Spring 5-20-2016

HIVE - An Agent Based Modeling Framework

Roohi Bharti
San Jose State University

HIVE - An Agent Based Modeling Framework

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Roohi Bharti

May 2016

The Designated Project Committee Approves the Project Titled

HIVE - An Agent Based Modeling Framework

by

Roohi Bharti

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2016

Dr. Jon Pearce	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ABSTRACT

HIVE - An Agent Based Modeling Framework

by Roohi Bharti

This thesis begins by defining agent based modeling. Agent based models are used to model the emergent behavior of complex systems with many interacting components, known as agents. Several model examples are given using NetLogo, which is a popular agent-based modeling platform. A model of concurrent computation is described that uses message passing as the only form of communication between the model's components, which are called actors. The model is called an actor model. Actors are primitive objects of concurrency in an actor model. In particular, we describe the actor model implemented by Akka, which is Scala's new actor library. To explore the relationship between actors and agents, we develop an agent based modeling framework called Hive. Hive is inspired by NetLogo. Like NetLogo, Hive is implemented in Scala and uses actors to represent agents. Unlike NetLogo, which uses Scala's deprecated actor library, Hive uses Scala's new Akka library.

ACKNOWLEDGMENTS

I would like to take an opportunity to thank Dr. Jon Pearce for his guidance and support during the research and development of this project. Also, I would like to thank committee members Dr. Thomas Austin and Dr. Robert Chun for monitoring the progress of the project and giving their valuable time.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1

AGENT BASED MODELING

As the world is becoming more interconnected and complex, our ability to understand it must as well. Simple models that simulate real life situations or problems are no longer sufficient for facilitating this understanding. There is a constant need to develop systems that can simulate real world problems and provide the ability to experiment with ‘‘complex systems’’.

1.1 What are Complex Systems ?

A complex system is composed of multiple interacting components or entities. If the system contains many components, then there will be many interactions and numerous relations between components. That makes the behavior of a complex system difficult to predict because the overall state of the system depends upon the states of individual components, and vice-versa.

To analyze the behavior of complex systems, we need to understand the behavior of the individual components that interact in simple, uniform ways with neighboring components [22]. The aggregate behavior of these interacting components is nonlinear. By this, we mean that though individual components of a complex system may behave uniformly, their aggregated behavior may lead to unpredictable and surprising effects on the overall behavior of the system.

Complex systems are an emerging field of science that focuses on studying the collective behaviors of these systems [23]. Various real world systems can be understood as complex systems. For example, our economies, our schools and our societies all are

examples of complex systems. Consider a market system. Vendors and customers are the components that interact by bargaining over the prices of commodities. After a substantial amount of time, we can observe pricing patterns for each commodity. For example, prices that were initially random, regional, and fluctuating will eventually settle to an equilibrium price that balances supply and demand.

Another example of a complex system is an ant colony. An ant colony will have ants as individual entities or components. Ants have simple individual behavior: an ant searches randomly for food. When the ant finds food, it will eat some and carry the rest back to the nest, while dropping a chemical trail. When an ant searching for food discovers one of these trails, it follows it to locate the food source. The cumulative behavior of these ants can lead to interesting patterns, such as sorting. Ants will usually exploit food sources in order of distance from the nest.

The aforementioned problems are difficult to understand, as they do not follow a direct cause and effect relation of the inputs and outputs of the system. A particular cause scenario of an entity in a complex system can affect other distant entities. There are various approaches to studying complex systems, one of which focuses on computer simulations that model agents as autonomous goal-oriented objects. This approach is called Agent Based Modeling and makes use of massively parallel distributed architectures called Multi Agent Systems. Using these approach, researchers can observe how the cumulative micro-behavior of thousands of agents leads to the overall emergent complex macro- behavior of the system.

1.2 What are Multi-Agent Systems ?

A Multi Agent System (MAS) can be defined as a type of distributed software system. A distributed software system is a system in which individual software

components or entities communicate and coordinate their actions by passing messages. These systems follow one of the following architectures:

1. Client-Server- In this architecture smart client communicates with server and request services.

2. Peer to Peer-In this architecture, responsibilities are uniformly divided among all the components, which are called peers. Peers act as both client and server.

