Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit

MICHAEL J. NORTH, NICHOLSON T. COLLIER, and JERRY R. VOS
Argonne National Laboratory

Many agent-based modeling and simulation researchers and practitioners have called for varying levels of simulation interoperability ranging from shared software architectures to common agent communications languages. These calls have been at least partially answered by several specifications and technologies. In fact, Tanenbaum [1988] has remarked that the "nice thing about standards is that there are so many to choose from." Tanenbaum goes on to say that "if you do not like any of them, you can just wait for next year's model." This article does not seek to introduce next year's model. Rather, the goal is to contribute to the larger simulation community the authors' accumulated experiences from developing several implementations of an agent-based simulation toolkit. As such, this article focuses on the implementation of simulation architectures rather than agent communications languages. It is hoped that ongoing architecture standards efforts will benefit from this new knowledge and use it to produce architecture standards with increased robustness.

Categories and Subject Descriptors: I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*Environments*; I.6.2 [**Simulation and Modeling**]: Simulation Languages

General Terms: Design, Standardization

Additional Key Words and Phrases: Agent-based Modeling and Simulation, Java, Python, Microsoft .NET

1. INTRODUCTION

Lu et al. [2000] and others have called for varying levels of simulation interoperability that range from shared software architectures to common agent communications languages. These calls have been at least partially

This research was supported by the U.S. Department of Energy under Contract W-31-109-Eng-38. The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-Eng-38a with the U. S. Deaprtment of Energy.

Authors' address: Center for Complex Adaptive Agent Systems Simulation, Decision and Information Sciences Division, Argonne National Laboratory, 9700 S Cass Avenue, Argonne, IL, 60439; email: {north,jvos}@anl.gov; nick.collier@varizon.net.

answered by several specifications and technologies. The alternatives include the Foundation for Intelligent Physical Agents' (FIPA) architecture specifications, the results of the Object Management Group (OMG) Agent Platform Special Interest Group (AP SIG), the Knowledge-able Agent-oriented System architecture (KAoS), the High-Level Architecture (HLA), and the Distributed Interactive Simulation (DIS) protocol, to name a just few [IEEE 1995a, 1995b; Bradshaw 1996, 1997; OMG 2000; IEEE 2001a, 2001b, 2001c; FIPA 2003]. In fact, Tanenbaum [1988] has remarked that the "nice thing about standards is that there are so many to choose from." Tanenbaum goes on to say that "if you do not like any of them, you can just wait for next year's model." This article does not seek to introduce next year's model. Rather, the goal is to contribute the authors' accumulated experiences from developing several implementations of an agent-based simulation toolkit to the larger simulation community. As such, this article focuses on the implementation of simulation architectures rather than agent communications languages. See Labrou et al. [1999] for a treatment of agent communications languages and Wooldridge and Jennings [1994] for a discussion of the theory underlying agent architectures and communications languages. It is hoped that ongoing architecture standardization efforts, such as those managed by FIPA, will benefit from this new knowledge and use it to produce architecture standards with increased robustness [FIPA 2003].

2. RELATED WORK

There is a significant amount of work related to agent-based simulation architectural interoperability. This related work can be classified into three main categories, namely de jure standards, de facto standards, and standards implementations. The focus here is on simulation rather than related technologies such as mobile agent systems. For a broader overview of the area, including mobile agents, see Flores-Mendez [1999].

2.1 De Jure Standards

De jure standards are "formal, legal standards adopted by some authorized standardization body" [Tanenbaum 1988]. The use of de jure standards may be either compulsory or voluntary, depending on the legal status of the standardization body [Tanenbaum 1988]. Four of the most important de jure simulation architecture interoperability standards are the results of the FIPA effort, the results of the Object Management Group (OMG) Agent Platform Special Interest Group (AP SIG), the Knowledge-able Agent-oriented System architecture (KAoS) group, and the HLA [Bradshaw 1996, 1997; IEEE 2001a; FIPA 2003]. A third de jure standard that will be briefly considered is the now obsolete Distributed Interactive Simulation (DIS) standard [IEEE 1995a, 1995b].

FIPA is a non-profit organization founded in 1996 [FIPA 2003], which has created, and continues to create, a suite of interrelated standards for intelligent physical agents. FIPA's mission is the "promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings" [FIPA 2003]. Their overall focus is broader than agent-based simulation, but this topic area

is included in their efforts. FIPA itself is organized into topical Technical Committees and Working Groups, with the Modeling Technical Committee most directly involved with agent-based simulation. Overall, the FIPA Technical Committees are spearheading a variety of standards including platform interoperability standards and notational standards [Bauer et al. 2001; FIPA 2003].

The FIPA platform interoperability standards define the required agent communication language, agent management facilities, and nonagent integration facilities [Poslad et al. 2000]. Currently, FIPA lists ten open source agent platforms as being FIPA conformant [FIPA 2003].[1] These implementations will be discussed further in the standards implementation section.

The OMG AP SIG seeks to "extend the OMG Object Management Architecture to better support agent technology" and "to promote standard agent modeling techniques that increase rigor and consistency of specifications" [OMG 2000]. Since the OMG AP SIG is working extremely closely with FIPA [OMG 2001], OMG AP SIG work will be considered along with FIPA.

KAoS seeks to "address two major limitations of current agent technology: 1. failure to address infrastructure, scalability, and security issues; and 2. lack of semantics and extensibility of agent communication languages" [Bradshaw 1997]. KAoS applies "commercial distributed object products (CORBA, DCOM, Java)" to address the first limitation [Bradshaw 1996]. The second limitation is "providing an open agent communication meta-architecture in which any number of agent communication languages with their accompanying semantics could be accommodated" [Bradshaw 1996]. KAoS infrastructure implementations will be discussed in the next section.

The HLA is an IEEE standard "general purpose architecture for simulation reuse and interoperability" that was first introduced by the U.S. Defense Modeling and Simulation Office (DMSO) [IEEE 2001a, 2001b, 2001c; DMSO 2004]. The HLA supersedes the earlier IEEE DIS standard by broadening the scope of applicability beyond just interactive simulations [IEEE 1995a, 1995b; Cavitt et al. 1997]. The HLA was originally designed "to support reuse and interoperability across the large numbers of different types of simulations developed and maintained" by the U.S. Department of Defense [DMSO 2004] and has subsequently been used worldwide [Lu et al. 2000; Sun et al. 2003].

The HLA defines a core set of services that are to be provided by all HLA compliant systems. Runtime infrastructures (RTIs) are software implementations of the HLA. These RTIs can optionally be verified by DMSO for adherence to the HLA interface specification. As before, HLA RTI implementations will be discussed further in the next section.

2.2 De Facto Standards

De facto standards "are those that have just happened, without any formal plan" [Tanenbaum 1988]. Such standards often result from accumulated market

[1]Following HLA terminology, this article describes a model as being "compliant" to a standard when the model meets the standard's requirements for implementation or interoperability. A toolkit or platform is described as "conformant" to a standard when models implemented using the toolkit are automatically "compliant."

share driven by widespread use of a given technology. De facto standards are not always recognized as such, but are often simply systems that are used widely enough to significantly influence later technologies. Important agent-based simulation systems that act as de facto standards include Swarm, NetLogo, and Repast [Wilensky 1999; ROAD 2004; SDG 2004].

2.3 Standards Implementations

Standards implementations are concrete instantiations of de jure standards or representative examples of de facto standards. FIPA conformant platforms, the HLA RTIs, Swarm, NetLogo, and Repast are example standards implementations [Wilensky 1999; FIPA 2003; DMSO 2004; ROAD 2004; SDG 2004].

