**A Java Reinforcement Learning Module for the Recursive Porous Agent Simulation Toolkit:**
**facilitating study and experimentation with reinforcement learning in social science multi-agent simulations.**

by

Charles Gieseler

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Vasant Honavar, Co-major Professor
Leigh Tesfatsion, Co-major Professor
Dimitris Margaritis

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the master's thesis of

Charles Gieseler

has met the thesis requirements of Iowa State University

_____
Co-major Professor

_____
Co-major Professor

_____
For the Major Program

## DEDICATION

To my parents, Mardi and Gene Gieseler, and to the Lamms, my second family, for believing in me even when I stopped believing in myself. Thank you for all your love and support.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

This work details a machine learning tool developed to support computational, agent-based simulation research in the social sciences. Specifically, the Java Reinforcement Learning Module (JReLM) is a platform for implementing reinforcement learning algorithms for use in agent-based simulations. The module was designed for use with the Recursive Porous Agent Simulation Toolkit (Repast), an agent-based simulation platform popular in computational social science research. Background, architecture, and implementation of JReLM are discussed within. This includes explanation of pre-implemented tools and algorithms available for immediate use in Repast simulations. In addition, an account of JReLM's use in an agent-based computational economics simulation is included as an illustrative application. Directions for further development and future use in ongoing agent-based computational economics work are discussed as well.

# CHAPTER 1. Introduction

Researchers in the social sciences are becoming increasingly adept at harnessing the power of modern computing through of a variety of computational tools and techniques. Among the selection of computational methods, agent-based simulation is gaining popularity world wide. This approach involves modeling with populations of individual actors and observing patterns that emerge from their interaction. Importantly, not only can these individuals act in a simulation environment, but they have the capacity to do so independently of each other or even a human user. That is, they are programs that exhibit some degree agency and autonomy. In addition, they will usually act to achieve some purpose or goal. This capacity for self-directed, goal-oriented action allows such programs to be used as metaphors for human actors. Such agents, as they are called, provide the foundation for building larger metaphors of human populations from the bottom up [1].

How agents direct their own actions directly affects what kind of patterns can emerge at the population level. Thus agent behavior is of central importance in agent-based simulations and this behavior can be driven by variety of techniques. In some cases, simple rule-based behavior will suffice. In other cases, more sophisticated behavior is required, especially when a dynamic environment calls for an adaptive response. Contributions to address these circumstances come from other fields such as evolutionary computation, cognitive modeling and machine learning.

This work presents an effort to facilitate the use of machine learning techniques in social science agent-based simulations. Presented here is the Java Reinforcement Learning Module (JReLM), a platform for the implementation and use of reinforcement learning algorithms

[1] Robert Axelrod and Leigh Tesfatsion provide an On-Line Guide for Newcomers to Agent-Based Modeling in the Social Sciences. Available at `http://www.econ.iastate.edu/tesfatsi/abmread.htm` (current as of November 2005)

in the Java programming language. This platform is designed specifically for use with a widely used agent-based simulation tool, the Recursive Porous Agent-based Simulation Toolkit. JReLM has been developed with the intention to eventually evolve into a full library of pre-implemented, easy-to-use algorithms.

JReLM began as an undertaking to support the work of the Agent-based Computational Economics group in the Department of Economics, Iowa State University. Agent-based Computational Economics (ACE)[2] is an approach to the study of economic systems that specifically makes use of simulation models of interacting agents. Agent-based methods can provide insightful tools for economic study and are gaining acceptance and importance in the economics community [16]. JReLM's development has been and is still being guided by the efforts of ISU's ACE group.

This first chapter covers a brief discussion of agent-based simulation platforms and an introduction to the Recursive Porous Agent Toolkit. Following chapters discuss the specifics of the JReLM platform and its use. Chapter 2 surveys the interface structure that provides the foundation for component-based, extensible implementation of algorithms. Chapter 3 discusses pre-implemented structures provided to support such implementations. Chapter 4 covers the initial pre-implemented reinforcement learning algorithms that are included in the first version of JReLM. Chapter 5 explains an illustrative application of JReLM in an economics agent-based simulation. Finally, chapter 6 concludes with a discussion of the future direction of JReLM including the expansion of pre-implemented reinforcement learning algorithms, learning process visualization, and the further integration of the module into the Repast package.

## 1.1 Overview of Repast

The Recursive Porous Simulation Toolkit (Repast) is an open source platform for building, managing, and analyzing agent-based simulations. Originally developed by Sallach, Collier, Howe, North [1], at the University of Chicago, Repast was inspired by the Santa Fe Institute's agent-based simulation platform, Swarm [9]. Over the years it has be maintained by a number

[2]Tesfatsion's ACE website provides an expansive source of ACE related information. It is located at `http://www.econ.iastate.edu/tesfatsi/ace.htm` (current as of November 2005)

of organizations, such as Argonne National Laboratory, however, it is now maintained by a dedicated, non-profit organization called the Repast Organization for Architecture and Development (ROAD). The Repast development project is currently hosted by sourceforge.net [3] where it is constantly being expanded, refined and improved by the open source community.

See figure 1.1 shows a screen capture of Repast as an application running a demonstration simulation model called HeatBugs.

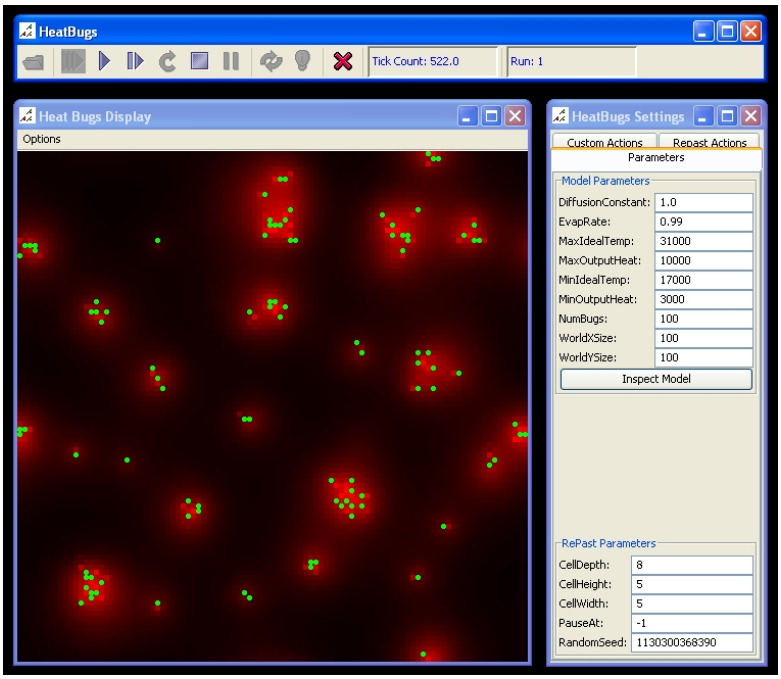


Figure 1.1 Repast running the HeatBugs demo model

One of the reasons that Repast was chosen as the base platform to develop for is that it is the primary simulation tool being used ISU's ACE group. However, Repast is also widely used and is recommended by some as the tool of choice for agent-based computational social science research. Gilbert and Bankes [4] surveyed some of the major platforms available as of 2002, with the particular needs of social science in mind. This survey included Repast, Swarm, Starlogo, Agentsheets and others. They briefly discussed the focus of each platform their implementation characteristics, their pros and cons. Repast was said to be a substantial

[3] Sourceforge is a prominent open source project management web portal, managing over 100,000 registered projects at the time of this writing. `http://sourceforge.net` (current as of November 2005)

library with the capability to build more sophisticated simulations when compared to some of the simpler platforms such as Starlogo and Agentsheets. However, they did mention that one of Repast's drawbacks was its complexity. Repast requires the user to learn the Java programming language whereas Starlogo and Agentsheets provide interfaces to build simulations visually. The purpose of JReLM is to mitigate some of this added complexity, specifically in regards to adaptive agent behavior.

A more in recent and in depth comparison of agent-based simulation platforms was done by Tobias and Hofmann [17] in 2004. For this comparison the authors proceeded in two stages. First, they pre-screened a selection of simulation frameworks found via internet search according to three criteria. First, the framework must allow the implementation of agent-based models grounded in social science theory. Second, the framework must be freely available. That is, a user must be able to obtain and use the framework without having to purchase a license. Finally, a Java version of the framework's source code must be available for customization and modification. Of the 21 simulation frameworks considered for investigation, only four Repast, Swarm, Quicksilver, and VSEit passed the pre-selection criteria. Once the four were selected, detailed evaluation proceeded using five categories containing numerous criteria and a rating system specifying how well each framework satisfied these criteria. In both a weighted and unweighted rating Repast came out on top. In the end, the authors themselves state

"...we can conclude with great certainty that according to the available information, RePast is at the moment the most suitable simulation framework for the applied modeling of social interventions based on theories and data."[17](pages 26-27)

For further information on Repast, Tesfatsion [15] provides an in-depth self-study guide along with a collection of resources for learning how to build and run simulation tools.

## 1.2 Adaptive agent behavior in Repast

A powerful aspect of Repast, and one of the primary goals in its development, is that it allows the open-ended definition of agents for customized simulations. From the Repast

homepage [4]:

"Repast includes a variety of agent templates and examples. However, the toolkit gives users complete flexibility as to how they specify the properties and behaviors of agents."

This flexibility allows Repast to be applied to a wide range of domains. However, it also means the burden of agent design and implementation is passed on to the simulation developer. While this may not be much of a problem where simple agents will suffice, in many applications it is useful to provide agents with more sophisticated capabilities; agents that can learn from experience and adapt to changing circumstances are desirable in many contexts.

RepastJ 3.1, the latest Java version of Repast, offers some infrastructure for creating agents with adaptive behavior. First, there is a sub-package for genetic algorithms. It is primarily an example of the use of evolutionary methods in the Repast environment rather than a module that can be used in general Repast simulations. As such, is grouped with other demonstration simulations. A second sub-package provides classes for building neural networks. It is an integration of the Java Object Oriented Neural Engine (JOONE)[7], a third-party framework for generating neural networks, and includes utilities for manipulating neural networks in a Repast simulation. An example showing the use of this neural network package is also included among the demonstration simulations.

To study other types of adaptation, in a multi-agent simulation context with Repast, programmers must implement the desired algorithms themselves or rely on third party libraries. Custom implementation can be time-consuming even for experienced programmers and confusing for novices. While third-party libraries do exist, at this time there are none that are specifically designed to integrate with the Repast simulation environment or allow the management learning through the Repast interface. The following chapters present a Java module developed to provide adaptive learning behavior, via reinforcement learning methods, specifically for agents in Repast simulations.

[4]The Repast homepage is located at `http://repast.sourceforge.net` (current as of November 2005)

# CHAPTER 2. Core Architecture

The Java Reinforcement Learning Module (JReLM) is a platform for developing and using reinforcement learning in Repast simulations. In the following section we go into the details of the Java classes and interfaces of JReLM. Here we distinguish between two types of users. The client programmer, or client for short, is a programmer using the components of the JReLM package in his or her own Java code. The end user, or user for short, is someone accessing features of JReLM through Repasts or JReLMs graphical user interface. The client programmer is usually someone who is building a Repast simulation while the end user is usually someone who is running a Repast simulation.

JReLM is implemented in the Java programming language. As such, Java terminology, conventions and references to common Java structures will be used throughout the following chapters. In the context of a Java class or interface, the term "method"' refers to a subroutine (or function in C++). In this document, method names are written as "*methodName()*." In addition, it is assumed the reader is familiar with the basic concepts of Object Oriented Programming (e.g. encapsulation, inheritance).

## 2.1 JReLM General Overview

The components of JReLM are motivated in part by Sutton's and Barto's [14] description of a reinforcement learning agent. The same structure and terminology have been maintained wherever possible. However, JReLM is focused on building units that learn from agent experience and then drive agent behavior based on that experience and not on the construction of the agents themselves. The particular details of agents may be tailored to individual simulations while leaving the implementation of reinforcement learning to JReLM. Such learning units are

referred to below as a learning engines, or learners for short.

JReLM is meant to provide an easy way for the client programmer to "drop" reinforcement learning algorithms into agents in Repast simulations. It accomplishes this in two ways. First, it provides a set of existing algorithms and tools that can be utilized in a wide variety of simulation contexts. Second, it provides a highly extensible platform for the implementation of further algorithms or tools.

In addition, JReLM is intended to be released as open source software. This means the source code and documentation will be made freely available to the public for use and modification. In fact, JReLM may be released for distribution with Repast itself as an included supplementary module. This is discussed further in chapter 6.

## 2.2 Specifics

This section details JReLM's specific Java structures and explains the relationship between them. Only the major components are discussed here.

Figure 2.1 shows a diagram outlining JReLM's package hierarchy. It may be helpful to refer back to this diagram as we go into the details of the JReLM components.

JReLM is divided into several subpackages. The edu.iastate.jrelm.core subpackage contains a collection of interfaces and classes for bridging the gap between the activities agents undertake in a simulation environment and the learning processes driving agent behavior from within. In addition, there are two classes to aid in building and managing learning agents. The edu.iastate.jrelm.rl subpackage contains interfaces for implementing reinforcement learning components as well as the pre-implemented algorithms provided by JReLM. The edu.iastate.jrelm.similarity and edu.iastate.jrelm.spillover contain structures for advanced optional features involving measuring the similarity of actions and the partial distribution of rewards to similar actions. The edu.iastate.jrelm.gui subpackage contains structures for graphical user interfaces. The edu.iastate.jrelm.util subpackage provides supporting structures for use in other JReLM components. Finally, edu.iastate.jrelm.demo subpackage contains a simple Repast model implementing an N-Armed bandit game. This is provided to demonstrate the

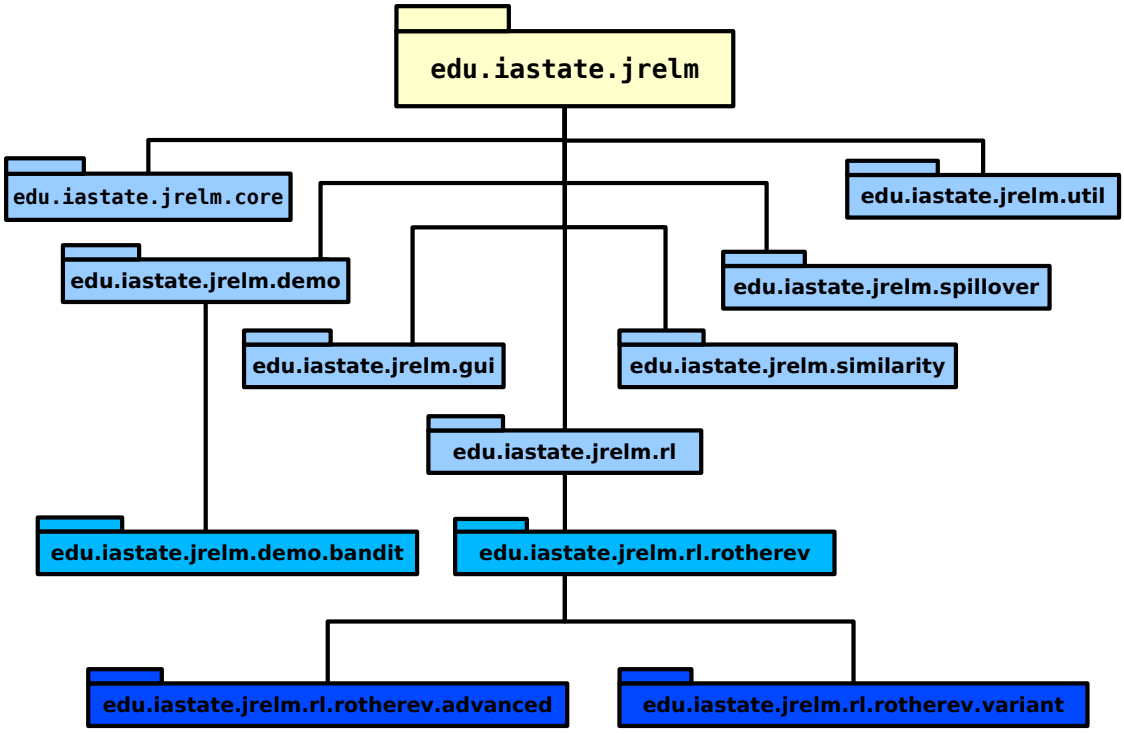


Figure 2.1 JReLM Package Hierarchy Diagram

mechanics of using JReLM.

