Chapter 1

Introduction

Phoenix is a distributed agent-based simulation platform designed for development of agent-based systems and simulations. We use agent based models and simulations in our study of complex systems, micro-simulations and bottom-up approaches to engineering.

This document explains the motivations, design, architecture of the new agent based simulation framework. It will also explain the motivations behind some of the design decisions so that we provide the right hooks for people looking to extend or modify the platform for themselves.

The platform is released with both the source code and the design documents as well as adequate explanations.

This documentation will also guide users to develop simple agent-based models and also include some case-studies of some models as well as possible ideas for extension of the platform.

Chapter 2

Motivation

In this section we discuss the motivation to create a new agent based platform.

2.1 Modelling Complex Systems

There are many approaches to modelling complex systems. We would like to classify them broadly into predictive models and generative models. (The generative models are similar to the approach to modelling as described by Epstine et.al in there book, generative social science.) The nature of complex systems do not allow for building of predictive models. Complex systems interact in intractable ways which are not easily succeptable to data observations. Thus, a lot of data and observations will not be avaliable to build and compare the built models.

2.1.1 An example to expalin complex system behaviour

Let us take a simple thought experiment to imagine complex systems. Imagine looking at a film-reel. A film reel is a series of photographs that are taken one after another forming a series of photographs. Each photograph forms a frame. By playing the reel at 30 frames per second gives us the illusion of moving picture, a movie (due to persistence of vision).

Linear systems are like a film reel. By following the frames on the reel one can deduce the story that the pictures are saying. In case a frame or a set of frames are missing in between them you can always see the frames at the extremities and *draw* new frames or deduce to a good extent, what the pictures in the missing frames would look like. This is because the film real follows a linear causality, i.e. it is ensured that after one frame with one picture, the next frame will hold the next logical frame. Given this guarantee one can go back in the reel to deduce lost pictures. One can also draw new pictures, at the end of a reel. We will also be able to tell where a scene began by tracing our steps back to the using the frames as the frames remain static. Thus, a linear, static, tractable system.

Now imagine instead of a single strip of film you are given a fabric of such photograph frames (as if on a checker pattern blanket where each box has a frame). To begin with how would one start reading such a piece? There is no definite start or end as any part of such a blanket can become the start or the end. There is another complication; imagine if the pictures do not have a causal relation ship, i.e. the adjacent pictures have nothing in common. There is no define way to establish a story. Given a starting point we have no idea as to which picture to jump next. This complicates the matter immensely as one cannot justify where the movie would start or end. One also fails to fill missing frames as we do not know how the frames are related. If a person starts filling frames randomly, there is no way for us to say if the scene is correct or, to make it worse, how did the scene even come about in the first place. In this blanket example, there are only two dimensions of frames, but, what if there were more dimensions? Thus, the system becomes intractable and non-linear. To add to our woes if the frames also kept shifting their places then the system becomes dynamic too!

Another reason for non-avaliability of data is that most complex system components are themselves areas of active research. For example behavioural economics still conducts experiments on how a community considers utility of various resources at its disposal.

However, one can still study complex systems using generative models. The objective of such models is in their ability in explaining the behaviour of the system i.e. it can explain the trajectory of certain outcomes of the complex system. This infomration can be used to understand or look for failure (or unwanted) states and thus test designs for such complex systems.

2.2 Complex Systems and Agent-Based Models

2.3 Other approaches to studying complex systems

2.3.1 Systems Dynamics

2.3.2 Complex Networks

Chapter 3

Design and Architecture of Phoenix

The framework provides infrastructure to create agents, agent behaviour, messaging, output and logging broadly forming the:

- Agent Definition Mechanism
- Communication Layer
- Visualisation or Output Module

The main components of Phoenix are:

- Agent Controller
- Agents
- Message Queues
- Output Module
- Logging

3.1 Agent Controller

AgentController(AC) is a container within which agents live and execute their behaviours. ACs communicate using AMQP protocol. An AC is responsible for the following:

- Agent creation
- Inter agent messaging
- Inter AC messaging.
- Writing simulation output.
- Gracefully shutting-down the simulation locally.

3.2 Agents

An agent represents a real world or abstract entity which is a vital actor in the simulation. Agents have a list of attributes which define the agent and their uniqueness with respect to other agents of the same type.

3.2.1 Agent Attributes

Agent attributes is a abstract class. For a particular type of Agent (e.g. Person) the class has to to extended and required set of attributes defined. The values of the attributes could be stored in a database or in a XML file.

3.2.2 Agent Behaviour

Agents have objectives, beliefs and preferences or biases by which they exist in the system. To achieve its objective an agent has to undertake certain actions to manipulate its data, interact with other agents etc, this is done through actions calls behaviours. An agent may have one or more behaviours behaviour which are executed in a defined order.