There are a large number of FIPA-conformant agent toolkits, with ten open source examples alone [FIPA 2003]. However, it should be noted that FIPA focuses quite heavily on mobile agents, generalized multi-agent systems, and other topics outside of strict agent simulation. Poslad et al. [2000] added support for the following features above and beyond the mandatory FIPA requirements while implementing the FIPA Open Source (FIPA-OS) platform:

- Multiple base classes for "producing agents that can then communicate with each other using the FIPA-OS facilities" were provided. Each base class offers a different level of customizable capabilities [Poslad et al. 2000].
- "Multi-layered support for agent communication" was provided [Poslad et al. 2000].
- "Message and conversation management" features were provided [Poslad et al. 2000].
- "Abstract interfaces and software design patterns" were used throughout the design and implementation of FIPA-OS [Poslad et al. 2000].
- Specialized "diagnostics and visualization tools" were provided [Poslad et al. 2000].

Providing multiple base classes along with abstract interfaces can increase the number of architectural options and overall development flexibility available to users. Advanced communications features and specialized development tools can speed model design and implementation. The advanced communications features can also reduce mismatches between independently developed components.

Multiple KAoS implementations currently exist. Bradshaw [1996] reports that several useful lessons were learned from creating the Gaudi KAoS implementation, including the following:

- Agent-based systems should be constructed in a modular way so that "all parts [are] replaceable" [Bradshaw 1996].
- Agent-based systems should be cross-platform so users can execute them "everywhere" [Bradshaw 1996].
- Agent-based systems need connections to allow users to "pull data from anywhere" [Bradshaw 1996].

In short, swappable cross-platform components with rich data services have a definite value for agent-based systems. Unfortunately for the current topic, KAoS focuses heavily on mobile agents and other topics beyond agent simulation [Bradshaw 1996, 1997].

There are many HLA RTI implementations. In fact, DMSO has verified 37 HLA RTIs and 9 more are in various stages of verification [DMSO 2005]. Several important lessons have been learned from this work, including the following, which are related to agent simulation [Bachinsky et al. 1998]:

- Extensibility in the form of "proper object-oriented design" is essential to allow the architecture to "embrace change and be prepared to adapt to changing requirements and technologies" [Bachinsky et al. 1998].
- Designing for testability using "well defined object interfaces" is required so that "components can be developed and tested for correct functionality in isolation or in concert with the overall system" [Bachinsky et al. 1998].
- The use of design patterns, such as the state pattern, is helpful for improving implementation quality [Gamma et al. 1994]. Design patterns are a widely used industrial approach to describing tried and true solutions for commonly faced software design problems [Gamma et al. 1994; Coplien 2001].

Despite these useful observations, most of the lessons learned from implementing HLA RTIs have focused on performance [Bachinsky et al. 1998; Lu et al. 2000]. Performance is certainly an important issue for agent-based simulation toolkit implementations, but it is far from the only concern. Furthermore, the HLA is neither agent-based nor even truly object-oriented. For example, according to Myjak et al. [1999], "the problem arises in the language of the HLA Specification, which uses object-oriented terminology yet only supports actions on public attributes." Agent-based extensions to the HLA have been developed, but they are not part of the core standard nor of the superseded DIS standard [Lu et al. 2000; Aronson et al. 2003]. Therefore, much remains to be learned about implementing agent-based simulation standards.

Swarm was one of the earliest of the agent-based modeling toolkits [Minar et al. 1996]. Swarm is implemented in Objective-C and has a Java wrapper [Burkhart et al. 2000], therefore, Swarm simulations can be written in either Objective-C or Java. Unfortunately, little has been published about lessons learned from the implementation of Swarm.

NetLogo is a free agent-based simulation environment that uses a modified version of the Logo programming language [Harvey 1997; Wilensky 1999]. NetLogo's developers have learned much from the StarLogo and StarLogoT efforts [Tisue and Wilensky 2004]. NetLogo was designed to provide a basic laboratory for teaching complexity concepts; however, it can also be used to develop more complicated applications. NetLogo provides a graphical environment to create programs that control graphic "turtles" that reside in a world of "patches," which is monitored by an "observer" [Wilensky 1999]. NetLogo also includes an innovative feature called HubNet, which allows groups of people to interactively engage in simulation runs alongside of computational agents

[Wilensky and Stroup 1999]. Tisue and Wilensky [2004] have offered several lessons from the development of NetLogo, including the following:

- Agent-based simulation environments and languages should be simple enough to have a "low threshold" for beginners [Tisue and Wilensky 2004].
- Similarly, such systems should have "no ceiling" that limits what experienced users can do [Tisue and Wilensky 2004].[2]
- These systems should include a large number of example simulations to help beginning and experienced users alike.

2.4 The Remaining Need

Despite the significant amount of related work, there remains an unmet need for practical experiences with creating multiple implementations of agent-based toolkits. This article documents the authors' experiences in creating three implementations of the Repast agent modeling toolkit.

3. REPAST

The Recursive Porous Agent Simulation Toolkit (Repast)[3] is a free open source toolkit that was developed by Sallach, Collier, North, Howe, Vos, and others [Collier et al. 2003]. Repast has an abstract feature set and three concrete implementations. Repast focuses on modeling social behavior, but is not limited to social simulation. All of the Repast implementations discussed in this article, including the source code, are available directly from the web at http://repast.sourceforge.net/download.html [ROAD 2004].

Repast was created at the University of Chicago in close collaboration with Argonne National Laboratory. Subsequently, responsibility for the ongoing development of Repast was assumed by the Repast Organization for Architecture and Design (ROAD) [ROAD 2004]. ROAD is a nonprofit volunteer group led by a board of directors that includes members from a wide range of government, academic, and industrial organizations.

The Repast user community is large and growing. For example, the most recent release, Repast version 3, has had several thousand downloads. These users have applied Repast to a wide variety of applications that range from social systems, to evolutionary systems, to market modeling, to industrial analysis. A sampling of Repast applications will be presented later in this section.

3.1 The Context of Repast

Repast is one of several available agent modeling toolkits such as Swarm, Ascape, NetLogo, and the Multi-Agent Simulator Of Neighborhoods (MASON) [Minar et al. 1996; Burkhart et al. 2000; Inchiosa and Parker 2002; George Mason University 2004]. Repast is differentiated from these toolkits in several respects. Unlike Swarm, Repast is available in both pure Java and pure Microsoft .NET forms. Swarm is distributed under the GNU General Public

[2]In practice, NetLogo's design team has been forced to find a good compromise between their competing "low threshold" and "no ceiling" goals.

[3]This article describes version 3 of the Repast toolkit.

License (GPL), which requires developers to make the source code for their entire model available to anyone who obtains a legitimate copy of the model's binary code. NetLogo is not open source. Both the executable code and source code for Repast are distributed free of charge under a variation of the Berkeley Software Distribution (BSD) license, which does not require user model source code to be released. Unlike the freely available versions of Ascape, Repast is being actively developed. MASON is a relatively new entrant into the field of agent-based simulation toolkits. MASON's "design owes a lot to other multiagent simulators in the Social Complexity and Robotics fields, particularly to Repast and TeamBots" [George Mason University 2004]. For reviews of Swarm, Repast, and other agent-modeling toolkits, see the 2002 survey by Serenko and Detlor, the 2002 survey by Gilbert and Bankes, and the 2003 toolkit review by Tobias and Hofmann [Gilbert and Bankes 2002; Serenko and Detlor 2002; Tobias and Hofmann 2004].

3.2 Example Repast Applications[4]

John Padgett, Doowan Lee, and Nicholson Collier's Hypercycle model uses Repast in combination with analytic methods to investigate autocatalytic coevolution of complex interconnected production and consumption systems [Padgett et al. 2003]. They seek to answer the question, "can self-sustaining cycles of economic production and consumption spontaneously emerge" [Padgett et al. 2003]? Their Hypercycle model has products that are exchanged throughout the Hypercycle world, rules that transform products into other products, and agents that use the rules to convert products into other products [Padgett et al. 2003]. The long-term goal is to model selected critical features found in real production and consumption systems such as Renaissance Florence [Padgett and Ansell 1993].

