DEMETER, PERSEPHONE, AND THE SEARCH FOR EMERGENCE IN AGENT-BASED MODELS

M. J. NORTH[*], Argonne National Laboratory, Argonne, IL USA and
the University of Chicago, Chicago, IL USA
T. R. HOWE, Argonne National Laboratory, Argonne, IL USA
N. T. COLLIER, Argonne National Laboratory, Argonne, IL USA and
PantaRei Corp., Cambridge MA USA
J. R. VOS, Argonne National Laboratory, Argonne, IL USA and
the University of Illinois, Urbana-Champaign, IL USA

In Greek mythology, the earth goddess Demeter was unable to find her daughter Persephone after Persephone was abducted by Hades, the god of the underworld. Demeter is said to have embarked on a long and frustrating, but ultimately successful, search to find her daughter. Unfortunately, long and frustrating searches are not confined to Greek mythology. In modern times, agent-based modelers often face similar troubles when searching for agents that are to be to be connected to one another and when seeking appropriate target agents while defining agent behaviors. The result is a "search for emergence" in that many emergent or potentially emergent behaviors in agent-based models of complex adaptive systems either implicitly or explicitly require search functions. This paper considers a new nested querying approach to simplifying such agent-based modeling and multi-agent simulation search problems.

Keywords: emergence; exploration tools; infrastructure; agent-based modeling; multi-agent simulation

INTRODUCTION

In Greek mythology, the earth goddess Demeter was unable to find her daughter Persephone after Persephone was abducted by Hades, the god of the underworld. Demeter is said to have embarked on a long and frustrating, but ultimately successful, search to find her daughter. Unfortunately, long and frustrating searches are not confined to Greek mythology. In modern times, agent-based modelers and multi-agent simulationists often face similar troubles when searching for groups of agents. This paper addresses this issue by first considering the kinds of searches commonly found in agent-based models and multi-agent simulations. It then grounds the need for search in Holland's properties and features of Complex Adaptive Systems (CAS) (1995). Building on this foundation, this paper then introduces both Repast Simphony (Repast S) and the new Repast S nested querying approach to simplifying many agent-based modeling and multi-agent simulation search problems. Finally, this paper presents some conclusions.

OLD FRIENDS

Several kinds of searches appear to be common in agent-based models and multi-agent simulations, including queries for:

- agents that are to be connected to one another either at simulation startup time or later during simulation execution (e.g., finding a set of friends with specific properties for each person in a social network model);
- appropriate target agents while defining agent behaviors (e.g., finding the set of people that are physically near a specific infected person at a given point in time in a contagious disease propagation model);
- agents that approach or connect to a given agent (e.g., finding students that walk within a certain distance of a hall monitor in a elementary school simulation); and
- agents that change properties, state, or attributes (e.g., alerting a guard when a nearby person unexpectedly draws a weapon in a terrorism simulation).

[*] *Corresponding author address*: Michael J. North, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439; e-mail: north@anl.gov.

Are these types of queries needed for systems with emergence and if so, why? If they are needed, how can these queries be efficiently and conveniently embodied in agent-based models and multi-agent simulations?

THE SEARCH FOR EMERGENCE

The three properties and four mechanisms common to all CAS suggest an answer to the questions posed at the end of the previous section (Holland 1995)[1]:

- The *nonlinearity property* is present when components or agents exchange resources or information in ways that are not simply additive (e.g., rumors can be dramatically transformed when retold).
- The *diversity property* is present when agents or groups of agents differentiate from one another over time (e.g., people are unique).
- The *aggregation property* is present when a group of agents is treated as a single agent at a higher level (e.g., people form clubs).
- The *flows mechanism* is present when resources or information are exchanged between agents such that the resources can be repeatedly forwarded from agent to agent (e.g., rumors spread from person to person).
- The *tagging mechanism* is present when identifiable flags are present, allowing agents to identify traits of other agents (e.g., a scowling person *may* be hostile).
- The *internal models mechanism* is present when formal, informal, or implicit representations of the world are embedded within agents (e.g., each person has a view of who likes who in a club).
- The *building blocks mechanism* is present when an agent participates in more than one kind of interaction where each interaction is a building block for larger activities (e.g., a person can be in a club and also work in an office).

