.gov.

directive, specialty, or ritual is carried out in a particular situation. Similarly, the creative use of iterators allows diverse agents to receive closures customized to their history and/or circumstances.

The existence of metaobjects allows routine activities to proceed as expected while possible changes in circumstances that would require the construction of a more innovative response are monitored. The metaobjects allow agents to have unique or path-specific capabilities or action orientations.

As such implementations of these dynamic capabilities are incorporated into Repast S, the toolkit will be able, through Groovy, to provide systematic support for IA models. An IA reference application (Interpretive Heatbugs) has been ported to Groovy, to both explore such capabilities and illustrate their use. All such innovations contribute to the forward potential of social agent simulation and the tools necessary to its realization.

As an added example of Groovy/Repast S integration, we introduce ReLogo, a pathway from the NetLogo (Wilensky 1999) “multi-agent programmable modeling environment” to Repast S via Groovy. NetLogo promises easy entry into agent-based modeling for anyone: from elementary school children, for whom it was originally designed, to advanced researchers.

ReLogo involves the creation of a NetLogo lexer and parser in ANTLR (Parr 1989) and the emission of Groovy translations of NetLogo code. The interpreted nature of the NetLogo language makes it especially useful to use a dynamic language such as Groovy as the target language. The expressiveness of Groovy results in easily readable code that retains a lot of the structure of the original agent code. As a consequence, the code can be easily extended and modified within the Repast S environment.

This paper is structured as follows. Section 2 gives a brief introduction to the Groovy dynamic programming language, focusing on the dynamic features that are most relevant to agent modeling. Section 3 presents Groovy integration with Repast S, specifically focusing on some aspects of the Repast S Agent wizard and the visual agent behavior editor, as well as the use of Groovy categories to carry out a variety of mathematical operations, including automatic unit conversions and matrix and calculus operations. The Groovy/Repast S port of Interpretive Heatbugs is discussed in Section 4, which highlights some of the dynamic Groovy capabilities. Finally, ReLogo is briefly presented in Section 5, and conclusions are given in Section 6.

It is important to note that Repast S and its related tools are still under development. This paper presents the most current information at the time it was written. However, changes may occur before the planned final release.

GROOVY

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 it is not a characteristic of every dynamic

language, there is usually a great reduction of boilerplate code, making the code much more expressive and readable by humans.

An important point to note, though, is that a lot of the ongoing debate that pits dynamic languages against static languages ignores the reality that some 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, such as closures, dynamic typing, and the metaobject protocol, to one of the most widely used, robust, and well-supported (statically typed) language, Java, via seamless integration. In fact, since JSR-241 (Java Specification Request), Groovy is the second standard language for the Java platform (the first one being Java).

According to the Groovy Web site (<http://groovy.codehaus.org/>), Groovy:

- “is an agile and dynamic language for the Java Virtual Machine”
- “builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk”
- “makes modern programming features available to Java developers with almost-zero learning curve”
- “supports Domain Specific Languages and other compact syntax so your code becomes easy to read and maintain”
- “makes writing shell and build scripts easy with its powerful processing primitives, object-oriented structure and an Ant DSL”
- “increases developer productivity by reducing scaffolding code when developing web, GUI, database or console applications”
- “simplifies testing by supporting unit testing “
- “seamlessly integrates with all existing Java objects and libraries”
- “compiles straight to Java bytecode so you can use it anywhere you can use Java”

There are a number of capabilities that Groovy brings to the realm of agent-based modeling. First of all, closures allow agent behaviors to be specified as first-class Closure objects. As taken from the Groovy Web site:

A closure in Groovy is an anonymous chunk of code that may take arguments, return a value, and reference and use variables declared in its surrounding scope. In many ways it resembles anonymous inner classes in Java, and closures are often used in Groovy in the same way that Java developers use anonymous inner classes.

The closures can be customized by context, arguments, agent state, composition with other closures, or additional embedded closures. In whatever way the customization is achieved, the effect is to calibrate the way in which the agent behavior is carried out in a particular situation. In addition, closures can be passed around between agents, enabling agents to acquire capabilities or learn from each other.

The metaobject protocol in Groovy allows for the specification of agent metaclasses. Metaclasses can be used to interrupt routine agent activities when changes of agent circumstances require the agents to construct more innovative responses. Furthermore, with Expando Metaclasses, agent classes can be modified as necessary at run time. These capabilities allow agents to have unique or path-specific capabilities or action orientations.