George Kampis of the Eotvos University, Budapest and Laszlo Gulyas of the Hungarian Academy of Sciences are applying Repast to investigate evolutionary emergence [Kampis 2002; Kampis and Gulyas 2003, 2004]. They are using Repast to answer the question, "how is it possible to produce sustained evolution in an artificial system" [Kampis and Gulyas 2004]? They state the following:

> We developed an agent-based simulation model using the Repast package. Organisms are agents that selectively feed, reproduce and die, based on their phenotypic properties described in variable length records. As adaptation progresses, new property sets extend the records, and as a result, selection can spontaneously switch between the defining properties of an interaction. The aim is to develop functionally disjoint subpopulations specialized for the use of different property sets. The first results have recently been reported, showing the possibility of progressive evolution productive of new selection effects, as an illustration for the causal principles of embodiment.

Randal Picker of the University of Chicago Law School has used Repast to investigate the endogenous emergence of social norms and the resulting

[4]Many of these examples, as well as Repast itself, are discussed in more detail in North and Mascal [2005].

reification of selected norms as law [Baird et al. 1998]. He has sought to answer the question: How do conventions emerge and eventually become laws [Baird et al. 1998]? The agents in his Endogenous Neighborhoods and Norms model are individual people. The agents' environment is a forum for interaction. The agents adopt or change norms based on the relative success they experience using those social norms. Success is itself dependent on the level of adoption of the underlying norms.

Lars-Erik Cederman of the Swiss Federal Institute of Technology Zurich is using Repast to study state formation and nationalist movements [Cederman 2001, 2002]. He seeks to answer multiple questions, including how do national borders emerge and why do they take the shapes that they do [Cederman 2001, 2002]? In some of his Repast models, each agent represents a nation or fiefdom existing on a grid containing variable resources. The nations or fiefdoms can interact peacefully by forming alliances or by merging, along with interacting by attempting to invade and conquer neighboring states.

P. Jeffery Brantingham of the University of California, Los Angeles is applying Repast to investigate stone tool assembly by ancient peoples [Brantingham 2003]. He seeks to answer questions such as: How did ancient people build their tools and why did they use the approaches that were chosen? His Repast model "dispenses with assumptions that raw material type and abundance play any role in the organization of mobility and raw material procurement strategies" [Brantingham 2003]. Brantingham reports that his Repast model shows that "richness-sample size relationships, frequencies of raw material transfers as a function of distance from source, and both quantity-distance and reduction intensity-distance relationships are qualitatively similar to commonly observed archaeological patterns." This success has lead Brantingham to interesting findings, including the "possibility that Paleolithic behavioral adaptations were sometimes not responsive to differences between stone raw material types in the ways implied by current archaeological theory."

Craig Stephan and John Sullivan of Ford Motor Company are studying the "growth of a hydrogen transportation infrastructure" [Stephan and Sullivan, 2004]. In the long run, they are seeking to answer several questions, including discovering what factors might lead to the successful emergence of a self-sustaining hydrogen transportation infrastructure [Stephan and Sullivan 2004]. In their model, the agents are drivers who chose what types of vehicles to buy (e.g., hydrogen-fueled cars versus gasoline-fueled cars) as well as fueling stations that chose to offer different fuels.

Michael North, Charles Macal, and others at Argonne National Laboratory are using Repast to model electric power markets. Their Electricity Market Complex Adaptive Systems (EMCAS) model seeks to answer questions about electric power market stability and efficiency [North et al. 2003]. EMCAS represents the behavior of an electric power system as well as the producers and consumers that operate within it. The agents include generation companies that offer power into the electricity marketplace and demand companies that buy bulk electricity from the marketplace.

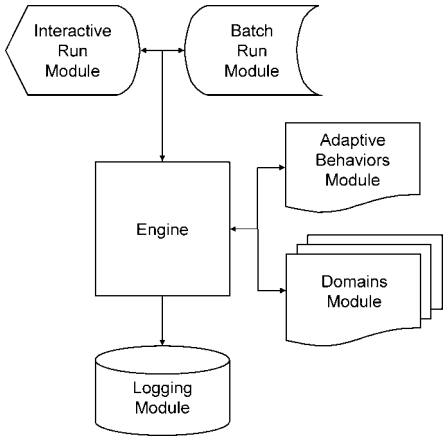

Fig. 1. The Repast components.

4. THE REPAST FEATURE SET

The Repast system supports a rich feature set. This feature set is divided into a set of related modules. These modules are classified as either fixed or flexible.[5]

All of the components in a fixed module are required for a full Repast implementation. Components in flexible modules are optional. Implementers can decide which components are included and which are not included in flexible modules. Implementers are free to add appropriate components to both fixed and flexible modules.

There are varying ways to implement both fixed and flexible modules. For example, there may be one implementation component for each module component or there may be one implementation component that takes on the role of two or more module components.

There are six modules in the Repast feature set. These modules are the *Engine*, the *Logging Module*, the *Interactive Run Module*, the *Batch Run Module*, the *Adaptive Behaviors Module*, and the *Domains Module*. A component diagram for Repast is shown in Figure 1.

4.1 The Engine Module

The *Engine Module* is a fixed module that is responsible for controlling the activities in a simulation. It contains *Engine Controller*, *Scheduler*, *Action*, and *Agent* components.

Controllers work with *Interactive Run* and *Batch Run* components to initiate, start, pause, step, stop, and restart simulation runs. *Schedulers* are responsible for managing the flow of time in a simulation using *Actions*. *Schedulers* are commonly implemented as discrete event clock managers [Law and Kelton 2000]. *Actions* are individual events that occur in a simulation. *Agents* cause actions to occur by registering them with a *Scheduler*. *Agents* are autonomous

[5]Flexibility is confined to the module level to keep the feature set as small as possible. Flexible components in otherwise fixed modules need not be included in the feature set, since these components are optional and implementers are free to add optional components to any module.

components that manage their own activities and coordinate with other *Agents* through one or more *Schedulers*. *Agents* are normally implemented as objects with specialized properties, including self-direction.

4.2 The Logging Module

The *Logging Module* is a fixed module that is responsible for recording simulation results. Two types of logging components, namely *Data Loggers* and *Object Loggers*, are required. These types differ based on the complexity of the inputs.

Data Loggers are the simplest logging components. *Data Loggers* simply record primitive values such as integers, floating point numbers, or strings to a specified location.

Object Loggers are more sophisticated than *Data Loggers*. *Object Loggers* record the state of full objects or sets of objects, rather than just primitives, to a specified location. Agents can be logged using object loggers since agents are normally implemented as objects with specific properties.

4.3 The Interactive Run Module

The *Interactive Run Module* is a fixed module that is responsible for managing simulation runs under the direct control of a user. The components in the *Interactive Run Module* usually act as intermediaries between users and *Engine Module* components.

User Interface Controllers act as intermediaries between users and *Engine Controllers*. *User Interface Controllers* offer users options to initiate, start, pause, step, stop, and restart simulations and then communicate these requests to the *Engine Controllers*. *User Interface Controllers* also show the current master *Scheduler* clock time and overall simulation status.

Environment Visualizations show and allow editing of the status of the overall model and agents in the system. These visualizations normally include various types of interactive multidimensional displays and a range of agent layout techniques.

Similar to *Environment Visualizations*, *Probes* show and allow editing of the status of the overall model and individual agents in the system. *Probes* are differentiated from *Environment Visualizations,* since *Probes* are textual while the visualizations are visual or abstractly spatial. Agents can have rich, recursively nested properties. Therefore, probes are expected to allow deep recursive exploration of agent properties.

Graphs show detailed visual results traces at both the level of individual agents and the overall model. Graphs often display results recorded in logs.

Much like *Graphs, Reports* show detailed results traces and often display results recorded in logs. However, *Reports* are textual.

4.4 The Batch Run Module

The *Batch Run Module* is a fixed module that is responsible for completing a set of simulation runs without requiring the direct intervention of a user. As with the *Interactive Run Module*, components in the *Batch Run Module* usually act as intermediaries between users and *Engine Module* components.

Each *Batch Controller* manages an individual *Engine Controller* during the course of one simulation run. A *Batch Controller* uses its *Engine Controller* to complete one model run using one model parameter combination. A model parameter combination is given to the *Batch Controller* by the *Parameter Sweep Framework*.

