RePast: An Extensible Framework for Agent Simulation

Nick Collier
Social Science Research Computing
University of Chicago
Chicago, IL 60637
nick@src.uchicago.edu

Abstract
RePast is a software framework for agent-based simulation created by Social Science Research Computing at the University of Chicago. It provides an integrated library of classes for creating, running, displaying, and collecting data from an agent-based simulation. This paper is an overview of RePast's design, features, and capabilities and describes the implementation of some of its key abstractions.[1]

1 Introduction

The University of Chicago's Social Science Research Computing's RePast is a software framework for creating agent-based simulations using the Java[2] language. It provides a library of objects for creating, running, displaying, and collecting data from an agent-based simulation. In addition, RePast includes several varieties of charts for visualizing data (e.g. histograms and sequence graphs) and can take snapshots of running simulations and create QuickTime movies of such. RePast borrows much from the design of the Swarm[3] simulation toolkit and can properly be termed a “Swarm-like” simulation framework.

At its heart, RePast behaves as a discrete event simulator whose quantum unit of time is known as a *tick.*[4] The tick exists only as a hook on which the execution of events can be hung, ordering the execution of the events relative to each other. For example, if event x is scheduled for tick 3, event y for tick 4, and event z for tick 5, then event y will

---

1 This paper corresponds to RePast version 1.4b
2 http://java.sun.com
3 http://www.swarm.org
4 In its implementation, RePast is more like a discrete time simulator, but this will change in the next

execute after event x and before event z. If no events are scheduled for a certain tick, then it is as if that tick never occurred. Ticks are merely a way to order the execution of events relative to each other.

Many RePast models have fairly simple schedules and scheduling requirements. However, the RePast scheduling mechanism allows for more sophisticated dynamic schedules such that the execution of an event can itself schedule other events for execution in the future. For example, assume that the modeler is simulating the transport of cargo and baggage by an airline through a network of airports. The "flight arrival" event might schedule the "baggage unloading event" and so on. This flexibility in the scheduling of events by no means prohibits the creation of more linear models where the same event executes each tick. In fact, novice users of RePast can easily create models without any knowledge of this scheduling mechanism. The creation and scheduling of events still occurs, although it is hidden from the user.

2 Agent-based Simulation with RePast

What then does this notion of ticks and discrete events mean in the context of agent-based simulation? The typical RePast model contains a set of agents. These agents may or may not be homogenous, or perhaps these agents are arranged in a hierarchy (a firm and its employees, for example). Regardless of their composition, each agent has some behavior, the interactions of which a modeler is interested in exploring. Assume that a modeler is working with a spatial iterated prisoner's dilemma type cooperation game played on a two dimensional grid. The agent's behavior would then take place in two phases. In the first, each agent would find its neighbors on the grid and play the game

---

release. Regardless, it appears to the user as a discrete event simulator.

with each neighbor using its current strategy. After all the agents had executed this first phase, each agent would then choose a strategy for the next round based on the payoffs from phase one.

In this sort of simulation, the same set of agent behavior gets executed every tick, and what gets scheduled then is an event or, as it is called in RePast, an action. This action first executes the phase one behavior for each agent and then the phase two behavior. As mentioned above, the modeler can have more or less direct control over the scheduling of events in her model. This particular example is amenable to RePast's auto-scheduling which allows the user to define three phases of behavior, a preparatory, an execution, and a post- or cleanup phase. RePast then schedules these in the appropriate order to occur every tick. Here, the neighbor finding and game playing would constitute the execution phase, and the strategy choice would constitute the post- or cleanup phase. Alternatively, the modeler can manually schedule this two-phase behavior as a single action. The first part of the action executes the neighbor-finding and game playing and the second part, the strategy choice.[5]

RePast also allows for more complex dynamic scheduling. A simulation of this sort typically schedules a single action to execute at the first tick, and this action then schedules another action to execute at some future tick. The mousetrap demonstration[6] simulation included with the RePast distribution is a good example of a dynamically scheduled simulation. What is being modeled here is a field of mousetraps on which some number of ping-pong balls are resting. When a mousetrap triggers, it throws its

---

5 The Endogenous Neighborhoods and Norms (enn) demonstration model in the RePast distribution is similar to this example (manually scheduled).