Finally, Groovy's tight integration with Java provides the ability to (a) seamlessly incorporate Groovy into the sophisticated agent-based modeling platform Repast S and (b) take advantage of the vast amounts of existing Java libraries that support agent-based modeling.

GROOVY INTEGRATION IN REPAST S

Visual Agent Behavior Editor

Repast S includes a point-and-click agent editor. The visual agent behavior editor, the new project wizard, the new agent wizard, and related components are modified forms of Alexander Greif's free and open-source Flow4J-Eclipse components (Greif 2006) that have been adapted specifically for agent-based modeling. Greif (2006) has made the Flow4J-Eclipse system available under a BSD-style free and open-source license. The Repast project team has built on Greif's contribution to create the above-mentioned Repast S components. According to the Flow4J home page (Greif 2006):

Flow4J-Eclipse is an Eclipse Plug-in for modeling process flows in a drag and drop manner. A process flow can contain process steps (I call them flowlets), which can be linked together [in]to a complex flow.

Flow4J-Eclipse used two types of "flowlets" (Greif 2006) — namely "Control Flowlets like Start-, Decision- and Jump- Flowlets which are configurable in Eclipse" — that "tell 'how' the process should flow" and "Task Flowlets [that] accomplish a specific task that is wrapped in a Java class," which can come from many sources, including Java itself or "scripting languages like Jython, Groovy, JavaScript, etc." Grief (2006) notes the following:

After designing the flows in the Eclipse Plug-in, the flow's Java source code is automatically created, and is immediately ready for compilation and deployment in a Java web/application. The flow's generated Java source code is highly optimized and lightning fast.... Web/applications can execute flows in two ways: (1) from inside any Java code [and] (2) triggered by a HTTP request if the flow is deployed in a web application.

The Flow4J-Eclipse editor works by presenting users with a flowchart-drawing interface that is used to create a process diagram (Grief 2006). The process diagram is saved in a platform-independent XML file. Following the Eclipse development model, the Flow4J-Eclipse system registers a builder with Eclipse that is notified whenever a Flow4J-Eclipse XML file changes (Grief 2006). This builder creates a Java source file from the freshly saved XML file. Eclipse itself then automatically generates a Java binary class file from the new Java source code file. The Flow4J-Eclipse system also includes an Eclipse wizard for creating new Flow4J-Eclipse projects, an Eclipse wizard for creating new flowchart XML files, and supporting tools such as a

project menu option to add the Flow4J-Eclipse nature to a standard project (i.e., activate the Flow4J-Eclipse builder for the project).

Repast agent editor adapts Greif's Flow4J-Eclipse to agent-based modeling in multiple ways:

1. The Repast agent editor's builder generates Groovy source code rather than Java source code to take advantage of a number of Groovy features.
2. The Repast agent editor has enhanced the Flow4J-Eclipse property views with step-by-step form-style inputs.
3. The Repast agent editor redefines the Flow4J-Eclipse flowlets to correspond to agent behavior primitives and properties rather than "process flows" (Greif 2006):
 - a. Flow4J-Eclipse "Flows" have been redesigned to support Repast agent class definitions, including user input for super classes and implemented interfaces.
 - b. Flow4J-Eclipse "Template Flowlets" have been redesigned as "Property Components" to support agent attributes.
 - c. Flow4J-Eclipse "Start Flowlets" have been redesigned as "Behavior Components" to support Repast scheduling and watchers (Howe et al. 2006; Parker et al. 2006).
 - d. Flow4J-Eclipse "Task Flowlets" have been redesigned as "Task Components" to support Repast agent activities.
 - e. Flow4J-Eclipse "Decision Flowlets" have been redesigned to use Groovy logical conditions instead of Flow4J-Eclipse predicate lookup tables.
 - f. Flow4J-Eclipse "Call Flowlets" have been redesigned as "Loop Components" to loops instead of method calls.
 - g. Flow4J-Eclipse "Jump Flowlets" have been removed.
4. The Repast agent editor includes the ability to embed user-specified comments into the generated Groovy source code.
5. The Repast agent editor's XML storage schema has been modified versus Flow4J-Eclipse to reflect the above-cited differences in the agent behavior primitives and properties (Greif 2006). Furthermore, to differentiate the contents, the Repast agent editor uses a different file extension for its XML files than the Flow4J-Eclipse system.
6. The Repast agent editor automatically generates a set of supporting components within each agent. For example, each generated Repast agent class includes an attribute and method for automatically assigning the agent a human-readable run time identifier.
7. The Repast system has expanded the functionality of the Flow4J-Eclipse wizard for creating new projects by adding a variety of features, such as the following:
 - a. Repast Score file input specification and generation has been added.
 - b. A large number of Repast-specific supporting files and directories are now generated for tasks such as defining batch runs, defining legacy model descriptors, and building model installations.