The *Parameter Sweep Framework* iterates over a range of model input parameters to complete a set of simulation runs. An example is a simple social network model that is run 20 times with between 10 and 20 people in each run. In this case, the *Parameter Sweep Framework* will execute the model 20 times for each of the 11 population counts,[6] resulting in a total of 220 (20×11) simulation runs.

4.5 Adaptive Behaviors Module

The *Adaptive Behaviors Module* is a flexible module that is responsible for providing adaptive components for implementing agent behaviors. The components can include genetic algorithms, neural networks, other artificial intelligence tools, and regression tools for building agents that can learn and adapt [Goldberg 1989; Rich and Knight 1991; Ginsberg 1993; McClave and Benson 1994; Mitchell 1996].

4.6 Domains Module

The *Domains Module* is a flexible module that is responsible for providing area-specific functions. In some cases, these functions may be sophisticated enough to become modules in their own right. Example components in the domains module include tools for general networks, social systems, geographical information systems (GIS), systems dynamics, and computational game theory.

5. THE REPAST IMPLEMENTATIONS

There are currently three implementations of Repast. These implementations are Repast for Java (Repast J), Repast .NET, and Repast for Python Scripting (Repast Py). Repast J was the original implementation; Repast Py was developed using Repast J Repast .NET is the most recent implementation.[7] Independent descriptions of the three implementations will be provided in this section and the similarities and differences will be pointed out in a later section.

5.1 Repast for Java

Repast J is a Java language implementation of the Repast specification [Foxwell 1999].[8] This implementation thus provides a software framework for creating

[6]The population counts go from 10 to 20 for a total of $20 - 10 + 1 = 11$ combinations.

[7]As will be discussed later, Repast Py presents the full Repast feature set using a visual interface and a simple programming language. However, since Repast Py was built using Repast J, Repast Py can be considered in some sense "half of an implementation" of the Repast feature set. The authors decided to describe their work as "three implementations" rather than "two and a half implementations" of the Repast feature set for brevity.

[8]Java version two was used in this article. The replacement for Java version two, namely Java version five, was released on September 29, 2004 [Sun Microsystems 2005]. The Java version following version two is version five, despite the skip in numbering.

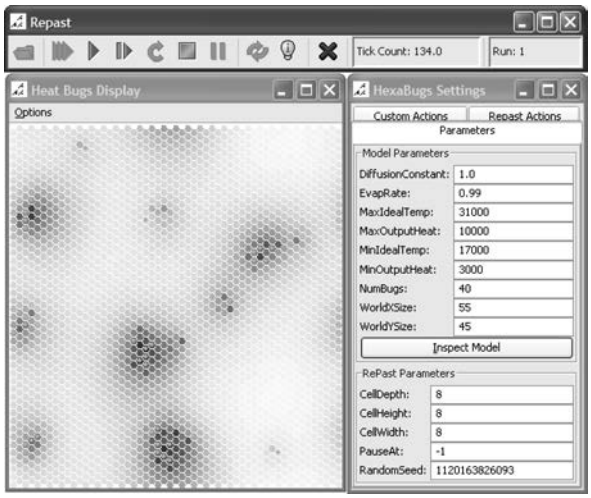

Fig. 2. The repast J HexaBugs Model, a typical Repast simulation, has a GUI controller toolbar at the top, simulation parameters on the right and a visualization of the simulation on the left.

agent-based simulations where the common infrastructural abstractions expressed in the framework are implemented as Java classes. The user then creates a simulation by combining and extending these classes. An example Repast J model is shown in Figure 2.

Java focuses on cross-platform compatibility [Foxwell 1999].[9] Java thus provides one language that can run on many different computer systems.[10] Java programs inherit this value proposition. Developers who choose Java use one language, but can run the resulting programs on nearly any type of computer.

The Repast J classes are organized into Java packages that roughly express the Repast specification. It is important to note that most of the classes described below are extensible so that features such as custom user interfaces, for example, can be easily created. A description of the packages relevant to the specification follows.

5.1.1 *The Engine Package.* The classes in the *Engine Package* are responsible for setting up, manipulating, and driving simulations. The *Engine Module*, as well as the *Controller* portions of the *Interactive* and *Batch Run Modules*, are implemented by this package. More specifically, this package contains the controller hierarchy by which an agent simulation is started, paused, stepped, stopped, and restarted. There are different controllers for handling user interaction with a simulation through an interface and for automating such interaction with a batch run mechanism. In addition, the *Engine Package* contains the

[9]The term "cross platform" used in this sentence refers to executing binary programs without modifications on many different computer systems such as Microsoft Windows, Apple Mac OS, Linux, and Sun Solaris.

[10]This is not strictly true. There are specialized languages such as Jython that can be compiled to Java binary code. However, this is by far the exception rather than the rule. The overwhelming majority of Java binary code was generated from Java language source code.

classes that make up a discrete event scheduling mechanism, including classes for the *Schedule* itself and the primitive *Actions* that are scheduled.

Agents are flexibly defined in Repast J. Agents can be built using one of a variety of Repast J base classes or they can be constructed using generic objects.

5.1.2 *The Analysis Package.* The classes in the *Analysis Package* are responsible for gathering, recording, and charting data. With its gathering and recording responsibilities, this package implements the *Logging Module*, while the charting functionality implements the graphing portion of the *Interactive Run Module*.

5.1.3 *The GUI Package.* The *Graphical User Interface* (*GUI*) *Package* classes are responsible for the graphical animated visualization of the simulation as well as providing the capability to take snapshots of the display and make QuickTime movies of the visualization as it evolves over time. The *GUI Package* uses the Model-View-Controller (MVC) design pattern to modularize functionality [Gamma et al. 1994].[11] The various display classes work in conjunction with the classes in the *Space Package* to display these *Spaces* appropriately. *Spaces* work in conjunction with the display classes in the *GUI Package* to present visualizations of the *Spaces* and the agents that they contain. In addition, this package handles agent probing. This package implements all of the *Interactive Run Module* except for the GUI Controller, which is implemented in the *Engine Package*.

5.1.4 *The Parameter Package.* The *Parameter Package* is responsible for defining parameter spaces and iterating through them (i.e., running the simulation with different sets of parameters each time in an automated way). This functionality is highly extensible so that custom parameter formats and parameter sweeps can be easily created. Additionally, this package implements the *Parameter Sweep Framework* component of the *Batch Run Module*.

5.1.5 *The Adaptation Package.* The *Adaptation Package* implements the *Adaptive Behaviors Module*. This package includes tools such as genetic algorithms and neural networks.

5.1.6 *Domains Module-Related Packages.* Repast J provides several packages beyond the more infrastructural ones described above. These packages provide the basis for constructing particular types of simulations and agents, such as simulations with agents that interact over a grid or torus topology.

The *Network Package* contains the core classes used to build network simulations. These include default node and edge classes, various specialized records for recording network data, and so forth. In addition, the *NetworkFactory* class is used to load networks from a file in a variety of formats as well as to generate small world, random density, and square lattice networks.

In an agent simulation, agents often have some sort of spatial relationship to each other. The *Space Package* contains base classes for creating such

[11]The MVC design pattern uses a *Model* that takes directions from a *Controller* to produce results shown by a *View*.

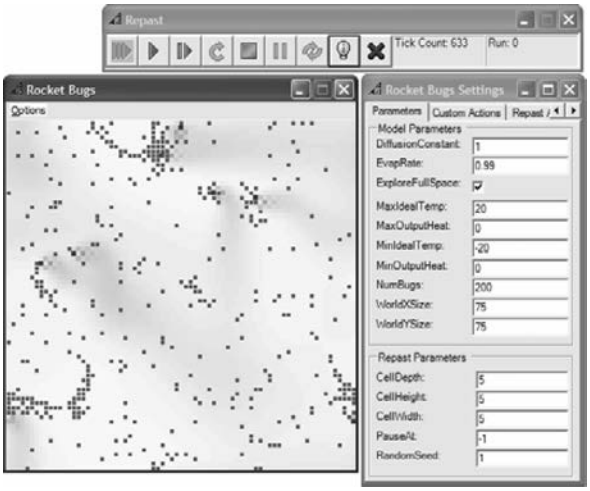

Fig. 3. The Repast .NET RocketBugs model, a typical repast .NET simulation, has a GUI controller, parameters, and a visualization.

relationships as well as several particular instantiations including grids, tori, and multiple occupancy grids, among others.