6 This is a port of the Swarm simulation of the same name and is itself a simulation of a demonstration of nuclear fission that used to take place in high school gymnasiums.

balls into the air and they then land on other mousetraps causing them to trigger. This cascading reaction continues until no more balls are in the air. The schedule here is fairly simple although its actual implementation in code is more complex than the linear example given above. The agent (the mousetrap) behavior is the triggering of other mousetraps, and this trigger behavior is defined as follows: find some number of neighboring mousetraps some distance away from the triggered mousetrap; schedule the execution of this trigger behavior on these mousetraps.[7] The simulation begins with a single trigger behavior executed on a randomly chosen mousetrap scheduled for the first tick. All the remaining behavior is scheduled dynamically and stochastically from within the simulation itself.

In short, a RePast simulation is primarily a collection of agents of any type and a model that sets up and controls the execution of these agents' behaviors according to a schedule. This schedule not only controls the execution of agent behaviors, but also actions within the model itself, such as updating the display, recording data, and so forth. Scheduling can be automated via the model or manually implemented by the modeler. In addition, this model is typically responsible for setting up and controlling simulation visualization, data recording and analysis. The model is said to be composed of these additional components (the schedule, the display, and so forth).

3 History

RePast was initially conceived of as a library of Java classes that would work together with and simplify the Swarm simulation framework. This initial conception was the result

---

7 The number of mousetraps, corresponding to the number of ping-pong balls, and their distance from the triggered mousetrap is a user-specifiable parameter of the model.

of University of Chicago researchers' concerns with the complexity of both Swarm and Objective-C and our respect for the maturity and elegance of the Swarm API. This notion of RePast as an extension to Swarm was soon abandoned for a variety of reasons and made partially redundant with the release of Java Swarm (a Java layer running on top of the Swarm kernel and released by the Swarm Development Group). Prior to the release of Java Swarm, we had begun some exploration into developing an independent framework completely written in Java, but borrowing several of the key abstractions present in Swarm. Convinced of this framework's viability and usefulness to University of Chicago researchers, the initial exploration grew into the current version of RePast. Today, RePast is used inside the University of Chicago and far beyond.

4 Design Goals

RePast's design goals grew out of our constituency's (University of Chicago researchers) concern with ease of use and the desire for a short learning curve, as well as our concerns about extensibility and robustness. We tried to meet these larger goals through the following design goals: abstraction of simulation infrastructure, extensibility, and "good enough" performance.

Abstraction

RePast abstracts most of the key elements of agent-based simulation and represents them as a Java class or classes. These classes cooperate to make a framework for creating agent-based simulations. Much of the design of this cooperation makes use of design

patterns[8] and achieves some small measure of elegance and clarity due to it. The current 1.4beta version of RePast provides a ready-to-use class or classes for most of the common infrastructural abstractions of an agent-based simulation (e.g., scheduling, display, data collection, and so forth) and a variety of generic components for constructing representational elements. These generic components include such things as agent spaces (grids, torii, "soups," etc.) and a few generic agent types. RePast is particularly strong in its support for network (social and otherwise) simulations. This support is both infrastructural (graph layouts, network generation, saving and loading) and representational (default node and edge classes).

Extensibility

As a design goal, extensibility grows largely from the success of the design and implementation of key abstractions, such that the modeler can use these abstractions as the basis for her own models. In making use of some of the Swarm abstractions, RePast inherits a time-tested design that contributes to its extensibility. By implementing some of these abstractions using design patterns extensibility is again enhanced. For example, the scheduling mechanism (the Schedule object and the various action classes) is implemented according to the composite design pattern, which allows client code to treat individual actions and the compositions of those actions uniformly. This provides clarity to the scheduling mechanism and allows it to be easily extended in the future. Extensibility is also provided by the use of Java as an implementation language. Object-oriented languages easily lend themselves to the creation of extensible frameworks

---

8 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.

through the use of inheritance and composition.