- c. Eclipse launch scripts are added for executing interactive model runs, executing batch model runs, and starting Repast in default mode.
- d. Repast library dependencies are included in the generated project.
8. The wizard for creating new flowchart XML files now creates Repast-specific files.
9. The Flow4J-Eclipse nature has been modified to include Repast library dependencies.

Figure 1 shows an example flowchart in the agent editor, while Figure 2 shows the corresponding Groovy code. Please see North et al. (2007) for more details on how to use the agent editing system.

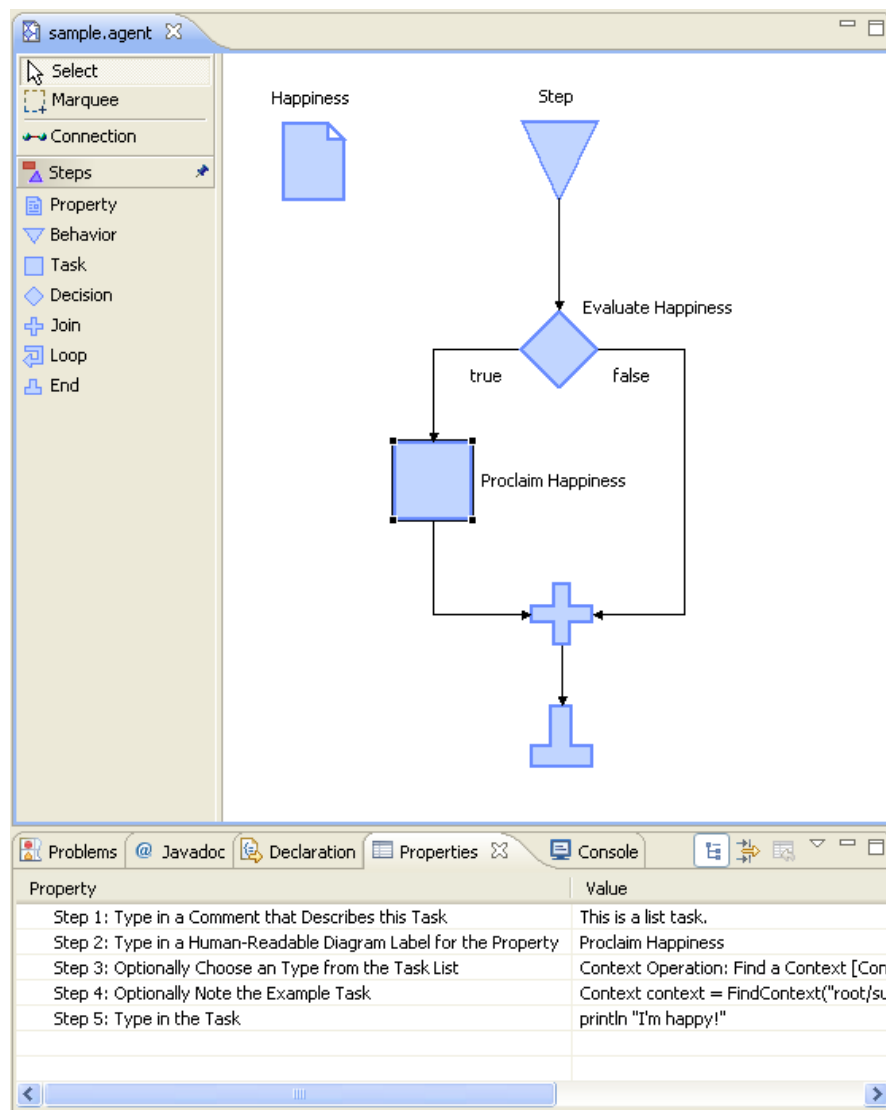


Figure 1 Example flow chart and Properties pane in the Repast S visual agent behavior editor

```

...
/**
 *
 * This is an agent property.
 * @field happiness
 *
 */
@Parameter (displayName = "Happiness", usageName = "happiness")
public def getHappiness() {
    return this.happiness;
}
public void setHappiness(def newValue) {
    this.happiness = newValue;
}
public def happiness = 0;

/**
 *
 * This is the step behavior.
 * @method step
 *
 */
@ScheduledMethod(
    start = 1d,
    interval = 1d
)
public void step() {
    // Define the return value variable.
    def returnValue
    // Note the simulation time.
    def time = GetTickCountInTimeUnits();
    // Use the Repast Symphony Groovy math tools.
    use (MathOperations.mathCategories()) {
        // Make a decision.
        if (happiness > 0.5) {
            // This is a list task.
            println "I'm happy!";
        } else {

        }
        // Exit this scope.
        return;
    }
}
...

```

Figure 2 Condensed parts of the automatically generated Groovy code corresponding to the Repast S visual agent behavior editor flowchart in Figure 1

JScience and Groovy Categories

JScience is a Java library developed to “provide the most comprehensive Java library for the scientific community” (JScience 2005). Users developing a Repast S model could reference the JScience library like they would any other Java library, but, as will be demonstrated below, our intention was to make this process more user-friendly and create more human-readable code. To do this, we employed Groovy Categories and simplified the use of a subset of the JScience capabilities: the scientific unit and unit conversion modules and linear algebra modules.

Categories are useful for situations in which one would like to define additional methods on classes that are not under one’s control. Specifically, a Groovy category is a class that contains a set of static methods (called *category* methods). Each of these methods is made available on the class of the method’s first argument. The following example helps to clarify these concepts.