3. Pipeline Architectures- I: In this architecture, components known as filters are connected through data pipes. A component (also called a filter) is capable of reading a message from a data pipe, which serves as a message queue. A filter reads a message from a pipe, processes it, and places the output in another data pipe, which acts like an input to another filter.

A multi-agent system is a peer-to-peer type of distributed software system. Peers are called agents. All agents have access to a common environment.

A Multi Agent System consists of two components: the Agents and the Environment.

1.2.1 Agents

Agents are considered to be autonomous computational individuals or objects. Agents have following properties [26]:

1. Agents are goal-oriented and autonomous.

2. Agents are decentralized and work in a distributed manner.

3. Agents interact with other agents through messages.

4. Agents are reactive and proactively react to the developments in the system. They are able to react to the messages received from the environment or other agents.

5. Agents can exhibit mobile or stationary behavior.

To successfully interact, agents will require the ability to cooperate, coordinate and negotiate with each other, much as people do in the real world.

1.2.2 Environment

An environment is the medium in which an agent operates [27]. The responsibilities of an environment are as follows [27]:

1. An environment should be able to provide a shared common space in which agents can move and operate.

2. An environment should be able to manage the resources and service of a Multi Agent System. Resources in a system are objects or data with a specific state. Services define any functionality of the environment.

3. An environment should be able to provide a means of communication to its agents. Multi Agent systems use a message passing mechanism to carry out indirect communications between agents. .

4. An environment defines different types of rules and laws for the entities in Multi Agent Systems. A certain rule might restrict a particular type of agent from accessing a specific resource.

5. An environment should be observable, and agents should have access to the resources and services of the environment. An agent should be able to observe

the actions and reactions of other agents and new run-time developments in the environment. This requires the environment to have observable properties.

1.2.3 Applications of Multi-Agent Systems

Multi Agent systems are used in various research domains and are an engaging way of understanding and modeling complex systems. They are being adopted as a modeling paradigm to analyze and model complex systems that represent various real-world problems. They can be used to solve complex problems that are difficult or impossible for a monolithic system to solve.

One example of a Multi Agent System is Air Traffic Control, which consists of autonomous controllers and pilots who communicate with each other to manage and direct air traffic. Another example of a Multi Agent System is a social network, which consists of independent social agents that communicate with each other to share thoughts and increase networking. The individual interactions of these agents lead to emergent social behaviors such as trends, uprisings, and economies.

1.2.4 Advantages of Multi Agent Systems

1. Distributes computational resources and capabilities across a network of interconnected agents, as compared to centralized system, which causes resource limitations and performance bottlenecks.

2. Multi Agent Systems efficiently retrieve, filter, and globally coordinate information from sources that are spatially distributed.[32]

1.3 Agent Based Modelling- An Application of Multi Agent Systems

Agent based modeling is a form of computational modeling wherein a complex phenomenon is modeled in terms of agents and their interactions. Interactions include fighting, bargaining, playing, etc.

Agent Based Models are built with the goal of understanding the cumulative behavior of agents, which obey simple rules and need not be intelligent. Examples of cumulative behavior include self-organization, markets, wars, revolutions, trends, and crime waves.

1.4 Properties of Agent Based Modeling

1.4.1 Model

A model is defined as conceptual presentation of an observed phenomenon. A model's goal is to explain the observed phenomenon[29].

1.4.2 Agents

1. Agents are autonomous entities with predefined attributes and behavior.
2. Agents are goal-driven and quit when the goal has been achieved.
3. Agents may be mobile or stationary.
4. Agents interact with each other and with the environment in which they exist.
5. An agent can give birth to other agents.
6. Agents also have memory, which enables an agent to learn from other agents' behavior or from the environment.

7. An agent has defined rules that it follows to exhibit its behavior [30]. Agents' interactions can lead to complex network effects and patterns.

8. An agent interacts with other agents by passing messages.

9. Agents' behavior can be represented as State Determined Automata or a Finite State Machine. A transition in state will occur upon receiving a particular message.

Figure 1: Ant Agent Based Model

The above figure demonstrates an ant society, which can be modeled using Agent Based Modeling.

An ant society consists of bunch of ant agents which are defined using following rules.

1. An ant agent will look for food in vicinity of its home and move towards the food source.

2. An ant agent will then consume some of the food upon reaching the food source

and carry the rest of the food back home. While coming back, it leaves a pheromone trail for other ant agents to follow.