In principle, every one of these properties and mechanism can make use of searching or querying for agents. For example:

- The *nonlinearity property* can arise from many sources including the nonlinear results of queries for agent lists (e.g., making lists of the people in each region of a geographical space can produce nonlinear results).
- The *diversity property* can be leveraged by using partial matches in queries for agent lists (e.g., finding all the active Artificial Intelligence (AI) researchers in given geographical region).
- The *aggregation property* can arise by grouping agents using queries (e.g., a selected subset of the active AI researchers in a given geographical region might form a professional association for that region).
- The *flows mechanism* can use querying to identify future sources of and targets for flows (e.g., finding a list of AI researchers to notify about a given conference).
- The *tagging mechanism* provides sets of agent attributes for querying (e.g., tagging people as active AI researchers allows them to be candidates to receive AI conference announcements).
- The *internal models mechanism* can use querying as a tool for model building (e.g., people may form ideas about AI by finding and contacting active AI researchers).
- The *building blocks mechanism* can be leveraged much like the diversity property by using partial matches in queries for agent lists (e.g., finding all of the active AI researchers in given geographical region and then contacting them for both a conference announcement and annual association dues).

This discussion suggests that querying can play integral role in CAS. As such, we are faced with what might be called a "search for emergence" in that many emergent or potentially emergent behaviors in agent-based models of CAS either implicitly or explicitly leverage or even require search functions. This paper builds on this idea by offering a new nested querying approach to simplifying such agent-based modeling and multi-agent simulation search problems. This new approach is being implemented within Repast S.

---

[1] Please see North and Macal (2005) for further exploration of these ideas relative to artificial life.

REPAST S[2,3]

Repast (ROAD 2005) is a widely used free and open source agent-based modeling and simulation toolkit with three released platforms, namely Repast for Java, Repast for the Microsoft .NET framework, and Repast for Python Scripting. Repast S extends the Repast portfolio by offering a new approach to simulation development and execution including a set of advanced computing technologies for applications such as social simulation. North, Howe, Collier, and Vos (2005a and 2005b) provide an overview of the Repast S runtime and development environments.

As discussed in North, Howe, Collier, and Vos (2005a and 2005b), once a model's *essence* is completed the *accident*, in the Aristotelian sense, of Repast S model software design and development is nominally intended to proceed as follows[4]:

- The modeler designs and implements model pieces, as needed, in the form of plain old Java objects (POJOs), often using automated tools.
- The modeler uses declarative configuration settings to pass the model pieces and legacy software connections to the Repast S runtime system.
- The modeler uses the Repast S runtime system to declaratively tell Repast S how to instantiate and connect model components.
- The Repast S runtime system automatically manages the model pieces based on (1) interactive user input and (2) declarative or imperative requests from the components themselves.

Repast S is designed to use two major types of declarative specifications, namely model and scenario descriptors, to integrate models. Model descriptors define what *can be* in a model such as the allowed agent types, permitted agent relationships, and automated notification (*watching*) information. Scenario descriptors define what *actually is* in a model such as agent data sources, visualizations, and logging. Model and scenario descriptors are stored in separate XML files. Model descriptors are to be created at model development time while scenario descriptors are expected to be created at run time. The Repast S development environment is expected to provide both a wizard for creating and a point-and-click editor for modifying model descriptors. The Repast S runtime environment includes a point-and-click panel for creating and maintaining scenario descriptors.

Repast S uses annotations, a new feature in Java 5, to declaratively mark code for later operations. Annotations are metadata tags that are compiled into binary class files. Like comments, annotations are not directly executed. Unlike comments, annotations can be stored in the compiled versions of source code[5]. This storage allows executing Java programs such as the Repast S runtime system to read and act on the encoded metadata. This allows Repast S developers to declaratively mark or annotate code at design time for special processing by the Repast S runtime system. This facility is used for tasks such as declaring *watchers* as discussed later.

CONTEXTS AND PROJECTIONS[6]

Repast S represents ABMS spaces and places through the use of *contexts* and *projections*. Repast S contexts are hierarchically nested named containers that hold model components. The model components can be any type of POJO, including other contexts, but are often expected to be agent objects. Each model component can be present in as many contexts as the modeler desires. The hierarchical nesting means that a model component that is present in a context is also present in all of that context's parent contexts. Of course, the converse is *not* true in the general case. The hierarchical nesting structure itself can be declaratively or imperatively specified by the modeler. Context membership and structure is completely dynamic and agents can be in any number or combination of

---

[2] This discussion of Repast S follows North, Howe, Collier, and Vos (2005a, 2005b, 2006a, and 2006b).