Figure 3 shows an example of the Java code needed to define Amount class objects x and y with the values 2 kilometers and 5,000 feet, respectively, along with the calculation of their sum, z , which properly accounts for the unit conversions by setting z to 3.524 kilometers.

```
Amount x = Amount.valueOf(2.0, SI.KILO(SI.METER));
Amount y = Amount.valueOf(5000.0, SI.FEET);
Amount z = UnitsOperations.addition(x,y);
```

Figure 3 Java code utilizing the Amount JScience class

Figure 4 shows the Groovy version of Figure 3 using categories.

```
def x = 2.kilometers
def y = 5000.feet
def z = x + y
```

Figure 4 Groovy code corresponding to Java code in Figure 3 (at end $z = 3.524$ km)

This simplification is possible for two main reasons. The first is that Groovy (unlike Java) allows one to override *operators*. Hence, in the example above, we were able to override the “+” and the “.” operators, resulting in intuitive and human-readable code. The second reason we can simplify is because we can enclose any lines of code within a closure and, by using the *use* keyword and referencing the appropriate Groovy category, [for example, `use(UnitsCategory) {lines of code}`], we can force the operators (or methods) within those lines of code to be overridden by the category’s [in this case, `UnitsCategory`’s] static methods. The lines of code in Figure 4 are located within such a closure.

In an analogous way, by using the `MatrixCategory`, which references the JScience linear algebra classes, matrix operations (e.g., matrix addition, subtraction, multiplication, and exponentiation) can be written in the simple syntax shown in Figure 5.

```

def c = a + b
def d = a - b
def e = a * b
def f = a ** 2

```

Figure 5 Matrix operations using Groovy categories (a and b are matrices)

In addition, we have also included a `CalculusCategory`, which allows for calculating derivatives and integrals of closures.

Such capabilities are built into Repast S, enabling the model developer to tap into these powerful functionalities and, at the same time, create human-readable and thereby more maintainable code.

INTERPRETIVE HEATBUGS

The purpose of the IA research program is to incorporate endogenous meaning attribution as a means of orienting agent communication and action selection into agent-based modeling (Sallach 2003). Interpretive Heatbugs (IHB) is an IA reference application in which interpretive mechanisms (prototype reasoning, situation definition, and orientation accounting) are developed, illustrated, and made available to other researchers (see Sallach and Mellarkod 2005; Mellarkod and Sallach 2005; Sallach and Ozik 2007). Similar to its widely known generic heatbug predecessor (Swarm Development Group 1997), it is designed to provide a simple introduction to the IA paradigm.