Creating agents using real geographic data is important for many applications [Brown et al. 2005]. Storing the results back into geographic data files is just as critical [Brown et al. 2005]. The *GIS Package* provides tools to create Repast J agents from, and to store simulation results back to, standard GIS data sources.

5.2 Repast .NET

Repast .NET is a Microsoft .NET implementation of the Repast specification [Foxwell 1999]. Repast .NET is written in the C# language [Archer 2001]. As with Repast J, the Repast .NET implementation provides a software framework for creating agent-based simulations. However, Repast .NET expresses the common infrastructural abstractions expressed as .NET Framework classes. The user then creates a simulation by combining and extending these classes. An example Repast .NET model is shown in Figure 3.

These classes are organized into .NET namespaces that express the Repast specification.[12] As before, most of the classes described in the following are extensible so that features such as custom agent behaviors, for example, can be easily created. The details of the Repast .NET framework follow.

Microsoft .NET focuses on providing multilingual functionality [Archer 2001], accordingly, Microsoft .NET provides many languages that all work together seamlessly on one computer platform.[13] Microsoft .NET programs

[12]Namespaces play the same role in .NET framework applications as packages play in Java applications. Both terms are used in this article rather than just one, to respect the differences in nomenclature between the .NET and Java development communities.

[13]As with Java's single language development, this sweeping statement is not strictly true. There is an effort underway that provides tools to execute Microsoft .NET programs under Linux. However,

inherit this value proposition. Developers that choose Microsoft .NET can use nearly any language but can only run the resulting programs on one type of computer. Thus in some sense, Java and Microsoft .NET present converse tradeoffs.

One of the main features of the Microsoft .NET framework is its ability to seamlessly support the development of a single program in several languages without complicated binary linking schemes. For example, C#, Managed C++, Visual Basic .NET[14], and even Managed FORTRAN can all be used together to write a single program. In fact, the Repast .NET model shown in Figure 3 uses a *Model* written in Managed C++, a *Space* written in Visual Basic .NET, and *Agents* written in C#.

5.2.1 *The Engine Namespace.* *Engine Namespace* classes are responsible for setting up, manipulating, and driving simulations. The *Engine Namespace* implements the *Engine Module* and the *Controller* portions of the *Interactive* and *Batch Run Modules*. Similarly to Repast J, the *Engine Namespace* contains the discrete event scheduling mechanism including classes for the *Schedule* and the *Actions* that are scheduled. As with Repast J, *Agents* can be created based on one of several parent classes or they can be based on generic objects.

5.2.2 *The Analysis Namespace.* *Analysis Namespace* classes are responsible for gathering, recording, and charting data. As with Repast J, this namespace implements the *Logging Module*, and the graphing portion of the *Interactive Run Module*.

5.2.3 *The GUI Namespace.* The classes in the *GUI Namespace* are responsible for the graphical animated visualization of simulations as well as for providing the capability to take snapshots of the displays. These classes provide visualizations for the classes in the *Space Namespace* as an addition to handling agent probing. The *GUI Namespace* implements the *Interactive Run Module,* excluding the *Engine Package*'s *GUI Controller*.

5.2.4 *The Parameter Namespace.* The classes in the *Parameter Namespace* are responsible for defining parameter spaces and automatically running the simulation multiple times with different sets of parameters each time. Custom parameters and parameter scans can be easily created. The *Parameter Namespace* also implements the *Batch Run Module*'s *Parameter Sweep Framework*.

5.2.5 *The Adpatiation Namespace.* As with Repast J, the *Adaptiation Namespace* implements the *Adaptive Behaviors Module*. As before, this namespace includes genetic algorithms and regression tools to be used to create adaptive agent behaviors.

5.2.6 *Domains Module Related Namespaces.* The Repast .NET *Domains Module* namespaces provide tools for constructing specalized types of simulations and agents. For example, the *Space Namespace* contains base classes for

---

the overwhelming majority of Microsoft .NET code will be executed solely under Microsoft Windows for the near future.

[14]Visual Basic .NET is the newest incarnation of Microsoft's Visual Basic language, built as a .NET language.

representing many relationships between agents, such as those found in grids, tori, and multiple occupancy grids, among others.

5.3 Repast for Python Scripting

Repast for Python Scripting (Repast Py) is a rapid application development (RAD) tool for producing Repast simulations in which agent behavior is scripted using the Python computer language [Lutz and Ascher 1999]. As a RAD tool, Repast Py differs significantly from Repast J and Repast .NET. In Repast Py, user services are presented in a visual manner through a separate application whereas Repast J and Repast .NET are frameworks that are accessed through standard programming languages such as Java or C#. For example, in Repast J and Repast .NET the user typically constructs their agents as Java or C# classes, but in Repast Py the user employs a point-and-click interface to set the properties of agent components. In general, much that would have to be manually coded (e.g. data logging) in Repast J or Repast .NET has been replaced by a point-and-click component based interface in Repast Py. Although Repast Py is a separate application, it is implemented in Java and makes extensive use of the Repast J implementation.[15] In fact, Repast Py produces simulations as compiled Java byte-code that is executed by the Repast J implementation. An example Repast Py model is shown in Figure 4. Repast Py can also generate Java source code that can be used with Repast J. However, unlike Repast J and Repast .NET, Repast Py is not fully object-oriented, since it only supports one level of inheritance.

Repast Py adapts the general Repast notion of how to organize the internal implementation of an agent simulation, but adds a component-based visual "point-and-click" interface as shown in Figure 5. Specifically, the *Model* part of the simulation is responsible for initializing the agents and any other required elements and defining what should occur at each time step of the simulation. The agents themselves then perform the actual behavior that drives the simulation. Repast Py adapts this notion by providing components for models and agents. Each component can be thought of as providing a generic description of a piece of the final simulation (i.e., the *Model*, a particular agent type, a chart, and so on). This composite description is then compiled into the actual simulation code itself.

These generic components are specialized by setting the values of the component's properties. For example, the component that describes and produces a description of an agent type based on a GIS input file has a data source property. The value of this property is the GIS data file that provides both a partial structure of the agent type and the data encapsulated by each instantiated agent of that type. Thus, different GIS input files will result in different agent types. It is through a component's properties that standard Repast services are exposed. For example, the schedule property allows the user to schedule the execution of agent and *Model* behavior. The result is the same as if the appropriate scheduling calls had been made, but Repast Py allows the user to do this

[15]Please see the previous discussion about Repast Py as "half an implementation" of the Repast feature set.

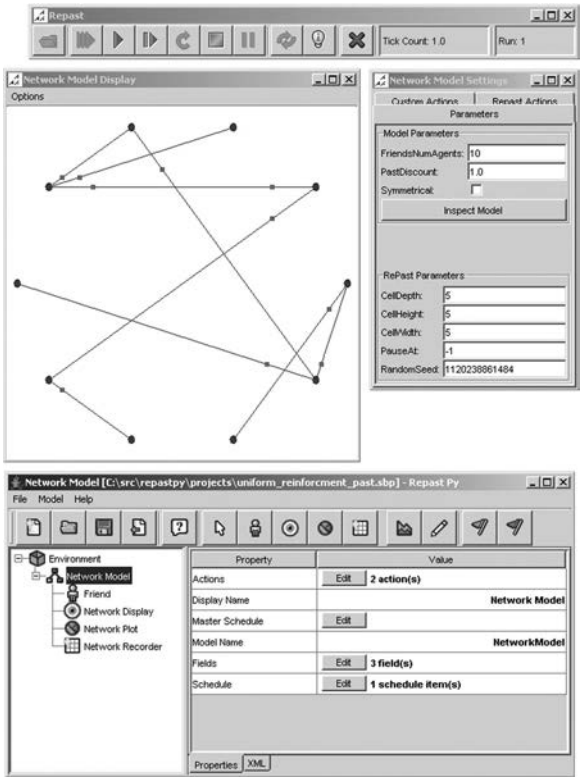

Fig. 4. A Repast Py network Model (The bottom window shows the Repast Py Application itself and the top three windows show the resulting Repast simulation).