### 2.2.1 edu.iastate.jrelm.core

Figure 2.2 shows the class diagram for the the core package. The following subsections explain all of the interfaces and classes within this subpackage, except for the JReLMAgent and BasicLearnerManager which are covered in sections 3.1.3 and 3.1.4 respectively.

#### 2.2.1.1 Action

To begin with, we will look at JReLM Actions. Action is a Java interface for classes that implement specific agent behavior choices. They are internal representations of basic operations that an agent can perform. In a given Repast model an agent can, presumably, perform operations that allow it to interact with the simulation environment. Since Repast allows for open-ended definition of models, it is impossible to predict all the operations that

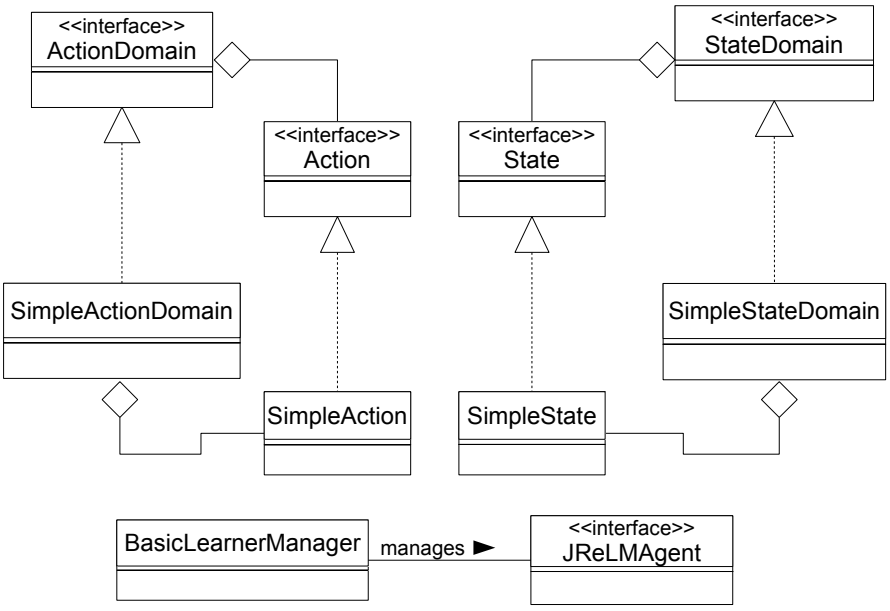


Figure 2.2 Class diagram for edu.iastate.jrelm.core

might be offered to an agent as well as how to interact with those operations as Java classes or methods. As such, it is useful to distinguish between external operations and their internal representations.

Classes that implement the Action interface can vary in sophistication. They may simply indicate an operation choice or they may fully encapsulate an operation, containing all needed data and subroutines. In addition a single Action object may actually represent a series of operations or a strategy in the simulation environment. However, all Actions must be able to be uniquely identified. Any class implementing the Action interface is required to have an associated identification object and must return this object when the Action's *getID()* method is called. The type of object to use as identification can be specified by the client when Action is implemented. For example, an Action may be identified by a numerical value, string or a more complicated, custom-built identifier.

It should be noted that JReLM Action is distinct from the Repast BasicAction class and all other BasicAction related classes. A BasicAction is an operation on the simulation level that can be executed by a models schedule. JReLM Action is intended to represent agent level operations. It is an internal representation of an activity an agent knows about and can

perform in a given simulation. This interface was given the name Action because an actor (i.e. an agent) performs the represented operation.

#### 2.2.1.2 ActionDomain

ActionDomains represent the space of possible actions that an agent can choose from. Classes implementing this interface manage all operations available to an agent in the form of Action objects. The use of ActionDomain separates the representation of action choice space from the specific collection of simulation operations offered to an agent. So a learner can be used in any agent action choice domain that can be represented by a class implementing the ActionDomain interface.

ActionDomain leaves open how Actions may be organized and managed. Implementing classes may organize Actions by table, list, function or whichever form is most appropriate for a specific domain. Moreover, the client may define the type of Action identifiers, just as with the Action interface.

One limitation, however, is that ActionDomains are required to be finite and contain discrete, uniquely identifiable Actions. Future versions may allow for infinite and/or continuous domains. These might be classes that generate Actions on-the-fly instead of storing a collection of predefined operations.

Using Actions and ActionDomains to internally represent the space of specific simulation activities allows algorithms implemented in JReLM to learn over arbitrary sets of agent operations in a wide variety of simulation environments. Thus a learner does not need to know any of the specifics of the simulation or even the encapsulating agent. This relieves the client from having to implement the learning algorithm to suit every particular simulation. However, the burden of implementing the domain of action as an ActionDomain class still remains. This is a necessary result of the design for wide applicability. Indeed a learner cannot implement all of the details of using reinforcement learning in a simulation without being customized to that particular simulation. However, JReLM does provide a SimpleActionDomain class to ease this burden for certain types of domains, which will be discussed later.

#### 2.2.1.3 State

Just as the Action and ActionDomain interfaces separate the representation of agent operations and their implementation, the State and StateDomain interfaces perform the same function for the states of the world that an agent inhabits. The State interface is for classes containing information about the simulation environment. That is, a State object contains information regarding the salient features of the environment that are relevant to the learning and action choice processes. Depending on the context, an agent may have full access to all aspects of the world or some aspects may remain hidden. Thus States may represent complete or partial views of an agent's environment. Like Actions, States must be uniquely identifiable and must provide an identification object upon request (via the *getID()* method).

#### 2.2.1.4 StateDomain

The StateDomain interface represents the space of possible world states an agent may encounter. Classes implementing StateDomain manage collections of States and provide the context for agent Actions. StateDomain is similar to ActionDomain in that its implementation and State identifier type is left open to suit the particular simulation contexts. This allows state-based learners to be used in any simulation context where the space of world states can be represented by a class implementing StateDomain. However, StateDomain also has the same limitations as ActionDomain. StateDomain classes must implement a discrete, finite collection of uniquely identifiable States.

### 2.2.2 edu.iastate.jrelm.rl

In the following subsections we cover the main classes in the edu.iastate.rl subpackage. These form the foundation for the reinforcement learning components implementing the specific learning methods. See figure 2.3 for a class diagram of this package. The SimplePolicy, SimpleStatelessPolicy and SimpleStatelessLearner classes are not discussed here, but are included in this diagram for completeness (see sections 3.3.2, 3.3.3, 3.3.1 respectively).

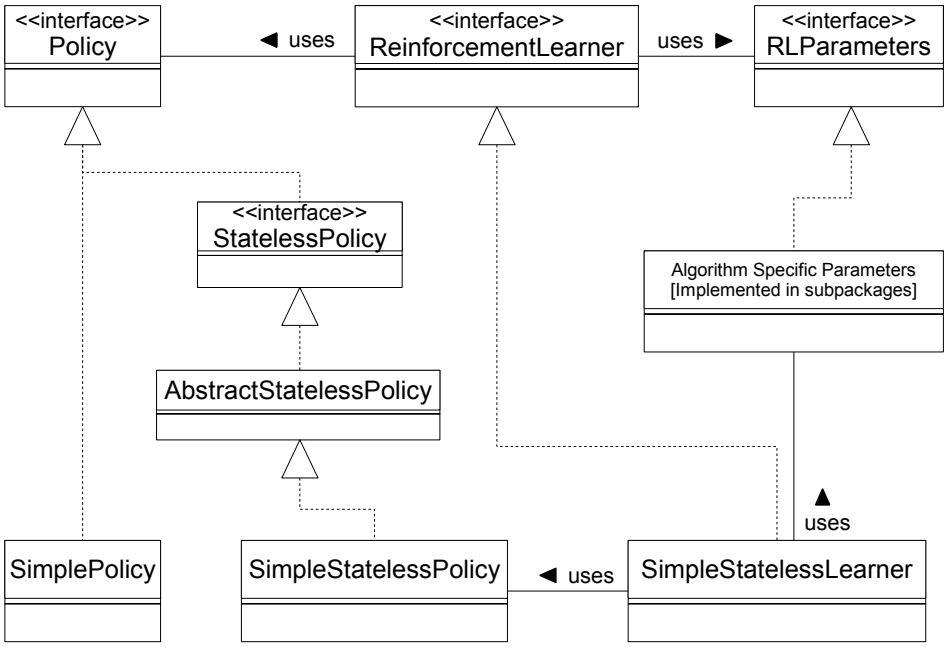


Figure 2.3 Class diagram for edu.iastate.jrelm.rl

#### 2.2.2.1 ReinforcementLearner

The ReinforcementLearner interface is for building reinforcement learning engines. Classes implementing this interface are responsible for driving the learning process of specific algorithms. Such a learning engine will require parameter settings for the specific algorithm and a policy. These are given in the form of RLParameters and RLPolicy, usually when the learner is first constructed.

Though different learning engines will have different functionality, the ReinforcementLearner interface dictates the methods that must be common among them. One such method is *chooseAction()*, which solicits a choice of action from the learner. When called, the learner selects an action from a given ActionDomain according to the current state of its policy. Another method is *update()*, which accepts a reward value and initiates the learning process. Upon calling update, the learner processes the reward and updates its policy according to its specific learning algorithm. The allowable type of reward value may specified by individual learning algorithms. For example, one algorithm may accept a floating point numerical value while another may accept a string.

Additionally, all ReinforcementLearners have methods to access and set the current parameters and policy in use. Parameter settings may be accessed through *getParameters()*, which returns an RLParameters compatible with the specific algorithm. Parameter settings may be changed using *setParameters()*, which accepts RLParameters compatible with the specific algorithm.

#### 2.2.2.2 RLParameters

Most reinforcement learning algorithms will have specific parameter values that must be set before the learning process starts. RLParameters is the interface for classes managing these settings. For every class that adheres to ReinforcementLearner and implements a specific reinforcement learning algorithm, there is a corresponding class that adheres to RLParameters and collects the settings for that algorithm. ReinforcementLearners will usually require an RLParameters object when constructed.

In addition RLParameters is an important component used in the graphical user interface. It plays a central role in communicating changes in parameter settings from the user to a learner. RLParameter objects are used by the BasicSettingsEditor to build custom parameter displays and store input settings. This will be discussed in 3.4.1 where BasicSettingsEditor is introduced.

#### 2.2.2.3 Policy

A central component of most reinforcement learning algorithms is the policy, a mapping from state-action pairs to probability values. JReLM's Policy interface is for building classes that act as policies for learners. During the learning process a learner updates its policy as dictated by the implemented algorithm. In this regard the policy simply stores and retrieves probability values associated with a state-actions pair. However, the Policy also serves as the basis for action selection. A learner may query the policy for a new choice of action with a State from StateDomain representing the current state of the world. The Policy then generates the next action choice. Specifically, when *generateAction()* is called, a Policy selects a new

Action object from the given ActionDomain according to the current action choice probability distribution associated with the desired State. This is required behavior and is expected of all classes implementing this interface.

At the least, a policy requires domains of action and world states. As such most Policies will require an ActionDomain and a StateDomain to be provided during construction.

#### 2.2.2.4 StatelessPolicy

Some reinforcement learning algorithms do not require explicit representation of the of the state of the world. Such algorithms may simply maintain the a single action choice probability distribution for all states. The StatelessPolicy interface is for building this kind of policy. StatelessPolicies simply retrieve and modify probabilities for all Actions in an ActionDomain and do not make use of States or StateDomains. A StatelessPolicy will still require a domain of action however, and implementing classes will likely require an ActionDomain be given during construction.

## 2.3 Flow of JReLM interaction

Figure 2.4 shows the general interaction of the basic JReLM learning components embedded in a Repast agent. Red lines indicate active interaction whereas blue lines indicate passive interaction. Active interaction processes involve making and carrying out decisions. Passive interaction processes involve observation and learning from experience. Here, active processes or components in the agent that are concerned with performing actions are referred to as "Effectors." Passive agent processes or components that are concerned with observation are referred to as "Sensors." Purple lines indicate both passive and active interactions; they occur when both choosing actions and learning from resulting feedback.

In this diagram we can see that the ReinforcementLearner component is the face of JReLM that the agent "sees" and is all that most agents will need to interact with. The typical flow begins when an agent requests a new choice of action from the ReinforcementLearner. The ReinforcementLearner in turn requests a new action from the Policy component. Policy selects

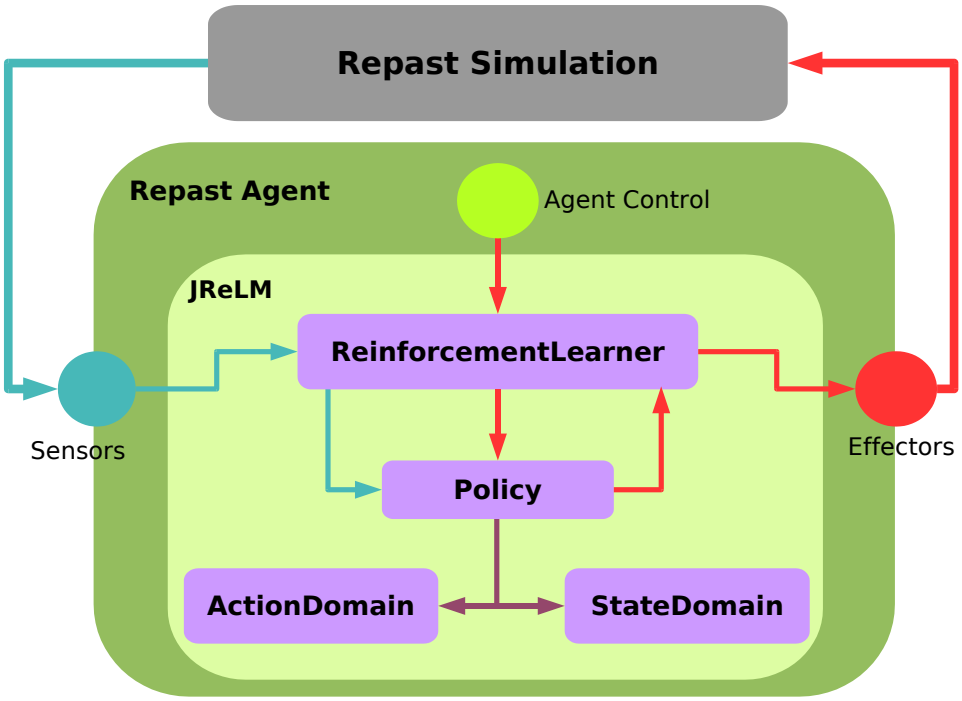


Figure 2.4 JReLM Interaction

a new action from the ActionDomain, possibly considering the current state of the world as related to the StateDomain. The selected choice is passed back to the agent via the ReinforcementLearner and the agent performs the action in the simulation. Now the second part of the cycle begins. The agent receives the results of its actions from the simulation and then passes relevant feedback (or "reward") to the ReinforcementLearner. The ReinforcementLearner updates the policy according to its particular learning method and the Policy adjusts selection likelihood values for all actions or action-state pairs in the ActionDomain and StateDomain.