IHB uses the familiar heatbugs environment, in which heat-emitting bugs require temperate zones and flee from settings that are uncomfortably hot or cold. Because each bug emits a small amount of heat, congregations of bugs initially create the needed warmth, while overcrowding creates excessive heat. These competing influences give rise to the often-observed complex, churning patterns. To this dynamic of temperature fluctuations driving the bug movements, IHB adds the capability of bugs to ignore, engage in voluntary exchange with, or undercut each other, and it also adds ethnic and religious identities that mediate the decisions to help or hinder.

The IHB application explores the role of interpretation: in the use of force as a means of improving bug circumstances; in the bugs' decisions to request gifts of energy (a placeholder for health and/or wealth); and in the decisions to convey all, part, or none of the energy requested. These decisions draw on the projection of a bug's comfort levels on others, as well as the prototype categorization of all the bugs accessible for interaction. Of particular interest is the emergence of endogenous evaluative classification in bug orientation and action.

Consistent with the concept of a reference application, the interpretive mechanisms that support the acts of aggression, requests, and voluntary contributions are designed to support comparable decisions in a variety of complex social applications (e.g., models of cultural conflicts, such as genocide and ethnic cleansing, as well as diversifying markets and a range of extended cultural processes). The IHB application, which was initially implemented in the

J programming language (Iverson and Hui 1990), has been ported to the Repast S platform (see Figure 6) by using Groovy’s dynamic capabilities and making these mechanisms available for the development of increasingly socially “thick” and interactive agent models. What follows is an illustrative set of examples of Groovy-isms used within IHB.

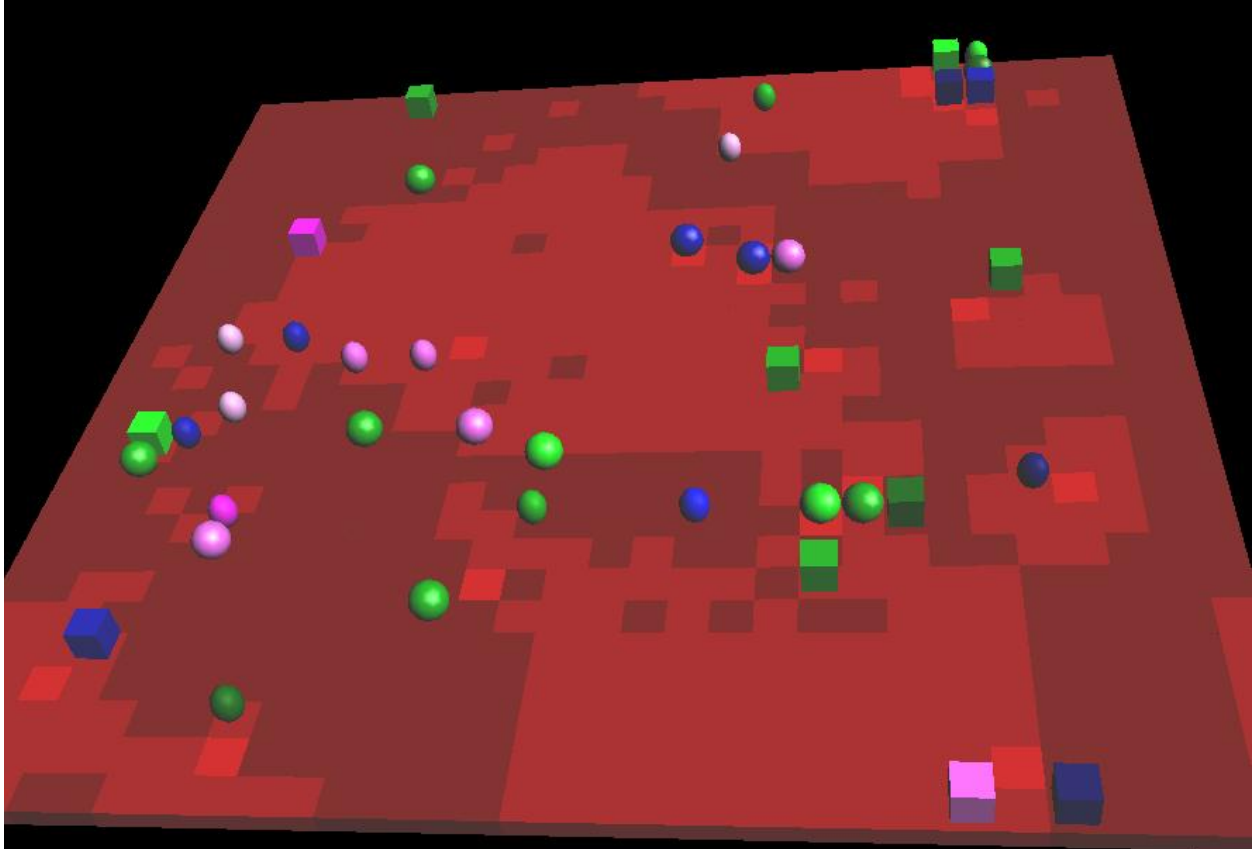


Figure 6 Screenshot of the 3D grid projection from the Groovy/Repast S port of IHB