"Good Enough" Performance

"Good enough" performance refers to a level of performance that is acceptable when weighed against the other benefits of the toolkit. While performance optimizations were not part of the initial design, care was taken to minimize object creation and achieve acceptable display speed. This is not to say that performance concerns are ignored; there are incremental performance improvements with every release. RePast offers performance comparable to similar frameworks and will only get faster with the continual improvement of Java virtual machines.

As a result of these goals, Repast is robust, extensible, and fairly easy to use, although the modeler must still learn Java.[9] However, choosing Java as an implementation language has its own benefits. Java eliminates the kind of memory leaks associated with C, C++, and Objective-C, a particular problem for long-running simulations. Java is well documented and there are many instructional books devoted to it, and its cross platform design also makes installation and setup on a variety of platforms quite simple.

5 Package overview

Java allows the programmer to organize his or her code into packages. The package system is primarily used to avoid namespace conflicts so that two Java classes with the same name will not be confused. However, it is also used to partition code into coherent units. RePast consists of 210 classes organized into 9 packages as well as several

demonstration simulations. A modeler will always use a model class from the engine package and classes from other packages where required. The most important of these packages are described below.

Analysis

The classes in the analysis package are used to gather, record, and chart data. Using these classes a modeler defines data sources and hooks up recording or charting classes to these sources. Data can be easily recorded in a tabular or customized format and charted in a sequence graph, histogram or user-defined plot.

Engine

The engine classes are responsible for setting up, manipulating, and driving a simulation. The SimModel interface is the super-class for all models written with RePast. A partial implementation of SimModel, the SimModelImp, class is provided and can be used as the base class for most, if not all, models written with Repast. Alternatively, the SimpleModel class can be used to automate event scheduling as described above. The controller classes (BaseController, Controller, and BatchController) are responsible for handling user interaction with a simulation either through a GUI or by automating such interaction through the use of a batch parameter file. In addition, the engine package contains the classes that make up the scheduling mechanism.

Event

---

9 This should change soon with the introduction of the *Evolver* rapid simulation development environment.

As mentioned above, the schedule is responsible not only for the execution of agent actions, but also actions within the model itself, such as display updates and so on. However, not all communication between parts of a model is done via the scheduler. A small portion is performed using an event mechanism; these classes together with those in the engine package constitute this event mechanism. These classes are used internally by RePast and are not of real concern to the modeler.

Games

The games package contains a few classes for implementing prisoner's dilemma-type cooperation games.

Gui

The gui 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 various *Display classes work in conjunction with the classes in the space package to display these spaces appropriately. Via a DisplaySurface, the LocalPainter class handles the actual display of these spaces on the screen, and the DisplaySurface itself allows for the probing of the displayed objects. Probing, left clicking on the visualization of a simulation object, introspects that object (an agent for example) and displays its current parameters in a separate window. The gui package also contains the graph layouts used to visualize networks and an extensible Display class that can be used to build custom displays.

Network

The network package contains the core classes used to build network simulations. These include default node and edge classes working together such that a node "knows" its incoming and outgoing edges and an edge "knows" its source and target node. The NetworkFactory class is used to load networks from a file in a variety of formats as well as to generate networks (Small World, Random Density, and Square Lattice). Networks can be recorded as adjacency matrices in a variety of formats using the NetworkRecorder. In addition there are some utility classes that can be used to collect some simple yet useful network statistics.