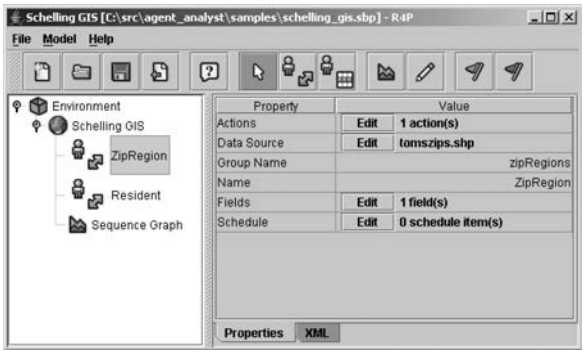

Fig. 5. The Repast Py Visual Interface (In this case, the user is creating a custom ZIP code region or "ZipRegion" Agent by setting property values through the visual interface).

in a completely different environment using a visual orientation as shown in Figure 6.

Many of the capabilities in the Repast feature set are visually encapsulated entirely in a point-and-click interface. For example, instead of writing the lines of Repast J code shown in Figure 7, the visual interfaces shown in Figures 6

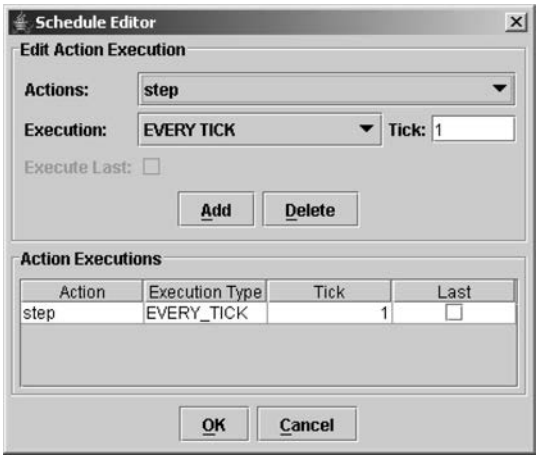

Fig. 6. Repast Py schedule editor.

```
// Create the required discrete
// event schedule.
private void buildSchedule() {

   // Schedule the agent actions.
   this.schedule.
      scheduleActionAtInterval(
      1, this.VectorAgent, "step");

   // Schedule the GIS
   // visualization update.
   this.schedule.
      scheduleActionAtInterval(
      1, this.updateDisplay(),
      Schedule.LAST);
}
```

Fig. 7. Example Repast J scheduling code.

and 8 are used. The schedule editors shown in Figures 6 and 8 are invoked from Repast Py agents or models. They allow the user to visually schedule the agent or model's behavior. This is in contrast to Repast J's need for scheduling code of the kind in shown Figure 7.

Repast Py allows the user to define domain-specific behavior through the *Actions Property*. The *Actions Property* is essentially a list of behaviors or actions that are edited in the *Actions Editor*. *Actions Properties* are associated with each simulation component such as an agent.

Actions are written in a special subset of the Python language [Lutz and Ascher 1999]. A subset is used because the entire Python language is not necessary for specifying agent behavior. For example, Python features such as class and function declarations or the dynamic manipulation of a class' structure are

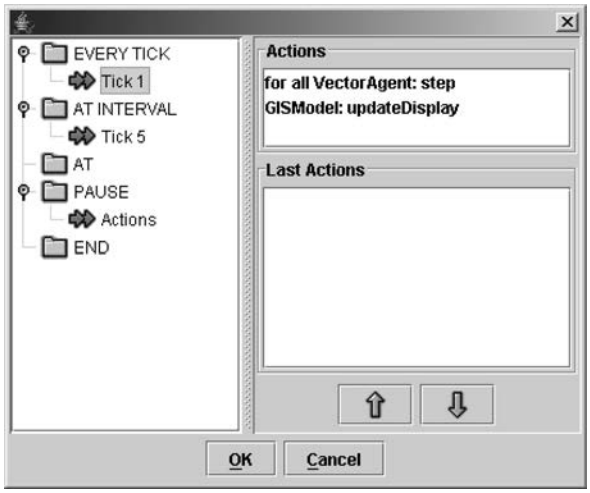

Fig. 8. The Repast Py schedule view.

not necessary when defining an agent's behavior. This subset is "special" because it integrates well with Java, and thus with Repast J as a whole. As a result, it is easy to use pieces from the Repast J framework when scripting agent behaviors and to automatically compile these Python scripts into executable code when running the resulting simulation.

6. CONCLUSIONS

Both technical and conceptual lessons were learned during the development of the three Repast implementations. These lessons point the way to the future work discussed in the next section.

6.1 Technical Lessons Learned

Based on the authors' comparative experiences implementing Repast J, Repast .NET, and Repast Py, several technical issues were found. These issues are summarized in Table I. The relative advantages and disadvantages of the three Repast implementations are summarized in Table II. These issues and comparisons have also contributed to some of the conceptual conclusions offered in the next section.

The use of design patterns such as the MVC pattern simplified the development process. In particular, MVC allowed the Repast .NET implementation to be built in testable stages, namely *Model*, then *Controllers*, and finally *View*s. It also largely eliminated the need to maintain redundant internal states within the toolkit modules.

The fact that both Java and Microsoft .NET are object-oriented provided a natural way to separate functions throughout the implementations and provides a natural way to build up agent behaviors in stages. This enriched the development and use of both implementations.

Microsoft .NET allows developers to be more expressive than Java in controlling the details of object inheritance.[16] This allowed the Repast .NET implementation to have significantly stronger type checking and fewer errors.

Microsoft .NET Properties are useful for simplifying object interfaces, and logically combining data access and writing

Microsoft .NET's ability to embed metadata tags into code has the potential to make development easier and to increase the interoperability of code modules.[17,18]

Microsoft .NET's reduction in virtual machine language restrictions compared to Java made cross-language development easier. This is particularly true when simplified languages such as Visual Basic .NET are included.

The Microsoft Visual Studio development environment is quite weak compared to free and open source tools such as Eclipse [Archer 2001; Eclipse 2004]. In particular, Visual Studio 2003's lack of refactoring, tools made iterative development difficult compared to Eclipse [Fowler et al. 1999].[19,20] This will be addressed to some extent in the next version of Visual Studio, but preview versions indicate that the improvements will still lag significantly behind the features available for Java.

Java has a much larger large set of third-party libraries than Microsoft .NET. This is particularly true in the free and open source arena. This made Java development much faster since existing code could be reused in many cases.

Differences were found in the underlying implementations of the Microsoft .NET and Java mathematics libraries. The differences caused otherwise identical calculations to produce divergent results, even for relatively simple equations. As might be expected, cumulative use of such calculations was found to produce widely divergent results. This complicates simulation cross verification between Repast J and Repast .NET.

Compared to Repast J and Repast .NET, Repast Py opted for a user-workflow-oriented approach over a component-oriented approach. This meant that the underlying simulation modules are organized to make model construction easier for users rather than to provide a component pallet for complex model design.

Overall, Microsoft .NET is a strong platform, has a good library of basic classes, and appears to be fast except for graphics operations. Java is also a strong platform with a solid built-in class library. On a comparative basis, Java 2 tools and libraries are much more mature, but the Microsoft .NET core

[16]Specifically, Microsoft .NET allows explicit control of virtual versus standard functions and control of method overriding by offering both *override* and *new* operators.

[17]Code metadata tags mark classes, methods, and fields with descriptive information that can be used by programs that work with code such as compliers and simulation runtime environments.