3. Other ant agents might follow the pheromone trail to reach the food source and repeat the process of consuming food.

1.4.3 Environment

An environment acts as medium of interaction between agents. The environment provides resources and services to agents. An agent can interact with fellow agents and with the environment. An environment can have static attributes to provide resources to agents. For example, in case of the Ant Agent Based Model, the environment provides a way for ants to leave pheromone trails for other ants. These pheromone trails will be static attributes and will act as a resource for other agents. Also, food sources will act as a resource for all the ants in the environment.

1.4.4 Emergence

Agent Based Modeling captures the emergent behavior that results from the interactions of the individual agents. Emergent behavior develops over time and can be complex and unpredictable, even when the behavior of the agents is simple and predictable. For example, an ant's decision to move towards a particular food source instead of another will have cumulative effect on the overall behavior of the model that will emerge at the later stage of the model's life cycle. This decision will affect the decisions of other ants, and vice-versa. Thus, emergent behavior cannot be calculated by simply taking the sum of individual behaviors.

1.4.5 Network Structure

Network structure in Agent Based Modeling focuses on information (such as which agents can interact with each other) and captures the network relationship between them. A network topology determines the structure of network interactions. Examples of network structures include fully connected networks and randomly defined networks. A network structure can exhibit different properties and lead to differences in interactions between agents. For example, an ant agent might move to another ant agent base or home after consuming some of the food, which might lead to a difference in the behavior of ant agent societies.

1.4.6 Stochasticity

Stochasticity means Agent Based Models are random in nature. This means homogeneous types of agents, which exhibit similar behavior and follow similar rules of interaction, might react to a situation differently. For example, if two ant agents are situated at the same distance from two food sources, one ant may decide to move towards a different food source instead of moving towards the same food source.

1.5 Why Agent Based Modelling?

[1]Agent Based Modeling supports an understanding of complex systems. The behavior of complex systems cannot be understood purely by representing the system with a mathematical model of linear equations. This solution only provides a crude approximation of the overall emergent behavior of the system. With emerging techniques and advancement in research domains, new ways have been developed to solve complex problems. Agent Based Models enhance the understanding of complex systems using computer-based simulations. These simulations can be as simple as

representing individual components as agent objects in an object-oriented environment, or they can be represented by powerful computer networks that represent underlying complexities. With this development, experts can experiment with different models to foresee a situation based on certain input conditions or analyze trends based on various factors.

1.6 Domains where Agent Based Modeling is used -

[2] Agent Based Modeling provides a basis for decision-making in complex real-life systems. For example, Agent Based Modeling simulations are used to gain experience in complex military operations. Military staff can train by simulating a war game, which is a multi agent system. Thus, staff with actual experience and virtual experience can fine-tune their strategies on the battlefield.

Another emergent field that makes extensive use of Agent Based Models is the field of marketplaces. It is important for fast-changing markets to simulate or predict the performance of a product.

In a few hours, Agent Based Models can simulate a situation that could actually take several years to happen. Thus, they can help in making crucial decisions with confidence.

Social phenomena are still another field that make use of Agent Based Modeling. Agent Based Modeling can be applied to a diverse range of public policy domains. It is gaining popularity in entertainment applications. These could be computer games, with players as humans or agents. Other entertainment fields that are using Agent Based Modeling are interactive media, television, and even books. One of the famous examples of this is the film ‘‘The Lord of the Rings: The Two Towers’’.

1.7 Net Logo - An Agent Based Modeling Tool Kit

Net Logo is a procedural programming language that is derived from LOGO, which in turn is a dialect of LISP. Language LOGO was designed for learning and was intended to be used by Children[9].

Net Logo provides a multi-agent programmable modeling environment [3], allowing a user to simulate various Agent Based Models. This language has been written for research and education purposes. A simulated model is capable of taking instructions from the user. This concurrent model allows looking into both micro-level developments within an agent and macro-level interactions in between agents. Net Logo includes rich library support for various models, a GUI to visualize the agentâĂŹs movement, and real time charts and visualizations [9].

1.7.1 Net Logo Agents

In Net Logo, agents are autonomous entities that are programmed using simple rules.