3.3 Messaging

The messaging subsystem (or communication channel) is a vital component in a distributed system which facilitates information sharing and synchronisation. In Phoenix the communication is entirely confined to ACs. Our messaging system had to satisfy the following requirements:

- High throughput - A few thousand messages should be delivered, end-to-end, in a second.
- High availability - The system should not crash under high load.
- Asynchronous messaging - Synchronous messaging will slow down a large system. A asynchronous technique which works like mail boxes is desired.
- Platform independence - It should not be tied to a particular programming language or a operating system, this will enable us to build heterogeneous ACs.

RabbitMQ is our choice of the message queuing system. It is based on the Advanced Message Queuing Messaging Protocol (AMQP) standard and is open source under Mozilla Public License. RabbitMQ has interfaces in languages such as Java, .Net, Python and others and supports multiple schemes of communication including 1-to-1, 1-to-many, Store-and-forward, file-streaming and others. Messages are transmitted in the form of binary data and hence any form of encryption/decryption has to be implemented at the client side.

3.4 Output System

This is work in progress. Currently only *log4j* logs are available for output analysis. We plan to include a graphing system.

3.5 Logging

The framework uses *log4j*, a java based logging utility under Apache License, Version 2.0 to provide detailed logging of all system level actions by the various entities.

Chapter 4

Implementation of Phoenix

The framework is implemented in Java. Java provides ease of programming and has a wide range of helpful libraries. In this section the code is explained to some extent.

4.1 AgentController

AgentController is an abstract class. The *AgentController* is responsible for running agents, writing the output, communicating with other ACs and so on. *<NEED TO EXPLAIN FURTHER>*

4.1.1 Agent

Agents are implemented as Java threads. Agent is an abstract class with at least the following attributes:

- *AID* - a unique identifier for the agent.
- *objectiveFlag* - indicates whether the agent has achieved its objective.
- *statusFlag* - a true value indicates that the agent has finished his task for the current tick, and a false value indicates that the agent hasn't finished execution.
- *compositeBehaviour* - one or more behaviours which the agent will run depending on the agents preferences and objectives.

The framework enables the definition of several types of agents, and the creation of a large number of agents of each type.

Every agent will have a set of behaviours, and choose to exhibit(run) a particular behaviour based on its internal logic. Behaviours are implemented using the composite design pattern. Behaviour is an abstract class with a method called 'run', which receives all the parameters of the agent in an object of type *AgentAttributes*. Every agent type will have its specific behaviour implementation, for example *PersonMoveBehaviour* (to move a person agent), *VehicleMoveBehaviour* etc.

The parameter (*AgentAttributes*) defines the current state of the agent. For example, a person agent will have attributes like health, speed, curiosity etc. *AgentAttributes* is itself an abstract class, and each agent type will have its own implementation of this class. *Person* agent type will have *personAttributes*, *Vehicle* will have *VehicleAttributes* and so on.

Every agent contains a 'run' method. While the implementation of this method, other attributes of the agent, its preferences and behaviours depend on the simulation to be run, the logic of the agent will be internalised within this method.

4.2 Messaging

Each AC runs in its own JVM and inter AC communication is through the *RabbitMQ* message queue. Every AC has an input queue, which is identified by the host IP address and a queue name. To communicate with another AC, a AC has to write the message to the recipients message queue. The AC is notified as and when a new message is received through a listener implemented in a method called *receivedMessage(Message).*

4.2.1 Message

A message is a has the following fields:

- type - is an integer and identifies the type of message
- content - is an Integer to indicate the status of the sending AC
- sender - is a string and identifies the sender (is the host name of the sender)

4.2.2 QueueManager

Manages the delivery of a message to the recipient and listening to incoming messages.

- queueUser - the AC using this queue.
- queueParameters - parameters of the input queue for the AC

QueueUser

This is an interface using which any class can receive messages on an input queue. Any class which wants to receive and send messages should implement this interface. *QueueUser* enforces the 'Observer' design pattern on the implementing class. Every AC (which implements *QueueUser*) registers with a *QueueManager*, and whenever a message is received *QueueManager* notifies the AC by calling the *receivedMessage(Message)* method of the AC.

QueueParameters

This defines the the parameters of the input queue.

- *queueName* - name of the queue.
- *username* - username for RabbitMQ
- *password* - password for RabbitMQ

4.2.3 Database Module

<NEED TO COMPLETE THIS>

4.3 High Level Diagrams

Chapter 5

Extending Phoenix and Module Architecture

5.1 Current Modules

5.2 Future Modules

Chapter 6

Using Phoenix

6.1 Basic Configuration

6.2 High Avaliability Configuration

Chapter 7

Performance of Phoenix

Chapter 8

Example Models using Phoenix