The bugs make assessments of the bugs that they come into contact with. This involves creating prototype clusters that group a bug’s known bugs into categories on the basis of each bug’s idiosyncratic history and observations. The current implementation of IHB employs a hierarchical agglomerative clustering algorithm for these groupings and uses the standard Euclidean metric. However, it also includes the flexibility to introduce any user-defined metric via closures, as is shown in the Groovy code snippet in Figure 7.

```
protected def cluster(int maxClusters, def ptsToCluster, Closure metric) {
  ...
}
```

Figure 7 Groovy code illustrating the `cluster` method taking a Closure object as a parameter

Depending on the type of bug and its situated context, a bug will adopt a particular set of behaviors for shoving, asking, and giving. As the bugs' shove, ask, and give rules are all implemented as closures, the bugs' behaviors can be assigned and modified easily by using rule dispatchers, which pass bug behaviors, specified as closures, to each bug as necessary. This allows for not only a flexible way in which to determine bug behavior but also an intuitive and straightforward way to encapsulate bug behavior, in a first-class Closure object, instead of requiring a lot of scaffolding code (for example, in a method within a first-class object).

One of the most powerful uses of closures in IHB has to do with a bug's `more` and `all` methods. In many situations, a bug's action is contingent on whether most or all of the bugs under consideration fit some set of criteria, where these criteria are implemented as closures. Instead of having to create different implementations of `more` and `all` corresponding to each combination of the many criteria that a bug can employ, the set of criteria are passed to the `more` and `all` methods as closures. Then Groovy's handy iterating mechanisms iterate the bugs under consideration. This is a good example of using combinations of basic closures to implement complex agent behaviors. The closures can be composed, combined, and reordered to provide flexibility in creating new and adaptive agent behaviors.

RELOGO

ReLogo is a pathway from the NetLogo (Wilensky 1999) "multi-agent programmable modeling environment" to Repast S that uses Groovy. As previously mentioned, NetLogo promises an easy entryway into agent-based modeling (Wilensky 1999). Repast S is a very sophisticated agent-based modeling platform, offering many advanced features for agent storage, display, and behavioral activation and new facilities for data analysis and presentation, while also allowing for the integration of external (legacy) models. Thus, ReLogo offers the ability to go from an exceedingly user-friendly and intuitive environment for model development and exploration in NetLogo all the way to the enterprise-level models developed in Repast S.

ReLogo involves the creation of a NetLogo lexical analyzer or "lexer" and parser in ANTLR and the emission of Groovy translations of NetLogo code. The interpreted nature of the NetLogo language makes it especially useful to employ a dynamic language like Groovy as the translation target language. The expressiveness of Groovy results in easily readable code that retains a lot of the structure of the original agent code. As a consequence, the code can be easily extended and modified within the Repast S environment.

Although ReLogo is still under development, it is nonetheless possible to present a simple example to illustrate some of the benefits of using Groovy. One of the most common patterns observed in NetLogo models is that of sending a block of code to a set of agents. In the following NetLogo code (Figure 8), rabbit agents are created, and a block of code specifying the created rabbit's color, position, and energy is sent to each created rabbit. (Note that the block of code is interpreted by each rabbit individually; thus, each rabbit sets its own random position and energy.) Implementing this type of behavior in the static Java language would result in convoluted code that would be difficult to maintain or even read. Even if the resulting code was brought over to the Repast S environment, editing, modifying, or debugging it would become an unenviable task.

```

create-rabbits number [
  set color white
  setxy random-xcor random-ycor
  set energy random 10
]

```

Figure 8 Sample NetLogo code with code block (Rabbits Grass Weeds model) (Wilensky 1999)