This completes our coverage of the basics of JReLM. The next chapter will go into the some of the pre-implemented and peripheral structures that provide extra functionality and support the use of the core features. This will include support classes, utility classes, the graphical user interface, and some optional advanced features.

# CHAPTER 3. Pre-implemented supporting structures

Much of JReLM is a structure of interrelated interfaces. However, some pre-implemented support classes are provided to help make the framework easier to use. SimpleAction, SimpleActionDomain, SimpleState, SimpleStateDomain are provided to help quickly create simple action and state spaces. SimpleLearner and SimpleStatelessLearner provide easy access to all pre-implemented algorithms included in JReLM. These classes may be used individually or in conjunction with each other. As a group, they hide much of the underlying complexity of JReLM and provide a way to quickly make use of its features.

## 3.1 edu.iastate.jrelm.core

### 3.1.1 SimpleAction and SimpleActionDomain

The task of building Action and ActionDomain classes may be daunting for the novice and can be time-consuming for the experienced Java programmer. To ease this burden JReLM provides SimpleAction and SimpleActionDomain, implementations of the Action and ActionDomain interfaces. These are convenience classes that take care of the details of managing Actions.

In many simulation contexts the natural representation of agent action choices may be simple. For example, in an N-Armed Bandit game an agent must choose one of n levers, or arms, on a simulated gambler. Each lever yields differing rewards with differing probabilities. The agent must learn which arms are the best to select. In this case an action choice is an integer between 1 and n and the domain is the list of these integers. If one wanted to use a JReLM learner in this situation, it would be helpful to be able to just give it the list of integers as an action domain.

SimpleAction is a simple implementation of the Action interface that functions as a wrapper around an object that represents an action choice for an agent. That is, a SimpleAction stores a given action choice object and provides the functionality required by the Action interface, allowing it to be used with other components of JReLM. So in the N-Armed Bandit simulation, n SimpleActions could be created, each containing a number indicating a lever could be created. This would put the lever selection choice into a format usable by any ActionDomain, ReinforcementLearner, or RLPolicy.

Along the same lines, SimpleActionDomain is an implementation of ActionDomain that accepts an arbitrary collection of objects and manages them as action choices. It allows the client to bypass the inconvenience of implementing an ActionDomain class just to wrap around an existing collection of action choices. In many cases the action domain can easily be represented by a Java Collection (e.g. a Vector or an ArrayList), where elements in the Collection represent action choices. SimpleActionDomain is given such a Collection during construction and builds a ActionDomain around it. Internally each object is wrapped in a SimpleAction since ActionDomain requires that action choices be represented by some type of Action. SimpleActionDomain also contains functionality to manage the action choices and fulfill the other requirements of ActionDomain. In effect this allows the client to use a general Collection as a domain that is compatible with other components of JReLM.

Together, SimpleAction and SimpleActionDomain constitute a bridge between arbitrary objects used as action choices and the rest of JReLM. There are, however, some restrictions limiting SimpleActions and SimpleActionDomains applicability.

Whereas generic Actions and ActionDomains can use any type of Java Object for identifiers, both SimpleAction and SimpleActionDomain may only use integer. That is, SimpleActions in a SimpleActionDomain are identified by unique integer index starting at zero. All objects in the given Collection are wrapped in a SimpleActions and are assigned an index based on the order they are read in. This means that SimpleActionDomain can only be used in simulations where the space of possible actions can be represented by a finite list of unique, discrete actions, each of which can be mapped to a positive integer.

Another restriction is that a multidimensional Collection must be collapsed into a single dimensional form before being given to SimpleActionDomain. For example, say that we have a Vector $\mathbf{y}$ containing n elements, each of which is another Vector $\mathbf{x}_i$, for $1 \leq i \leq n$. Each $\mathbf{x}_i$ contains three Strings, which are the actual action choices. If the original Vector $\mathbf{y}$ is given to SimpleActionDomain, then a SimpleAction object will be wrapped around each $\mathbf{x}_i$. The domain will contain n SimpleActions, each containing a Vector, rather than $3 \times n$ SimpleActions, each containing a String. To set up the domain correctly one could fill another Vector $\mathbf{z}$ with all of the String elements of each $\mathbf{x}_i$ and then construct a SimpleActionDomain with $\mathbf{z}$.

### 3.1.2 SimpleState and SimpleStateDomain

The SimpleSate and SimpleStateDomain classes fulfill the same role for States and StateDomains that SimpleActions and SimpleActionDomains fulfill for Actions and ActionDomains. That is, they simplify the construction of States and StateDomains. SimpleState acts as a wrapper around any Object representing a single state of the world and SimpleStateDomain is built around an arbitrary Collection representing all the possible states of the world. Like Action and ActionDomain, these classes bridge between native Java Objects and Collections and the components of JReLM.

SimpleState and SimpleStateDomain are used in the same way as SimpleAction and SimpleActionDomain. They also suffer from the same limitations. In fact, they are basically the same classes as their Action based cousins, except for a difference in class and method names. The purpose of SimpleState and SimpleStateDomain as separate classes is, given arbitrary Objects, to distinguish between which are states of the world versus which are action choices.

### 3.1.3 JReLMAgent

JReLMAgent is a simple interface for designating an agent that uses JReLM learning components. All that is required of an implementing class is that it provides a Sting identifier and gives access to the ReinforcementLearner it is using. This is primarily used in tracking agents and accessing their learning components through the graphical user interface (see sections 3.1.4

3.4.1).

### 3.1.4 BasicLearnerManager

The BasicLearnerManager is a JReLM component that can be used to manage a group of ReinforcementLearners. It may be used as a tool to track and retrieve all the learners in simulation model. Internally the BasicLearnerManager really manages a collection of JReLMAgents, each registered under its String identifier. Essentially the manager acts as a registrar that maintains a Hashtable of agents keyed on String IDs. For ease of use, however, the client may also register bare ReinforcementLearners (learners without an encapsulating agent). When registered, a bare learner is wrapped in a new WrapperAgent (section 3.2.2), a class implementing a simple JReLMAgent. This way, all the ReinforcementLearners used in a simulation may be tracked whether or not they are used in full agents. However, this means learners are always accessed first by retrieving the encapsulating agent via the agents ID and then getting the learner from the agent.

Clients may initialize the manager with a given list of agents/learners during construction or may register and unregister agents/learners individually after the manager has been constructed. A bare ReinforcementLearner may be registered under a given String identifier. Alternatively, it may be registered anonymously, in which case it will be assigned an auto-generated ID. If a list of learners is given during construction, they will be registered anonymously.

Besides tracking learners, BasicLearnerManager automatically groups agents according to the type of ReinforcementLearner they use. These groups are stored as Vectors of agents and can be retrieved by the Class of ReinforcementLearner. For example, say that a simulation uses two learning algorithms, A and B, and each is instantiated by a class implementing the ReinforcementLearner interface, LearnerA and LearnerB respectively. Within this simulation there are five agents. Agents One, Two and Three use a learner of type LearnerA. Agents Four and Five use a learner of type LearnerB. Once all agents are registered with a manager, the client may give the manager the Class object for LearnerA and receive a Vector containing agents

One, Two and Three. In addition, the manager maintains a list of all ReinforcementLearner types that have been used in forming the groups. This list may be accessed by the client to aid in sifting the different groupings. The grouping feature helps agent organization in simulations where multiple types of learning algorithms are used. It is useful when one wishes to observe or manipulate all agents using the same algorithm.

Finally, the BasicLearnerManager also allows for the direct manipulation of learner settings. A client may give new learning settings to any registered agent through the use of an RLParameters object compatible with that agents learner. Settings may be given to a particular agent, specified by a String ID or to an entire group of agents using the same type of ReinforcementLearner. This feature is useful if one wants to manipulate learning settings without having to retrieve the actual agents.

It should be noted that the BasicLearnerManager does not allow agents to be registered twice, even under different learning methods. If the client wishes to register a single agent that is making use of more than one learning method, the best approach would be to acquire those learners from the agent and register them separately, but each using the same host agent ID for each.

## 3.2 edu.iastate.jrelm.util

### 3.2.1 ModifiedEmpiricalWalker

RLPolicies are responsible for choosing Actions from an ActionDomain according to the current probability distribution. To do this, it helps to make use of a specialized class that can select random events according to a given probability density function (pdf). The Colt Package is a Java library developed by CERN to provide "a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java" [1]. Colt provides a subpackage containing pseudo-random number generators and a variety of probability distributions based on these engines. In particular, JReLM makes use of the EmpiricalWalker class.

EmpiricalWalker is an implementation of an algorithm for a discrete empirical distribution

[1] `http://dsd.lbl.gov/~hoschek/colt/` (current as of November 2005)

originally developed by Alastair J. Walker [19]. It is a port from an implementation in C written by James Theiler, as part of the GNU Scientific Library (GSL) version 0.4.1. EmpiricalWalker is constructed with a pdf given in the form of an array of floating point real values, each representing the probability of an event. When invoked, EmpiricalWalker selects an event according to the pdf and returns the corresponding index in the pdf array. Thus RLPolicy classes can manage changing probability values while using EmpiricalWalker to actually make action selection based on those probabilities. However, we ran into a problem when first testing with EmpiricalWalker.

When EmpiricalWalker is given a uniform pdf of a certain size, it fails to generate events in a uniform manner. It will only select the first and last event and the first is selected the majority of the time. Fortunately, cern is an Open Source library and its source code is easily accessible. After examining the code for EmpiricalWalker, we traced the problem to a floating point error that arises when the pdf values are normalized. The values of the pdf array are summed and then each value is divided by this sum. A small floating point error occurs with each summation. The result is that uniform values add up to a value slightly larger than 1, making each value slightly smaller when it is normalized. Later the values in the pdf are classified as "small" if they are less than the mean, calculated as one over the size of the pdf array, or "big" if they are greater than or equal to the mean. In a uniform distribution each value should be equal to the mean and thus classified as a big value. But since normalization shrunk each value by a small amount, each is less than the mean and is classified as small. This interrupts initialization prematurely. The result is that the last event choice correctly points to the last index in the pdf, but every other event choice points to first index in the pdf. So when EmpiricalWalker is asked for an event, it chooses according to the distribution, but still returns the first event most of the time, since most choices point to the first event index.

ModifiedEmpiricalWalker is a class, included in the edu.iastate.jrelm.util subpackage, specifically built to address this issue. This class extends from Colt's EmpiricalWalker and leaves much of the original functionality intact. However, it overrides methods used in setting the pdf and makes slight changes to allow uniform pdfs to be generated and used correctly. Most of the

code in ModifiedEmpiricalWalker is copied directly from EmpiricalWalker. But the sum of the pdf values is rounded to the nearest integer to compensate for floating point truncation errors accrued during the summation. When the pdf contains uniform values, this summation rounds to exactly one and the values are normalized correctly (they are all divided by one and remain unchanged). Initialization then proceeds correctly and EmpiricalWalker generates events in a uniform manner. though this may not be the best solution to the problem, it suffices for the functionality required for the initial development of JReLM. Future versions of the module will likely include a more elegant solution or the use of an entirely different method of random event generation.

Like EmpiricalWalker, ModifiedEmpiricalWalker use and pseudo-random number generator in generating events according to the pdf. This generator must be supplied to ModifiedEmpiricalWalker upon construction. Specifically it requires a generator of type RandomEngine, the general class of pseudo-random number generators included in the Colt library. Though this further couples the ModifiedEmpiricalWalker to the Colt, this requirement is in place to remain as true to the original EmpiricalWalker as possible. In addition, this allows the client to specify the type of generator used, as long as it is one of the types included in Colt.

### 3.2.2 WrapperAgent

WrapperAgent is simply a "dummy" implementation that satisfies the basic requirements of the JReLMAgent interface. Its only function is to store a ReinforcementLearner and an associated agent identifier. Currently it is used by the BasicLearnerManager (section 3.1.4) to register bare learners to alongside full agents.

## 3.3 edu.iastate.jrelm.rl

### 3.3.1 SimpleStatelessLearner

SimpleStatelessLearner combines all of JReLM's pre-built ReinforcementLearners that implement stateless algorithms. It is intended to provide a way to quickly and easily use such algorithms in simpler simulation environments. When constructed, SimpleStatelessLearner

will accept a variety of RLParameters. The specific type of RLParameters determines which learning algorithm to use. For example, if SimpleStatelessLearner is constructed with VREParameters, it will act as VRELearner. Once it has been set to use a particular learning algorithm, however, it may not be changed. That is, the algorithm chosen by the type of RLParameters provided during construction will be the algorithm used for the life of the SimpleStatelessLearner object. It must also be given a domain of action choices in the form of a Collection of objects or an existing SimpleActionDomain. Internally, SimpleStatelessLearner takes care of building a SimpleDomain, if needed, a compatible RLPolicy, and the chosen type of learning engine. In this way it shortcuts a few of the steps in constructing a full ReinforcementLearner. By accepting a Collection as an action choice domain, it circumvent the hassle of building custom ActionDomain classes for simpler simulation environments, just as in SimpleActionDomain. For some algorithms, a SimpleStatelessLeaner may be constructed with an existing policy. This allows the client to construct learners with knowledge learned from previous simulation runs. In this case the type of RLPolicy given must be compatible with the type algorithm indicated by the RLParameters. SimpleStatelessLearner does have its limitations. Since it uses a given or internally constructed SimpleActionDomain, it can only be used in the same simulation contexts as SimpleActionDomain itself.