Space

In an agent simulation, agents often have some sort of spatial relationship to each other. The space package contains classes that instantiate various sort of spacial relationships. The classes themselves are essentially container classes that represent various types of spaces (two-dimensional grids, torii, single or multiple occupancy, and so forth) accessible through the appropriate interfaces. For example, the grid spaces allow objects to be inserted and retrieved based on x and y coordinates. Spaces work in conjunction with the display classes in the gui package to present a visualization of the space and the objects (e.g., agents) that it contains.

Util

Util, the utilities package, contains a variety of utility classes used both internally by RePast and by the modeler. The two most important classes here are Random which encapsulates a large number of random number distributions and operations on them, and

SimUtilities which contains a number of static methods that shuffle lists, display dialogs, update probes and so forth.

In addition to its own classes, Repast also makes use of those in external libraries, most notably the Colt library.[10] The Colt library provides random number generation for RePast (encapsulated in the Random class) through its implementation of the MersenneTwister (MT19937), one of the strongest pseudo-random number generators. Various other random number generators and distributions are also found in the Colt library and are thus available for use with RePast. RePast also make use of the byte code generation facilities in the Trove library.[11]

6 Inside RePast

This section discusses how the scheduling and display mechanisms are implemented. The scheduling mechanism is a good example of how RePast's internals are implemented, while the display mechanism is a good example of how a modeler is able to compose his or her model using the pieces that RePast provides.

Scheduling Mechanism

The scheduling mechanism is responsible for all the user-defined state changes within a RePast simulation. As described above, scheduling consists of setting up and executing actions (agent behavior and so forth) at some specific time relative to other actions. As implemented in Java, this translates into setting up and executing method calls on objects

10 http://nicewww.cern.ch/~hoschek/colt/index.htm
11 http://opensource.go.com/Trove

at some specified time. RePast represents these method calls separately from the objects themselves through the BasicAction class. A BasicAction consists of a single abstract public void execute() method. Any classes that sub-class a BasicAction must implement this method, and it is in this method that the actual method call or calls to be scheduled should occur. So, for example, if your agent behavior is encapsulated by a *step* method, then the BasicAction's execute method would iterate through all the agents and call this *step* method on each one. This BasicAction, and not the *step* methods themselves, then gets scheduled for execution at some specific tick.

BasicAction-s can be created in two ways, either by the modeler or implicitly by a Schedule object. In the first, the modeler will sub-class a BasicAction, implementing the execute method accordingly. This sub-class is usually created as inner class (anonymous or otherwise). In the second, the modeler provides an object reference and the name of the method she wishes to execute as arguments to a Schedule object's schedule method. The schedule object will then dynamically create and load the byte-code for a BasicAction class whose execute method calls the named method on the specified object. For example, suppose a model class contains a method named “run” in which all the agents are iterated through calling a method named “step” on each agent. To schedule this run method, the modeler passes the name of the method, that is, “run,” and a reference to the model to the Schedule object. No sub-classing is necessary; the Schedule object does all the work. Furthermore, because the byte-code for the BasicAction is dynamically created, there is no performance penalty, as there might be with a solution that relied on reflection.

As mentioned above, the scheduling mechanism implements the composite design

pattern (see figure 1). The scheduling objects are composed into a tree structure that represents a part-whole hierarchy. In this context the BasicAction class (that is, anonymous and otherwise user-defined classes that inherit from BasicAction) is the component and the Schedule and ActionGroup classes are the composites or containers. As containers, the Schedule and ActionGroup objects primarily store the primitive or leaf components, although they can also store other Schedule or ActionGroup objects. Schedule objects store BasicAction-s and associated information about when to execute these BasicAction-s. ActionGroup-s allow for the grouping of BasicAction-s into groups of conceptually similar actions and provide methods to determine the order of execution of the BasicAction-s within that ActionGroup.

The operation that all these objects share is the execute method mentioned above. The primitive components implement the execute method to call the actual methods on the simulation objects, while the containers implement execute to call execute on their children. In addition to this execute method, the Schedule object contains other methods to add BasicAction-s to itself together with associated scheduling information. BasicAction-s can be scheduled to execute every iteration beginning at specified time, once at some specific time, repeatedly at a specified interval, once at a pause, or at the end of a simulation run. Regardless of how a BasicAction is created the methods for scheduling them are the same.