On the other hand, Figure 9 shows how this agent behavior can be implemented with Groovy. The `createRabbits` method can take as parameters the *number* variable as well as the code block (as a closure). As one can see, the resulting code closely mirrors the original code in structure and in the function of the resulting code units, allowing the modeler to carry over many of the mental constructs that may have developed during model development in the NetLogo environment.

```

createRabbits(number){
  setColor(Color.white)
  setXY(randomXcor(),randomYcor())
  setEnergy(nextIntFromTo(1,10))
}

```

Figure 9 Groovy implementation of code example in Figure 8

CONCLUSIONS

This paper demonstrates the many benefits Groovy brings to the agent-based modeling realm. Groovy integration in Repast S simplified the use of the visual agent behavior editor as well as the incorporation of physical unit conversions, matrices and calculus operations into agent-based models. The Groovy and Repast S implemented Interpretive Heatbugs reference application demonstrated some of the flexibility and clarity achieved by the use of closures. Finally, this papers shows how having Groovy as the target language greatly improves the ability to move from the NetLogo to the Repast S environment via ReLogo. The authors believe that all such innovations will contribute to the forward potential of social agent simulation and the tools necessary for its realization.

ACKNOWLEDGMENTS

The authors wish to thank Alexander Greif for contributing Flow4J-Eclipse to the software development community. They also wish to thank Uri Wilensky, Seth Tissue, and Bill Rand for the helpful discussions and support received on the ReLogo project. In addition, they want to thank Eric Tataro for his helpful contributions. This work is supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

REFERENCES

- Greif, A., 2006, *Flow4J-Eclipse Home Page*; available at <http://flow4jeclipse.sourceforge.net/>
- Howe, T.R., N.T. Collier, M.J. North, M.T. Parker, and J.R. Vos, 2006, “Containing Agents: Contexts, Projections, and Agents,” in *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*, ANL/DIS-06-7, co-sponsored by Argonne National Laboratory and The University of Chicago, Sept. 21–23.
- Iverson, K., and R. Hui, 1990, *J Software Home Page*; available at <http://www.jsoftware.com> .
- JScience, 2005, *JScience*, home page; available at <http://jscience.org> .
- König, D., A. Glover, P. King, G. Laforge, and J. Skeet, 2007, *Groovy in Action*, Manning Publications.
- Mellarkod, V.S., and D.L. Sallach, 2005, “Interpretive Heatbugs: Design and Implementation,” in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, ANL/DIS-06-5, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15.
- North, M.J., T.R. Howe, N.T. Collier, and J.R. Vos, 2005a, “The Repast Symphony Development Environment,” in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, ANL/DIS-06-5, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15.
- North, M.J., T.R. Howe, N.T. Collier, and J.R. Vos, 2005b, “Repast Symphony Runtime System,” in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, ANL/DIS-06-5, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15.
- North, M.J., E. Tatar, N.T. Collier, and J. Ozik, 2007, “Visual Agent-Based Model Development with Repast Symphony,” in *Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence*, ANL/DIS-07-2, co-sponsored by Argonne National Laboratory and Northwestern University, Nov. 15–17.
- Parker, M.T., T.R. Howe, M.J. North, N.T. Collier, and J.R. Vos, 2006, “Agent-Based Meta-Models,” in *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*, ANL/DIS-06-7, co-sponsored by Argonne National Laboratory and The University of Chicago, Sept. 21–23.
- Parr, T., 1989, ANTLR, home page; available at <http://www.anltr.org>.
- ROAD (Repast Organization for Architecture and Design), 2005, *Repast, News*, Chicago, IL; available at <http://repast.sourceforge.net>.

- Sallach, D.L., 2003, "Interpretive Agents: Identifying Principles, Designing Mechanisms," in *Agent 2003, Conference on Challenges in Social Simulation*, Proceedings of the Conference, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 2–4.
- Sallach, D.L., and V.S. Mellarkod, 2005, "Interpretive Agents: A Heatbug Reference Simulation," in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, ANL/DIS-06-5, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15.
- Sallach, D.L., and J. Ozik, 2007, "Interpretive Heatbugs," Argonne National Laboratory, Argonne, IL, in preparation.
- Swarm Development Group, 1997, *Swarm*, Albuquerque, N.M.; available at <http://www.swarm.org/examples-heatbugs.html> . Albuquerque, NM.
- Wilensky, U., 1999, *NetLogo*, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL; available at <http://ccl.northwestern.edu/netlogo/>.