### 3.3.2 SimplePolicy

SimplePolicy is a class providing a basic implementation of the Policy interface. Essentially it is a mapping between State-Action pairs and probability values. A separate action choice probability distribution is maintained for each State. Thus the likelihood of choosing any particular Action depends on the current state of the world. SimplePolicy requires at least a domain of action choices and a domain of world states. That is, classes implementing the ActionDomain and StateDomain interfaces must be provided when a StatelessPolicy class is constructed.

SimplePolicy uses ModifiedEmpiricalWalker to select Actions from the ActionDomain according to the probability distribution associated with the given State. It initializes the Modi-

fiedEmpiricalWalker with a pseudo-random number generator of type MersenneTwister. This is a type of generator in the Colt library that implements the Mersenne Twister pseudo-random number generation algorithm [8].

### 3.3.3 SimpleStatelessPolicy

Just as SimplePolicy with Policy, SimpleStatelessPolicy provides an convenience implementation of the StatelessPolicy interface. Like SimplePolicy, it provides a mapping to a probability distribution. However, since it does not make use of world states, the mapping is simply from action choices to selection likelihood values. That is, it maintains a single action choice probability distribution for all states of the world. As such, SimpleStatelessPolicy only requires an ActionDomain upon construction.

Again, ModifiedEmpiricalWalker is used to select Actions from the ActionDomain in accordance with the probability distribution maintained for all states. Like SimplePolicy, the SimpleStatelessPolicy initializes this with a MersenneTwister pseudo-random number generator.

## 3.4 edu.iastate.jrelm.gui

### 3.4.1 BasicSettingsEditor

The BasicSettingsEditor provides an elementary graphical user interface to allow the end user to manipulate the settings for ReinforcementLearners used in a simulation. It is used in conjunction with a BasicLearnerManager (3.1.4), which is given to the editor during construction. When a model using BasicSettingsEditor is loaded into Repast, a separate JReLM settings window appears. The settings for any ReinforcementLearner registered with the given manager will is then available for display and can be accessed through this window. Figure 3.1 illustrates how the BasicSettingsEditor works with the BasicLearnerManager to communicate between the end user and the ReinforcementLearners of a simulation.

The editor is split into three major display sections. See figure 3.2, which shows the window generated by BasicSettingsEditor for the purpose of viewing and editing learning settings for

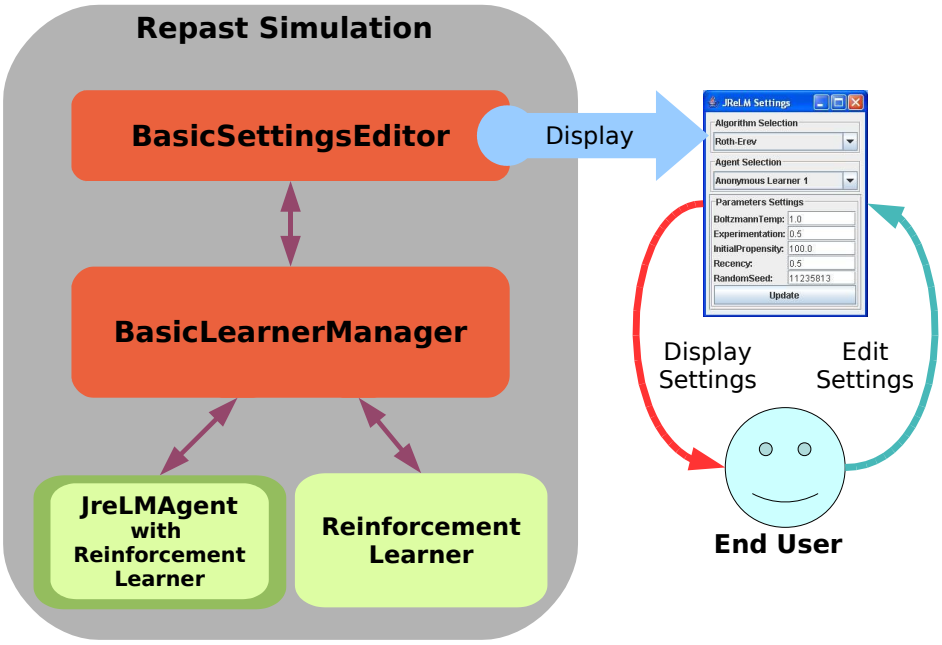


Figure 3.1 Interaction diagram for the JReLM GUI

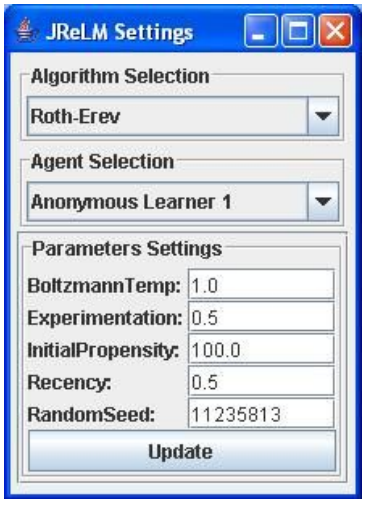


Figure 3.2 JReLM learning settings window (generated by BasicSettingsEditor)

agents with JReLM ReinforcementLearners. At the top is the "Learning Method Selection", which displays a drop-down list of all the JReLM learning methods being used in the simulation. More precisely, it has a list of all the learning methods that have been registered with the BasicLearnerManager this editor is linked to. Selecting a learning method sets the learning context and alters what the other sections will display. Only one method may be selected at a time.

The second section, in the middle of the window, is "Agent Selection". This displays a drop-down listing of all registered agents that are using the selected learning method. The listing of agents displayed here will change as different learning methods are selected.

Finally, displayed at the bottom of the window, the third section "Parameter Settings" shows the parameter settings fields for the selected learning method and agent. This is where the user may manipulate parameters that will affect the agent's learning. To change settings, the user enters the desired value into the parameter fields and then clicks on the "Update" button. This button signals the editor to send the current values in the parameter fields to the manager which then updates changes in the selected agent. The available settings displayed in the parameter settings panel will change as the different learning methods are selected and specific values for these settings will change as different agents are selected (only if different agents have different settings).

BasicSettingsEditor also includes a feature to disable the parameters settings section from within a Repast model, preventing the user from the parameter fields. This feature, accessed through the *setEditorEnabled()* method, will disable just the parameter settings fields without disabling the entire window. This will allow the user to browse the settings of different agents and still lock out changes while a simulation is running.

One challenge presented in developing even a simple interface stems from Repast's open agent design policy. Since the client is left free to define agents to suit particular simulation contexts, there is no common interface dictating how to access or communicate with the components of agents used in Repast. Thus, the problem is how to access JReLM ReinforcementLearners embedded in arbitrary types of agents. Since we do not know what kind of

agents will use JReLM, how can we get to the JReLM components within these agents?

Two mechanisms have been included in JReLM to circumvent this problem. First, the primary function is to mark agents that use JReLM learning components and to provide a method of accessing those components. Since JReLMAgent is just an interface, it only specifies additional functionality that needs to be in place for an agent to be compatible with a BasicLearnerManager and BasicSettingsEditor. An agent can implement the JReLMAgent while extending from other classes and implementing other interfaces. Thus JReLMAgent does not interfere with Repasts open design policy. The client is still free to design agents to fit a particular simulation context; he only needs to provide minimal additional functionality to be used with the manager and editor. The second mechanism is the flexibility of registration in the BasicLearnerManager. Recall that ReinforcementLearners may be registered in raw form. Learners that are embedded in client designed agents may simply be registered separately from the actual agent if the client does not wish to implement the JReLMAgent interface. This also allows manipulating ReinforcementLearners that are not incorporated into agents at all.

Another challenge to the interface is that a simulation model may contain a variety of different learning algorithms. Individual agents or groups of agents may be outfitted with different ReinforcementLearners, especially in situations where learning methods are being compared. An interface must be able to display all the different learners being used and to indicate which agents are using which learners. In addition, the interface must be able to dynamically display the specific parameter settings for each different type of learner. BasicSettingsEditor works in conjunction with BasicLearnerManager to handle the multiplicity of learner types. BasicLearnerManager already provides convenient functionality to compile a list of all the types of registered ReinforcementLearners as well as lists of agents using a particular type. BasicSettingsEditor simply displays these lists, allowing the user to select the desired algorithm and then choose an agent from the corresponding group.

The editor employs part of Repast's GUI infrastructure to display specific learning settings for the selected agent. IntrospectPanel is a graphical component used by Repast to build custom simulation controls for a client's model. IntrospectPanel is always built around a given

Object. During construction, it inspects this Object and builds graphical widgets around the Object's properties. That is, IntrospectPanel builds graphical elements that are connected to certain fields in the Object that may be accessed through specially named methods. These methods are "accessor methods" that start with the word "get" or "set". A matching pair of accessor methods share the same property name. For example, *getSize()* and *setSize()* could be accessor methods for a property called size. IntrospectPanel will create widgets either for desired properties as specified by a list given with the Object or for all properties that can be inferred by examining all of the Object's methods. All of these property widgets will be displayed, resulting in a customized interface that allows the user to directly edit the properties of the given Object. Further, by passing IntrospectPanel an Object that implements Repast's DescriptorContainer, the client can specify the type of widget to be used with each property. A text field can be used for one property, while a slider can be used for another. When a user edits the value in a text field or moves the slider, it directly modifies the value of the field in the Object via the corresponding methods. The BasicSettingsEditor can use IntrospectPanel to quickly build interfaces customized to particular learning algorithms simply by building a new IntrospectPanel around an RLParameters Object. RLParameters already all the parameter fields for a particular algorithm and so is a natural choice as a base for a customized interface. All that is required is that the RLParameter contain the properly named accessor methods for each parameter field that should be accessible to the user. It should be noted, however, that changes that are made through an IntrospectPanel are immediate. Any mistaken values are set right away.

In some cases it may be useful to delay the writing of new parameter values in the BasicSettingsEditor, especially since the parameter settings panel may be changing as the user selects from multiple algorithms and agents. To help with this, JReLM provides DelayedIntrospectPanel. This class is simply an extension of IntrospectPanel with a modification to delay passing new settings to the given Object until the user presses an Update button. To build custom learning parameter displays, then, BasicSettingsEditor constructs a DelayedIntrospectPanel around an agent's RLParameters. The new learning settings are not given to an

agent until the user confirms the settings.

We have seen how the BasicSettingsEditor can automatically give the user access to all registered agents by coordinating with a BasicLearnerManager. We have also seen how the editor is able generate custom displays of specific learning settings on-the-fly using RLParameters and DelayedIntrospectPanel. This shows how BasicSettingsEditor may be used in a variety of simulation contexts with arbitrary numbers and types of agents and learning algorithms.

Though the BasicLearnerManager and BasicSettingsEditor combine to produce a user interface that is sufficient to address the challenges discussed earlier, there is much potential for further development.

## 3.5 edu.iastate.jrelm.similarity

This package provides interfaces and classes for implementing measures of similarity (or dissimilarity) among Actions. Some algorithms may use comparisons of action choices in the learning process (e.g. distributing partial rewards). The purpose of this package is to allow algorithms that use similarity to be decoupled from specific domains. The two major interfaces, SimilarityMeasure and DissimilarityMeasure, are provided to guide classes that formalize and encapsulate ideas of similarity. These can be used to implement custom measures that define similarity among action choices in specific domains. Thus learning methods that make use of SimilarityMeasures can be used across a variety of domains, as long as the proper domain specific implementations are provided. Currently, ARELearner (section 4.5.0.6) is the only learner in JReLM that makes use of this package.

## 3.6 edu.iastate.jrelm.spillover

Some learning methods make use of the notion of the partial distribution of rewards among action choices, also known as spillover. Spillover techniques usually give small pieces of the reward received to action choices that are similar to the last action chosen. The idea here is that actions that are similar to the last choice selected will probably produce similar results and thus should be partly reinforced as well. As such, similarity measures are often used in

distributing rewards. The edu.iastate.jrelm.spillover package provides structures to implement different spillover methods and it is expected that many implemented spillover techniques will use SimilarityMeasures (or DissimilarityMeasure) that are customized for particular simulation contexts.

The main component is this package is SpilloverWeightGenerator, which is an abstract class that provides a base for implementing spillover methods that can be used with other JReLM components. This mainly provides a guideline on what JReLM components will expect from a spillover technique. Specifically, such techniques should provide a *setReferenceAction()* method to set the Action against which other Actions in the same domain can be compared. In addition, an implemented spillover technique should provide a *generateWeight()* method that accepts an action and return a real value indicating the degree of similarity.

## 3.7 edu.iastate.jrelm.demo and the N-Armed Bandit Demo

The demo package (edu.iastate.jrelm.demo) contains pre-implemented demonstrations of the use of JReLM components. The purpose is to provide simple examples that will help the client learn how to use JReLM in more complex projects. The top level of this package includes components that may be reused in multiple demonstrations. Subpackages will include demonstration programs which can be implementations of commonly know "toy problems" or fully implemented Repast models. Currently the only component included at the top level is the RothErevAgent and the only demonstration subpackage is the N-armed bandit model.

### 3.7.1 RothErevAgent

As discussed in section 3.1.3, JReLMAgent is an interface used to build agents that can be tracked and managed with certain JReLM components. The RothErevAgent class, included in edu.iastate.jrelm.demo package, provides an example of an agent that makes use of the Roth-Erev algorithm (section 4.1) to drive its behavior. Specifically, it makes use of a RELearner (section 4.3.0.1) to chose actions and learn from experience.

RothErevAgent is designed to be a simple generic agent that can be used in variety of

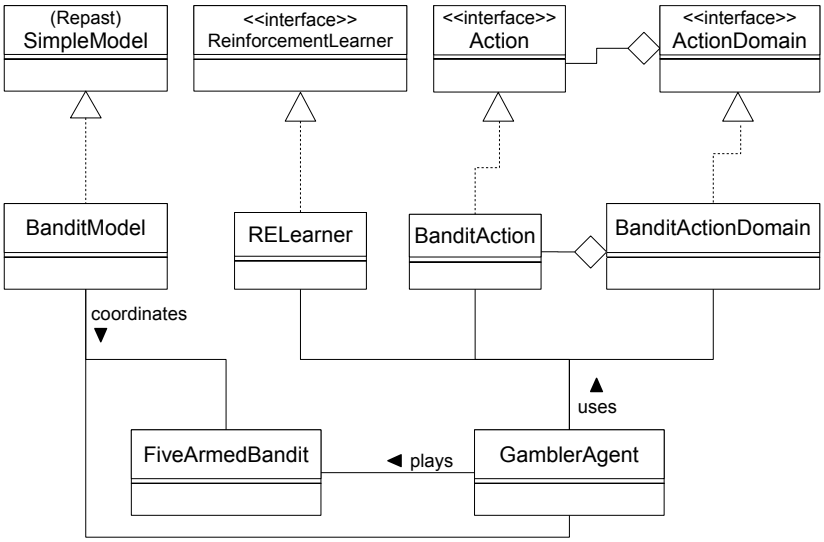


Figure 3.3 Class diagram for edu.iastate.jrelm.demo.bandit

contexts. When constructed, RothErevAgent must be given an ActionDomain and a set of Roth-Erev learning parameters (section 4.3.0.3) and optionally an agent identifier in the form of a String. The *act()* method is called to stimulate the agent into action and simply asks the agent's learner to select an Action from the domain. This Action is given to the agent which then passes it to the simulation model. Here the Action is acting as a message from the agent to the model stating what the agent does in the next period. The model then uses the agent's *receiveFeedback()* method to tell the agent the result of its action and this result is passed directly to the learner. RothErevAgent is an example of a passive agent that requires stimulation from a simulation model to operate. One could easily build a more complex and proactive agent using RothErevAgent as a starting point.