[3] It is important to note that Repast S and its related tools are still under development. This paper presents the most current information as of the time of the writing. However, changes may occur before the planned final release.

[4] It should be noted that iterative refinement through repeated phases of *essential* activities and *accidental* work or other combinations of phase interleaving may actually occur.

[5] More details on Java annotations can be found in Viswanath (2005).

[6] Contexts and projections are discussed in greater depth in North, Howe, Collier, and Vos (2006a).

contexts at any time. Furthermore, agents can themselves contain any number of contexts and can even be contexts. In addition, the contents of components within contexts (e.g., agent properties) can be declaratively logged at runtime.

In addition to supporting hierarchical nesting, contexts support *projections*. Repast S projections are named sets of relationships defined over the members of a context. For example, a Repast S network projection stores a network or graph relationship between the members of its context. The members of this context can then ask who they are linked to and who is linked to them. Similarly, the Repast S grid projection stores a set of Cartesian coordinates for each member of the context. The members of this context can ask where they are. Each context can support any mixture of projections. Also, projections can be declaratively visualized at runtime as shown in Figure 1. A wide range of projections will be included in the final release of Repast S, such as:

- networks including directed graphs, undirected graphs, weighted graphs, and trees;
- multidimensional discrete grids including both toroidal and bounded surfaces;
- multidimensional continuous spaces also including both toroidal and bounded surfaces;
- two dimensional geographical information systems surfaces; and
- three dimensional geographical information systems surfaces on occasion.

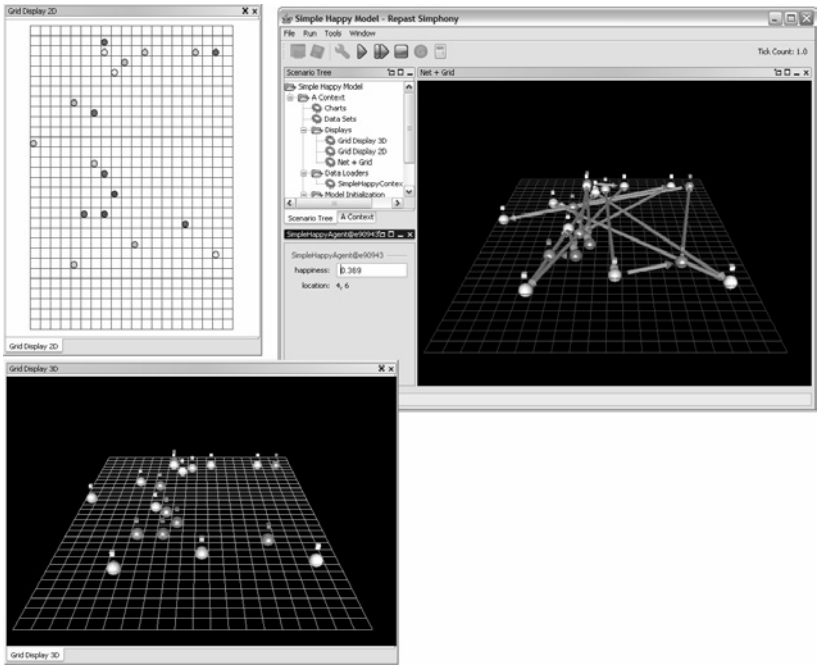

Figure 1: The Repast S Runtime System Showing a Context with a Network and a Two Dimensional Grid. The Network and Grid are Shown Using Several Different Visualizations

Repast S contexts and projections work with watchers. Repast S watchers are automated listeners or call back procedures that trigger based on complex queries. These queries define what kinds of agents to watch, where to watch for the tracked agents, what to look for in each tracked agent, and when to react to events in the tracked agents. Watchers are typically defined declaratively using Java annotations. Repast S contexts work directly with watchers by allowing watcher queries to use context names and properties to define what and where to watch. Similarly, projections work directly with watchers by allowing watcher queries to use projection names, properties, and relationships.

QUERYING

Simply put, Repast S querying provides support for finding complex subsets of agents. Queries are defined using the following predicates[7]:

[7] All of the predicates listed here are implemented except for, Hex. This predicate is currently under development.