[18]Java 5 includes support for the meta-data tags in the form of Java annotations.

[19]Refactoring is an industrial technique for code reorganization. Refactoring generally requires tool support to be efficient, since it often involves simultaneous, systematic, and repeated changes to many code modules.

[20]In some cases, certain third party tools can add refactoring support to Visual Studio.

Table I. Technical Lessons Learned

	Repast J (Java)	Repast .NET (.NET Compatible Language)
Object Oriented	+	+
Fine Grained Inheritance Control	−	+
First Class Properties	−	+
Metadata Tags	−[21]	+
Cross-language Development	−	+
Cross-Platform Development	+	−
Advanced Development Features (e.g., Integrated Refactoring)	+	−[22]
Large Variety of Free and Open Source Third Party Libraries	+	−

Table II. Advantages and Disadvantages of the Three Repast Implementations

	Repast J	Repast .NET	Repast Py
Cross-Platform Development	Windows, Linux, Apple OS X, and Many Others	Windows and Linux	Windows, Linux, Apple OS X, and Many Others
Cross Language Development	Java Only (For Practical Purposes)	C#, Managed C++, Visual Basic .NET, and Others	Python-like Language, but with Strong Java Integration
Extensibility	Very Extensible Through a Myriad of Third Party Java Libraries	Less so than Repast J, but Still Extensible Through .NET Language Libraries	Visual Development and Ease of Use Constraints Limit Extensibility, but Java Integration Allows for the Use of Third Party Java Libraries
Nature of Implementation	Java Language Library for Constructing Repast Simulations	.NET Framework Library for Constructing Repast Simulations	Stand–alone Application for Constructing Repast Simulations
Visual Programming	No	No	Simulation is Constructed Via a Point-and-Click Interface with Some Programming

language specification is richer.[23] Java's multiplatform approach will remain attractive to those who need to execute simulations on many different types of computers, particularly those who need to run models on both workstations and large clusters. Microsoft .NET's multilingual approach will appeal to those who

[21]As previously mentioned, meta-data tags are available in Java version five.
[22]As previously stated, Microsoft Visual Studio 2003 was evaluated. The next version of Microsoft Visual Studio is expected to include some advanced development features.
[23]Java 5 has essentially eliminated this gap.

wish to leverage existing language skills and the unique features of specialized languages.[24]

6.2 Conceptual Lessons Learned

In addition to technical lessons learned, the authors' experiences in creating three implementations of the Repast feature set have suggested several conceptual lessons. The conceptual lessons learned are as follows:

- As with FIPA-OS and the HLA-RTIs, object-oriented design is essential to allow the architectures to be flexible [Bachinsky et al. 1998; Poslad et al. 2000].
- As with FIPA-OS and the HLA RTIs, the use of design patterns is helpful for improving implementation quality [Bachinsky et al. 1998; Poslad et al. 2000].
- Choosing environments that allow seamless programming in different languages may help provide both the "low threshold" NetLogo seeks for beginners and the unlimited ceiling NetLogo seeks for advanced users [Tisue and Wilensky 2004].[25]
- As with the Gaudi KAoS implementation, choosing environments that are cross-platform makes systems usable for a wider audience than with single platform approaches [Bradshaw 1996].
- As with the Gaudi KAoS implementation, modular construction is essential [Bradshaw 1996].
- Toolkit developers should be keenly aware of differences in the fundamental details of the candidate implementation environments, such as differences in mathematics libraries.
- Toolkit developers should consider the availability and capability of the ecosystem surrounding an implementation environment when making their selections. For example, the availability of third-party libraries and the quality of development tools should be taken into account.
- Toolkit developers should consider the availability of advanced language features such as metadata tags and first class properties when selecting implementation languages.
- Toolkit developers should consider future growth plans for the candidate environments, such as F#, when selecting an implementation environment.

7. FUTURE WORK

There are several directions for future work. First, an optional visual programming tool should be added to supplement the textual languages used in Repast.

[24]For example, Microsoft's F# language promises to simplify most common programming tasks by eliminating tedious and repetitious coding details, at the possible expense of some expressiveness. Any lost expressiveness can be overcome by coding in a standard Microsoft .NET language such as C#. F# may be extremely useful for less technical modelers and for prototyping. Repast .NET's implementation makes F# immediately available to agent-based modelers.

[25]For example, Visual Basic .NET can be used by beginning programmers and C# can be used by more advanced programmers.

This will allow developers to explore the implementation of a fully visual environment. Second, higher-level languages such as Microsoft .NET framework F# should be explored to see if they can help support the type of "rapid-discovery social science" being sought by Sallach [2003]. Third, a Repast implementation in a nonprocedural language should be completed as suggested by Sallach [2004]. This will allow a full reexamination of the underlying feature set and the implementation details of the Repast agent-based simulation toolkit.

ACKNOWLEDGMENTS

The authors wish to thank David L. Sallach for his visionary leadership in founding the Repast project as well as pioneering the Repast nonprocedural implementation effort, Charles M. Macal for sustaining involvement in the project, and Repast contributors such as Thomas R. Howe.

REFERENCES

ARCHER, T. 2001. *Inside C#*. Microsoft Press, Redmond, WA.

ARONSON, J., MANIKONDA, V., PENG, W., LEVY, R., AND ROTH, K. 2003. An HLA compliant agent-based fast-time simulation architecture for analysis of civil aviation concepts. In *Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop*, IEEE Kisimmee, FL USA.

BACHINSKY, S. T., MELLON, L., TARBOX, G. H., AND FUJIMOTO, R. 1998. RTI 2.0 architecture. In *Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop*, IEEE, Orlando, FL.

BAIRD, D. G., GERTNER, R. H., AND PICKER, R. C. 1998. *Game Theory and the Law*. Harvard University Press, Cambridge, MA, USA.

BAUER, B., MÜLLER, J. P., AND ODELL, J . 2001. Agent UML: A formalism for specifying multiagent interaction. In *Agent-Oriented Software Engineering*, Ciancarini, P. and Wooldridge, M. Eds. Springer-Verlag, Berlin, Germany, 91–103.

BRADSHAW, J. 1996. KAoS: An open agent architecture supporting reuse, interoperability, and extensibility. In *Proceedings of the 1996 Knowledge Acquisition Workshop*, Banff, Alberta, Canada, University of Calgary.

BRADSHAW, J. 1997. An introduction to software agents. In *Software Agents*. J. Bradshaw, Ed. AAAI Press, Menlo Park, CA.

BRANTINGHAM, P. 2003. A neutral model of stone raw material procurement. *American Antiquity*, 487–509.

BROWN, D. G., RIOLO, R., ROBINSON, D. T., NORTH, M. J., AND RAND, W. 2005. Spatial process and data models: Toward integration of agent-based models and GIS. *Journal of Geological Society*.

BURKHART, R., ASKENAZI, M., AND MINAR, N. 2000. Swarm release documentation. www.santafe.edu/projects/swarm/swarmdocs/set/set.html.

CAVITT, D. B., OVERSTREET, C. M., AND MALY, K. J. 1997. A performance monitoring application for distributed interactive simulations (DIS). In *Proceedings of the 1997 Winter Simulation Conference*, ACM, Atlanta, GA, USA.

CEDERMAN, L.-E. 2001. Modeling the co-evolution of states and nations. In *Workshop on Simulation of Social Agents: Architectures and Institutions*, Chicago, IL, Oct. 4–6, 1997, Argonne National Laboratory.

CEDERMAN, L.-E. 2002. Endogenizing geopolitical boundaries with agent-based modeling. In *Proceedings National Academy of Sciences 99*(90003), 7296–7303.

COLLIER, N., HOWE, T., AND NORTH, M. J. 2003. Onward and upward: The transition to Repast 2.0. In *First Annual North American Association for Computational Social and Organizational Science Conference*. (Pittsburgh, PA USA), North American Association for Computational Social and Organizational Science.

COPLIEN, J. O. 2001. *Software Patterns Page*. www.hillside.net/patterns/.