### 3.7.2 edu.iastate.jrelm.demo.bandit

This subpackage contains an implementation of the classic N-Armed Bandit toy problem (figure 3.3 for the class diagram of this subpackage). N-Armed Bandit is game in which a learner plays against a device modeled after a slot machine gambling game (aka a one-armed bandit), except this device has n arms to select from. When pulled each arm will yield varying reward payouts with varying probabilities and some arms yield a better distribution of payouts

than others. The task of the learner is to learn, through repeated plays, which arms are the best to pull (which yield the best expected payout). For a more in depth overview of the N-Armed Bandit, see Sutton and Barto [14], chapter 2.1.

JReLM implements a variation of the N-Armed Bandit game, specifically 5-armed bandit, in six classes under the edu.iastate.jrelm.bandit subpackage. Though gambling terms have been carried over from the original metaphor, in this example it does costs the player nothing to play and so this is not an actual gambling game. The BanditModel class is a Repast Model that coordinates the game between the gambler (the player agent) and the bandit (the payout device). BanditModel is run as a Repast simulation. FiveArmedBandit is the payout device with five arm options for the player. Each arm can yield three different payout values and has a probability distribution governing the likelihood of each value. ModifiedEmpiricalWalker is used to choose payouts according to their distributions. The player in this game is the GamblerAgent which is an agent using an RELearner (section 4.3.0.1) to learn the best arm selections. This class must be initialized with the appropriate learning parameters in the form of an REParameters object (section 4.3.0.3) and is also a passive agent, like RothErevAgent. GamblerAgent provides an example of how an agent can use an ReinforcementLearner to drive its behavior.

GamblerAgent makes use of BanditActionDomain which is a class implementing JReLM's ActionDomain interface. GamblerActionDomain contains five GamblerActions, classes implementing the Action interface, representing all of the possible actions the GamblerAgent can perform (pulling one of the five arms). These two classes provide an example of how a custom domain of action can be implemented for a specific simulation context using the Action and ActionDomain interfaces

## 3.8 Comments on comments

Like most programming languages, Java allows the programmer to insert notes for clarification or extra information into the code. In addition to these standard comments, the *Javadoc* tool uses specially formatted comments to generate HTML documentation of Java packages.

These are called Application Programmers Interface (API) specifications and they provide an easy to use references to guide client programmers in the use of a Java packages.

The code within JReLM has been extensively commented in a effort to make it easy for the client programmer to understand, use and build upon. This includes both standard and Javadoc comments explaining the classes, the interfaces, their use and notes about implementation details (i.e. the learning algorithms, use of random number generators, etc). The JReLM API specification will be made available for distribution with the module.

# CHAPTER 4. Pre-implemented reinforcement learning algorithms

As we have seen thus far, JReLM includes infrastructure to support the construction and use of reinforcement learning algorithms. However, JReLM also provides pre-implemented learning algorithms. These serve two purposes. First, they demonstrate how the module may be used to implement desired reinforcement learning algorithms. They show how the module can be used as a base to ease the development of such learning components as well as to leverage supporting structures in JReLM and Repast. Second, they act as a library of ready-made algorithms that can simply be dropped into Repast simulations. This eases the burden on novice programmers and allows the client to circumvent extra implementation.

## 4.1 The Roth-Erev algorithm

To begin with, we will examine several variations of the Roth and Erev reinforcement learning algorithm and the corresponding JReLM implementations, RELearner and ARELearner. Alvin Roth and Ido Erev originally developed [11] and later refined [2] their algorithm to model how humans perform in repeated games against multiple strategic players. A variation of this algorithm was developed by Nicolaisen et. al. [10] in response to a problem found by Koesrindartoto [5] regarding the updating choice probabilities when an agent receives zero valued rewards. It should be noted that Roth and Erev originally presented the algorithm with equations illustrating calculation among a group of agents. For clarity, the equations below represent calculation from the perspective of a single agent. The reader should keep in mind that multiple agent learning via the Roth and Erev algorithm perform the same calculations, but perform them on an individual basis.

The basic Roth-Erev learning algorithm does not follow the traditional form of a rein-

forcement learning algorithm, as defined by Sutton and Barto [14], in that it is a stateless algorithm. That is, it does not look at states of the environment the learner is situated in, but instead bases learning solely on the last action chosen and the reward received for that action. In addition there is no explicitly defined policy or value function, though a probability distribution over action choice is maintained.

The basic Roth-Erev learning algorithm operates by associating *propensity* values with all possible action choices in the domain. These propensities are translated into a *probability* distribution that governs future action selection. Let $q_j(t)$ be the propensity for action choice j at time t, where j is one of n actions in the domain and t is a count of passing learning cycles. Then the probability that j is selected at time t is

$$p_j(t) = \frac{q_i(t)}{\sum_{m=1}^{n} q_m(t)} \tag{4.1}$$

During initialization, before the first learning cycle is entered, all action choice propensities are initialized with the same *initial propensity* value, $q_{init}$. As such, the initial probability distribution will be uniform.

The core of the basic Roth-Erev learning algorithm is the *reinforcement function*, which is responsible for updating action choice propensities. As an agent chooses actions, propensities are updated using received feedback (reward). When an agent performs action choice k at time t and receives reward $r_k(t)$, the propensity for action choice j at time t+1 is updated according to

$$q_j = \begin{cases} q_j(t) + r_k(t) & \text{if } j = k \\ q_j(t) & \text{if } j \neq k \end{cases} \tag{4.2}$$

Thus the propensity for the action k is reinforced positively or negatively depending on the resulting reward. Propensities for all other actions remain unchanged.

In [2] Roth and Erev presented a revised version of this algorithm with an alternative reinforcement function containing two new parameters

$$q_j(t+1) = [1 - \phi]q_j(t) + E_j(\epsilon, j, k, t) \tag{4.3}$$

Here $\phi$ is called the *recency parameter*. It determines to what degree the recent past has greater bearing on present action choice than the distant past. That is, it determines how quickly past rewards for actions fade from memory. It does this by decaying older propensity values each update period (*recency* is sometimes referred to as the *forgetting* parameter). As $\phi$ increases, the new propensity for action j, $q_j(t+1)$, gets less of its previous value, $q_j(t)$. The propensity value built up from previous rewards has a weaker influence on the new value. The decaying propensity value is supplemented by an experience value given by an *experience function* defined as

$$E_j(\epsilon, j, k, t) = \begin{cases} r_k(t)[1-\epsilon] & \text{if } j = k \\ r_k(t)\frac{\epsilon}{2} & \text{if } j = k \pm 1 \end{cases} \tag{4.4}$$

The experience function contains an *experimentation* parameter, $\epsilon$. This affects the how likely an agent is to choose actions similar to actions that have chosen in the past. Here part of the reward is distributed to the two actions most similar to the last action chosen. Assuming the action choices are ordered according to similarity, the two most similar actions are simply differ by one index value in the list of choices. This is a form of reward spillover that is meant to help strike a balance between exploration and exploitation.

Alternatively, if the action choices are not ordered according to similarity, the experience function takes the following form

$$E_j(\epsilon, j, k, t) = \begin{cases} r_k(t)[1-\epsilon] & \text{if } j = k \\ r_k(t)\frac{\epsilon}{N-1} & \text{if } j \neq k \end{cases} \tag{4.5}$$

where N is the total number of action choices available to the learner. Here the reward is partially distributed to all action choices and helps to encourage experimentation over the whole domain.

To summarize the basic Roth-Erev reinforcement learning algorithm operates as follows:

1. Initialize all action propensities to the initial propensity value $q_{init}$. Initialize the action choice probability distribution to a uniform distribution.

2. Generate choice probabilities for all actions using current propensities.

3. Choose an action k according to the current choice probability distribution.

4. Update propensities for all actions using the reward for the last chosen action.

5. Repeat from step 2.

## 4.2 Variation of Roth-Erev

Roth's and Erev's algorithm does, however, suffer from a problem that can hinder the learning process in contexts where the agent may receive zero valued rewards. When an agent performs an action and receives a reward value of zero, both experience functions 4.4 and 4.5 result in no changes in the action choice probabilities. Both versions of the experience function zero out, and all propensities are all equally reduced by the recency parameter. Thus, all action choices retain the same relative probability. This can slow the leaning process since the agent will need to choose at least one more action to change the choice probabilities. That is, each time the agent receives a zero valued reward, at least one more step is added to the overall learning process. In some contexts it may be desirable that an agent learn to avoid action choices that yield no reward as well as negative rewards. Koesrindartoto[5](2001) originally identified this problem and demonstrated its potential adverse affect in double-action experiments involving Roth-Erev learners. In experiments running 1000 auction rounds, certain parameter settings resulted persistently high market inefficiency. Auction participants would frequently make price offers that would fail to get them matched in trade and would thus yield zero profit. The choice probabilities for these offers would remain sufficiently high to promote their continued selection until the later trading rounds.

To address the problem of updating with zero-valued rewards, Nicolaisen et al.[10](2001) developed the following modification of equation 4.5:

$$E_j(\epsilon, k, t) = \begin{cases} r_k(t)[1-\epsilon] & \text{if } j = k \\ q_j(t)\frac{\epsilon}{N-1} & \text{if } j \neq k \end{cases} \tag{4.6}$$

Here unselected choices receive a portion of their old propensity values rather than a portion of the reward. Note that when $r_k(t) = 0$, equation 4.6 yields an experience value for the

selected action that is lower than the values for the unselected actions. The propensities for all choices will still undergo the forgetting effect, being degraded by recency in 4.2. However, the propensities for unselected actions will not degrade as much as for the selected action. Thus the new relative choice probability for the selected action will decrease more than the choice probabilities for non-chosen actions, encouraging the agent to select other actions in the future.

## 4.3 Implementation of the Roth-Erev algorithm

In this section and the following sections we examine the JReLM components implementing the family of Roth-Erev learning methods. Figure 4.1 shows the class diagram for this collection of algorithms.

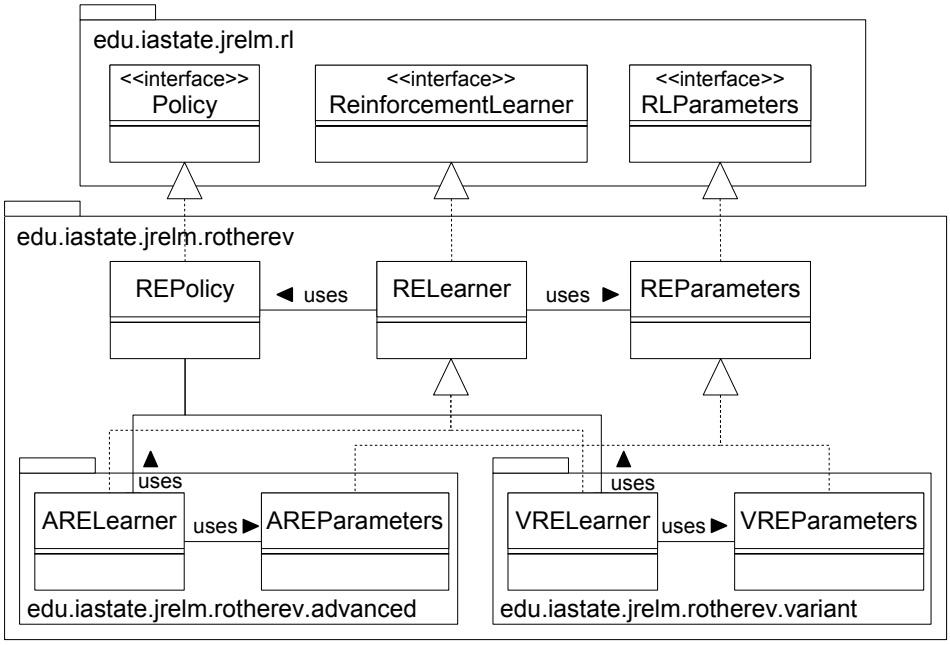


Figure 4.1 Class diagram for edu.iastate.jrelm.rotherev and subpackages

#### 4.3.0.1 RELearner

RELearner is the central component that implements the Roth-Erev reinforcement learning algorithm. Specifically, it implements the version presented in [2]. As a Java Class, it fulfills the ReinforcementLearner interface seen in 2.2.2.1 and is compatible with most other

JReLM components. RELearner's *update()* method accepts reward of type Double, an object representation of an 8-byte floating point number, allowing it to learn from both positive and negative feedback values. When *update()* is called, RELearner proceeds to update action choice propensities and probabilities according to the algorithm.

The algorithm originally described by Roth and Erev learned over a domain of strategies that guided an agent during upcoming an activity period or series of periods. In the context of the JReLM, learning occurs over Actions in an ActionDomain. As discussed in 2.2.1.1, Actions can be used to represent strategies or specific, individual activities. RELearner is compatible with any type of ActionDomain and Action, regardless of the underlying implementation details, so it may be used with a domain of strategies or agent operations.

RELearner does not factor similarity among action choices into learning. As such, it uses equation 4.5 for the generating experience values. This was chosen to allow wider applicability of the learner, since most domains will presumably be unordered.

The implementation of RELearner differs from standard Roth-Erev learning in one important aspect. Rather than generating probabilities from relative propensity values, as in equation 4.1, RELearner uses a Gibbs-Boltzmann distribution:

$$p_j(t) = \frac{e^{q_j(t)/T}}{\sum_{i=1}^{n} e^{q_i(t)/T}} \tag{4.7}$$

Here T is a temperature parameter that affects the "flatness" of the probability distribution. That is, it can be used magnify or dampen features, peaks and valleys, in the distribution. Note that as $T \to \infty$, $p_j(t) \to \frac{1}{n}$ for all j. The distribution approaches uniformity as T takes on higher values. As peaks are lowered and valleys are raised, differences in propensity have less and less bearing on the likelihood that a particular action will be chosen. However, features of the distribution become more distinct for lower values of T and are eventually accentuated as $T \to 0$, magnifying differences in propensity. Setting $T = 1$ prevents propensities values from being scaled in either direction and essentially reproduces the relative distribution. By default, T is set to 1 to mimic the behavior of the standard Roth-Erev algorithm.

The main purpose for using the Gibbs-Boltzmann distribution is to robustly handle negative

reward values. In some simulation contexts an agent may encounter large or repeated negative rewards. For example, an agent trying to sell a product in a simulated market may have continuing production costs whether it makes a sale or not. This cost may be communicated to the learner as a negative reward when few or no sales are made. After repeated periods of poor sales, it is possible that the propensity values for bad choices may become negative, resulting in negative probability values if a relative distribution is used. This problem can be alleviated somewhat by carefully choosing an appropriately high initial propensity value. However, it may be difficult to determine appropriate values in some contexts. Using the Gibbs-Boltzmann distribution ensures that action choice probabilities will remain well defined even if negative propensities are reached.