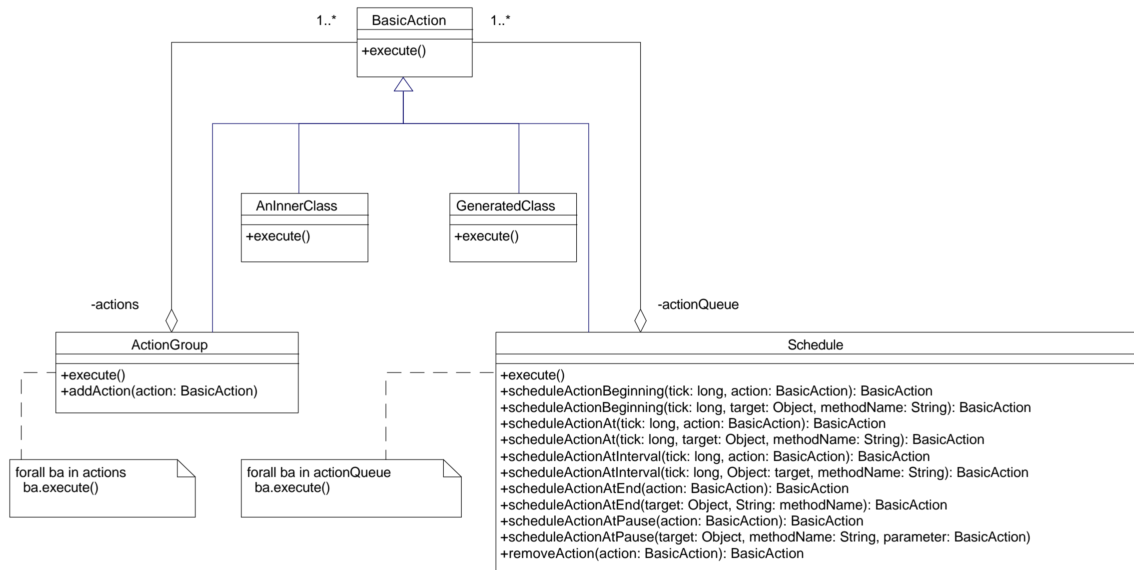

Figure 1

The master schedule that controls the execution of all its children will be created in a model implementing the SimModel interface primarily by extending SimModelImp. BasicAction-s will then be created and added to this Schedule object together with their associated scheduling information. When the scheduling mechanism is automated and hidden, the SimpleModel class creates a master Schedule and schedules some predefined methods representing the pre-, execution, and post- phases to this Schedule object. All the modeler must do then is fill in the implementation of these predefined methods. Schedules, BasicAction-s and so forth remain hidden.

Regardless of how the actual master Schedule is created and used, one of the controller classes mentioned above then begins a loop that calls, among other things, the execute method of this Schedule object. At this point the time or tick count is one. The Schedule object then builds the execution queue for the current time based on the scheduling information associated with each basic action, adding its child BasicAction-s to the queue if appropriate. It then iterates over this queue calling execute on the BasicAction-s in the queue. Consequently, the primitive or leaf component BasicAction-s will then call methods on actual simulation objects thus changing the state of the simulation. When the Schedule object finishes iterating over the execution queue, the tick count is incremented. It is this tick count against which BasicAction-s are scheduled for execution.[12] This design and implementation provides a clear and flexible scheduling mechanism. Complicated schemes of execution can be created through the composition of BasicAction-s, ActionGroup-s, and Schedule-s, while for simpler models or novice users the entire scheduling mechanism can be kept hidden.