- InContext: This predicate determines whether the object exists in a given context.
- InstanceOf: This predicate determines whether the object is of a particular class.
- Equals: This predicate determines whether the object is equal to a given object.
- PropertyEquals: This predicate determines whether a property in the object is equal to a given value.
- PropertyLessThan: This predicate determines whether a property in the object is less than a given value.
- PropertyGreaterThan: This predicate determines whether a property in the object is greater than a given value.
- NetworkAdjacent: This predicate determines whether the object is linked to a given object in a specified network.
- NetworkSuccessor: This predicate determines whether the object has an inbound edge from a given object in a specified network.
- NetworkPredecessor: This predicate determines whether the object has an outbound edge to a given object in a specified network.
- Touches: This GIS predicate determines whether the object touches a given object in space.
- ContainedBy: This GIS predicate determines whether the object is contained by a given object in space.
- InEnvelope: This GIS predicate determines whether the object is within a given envelope (bounding box) in space.
- And: This predicate implements intersection.
- Or: This predicate implements union.
- Not: This predicate implements negation.
- VonNeumann: This predicate determines whether an object is within the Von Neumann neighborhood of a given object in a grid.
- Moore: This predicate determines whether an object is within the Moore Neighborhood of a given object in a grid.
- Hex: This predicate determines whether an object is within the hexagonal neighborhood of a given object in a grid.
- Within: This GIS and non-GIS predicate determines whether the object is within a given distance of a specified object in a gis space, a non-gis grid or abstract continous space, or within a given path length in a network.

Searches using these predicates can be performed either imperatively or declaratively.

When used in an imperative mode, Repast S queries normally return a list scanning object or *Iterator*. These iterators can be used in programmed agent behaviors to operate on and react to members of the list. An example is shown in Figure 2. In this apocryphal example, Demeter starts her search for Persephone by asking linked friends if they have seen her daughter. The process beings by creating a network query that returns all of the people linked to Demeter. Each of these linked friends is then asked if they have seen Persephone. If they have seen her, then Demeter ask that friend about Persephone and remembers what she hears. Of course, this is a fanciful example. The real point is that constructing and using such queries is a straightforward process[8].

As a note, more direct approach to the search for Persephone that might be used by Zeus is shown in Figure 3. Here Zeus uses his power to directly find Persephone, regardless of where Hades has hidden her. He simply creates a query that uniquely specifies Persephone, and then searches for her starting with the main context.

[8] In the mythical story Zeus eventually ordered Hades to return Persephone to Demeter, but unfortunately there was a cost. Depending on the version of the story, Hades secretly fed Persephone some pomegranate seeds or Persephone voluntarily ate some pomegranate seeds that either way forced her to return to the underworld for several months each year. Demeter's periodic disappearance to search for her daughter is then said to be the cause of the seasons. It seems that even with Zeus' mythic help, search is not free! Clearly Repast S is not as powerful as a legendary Greek god, but we hope our search system has less costly results!

Querying treats grids, networks, GIS, and other environments in a consistent and uniform manner. An imperative example is shown in Figure 4. In this example three projection queries are defined. The first searches a network, the second searches a grid, and the third simultaneously searches both a grid and a network.

```
// Create a network query that lists all of Demeter's friends.
Query<Person> netQuery = new NetworkAdjacentQuery(demeter);
Iterable<Person> friends = netQuery.query(context);

// Scan the list of agents that were found.
for (Person friend : friends) {

      // Ask the next friend if they have seen Persephone.
      if (friend.seen("Persephone")) {

            // Find out what we can about Persephone.
            demeter.remember(friend.askAbout("Persephone"));

      }

}
```

Figure 2: An Example Imperative Use of a Repast S Query

```
// Create a query that can search the main content in the simulation.
Query<Person> zeusQuery = new AndQuery(
      new PropertyEqualsQuery("Name", "Persephone"),
      new PropertyEqualsQuery("Mother", "Demeter"));

// Immediately find Persephone (she is the first and only
// person in the list).
Person persephone = zeusQuery.query(mainContext).next();
```

Figure 3: Another Example Imperative Use of a Repast S Query

Repast S querying also allows agent subsets to be created in a declarative manner. This lays the groundwork for a declarative query language. A simple example that might be used by Hades is shown in Figure 5. In this example Hades is using a Repast S watcher to keep an on eye the people in Tartarus[9] to make sure that no one becomes content. If anyone in the Tartarus context starts to become content then Hades is notified through a call to the `quellContentment` method and nips the growing joy in the bud using the supplied `contentPerson` parameter. Unfortunately for the doomed, Repast S will make this an efficient process since the runtime system uses the listener design pattern to limit checking to only those times and objects for which the watched values change.

```
// Three kinds of relationships are created, each by changing one
// line of code.

// These lines create a network query.
Query<Agent> netQuery = new NetworkAdjacentQuery(myAgent);
Iterable<Agent> results = netQuery.query(context);

// These lines create a grid network query.
```