At this time, RELearner does not make use of a temperature schedule (or cooling function) as in simulated annealing techniques. However, this is a prime candidate for future expansion of the Advance Roth-Erev learner (see 4.5.0.6).

The boltzmann temperature is included in the set of learning parameters and may be set through the REParameters class discussed in section 4.3.0.3.

#### 4.3.0.2 REPolicy

JReLM's implementation of the Roth-Erev algorithm differs from the original in that it makes use of an explicitly defined policy. Specifically, RELearner works in conjunction with an REPolicy object, which is a customized SimpleStatelessPolicy (see 3.3.3) that maintains both action choice propensities and probabilities. Not only does REPolicy act as a repository for both action choice propensity values, it also selects new action choices according to the probability distribution, just as in SimplePolicy. Since Roth-Erev is a stateless algorithm, REPolicy is a stateless policy and probability values are simply maintained for each action choice rather than for state-action pairs.

Like SimplePolicy, REPolicy uses ModifiedEmpiricalWalker to select Actions according to the current choice probability distribution. A given seed value may be used to seed the MersenneTwister pseudo-random number generator within ModifiedEmpiricalWalker. This

allows the results of a simulation run to be replicated. Specifying the same seed value from one run to another can be used produce the same sequence of action choices from the policy and thus the same agent behavior.

RELearner may also be given an existing policy upon construction. This allows learned knowledge to be carried over between simulation runs, or to be shared among agents.

#### 4.3.0.3 REParameters

REParameters maintains and validates parameter settings for the Roth-Erev algorithm implemented by RELearner. This includes *experimentation*, *initial propensity*, *recency* and an optional seed value for the pseudo random number generator used in REPolicy. This class abides by the RLParameters interface and is designed to encapsulate parameter settings. Its primary function is to allow RELearner and other JReLM components to safely use, modify, and transfer parameter settings.

All initialization and modification of parameter settings must proceed through a REParameters object. An RELearner must be given an initial REParameters object when constructed. Once the initial parameter settings are in place, the client may modify them in two ways. First, the client may acquire this REParameters object from the learner and change the settings through that object. Or the client may construct a new REParameters object with the desired settings and give the learner.

One advantage REParameters provides is that groups of RELearners can all be initialized with the same settings simply by giving the same REParameters object to each learner. This is especially useful in simulations where it is important to start with a group homogeneous learners.

As discussed earlier in section 3.4.1, the BasicSettingsEditor uses an RLParameters objects to build displays for the end user and to communicate parameter settings to individual learners. In this regard, REParameters acts as a communications vessel between the RELearner and the graphical user interface. REParameters implements Repast's DescriptorContainer interface which allows it to explicitly declare which parameter settings are available to the end user and

how they should be displayed.

For a typical use the Roth-Erev algorithm in an actual agent, a client must first provide the collection of action choices in the form of an ActionDomain. The client then creates a new REPolicy, giving the policy the ActionDomain during construction. Next a new REParameters is created with the desired initial settings. Finally, a new VRELearner is constructed with the REPolicy and VREParameters. An agent may then solicit action choices from and learn though the learner.

## 4.4 Implementation of Variant Roth-Erev

JReLM implements the Variant Roth-Erev algorithm discussed in 4.2in two classes extending from the bases classes implementing Roth-Erev learning.

#### 4.4.0.4 VRELearner

VRELearner extends the RELearner, but uses the learning method of the Variation of Roth-Erev algorithm. Specifically, it overrides the experience function in RELearner and uses equation 4.6 in updating action choice propensities. Besides this modification, VRELearner operates and is used in the same way as RELearner.

However, in our work implementing and testing Variation of Roth-Erev algorithm [3], we have found a quirk involving the selection of initial propensity values that can be a potential pitfall or useful tool affecting the early behavior of the learner. Initial propensities that are high relative to the potential reward values can alter the assessment of action choices, at least in the initial stages. To see the issue, let's observe the modified experience function again.

$$E_j(\epsilon, k, t) = \begin{cases} r_k(t)[1-\epsilon] & \text{if } j = k \\ q_j(t)\frac{\epsilon}{N-1} & \text{if } j \neq k \end{cases} \tag{4.8}$$

Recall that in the first cycle of learning, all choices are initialized with the same initial propensity $q_{init}$ and when the first selected choice k is performed, it will receive reward $r_k(0)$. If $r_k(0) < q_{init}\frac{\epsilon}{(N-1)(1-\epsilon)}$, choice k will be updated with a lesser experience value in comparison to all other choices and thus it will have a lesser propensity value at the end of period 1. When

the probabilities are updated from the new propensities, k will have a lower likelihood of being chosen than all other choices. The result is that choice k is initially dampened, even if it results in a positive reward value, because of over over-inflated inittial expectations.

It appears that over time, the effect of decay induced by recency, along positive rewards, will diminish this dampening effect. That is, after enough experience, the learner will adjust it's assessment of action choices away from initial overestimations. The number of learning cycles needed for this depends on several factors. These include the magnitude of difference between $q_{init}$ and the potential positive rewards as well as the values of the recency and experimentation parameters. In addition, the sequence of choice selections will affect how quickly the dampening effect fades for any particular action choice.

When using the Variant Roth-Erev algorithm, the client or the user should be remain mindful of this dampening effect and carefully choose a value for $q_{init}$. Choosing an over-inflated initial propensity can inadvertently slow the learning process, initially derailing the learner from potentially beneficially action choices. However, this dampening effect can also be useful tool to purposefully tweak the learning process. A carefully selected $q_{init}$ can encourage initial exploration among action choices, above that induced by the experimentation parameter $e$. While $e$ remains constant, the dampening effect dissipates as experience is gained. This means the effect can be used to encourage extra experimentation at the beginning of the learning process while gradually returning to the base experimentation over time. In this way, it may be possible to use an exaggerated initial propensity to prevent a learner from being caught by local minima in the reward space during the initial stage of learning for certain contexts.

Though further discussion of this issue is beyond the scope of this work, it is a fertile topic for future investigation.

#### 4.4.0.5 VREParameters

The VREParameters plays the same role for the VRELearner that REParameters plays for the RELearner. It encapsulates the parameters required for VRELearner to operate, which

are, in fact, the same required for RELearner. Though it extends REParameters, it is actually an empty class since VRELearner does not require extra parameter settings than its parent. The purpose for VREParameters, then, is to designate when VRELearner should be used by other components. For example, SimpleStatelessLearner (see 3.3.1) encapsulates all included stateless algorithms in JReLM. SimpleStatelessLearner uses the learning method indicated by the type of RLParameters it receives upon construction. For SimpleStatelessLearner to act as a VRELearner, it must receive a VREParameters object.

## 4.5 Advanced Roth-Erev Learning

#### 4.5.0.6 ARELearner

ARELearner implements an "Advanced" version of the Roth-Erev learning method. Specifically, it includes features that allow client programmers to make their own custom version of the Roth-Erev learning method. It is meant to facilitate the investigation of unexplored modifications to the base learning method. This learner extends the RELearner, but implements the experience function using interchangeable components to allow for custom methods of reward distribution to be specified upon construction or even switched during runtime.

In ARELearner, the experience function (equation 4.4) used in RELearner is replaced with

$$E_j(k,t) = r_k(t)W(j,k) \tag{4.9}$$

Here W(j, k) is a weight function governing how much of the reward value action j receives based on how similar it is to the last chosen action k. W(j, k) is not a fixed function in the sense that it is not pre-defined in ARELearner. The weight function is actually defined by a SpilloverWeightGenerator (section 3.6) object that is given to the learner. This allows ARELearner to operate without having to know the details of how weights are generated, meaning the learner can be used with a variety of different weighting methods.

As discussed earlier, SpilloverWeightGenerator makes use of a SimilarityMeasure or DissimilarityMeasure (section 3.5)to compare Actions. Since the notion of similarity is necessarily closely tied to each specific simulation context, to take full advantage of ARELearner a client

will usually need to build a custom ActionDomain, SimilarityMeasure and SpilloverMethod for the particular simulation. Such customization is usually reserved for more experienced programmers, hence the "advanced" rating of this learner. However, ARELearner does default to standard Roth-Erev learning using experience equation 4.5. Thus it will act as an RELearner if no other spillover method is specified.

#### 4.5.0.7 AREParameters

AREParameters encapsulates the parameters settings for the ARELearner. This class extends from REParameters and manages settings for the same boltzmann temperature, experimentation, recency, and random seed settings. It addition it manages the collection of known spillover methods and tells the ARELearner which method to use in learning.

Internally AREParameters maintains a list of spillover methods that may be used with ARELearner. By default, it will contain all of JReLM's pre-implemented spillover methods that are compatible with Roth-Erev learning. As new spillover methods are added to the standard JReLM distribution, new SpilloverWeightGenerators will be added to AREParameters as applicable.

To add client defined spillover methods, and thus specify a custom experience function, the client first creates a new SpilloverWeightGenerator class and then gives an instance of this class to an instance of AREParameters. This can be done through *addSpilloverMethod()*, which will adds a single SpilloverWeightGenerator to the list. Alternatively, the client may specify the list of available spillover methods through *setSpilloverMethodList()*, which accepts Java List containing a collection of SpilloverWeightGenerator that may be used. Once the list of spillover options is complete, *setActiveSpilloverMethod()* is used to set which SpilloverWeightGenerator is used. The AREParameters object is then given to an ARELearner object, which uses the parameters to define the experience weight function and guide the learning process.

The user can also take advantage of ARELearner's advanced option via the graphical user interface. Like REParameters and VREParameters, AREParameters can be used with the BasicSettingsEditor (section 3.4.1) to display settings for the Roth-Erev parameters. In

addition, AREParameters leverages Repast's DescriptorContainer to display a drop-down list of the available spillover methods. The user may select a desired method and then click the "Update" button to set the corresponding SpilloverWeightGenerator to be used by the selected learner. See figure 4.2 for an example display of AREParameters with various spillover options.

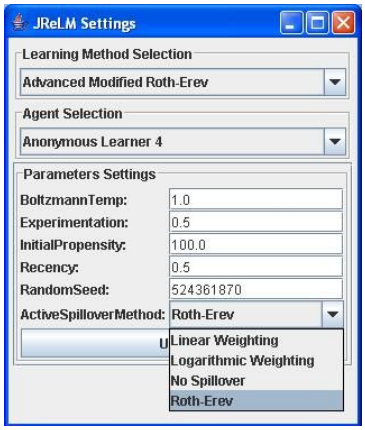


Figure 4.2 JReLM learning settings window displaying Advanced Roth-Erev learning settings

Custom spillover methods may contain parameter settings themselves and it may be desirable to make these settings available to the user along with the other learning parameters. AREParameters is intended, with further development to read, edit and display settings for available spillover methods in future versions of JReLM.

# CHAPTER 5. Illustrative Application

This chapter explores the use of JReLM in an agent-based economics simulation called the Raita Economy Model. This simulation is built upon Repast and is the first application of JReLM in full social science model.

## 5.1 The Raita Economy Model

The Raita Economy is an agent based simulation developed by Somani and Tesfatsion [13](2005). It is a Repast model designed to examine *market concentration* in relation to *market power* in a dynamic, single product economy. *Market concentration* is the degree to which the majority of market activity is performed by a minority of the participants. *Market power* is the degree to which a participant may profitably influence prices away from competitive levels. Measures of market concentration are often used as indicators of market power. The idea here is that the participants that are most active in a concentrated market are more likely to try to inhibit competition in their own interests. The Raita Economy is meant to examine the degree to which three common measures of market concentration can be used to predict the emergence of market power over time in a simplified dynamic production economy.

The context of this simulation is a market containing consumer and firm agents buying and selling a single product, raita (an Indian dish consisting of yogurt with cut-up vegetables or fruit) . While consumers are simpler reactive agents, firms are strategic agents that compete with one another for consumers' purchases. Consumer agents seek to maximize their utility by purchasing raita at the lowest price they can find. That is, they will always act to get the most raita for their buck. Firm agents seek to maximize their profits and may act by adjusting their price and production levels. In addition, a firm may purchase or sell extra production

capacity to bolster profitability or offset losses. This is called capacity investment. Thus firms have a richer set of action choices with which to compete with each other and possibly manipulate market conditions. In addition to this rich action set, the firms are able to learn from experience, allowing them to base present action choice on the results of past action choices. The processes of action choice selection and learning are driven by reinforcement learning components provided by JReLM.

Before going into the use JReLM in this context, we will first give an overview of dynamic market activities in the Raita Economy. Commerce proceeds through a series of trading periods. During the first trading period each consumer is given a lifetime endowment and shares holdings in each of the firms. At the beginning of each following trading period, each consumer receives income based on it's remaining endowment, share holdings, and any unspent income from the previous round. It then performs a multi-stage price discovery process to attempt to trade with a firm in the interest of maximizing its raita consumption. A firm may sell out of raita (stock out) before meeting the demand of all consumers waiting to trade with it. Thus a consumer may not be able to trade with the first firm that it chooses. If this happens, the consumer will try to trade with another firm offering the same price or, if there is no such firm, be forced to trade with a firm offering a higher price. In this way, a consumer will successively try to trade the current lowest priced firms available until it satisfies its demand or until the trading period ends. The price discovery process continues until all firms are sold out or until all consumers have met their demands. At this point the trading period ends.

The firms follow their own course during each trading period. In the initial period each firm begins with a nonnegative sum of money and a positive production capacity, the maximum it may produce during any one period. At the start of each trading period a firm submits a supply offer to the market. The supply offer consists of a choice of how much raita to produce and at what unit price to offer it at. The consumers then browse the supply offers during the price discovery process and match up in trade.

Every consumer has a minimum amount of raita it must acquire each period, its subsistence needs. If a consumer fails to meet its subsistence needs for raita, it "dies" and is removed from

the economy permanently. Similarly, if a firm's net worth drops to zero or below at any point, that firm is removed from the economy and may no longer participate in the market. If the number of firms or consumers drops to zero at any point, the simulation stops.