Display Mechanism

The display mechanism is responsible for displaying a visualization of a running simulation in real time. The mechanism primarily consists of the space classes from the space package, the displays corresponding to those spaces, the SimGraphics class, the various drawable interfaces (Drawable, Drawable2DNode, and so forth) associated with the display and spaces classes, a LocalPainter and a DisplaySurface, all from the gui package. As mentioned above, spaces are ordered containers for simulation objects, most

---

12 The above is a somewhat simplified, omitting the description of a few simple optimizations in the creation of the event queue.

likely agents. For example, the space class Object2DTorus represents a two- dimensional toroidal grid where each grid cell can contain an object. Each display class contains a single space and provides an interface and implementation for displaying the objects contained within that space. If the objects within a space are to be displayed, those objects must be of a certain type. The various drawable interfaces define these types, and as interfaces they can be implemented by any type of object. The displays also implement the Probeable interface, taking screen coordinates, converting those coordinates into coordinates relevant to their topology and returning a list of objects at those coordinates. The SimGraphics class is a wrapper around java.awt.Graphics2D and as such simplifies the drawing of circles, rectangles, text, colors and so forth. The LocalPainter is a container for displays, and it handles the actual drawing of these displays, double buffering, and Graphics2D manipulation. The DisplaySurface handles probing and is the public interface to the drawing mechanism and the LocalPainter in particular. (A modeler will add displays to a DisplaySurface, which then adds them to the LocalPainter.)

Creating a RePast display is thus a matter of deciding on a space or spaces, implementing the drawable interfaces appropriate to these spaces in the objects that the spaces contain, adding these spaces to the appropriate displays, and then adding these displays to a DisplaySurface. RePast provides all of these except, naturally enough, the agents or objects inhabiting the space. The modeler "plugs-in," that is, composes her model from these pieces, and while it sounds complicated the actual code is only a few lines long.

Given this structure, the actual drawing sequence is as follows. The scheduling mechanism calls the updateDisplay method on the DisplaySurface object.

Having received this call, the DisplaySurface object tells the LocalPainter to paint itself. The LocalPainter then creates a java.awt.Graphics2D object from an off-screen BufferedImage, and wraps a SimGraphics object around this Graphics2D object. It then calls the drawDisplay(SimGraphics g) method on each display it contains, passing this SimGraphics object as an argument. The display then either gets a list of all the objects in the space it contains from that space, or if a list of objects was added to the display, that list is used.[13] The display then iterates through this list, requesting some drawing information (coordinates, size, etc.) from each object in the list through the appropriate drawable interface, and prepares the passed-in SimGraphics object using this information. Each drawable object in the list is then told to draw itself, using the passed-in SimGraphics object. When the painter has finished iterating through all the displays, it draws the off-screen image to the screen, and the drawing is finished. Network visualization is largely similar with the addition of GraphLayout-s that layout the graph prior to drawing it.

The structure of the display mechanism is thus one of composition where each container delegates the actual drawing responsibilities to its children. This provides flexibility and extensibility, although the tight conceptual coupling between spaces, displays, and drawable interfaces means that such extensibility requires the implementation of several classes and interfaces.

7 The Future

13 In most cases, the modeler has the option of adding a list of objects to be displayed directly to the displays together with the space. The list of objects received from the space may often contain null objects if that space is sparsely populated. Consequently, it is frequently more efficient to use the passed-in list, rather than that received from the space.

Current work on RePast is focused on the *Evolver*, a rapid simulation development environment for creating network simulations. Using a drag-and-drop model, a simulation can be graphically composed out of various pieces (pre-defined models, agents, analysis components, etc.). Any desired behavior not included in the pre-defined components can be specified using NQPython (Not Quite Python), a Python-like[14] language specifically designed to integrate well with RePast and much simpler than Java. We also hope to expand *Evolver's* capabilities beyond network models. *Evolver* is scheduled for public release in late 2001, early 2002. In addition to work on the *Evolver*, updates and improvements to the scheduler are planned. Among these are support for fractional within-tick scheduling and the incorporation of FAST, a distributed parallel scheduling and execution engine.

---

14 http://www.python.org