DMSO. 2005. *High Level Architecture Home Page.* U.S. Defense Modeling and Simulation Office, www.dmso.mil/public/transition/hla.

DMSO. 2004. *HLA RTI Verification Status Board*. Defense Modeling and Simulation Office, www.dmso.mil/public/transition/hla/statusboard.

ECLIPSE. 2004. *Eclipse Home Page*. The Eclipse Project, www.eclipse.org/.

FIPA. 2003. *Foundation for Intelligent Physical Agents Publicly Available Agent Platform Implementations*. Alameda, CA, FIPA. http://www.fipa.org/resources/livesystems.html

FLORES-MENDEZ, R. A. 1999. Towards a standardization of multi-agent system frameworks. *ACM Crossroads 5*.

FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., AND ROBERTS, D. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Redwood City, CA.

FOXWELL, H. 1999. Java 2 software development kit. *Linux Journal*.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES. J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Wokingham, UK.

GEORGE MASON UNIVERSITY. 2004. MASON Home Page. George Mason University, Fairfax, VA, http://cs.gmu.edu/∼eclab/projects/mason/.

GILBERT, N. AND BANKES, S. 2002. Platforms and Methods for Agent-based Modeling. In *Proceedings of the National Academy of Sciences of the USA*, *99* (3), 7197–7198.

GINSBERG, M. L. 1993. *Essentials of Artificial Intelligence*, Morgan Kaufmann Publishers.

GOLDBERG, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Redwood City, CA.

HARVEY, B. 1997. *Computer Science Logo Style*. MIT Press, Boston, MA,

IEEE. 1995a. *IEEE Standard for Distributed Interactive Simulation—Application Protocols*, Institute of Electrical and Electronics Engineers, 1278.1-1995.

IEEE. 1995b. *IEEE Standard for Distributed Interactive Simulation—Communication Services*, Institute of Electrical and Electronics Engineers, 1278.2-1995.

IEEE. 2001a. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Framework and Rules*, Institute of Electrical and Electronics Engineers, P1516.

IEEE. 2001b. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Federate Interface Specification*, Institute of Electrical and Electronics Engineers, P1516.1.

IEEE. 2001c. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Object Model Template (OMT) Specification*, Institute of Electrical and Electronics Engineers, P1516.2.

INCHIOSA, M. E. AND PARKER, M. T. 2002. Overcoming design and development challenges in agent-based modeling using ASCAPE. In *Proceedings National Academy of Sciences*, *99* (3), 7304–7308.

KAMPIS, G. 2002. A causal model of evolution. In *Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning,* Singapore.

KAMPIS, G. AND GULYAS, L. 2003. Causal structures in embodied systems. *The European Research Consortium for Informatics and Mathematics News 53*.

KAMPIS, G. AND GULYAS, L. 2004. Out of interaction: A phenotype based model of species evolution. Accepted for the *5th International Workshop on Emergent Synthesis*.

LABROU, Y., FININ, T., AND PENG, Y. 1999. The interoperability problem: Bringing together mobile agents and agent communication languages. In *Proceedings of the 1999 Hawaii International Conference on System Sciences,* IEEE, Maui, Hawaii, USA.

LAW, A. M. AND KELTON, W. D. 2000. *Simulation Modeling and Analysis*, 3rd Ed. McGraw-Hill, New York, NY.

LU, T., CHUNGNAN, L., AND HSIA, W. 2000. Supporting large-scale distributed simulation using HLA. *ACM Trans. Model. Comput. Simul. 10*, 3, 268–294.

LUTZ, M. AND ASCHER, D. 1999. *Learning Python*. O'Reilly, Sebastopol, CA.

MCCLAVE, J. T. AND BENSON, P. G. 1994. *Statistics for Business and Economics*. Prentice Hall, Englewood Cliffs, NJ.

MINAR, N., BURKHART, R., LANGTON, C., AND ASKENAZI, M. 1996. The Swarm Simulation System, A Toolkit for Building Multi-Agent Simulations.

MITCHELL, M. 1996. *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*. MIT Press, Cambridge, MA.

MYJAK, M., SHARP, S., LAKE, T., AND BRIGGS, K. 1999. Object Transfer in HLA. In *Proceedings of the Simulation Interoperability Standards Organization Spring Simulation Interoperability Workshop*, IEEE.

NORTH, M. J. AND MASAL, C. M. 2005. Escaping the accidents of history: An overview of artificial life modeling with Repast. In *Artificial Life Models in Software*, A. Adamatzky and M. Komosinski, Eds. Springer, Heidelberg, Germany. 115–141.

NORTH, M., THIMMAPURAM, P., CIRILLO, R., MACAL, C., CONZELMANN, G., KORITAROV, V., AND VESELKA, T. 2003. EMCAS: An agent-based tool for modeling electricity markets. In *Agent 2003: Challenges in Social Simulation*, (University of Chicago, Chicago, IL USA), Argonne National Laboratory.

OMG. 2000. *OMG Agent Platform Special Interest Group Mission Statement*. Object Management Group, Needham, MA, www.omg.org/.

OMG. 2001. *OMG Unified Modeling Language Specification Version 1.5*. Object Management Group Needham, MA, http://www.uml.org/#UML1.5.

PADGETT, J. F. AND ANSELL, C. K. 1993. Robust action and the rise of the Medici, 1400–1434. *American Journal of Sociology 98*, 1259–1319.

PADGETT, J. F., LEE, D., AND COLLIER, N. 2003. Economic production as chemistry. *Industrial and Corporate Change 12* (4), 843–877.

POSLAD, S., BUCKLE, P., AND HADINGHAM, R. 2000. The FIPA-OS Agent Platform: Open Source for Open Standards. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*. Manchester, UK.

RICH, E. AND KNIGHT, K. 1991. *Artificial Intelligence*, McGraw-Hill, New York, NY.

ROAD. 2004. *Repast Home Page*. Repast Organization for Architecture and Design, Chicago, IL, repast.sourceforge.net/.

SALLACH, D. L. 2003. Social theory and agent architectures: Prospective issues in rapid discovery social science. *Social Science Computer Review 21*(Summer).

SALLACH, D. L. 2004. Repast for Oz/Mozart. M. North. Argonne, IL.

SDG. 2004. *Swarm Home Page*. Swarm Development Group, Santa Fe, NM, www.swarm.org/wiki/Main_Page.

SERENKO, A. AND DETLOR, B. 2002. *Agent Toolkits: A General Overview of the Market and an Assessment of Instructor Satisfaction with Utilizing Toolkits in the Classroom (Working Paper 455)*, McMaster University, Hamilton, Ontario, Canada.

STEPHAN, C. AND SULLIVAN, J. 2004. Growth of a hydrogen transportation infrastructure. In *Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence*, Chicago, IL, University of Chicago and Argonne National Laboratory.

SUN, X., LIU, F., AND XU, M. 2003. Research on interoperability of intelligent mobile agent for DIS. *ACM SIGSOFT Software Engineering Notes 28* (6), 9.

SUN MICROSYSTEMS. 2005. *J2SE Code Names*, Santa Clara, CA, java.sun.com/j2se/ codenames. html.

TANENBAUM, A. 1988. *Computer Networks*. Prentice Hall, Englewood Cliffs, NJ,.

TISUE, S. AND WILENSKY, U. 2004. NetLogo: Design and implementation of a multi-agent modeling environment. In *SwarmFest 2004*, Ann Arbor, MI, Swarm Development Group.

TOBIAS, R. AND HOFMANN, C. 2004. Evaluation of free Java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation 7*(1).

WILENSKY, U. 1999. *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

WILENSKY, U. AND STROUP, W. 1999. *HubNet*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

WOOLDRIDGE, M. AND JENNINGS, N. R. 1994. Agent theories, architectures, and languages: A survey. In *Proceedings of the 1994 Workshop on Agent Theories, Architectures & Languages*. Amsterdam, The Netherlands, Springer-Verlag.

Received January 2005; revised October 2005; accepted October 2005