## 5.2 Implementation of the Raita Economy

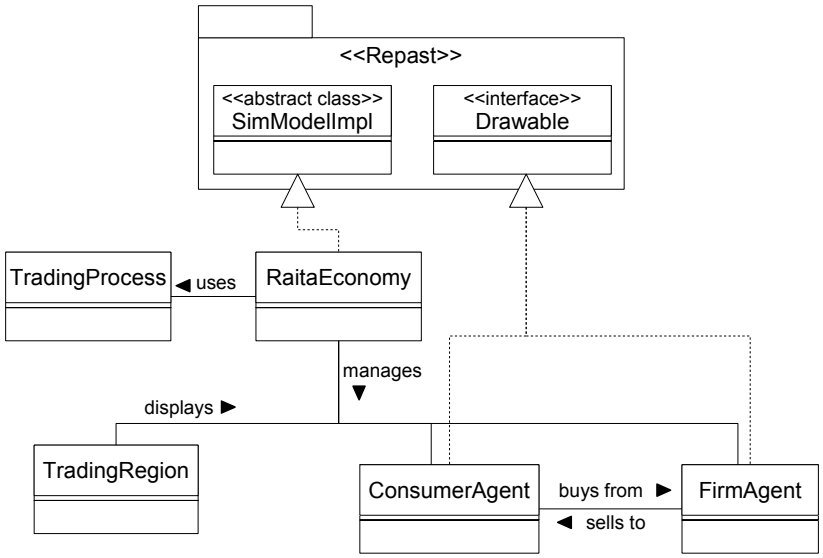


Figure 5.1 Class diagram for the Raita Economy simulation

Figure 5.1 shows a class diagram of the major classes of the Raita Economy model. RaitaEconomy is the actual model class that extends Repast's SimModelImpl class, a convenience class for building models. RaitaEconomy sets up and runs the simulation. TradeProcess encapsulates the trading processes and drives the market each trading period. The ConsumerAgent and FirmAgent classes implement the market participants. ConsumerAgents carry out simple utility maximizing behavior, while FirmAgents make use of JReLM components to for adaptive learning behavior.

## 5.3 JReLM in the Raita Economy

This section describes the specifics of how JReLM is used in the RaitaEconomy model. Refer to figure 5.2 for a diagram showing the use of JReLM components in the FirmAgent.

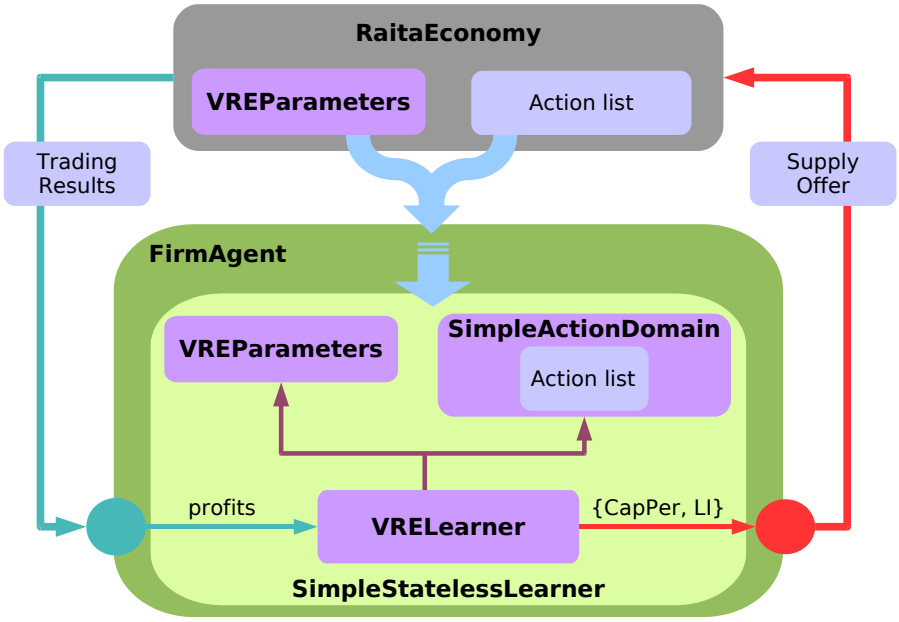


Figure 5.2 Interaction between the RaitaEconomy model and JReLM components in FirmAgent

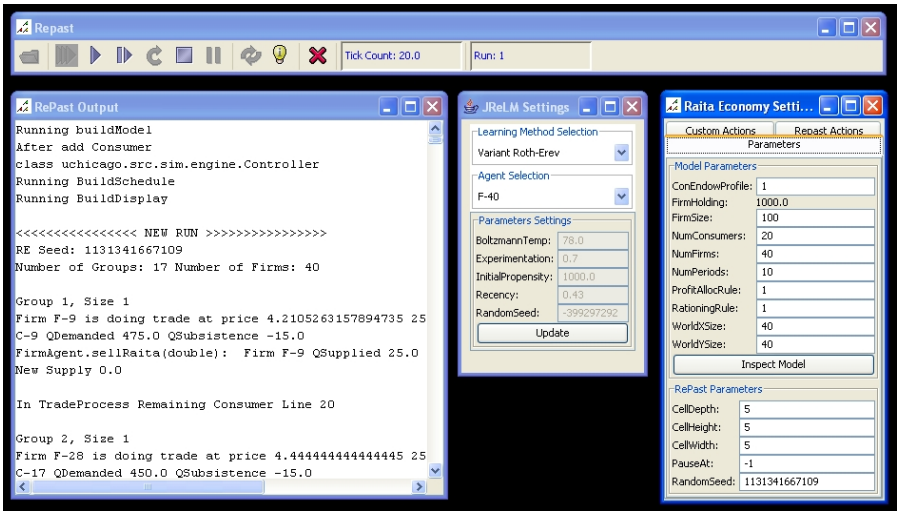


Figure 5.3 Screen capture of the RaitaEconomy Repast model using JReLM

Figure 5.3 shows Repast running the RaitaEconomy model with the JReLM settings window displayed.

Each FirmAgent contains a SimpleStatelessLearner that drives its action choices and performs learning based on the profits received from each trading period. The SimpleStatelessLearner within each agent is keyed to act as a VRELearner, so the firms learn using the variant of the Roth-Erev algorithm discussed earlier (section 4.2). All firms initially receive the same set of learning parameters, though these may be changed on an individual basis by the end user through the JReLM settings window (section 5.3).

In addition, each firm receives the same action domain. In the RaitaEconomy model, the action domain for FirmAgents is the space of possible supply offers which can be which can be sent to the market. This space can be expressed as

$$S = \{s_1, s_2, \ldots, s_i, \ldots, s_n\}$$

where S is the supply offer space of size n and $s_i$ is the ith supply offer. Each supply offer contains a production quantity r and a unit price p and takes the form $s_i = \{r, p\}$.

For a FirmAgent to make use of a JReLM learner for action selection, the set of possible firm actions must be formulated as a JReLM ActionDomain. To facilitate this, FirmAgents use an internal representation of a supply offers as an action choices that differs from the actual offers sent to market.

Instead of directly specifying a production quantity, an action choice contains a percentage of the agent's production capacity. If $Cap_n(t)$ is the maximum production capacity for firm agent n in trading period t, then the percentage of production capacity for desired production amount r, is $CapPer(r,t) = \frac{r}{Cap_n(t)}$. Firm agents always have a non-negative production capacity and since $0 \leq r \leq Cap_n(t)$, $CapPer(r,t)$ will always be a value between 0 and 1. Also, instead of containing an actual unit price, each action choice contains a "Lerner Index" taking the form

$$LI(r,p) = \frac{p - MC(r)_n}{p} \tag{5.1}$$

where $MC_n(r)$ is the marginal cost of producing quantity r for agent n and $p \geq MC_n$. LI is used to reconstruct a price once an action choice is selected and will always be set to a value between 0 and 1.

Thus, for each supply offer $s_i$ in the supply offer space S, we have a corresponding action choice $a_i$ in the action domain A.

$$s_i = \{r, p\} \rightarrow a_i = \{CapPer(r,t), LI(r,p)\} \quad \text{where } s_i \in S \text{ , } a_i \in A \text{ for } i = 1, \ldots, n \tag{5.2}$$

Since both the percentage capacity and the Lerner Index can take on real values, S and A are potentially continuous. However, the RaitaEconomy model uses a discretized version of A and by extension a discretized version of S. This is a two-dimensional collection of [x] percentage capacities and [y] LI values associated with each one. The fineness (or granularity) of the domain is determined by these x and y values. Currently, x and y are "hard-coded" and may not be changed by the user. Functionality for automatic domain construction will need to be added before customizable granularity can be supported.

For simplicity, the RaitaEconomy model maintains this action domain A as a simple listing of action choices (in the form of a Java ArrayList). This form was chosen because it is easy for the RaitaEconomy to maintain and it can easily be fed into a SimpleActionDomain for use with JReLM components.

The RaitaEconomy model assembles this list of action choices and passes it to each FirmAgent upon construction. A FirmAgent gives the list to its SimpleStatelessLearner which manages it as a collection of SimpleActions in a SimpleActionDomain. These are invisible to the FirmAgent. At the agent's request, the learner selects and a SimpleAction from the domain according to its current policy. It then unwraps the $\{CapPer(r,t), LI(r,p)\}$ pair from the SimpleAction and returns it as an action choice to the agent [1]. The agent then translates the action choice into an actual supply offer consisting of a production quantity and price.

The advantage to using action choices with percentage capacity and Lerner Index values is that they do not depend on an agent's specific production capacities and marginal costs.

[1] The FirmAgent specifically requests the action choice in unwrapped form via the SimpleLearner's *chooseActionRaw()* method. Using the standard chooseAction() method, specified from the ReinforcementLearner interface, would yield the action choice in the form of a SimpleAction.

That is, two different FirmAgents can use the same $\{CapPer(r,t), LI(r,p)\}$ to generate production level and prices values appropriate to their individual maximum production capacity and marginal cost. This means all agents can use the same action domain and yet individually transform the actions to cater to local production and cost constraints.

It should be noted that the purchase or sale of extra production capacity is not included in the definition of an action choice. This means that production capacity investments are not driven by learning. Instead such investments are automatic, based on the profit and production level during the last trading period. This does, however, affect the FirmAgent's learning problem, since adjustments in production capacity change the underlying domain for an agent. The same action choice will translate to different supply offers in the market after a change in production capacity. This means FirmAgent's must learn in a shifting domain.

### 5.3.1 Pseudo-random number generator seeding

To simulate independent actors in the market, each firm agent must be given a unique seed value for the underlying pseudo-random number generator being used for action choice selection. Recall that SimpleStatelessPolicy, and thus REPolicy (sections 3.3.3 and 4.3.0.2 respectively), uses a MersenneTwister pseudo-random number generator in selecting action choices. Given the same seed value, a pseudo-random number generator will always produce the same pseudo-random sequence of numbers. This means that, given the same seed value and sequence of rewards, each firm's learner will select the same sequence of action choices. Since the consumers are purely deterministic (they always try to buy at the lowest price), if all firms start with the same seed value they will all behave exactly the same.

Thus it is important that each firm receive a unique seed value for its learner. Although separate seeds can be set for each firm manually through the REParameters object or through the graphical user interface, this can be a tedious process. Currently, the Raita Economy model uses a simple method to automatically assign unique random seeds to each firm's learner.

Repast itself includes a pseudo-random number generator (actually an encapsulation the colt library generators) for use in client models. In addition, it provides a "RandomSeed"

parameter that can be set through the graphical user interface and be accessed by client models to initialize generators. A default seed value is provided when Repast is run and is itself initially generated pseudo-randomly, changing each time Repast is started. If a client model uses the RandomSeed seed parameter, the default value will be used unless the end user specifies another value before the model is loaded.

RaitaEconomy acquires this RandomSeed parameter value from Repast and uses it to seed an intermediary pseudo-random number generator [2]. It then uses this generator to generate individual seed values for the firm learners. Using this method, the learners will receive a varied distribution of seeds, while the results of a simulation run can be replicated by setting a single specific seed value through the Repast interface.

## 5.4 Further development of the Raita Economy

This concludes our discussion of the RaitaEconomy model and its use of JReLM. Currently, the model is still under development. Once the model is completed and JReLM itself is further tested and refined, it is our intention to proceed with experimental runs in pursuit of the model's objective. This will include a stage of learning parameter tuning to determine the appropriate settings to allow FirmAgents to learn self-interested behavior in the raita market context.

[2]The intermediary generator is standard type included in the Java language (specifically java.util.Random)

# CHAPTER 6. Conclusion and future work

As agent-based methods become more widespread in the field of computational social science simulation, it is important that supporting technologies be developed and made available to researchers. The Recursive Porous Agent Simulation Toolkit, or Repast, has become an acclaimed tool for fulfilling this need. One Repast's greatest strengths, its open agent architecture, can also be a weakness, since the burden of agent design and construction is left to the client programmer. When sophisticated agent behavior is required, such as the use of machine learning, it can be difficult and time consuming for the client to implement such behavior.

The preceding chapters have presented JReLM, the Java Reinforcement Learning Module for RepastJ, which has been developed specifically for supporting the use of reinforcement learning in computational, agent-based, social science simulations. JReLM accomplishes this goal in four major ways. First, JReLM provides a Java infrastructure for the implementation and use of reinforcement algorithms in RepastJ in simulations. Second, it provides a selection pre-implemented algorithms and supporting structures to allow learning methods to simply be "plugged into" simulation agents. Third, it provides tools for managing reinforcement learning methods and allowing the end user to control them through the Repast interface. Finally, every effort has been made to abide by object orient design principles and provide documentation of the module to promote extensibility and aid conceptual understanding.

This work has presented the initial stage of JReLM which will act as the core platform for future development. The following sections discuss some of the open directions future development as well as some of the projects that JReLM will be used.

## 6.1 Future development

Of course, one of the primary goals in the further development of JReLM will be the inclusion of additional pre-implemented reinforcement learning methods and supporting structures. This will include the further investigation and implementation of reinforcement learning algorithms appropriate for use in multi-agent simulations. The continuing goal will be to make JReLM useful in a wider variety of simulation contexts as well as give the client and user more options for convenient adaptive agent behavior. In addition, to the expansion of reinforcement learning services, there are a number of other near term tasks related to the further development of JReLM as a platform.

### 6.1.1 Testing and validation

Currently JReLM is still in the testing phase, specifically unit testing, which involves checking the operation of JReLM components for desired functionality. A suite of JUnit [1] test cases have been built and are being expanded to formalize and expedite this process. The tests are built from the bottom up in the same fashion as JReLM, focusing first on individual core components and then the more complex, aggregate components. This way, the unit tests can also be used check that components interact properly and produce the desired collective behavior.

In addition to unit testing, JReLM will need to undergo an initial validation. Most importantly this will involve checking the operation of RELearner and it's derivatives to make sure they are producing the correct behavior specified by the Roth-Erev learning algorithms. This can not be accomplished through unit tests alone since learning occurs in relation to an environment. That is, what constitutes correct learning behavior is tied to the specific environment that a learner is placed in. As such, a complete Repast simulation model will be needed to test JReLM Roth-Erev learning.

Initial validation is proceeding in two stages. First, the N-Armed Bandit Demo (see 3.7.2)

[1] JUnit is a Java package for building and running unit tests of Java classes. It is available at `http://www.junit.org` (current as of November 2005)

will be used as a validation platform for a single RELearner. The Demo is a suitable starting point since it is a very simple model and the environment is completely predictable. This means the specifications for correct learning can easily be calculated and compared to the actual behavior of the RELearner. Eventually, a version of the Bandit Demo may be combined with a framework of JUnit tests to form a general validation platform to be used on any algorithm included in JReLM. The second stage will be to observe learning behavior in a full multi-agent context. The Raita Economy simulation will be instrumental in this effort. It will allow us to observe multiple VRELearners interacting and yet still provide a simple enough context where a single agent's behavior can be traced. In addition, it will be the first research model using JReLM. This will give us a chance to see what sorts of issues may arise in actually using JReLM as an supplementary tool for experimental investigation.