[9] Many different geographies of the Greek mythical afterlife have been received from antiquity. The version referenced here divides the geography into two major areas, namely the Elysian Fields and Tartarus. The Elysian Fields or Elysium was a pleasant land reserved for the good. In contrast, Tartarus or Erebus was a foul depth reserved for the extremely evil. Three goddesses called the Furies were said to torment the evil in Tartarus. Here for simplicity we have Hades invoking the Furies. Other versions of the story include extra regions, additional gods, and different divine responsibilities.

```
Query<Agent> vnQuery = new VonNeumannQuery(myAgent)
results = vnQuery.query(context);

// These lines create a combination of a grid and a network query.
      Query<Agent> unionQuery = new UnionQuery(netQuery, vnQuery);
results = unionQuery.query(context);
```

Figure 4: Example Imperative Queries Showing Multiple Agent Relationships

```
@Watch(watcheeClassName = "Person",
      watcheeFieldName = "contentment",
      query = "in 'Tartarus'",
      triggerCondition = "$watchee.contentment > 0",
      whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
      public void quellContentment(Person contentPerson) {

            // Tell the Furies to nip the joy in the bud.
            furies.torment(contentPerson);

      }
```

Figure 5: An Example Use of a Repast S Query for a Declarative Behavior Trigger Query

CONCLUSIONS

In Greek mythology, the earth goddess Demeter is said to have embarked on a long and frustrating, but ultimately successful, search to find her kidnapped daughter. In modern times, agent-based modelers often face similar troubles when searching for agents that are to be to be connected to one another and when seeking appropriate target agents while defining agent behaviors. The result is a "search for emergence" in that many emergent or potentially emergent behaviors in agent-based models of CAS either implicitly or explicitly require search functions. This paper has presented a new nested querying approach to simplifying such search problems. This approach offers both imperative and declarative methods to find sets of agents with particular attributes in particular locales. In particular, the declarative approach introduces a temporal dimension to the process by explicitly allowing for searches that activate in the future.

REFERENCES

Holland, J. H., *Hidden Order: How Adaptation Builds Complexity*, Addison-Wesley, Reading, MA, USA (1995).

North, M.J., N.T. Collier, and J.R. Vos, "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit," *ACM Transactions on Modeling and Computer Simulation*, vol. 16, issue 1, pp. 1-25, ACM, New York, New York, USA (January 2006).

North, M. J., T. R. Howe, N. T. Collier, and J. R.. Vos, "The Repast Simphony Runtime System," *Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, Argonne National Laboratory, Argonne, IL USA (October 2005a, In Press).

North, M. J., T. R. Howe, N. T. Collier, and J. R. Vos, "The Repast Simphony Development Environment," *Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, Argonne National Laboratory, Argonne, IL USA (October 2005b, In Press).

North, M. J., T. R. Howe, N. T. Collier, and J. R. Vos, "Spaces, Places, and the Lernaean Hydra Of Agent-Based Modeling," *North American Association for Computational Social and Organizational Science 2006 Conference*, Argonne National Laboratory, Argonne, IL USA (Accepted for June 2006a).

North, M. J., T. R. Howe, N. T. Collier, and J. R. Vos, "Supporting Validation with Declarative Model Integration," *First World Congress on Social Simulation*, Kyoto, Japan (Accepted for August 2006b).

North, M. J., and C. M. Macal, “Escaping the Accidents of History: An Overview of Artificial Life Modeling with Repast,” in A. Adamatzky and M. Komosinski, eds., *Artificial Life Models in Software*, pp. 115-141, Springer, Heidelberg, FRG (April 2005).

ROAD, *Repast Home Page*, Available as http://repast.sourceforge.net/, Repast Organization for Architecture and Design, Chicago, IL USA (2005).

Viswanath, K., “Java Annotation Facility - A Primer,” *Java Developers Journal Online*, Available as http://java.sys-con.com/read/48539.htm, SYS-CON Media, (March 9, 2005).