### 6.1.2 Core functionality

One of the first items of further development of the JReLM core will be to find or build a reliable random event generator. As discussed in 3.2.1 the EmpiricalWalker in the colt library suffers from a flaw in handling uniform probability distributions that prevent it from being suitable for JReLM's needs. The ModifiedEmpiricalWalker was developed as work-around, however, it is based on the EmpiricalWalker implementation which itself is a port from an implementation in C. It would be desirable to either make use of another third-party library that provides a direct implementation of a random event generator or to write such an implementation specifically for JReLM.

Another desired addition to the core functionality is the ability for JReLM to read and write relevant information to and from a text file. Relevant information includes learning parameter settings, ActionDomain and StateDomain specifications, and snapshots of policies. The three main motivations for this are to allow for automated batch simulation runs, replicating simulation runs and reconstituting simulation runs.

Repast currently has the capacity to run batches of simulation runs with a given model and parameter settings. Having JReLM be able to read in settings to set up collections of learners

over a series of simulation runs would help it work with Repast in batch mode.

Allowing JReLM to read settings from a file would also assist in replicating simulation experiments without the user needing to hand enter the desired settings.

Finally, allowing JReLM to write settings to file would allow simulations runs to be stopped and started again even after Repast has been stopped and restarted. By writing policy data to a file, along with the associated parameters and domains, a learner can recover all learned "knowledge" from the time a previous run was stopped. This can assist in recovering from interrupted simulation runs or simply add flexibility in how and when simulations runs are made.

Reading and writing of all relevant information will likely be longer term goal, as it will be a time intensive undertaking. Two major decisions will need to be addressed first. First, a protocol for how, when and what data to read and write will need to be specified. Second, a specific form for data will need to be selected (e.g. XML). Finally, the shear volume of data may eventually pose a problem. As a first step, using text files for reading and writing just learning parameters may be sufficient. However, even a small collection of agents using modest sized action and state domains can generate a tremendous amount of data if their policies are recorded. It will be impractical, if not impossible, to use text files for large numbers of agents. The use of a database may be required to manage all relevant data. This is, of course, not anywhere near a short term goal for the development of JReLM.

### 6.1.3 Graphical user interface

Currently, JReLM's graphical user interface is displayed in a separate window that appears when a model using JReLM is loaded into Repast. While this is sufficient, it is desirable to have JReLM setting be integrated directly into Repast's control panel. Besides providing for a cleaner overall interface, this would emphasize to the user that JReLM is an tool integrated into Repast rather than feature provided by the current model being loaded.

Another addition to be added to the user interface is the use of "Info" buttons. These can appear next to the learning method selection panel and the parameter settings fields. When

clicked, an Info button will display information about the selected algorithm or associated parameter. This may include a brief explanation of the algorithm or how a parameter value is used and perhaps recommended settings. This information might either be displayed in a separate window that appears or perhaps in a separate panel within the JReLM settings window itself. The intention is to assist the user in learning how to use a particular learning method. This will add some burden to the implementation of new algorithms, as the required information will need to be written in the algorithm's RLParameters. However, this feature can greatly improve usability for the end user.

Finally, a major graphical user interface feature to be added to JReLM is the ability to visualize learning processes and related data. Some examples include graphing probability distributions of policies, mapping a history of selected actions, showing changes in learning parameter settings, and displaying similarity among actions in a domain. The intention is to help users gain insight into the macro-level patterns manifested in a simulation by providing a micro-level view of the internal workings of learning agents.

## 6.2 Agent-based Modeling of Electricity Systems

This section introduces the Agent-based Modeling of Electricity Systems (AMES) model and covers its use of JReLM for agent learning.

In 2003 the Federal Energy Regulatory Commission (FERC) released a proposal for a new Wholesale Power Market Platform, referred to as the WPMP, to be adopted by all wholesale power markets in the U.S [18]. This platform outlines a system using day-ahead and real-time markets, along with ancillary services, to coordinate the wholesale of power between generators and Load Serving Entities (LSEs), which be an electric utility, transmitting utility or Federal power marketing agency. The day-ahead market allows for LSEs to enter into contracts to buy electricity from generators to meet the next day's demand. The real-time market allows LSEs to buy electricity on the spot to service gaps in demand not covered by contracts from the previous day or resulting from spikes in demand. The WPMP also includes the use of locational marginal pricing and tradable financial transmission rights. Locational Marginal

Pricing (LMP) is the practice of setting prices according to the geographic location where electricity is injected into or withdrawn from the grid. The purpose of LMP is to accurately communicate the costs of transmission and congestion so as to encourage the efficient citing of new power generation and transmission facilities. All this is managed by an Independent System Operator (ISO) or Regional Transmission Organization (RTO).

AMES is a computational, agent-based simulation being developed by Koesrindartoto, Sun, and Tesfatsion [6] to test the economic reliability of the WPMP. Built upon Repast, this model provides representations of the transmission grid, the day-head and real-time markets and agents for the LSEs, generators and an ISO. A complex set of trading rules and transmission grid properties govern a how power is bought, sold and transmitted in a manner matching the actual WPMP and real-world physical grid constraints. In addition, AMES will use strategic learning agents that will adapt their behavior to changing market conditions to simulated the actions of real market participants. JReLM will provide the learning services for these agents.

In the initial stage, AMES will model a five-bus transmission grid with five generators and two LSEs with the intention to scale up to larger systems in the future. To begin with, the behavior of LSE and ISO agents will be rule-based, using self-interested purchasing strategies and enforcing the system operation guidelines respectively. For the generators, however, AMES will make use of JReLM to provide adaptive learning behavior. Specifically, the generators will use the Variation of Roth-Erev algorithm via the VRELearner. In the context of this simulation, the task of the learner will be to guide the generators in choosing supply offers, which consist of pricing schedules for varying levels of generation.

The Variation of Roth-Erev learning was chosen for use in AMES, in part, because of its success in previous work studying simulated electricity markets. Nicolaisen, Petrov, and Tesfatsion [10] examined market power and efficiency in an agent-based, simulated market with discriminatory, double-auction pricing. In this simulation, both buyer and seller agents used variant Roth-Er'ev learning to participate in the market. This study found that the market attained high efficiency and more over that it's microstructure was strongly predictive of the relative market power of buyers versus sellers.

AMES itself is being developed as part of a larger initiative. Decision Models for Bulk Energy Transportation Networks is a project funded by the National Science Foundation focused on studying integrated energy networks (including electricity, gas, coal, water). This is an interdisciplinary effort lead by primary investigator J. McCalley (from Iowa State University's Department of Computer and Electrical Engineering), and co-primary investigators S. Ryan (Industrial and Manufacturing Systems Engineering), S. Sapp (Sociology), and L. Tesfatsion (Economics and Mathematics). Most efforts of this sort have focused on narrow aspects of each network. However, this project aims to take a more holistic approach, providing empirically grounded models of the physical distribution and economic networks and the interrelations between them. The National Electric Energy System (NEES) model will focus on the structural ties between electricity, gas, coal and water systems, while the Agent Based NEES Market Model will overlay a market network model on top of the NEES. The models will be interconnected such that conditions in one will influence the other.

AMES will be used to inform the larger modeling effort. In regards to learning agents, as JReLM is expanded, AMES will be used to examine how robust the WPMP is to different methods of learning. That is, it will be used to look at how strongly market protocols influence market outcomes versus participant behavior driven by various learning methods. If the resulting dynamics of the market remain the stable over different types of learning, then the market structure is sound, independent of the particular behavior of the participants. Examination of market structure robustness in AMES will help to inform to what degree the larger NEES Market Model will need consider detailed, individual adaptive behavior.

## 6.3 Sandia National Labs N-ABLE

The National Infrastructure Simulation and Analysis Center (NISAC) is a program funded by the United States Department of Homeland Security, Information Analysis and Infrastructure Protection Directorate. NISAC is a joint effort between Los Alamos and Sandia National Laboratories to study issues regarding United States critical infrastructure. Specifically, NISAC provides modeling, simulation, and analysis services to examine the interdependen-

cies between national infrastructure systems and the ramifications of their disruption. Within NISAC, the Computational Economics Group (CEG) at Sandia National Labs focuses on the economic issues of critical infrastructure using in-depth analysis and a variety of computational tools.

In addition to using a selection of third-party tools, one of the the CEG's main efforts lies in developing its own sophisticated, agent-based, microeconomic, simulation platform. The NISAC Agent-Based Laboratory for Economics (N-ABLE)[12] is a tool for building and analyzing detailed models of the economic interdependencies and dynamics between infrastructure systems. N-ABLE is used to support broader NISAC efforts through the analysis of models informed by real-world national economic and infrastructure data.

Agent behavior is an issue of central importance for N-ABLE as an agent-based simulation platform. One of the goals in N-ABLE's further development is to provide agents with greater adaptive and cognitively inspired behavior. The work that has gone into JReLM is also contributing to this effort. JReLM's implementation of the family of Roth-Er'ev learning algorithms has been used to inform the initial expansion N-ABLE's adaptive agent behavior capabilities. In addition, the architecture and interaction structures for JReLM are providing an initial starting point in the design of a larger adaptive behavior sub-platform within N-ABLE. It is hoped that JReLM will continue to be of assistance as the offerings for agent behavior within JReLM continue to evolve.

## 6.4 Distribution with Repast

As an open source project, Repast flourishes from the contributions of programmers willing devote time and energy to its development. The intention driving behind JReLM is to provide a platform for reinforcement learning in social science simulation and in doing so supplement Repast as a tool. In the course of its development, the possibility of distributing JReLM with Repast itself has arisen. That is, JReLM would be included as a subpackage in the Repast distribution archive. This would make JReLM more convenient for client programmers and assist in its longer term integration with other components of Repast.

One of the conditions JReLM must meet for inclusion in Repast is that it must provided a demonstration model showing how it may be used. This was the primary purpose of including the N-Armed Bandit model. Also, open source software is typically released under a license dictating the terms of use, modification, extension and distribution. Though the exact terms of release for JReLM have not yet been decided, it will most likely be released under the same or some similar license as Repast.

There are still a few loose ends to attend to and further testing to perform before JReLM can be released for general use. However, it is hoped that an initial release version will be ready soon and included in Repast's next release.

## Bibliography

[1] Collier, N., Howe, T., and North, M. Onward and upward: The transition to repast 2.0. In *Proceedings of the First Annual North American Association for Computational Social and Organizational Science Conference* (Pittsburgh, PA USA, June 2003), Electronic Proceedings.

[2] Er'ev, I., and Roth, A. Predicting how people play games: Reinforcement learning in experimental games with unique, mixed strategy equilibrea. *American Economic Review 88*, 4 (1998), 848–881.

[3] Gieseler, C. Introduction to the modified roth-erev reinforcement learning algorithm and its c++ implementation in modrotherev. Working Paper, Sandia Report, Sandia National Labs, Albuquerque NM, 2005.

[4] Gilbert, N., and Bankes, S. Platforms and methods for agent-based modeling. In *Proceedings of the National Academy of Sciences of the USA* (Washington, DC, USA, May 2002), vol. 99, National Academy of Sciences of the USA, pp. 7197–7198.

[5] Koesrindartoto, D. P. Discrete double auctions with artificial adaptive agents: A case studyofan electricity market usinga double auction simulator. Master's thesis, Economics Dept., Iowa State University, 2001.

[6] Koesrindartoto, D. P., Sun, J., and Tesfatsion, L. An agent-based computational laboratory for testing the economic reliability of wholesale power market designs. In *Proceedings, IEEE Power Engineering Society Conference* (June 2005), vol. 1, pp. 931–936.

[7] Marrone, P. Java object oriented neural engine: The complete guide, all you need to know about joone, 2005. Available at http://www.jooneworld.com (current as of November 2005).

[8] Matsumoto, M., and Nishimura, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation 8*, 1 (January 1998), 3–30.

[9] Minar, N., Burkhart, R., Langton, C., and Askenazi, M. The swarm simulation system, a toolkit for building multi-agent simulations, 1996. For Swarm resources and community, see the SwarmWiki at http://swarm.org (current as of November 2005).

[10] Nicolaisen, J., Petrov, V., and Tesfatsion, L. Market power and efficiency in a computational electricity market with discriminatory double-auction pricing. *IEEE Transactions on Evolutionary Computing 5*, 5 (October 2001), 504–523.

[11] Roth, A., and Er'ev, I. Learning in extensive-form games: Experimental data and simple dynamic models in the intermediate term. *Games and Economic Behavior, Special Issue: Nobel Symposium 8*, 1 (1995), 164–212.

[12] Schoenwald, D. A., Barton, D. C., and Ehlen, M. A. An agent-based simulation laboratory for economics and infrasctructure interdependency. Computation, Computers, Information and Mathematics Center, Sandia National Laboratories, P.O. Box 5800, Albuquerque, New Mexico 87185-0318. Conference Paper, 2004 American Control Conference, June 2004.

[13] Somani, A., and Tesfatsion, L. Concentration and market power in a dynamic production economy: An agent-based computational study. Working Paper, Deparment of Economics, Iowa State University, Ames IA, 2005.

[14] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. MIT Press, 1998.

[15] Tesfatsion, L. Repast: A software toolkit for agent-based social science modeling: Self-study guide for java-based repast (repastj), 2005. Maintained online at http://www.econ.iastate.edu/tesfatsi/repastsg.htm (current as of November 2005).

[16] Tesfatsion, L., and Judd, K. L., Eds. *Handbook of Computational Economics: Volume 2, Agent-Based Computational Economics*. North-Holland (Handbooks in Economics Series), Amsterdam, Spring 2006, to appear. Contributors, contents and chapter abstracts available at http://www.econ.iastate.edu/tesfatsi/hbace.htm (current as of November 2005).

[17] Tobias, R., and Hofmann, C. Evaluation of free java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation 8* (2004). Available at http://ideas.repec.org/a/jas/jasssj/2003-45-2.html.

[18] Notice of white paper, April, 28th 2003. United States Federal Energy Regulatory Commission.

[19] Walker, A. J. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software 3*, 3 (September 1977), 253–256.

## ACKNOWLEDGEMENTS

I would like to thank my committee, Professor Vasant Honavar, Professor Leigh Tesfatsion, and Professor Dimitris Margaritis for their patience and advice. In particular I would like to thank Professor Tesfatsion for her contagious enthusiasm, keen insight, and refreshing openness. Without her encouragement and support, none of this would have even begun, let alone have seen the light of day.

To everyone in the Lunch Crew and the Statistics Contingency, thanks for your warm camaraderie, the endless entertainment, and your relentless promotion of me as "The Hippy from California."

I would also like to thank all the coffee shops that provided sanctuary and office space and the intrepid baristas whose tireless efforts fueled the thesis-forge fires.

Finally, I would like to thank Scott Hellon for inspiring me to view the world from different perspectives and for being a living example of how to persist in the pursuit of one's dreams.