Net Logo Agents are of two types-

1. Turtles- Turtles are agents that can move in the Net Logo world. They have changing coordinates.

2. The NetLogo environment is divided into a grid of patches. Patches are agents with fixed coordinates.

1.7.2 NetLogo Interface

The NetLogo user interface consists of controls, monitors, and a window that displays the agents as the simulation runs.

1.7.3 Programming NetLogo

NetLogo users can run interesting models from the model library, or they can build their own models by writing initialization and update procedures in the LOGO programming language. An initialization procedure describes how an agent initializes its state when a simulation begins. The state might include properties such as fitness, metabolism, and speed. A simulation is divided into cycles. During a cycle, NetLogo asks each agent to execute its update procedure. The update procedure describes how an agent updates its state. This typically involves interacting with a randomly selected neighbor agent, then moving to a randomly selected nearby position. Typical interactions include fighting, mating, and trading.

1.7.4 Ant Foraging Model Net Logo Simulation

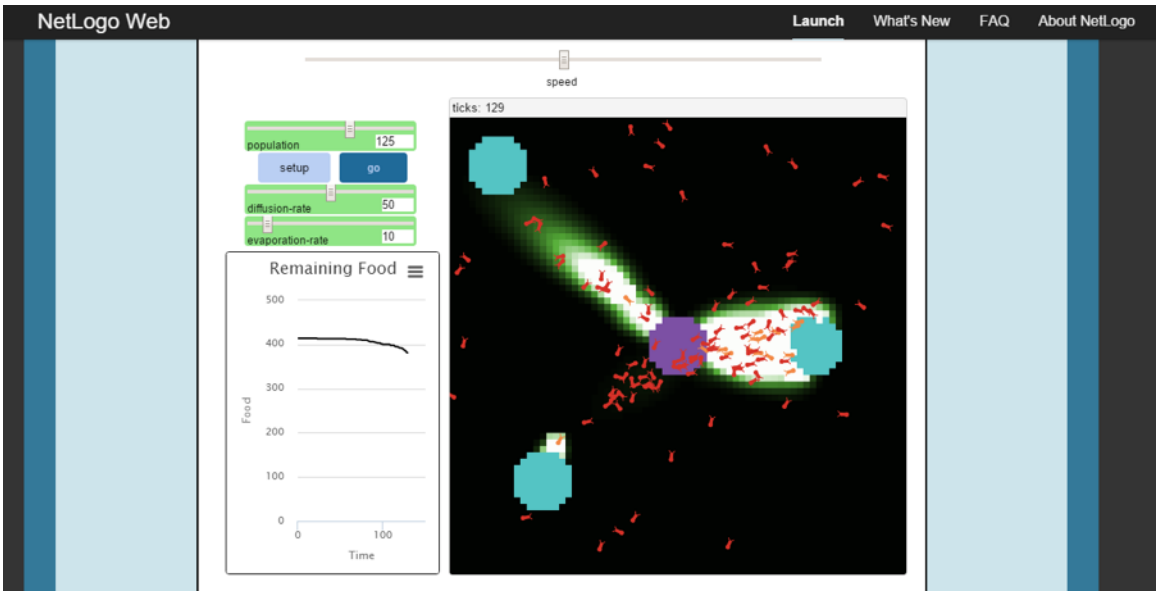

Figure 2: Ants moving in Ant nest looking for food

The ant foraging model can be implemented in NetLogo. In this case, ants are represented by ant-shaped turtles. The ant nest, pheromone trails, and sugar sources

are represented by colored patches.

The above screenshot shows the Ants moving out of Ant nest and moving towards different food source.

The ant update procedure is called look-for-food:

```
to look-for-food ;; turtle procedure
  if food > 0
  [ set color orange + 1   ;; pick up food
    set food food - 1      ;; and reduce the food source
    rt 180                 ;; and turn around
    stop ]
  ;; go in the direction where the chemical smell is strongest
  if (chemical >= 0.05) and (chemical < 2)
  [ uphill-chemical ]
End
```

Upon reaching a food source, an ant picks up the food and adjusts the remaining food at the food source. After this, an ant will again move in the direction where the pheromone trail (chemical) is strongest.

The pheromone trails have two properties that can be controlled by sliders on the user interface. Diffusion controls the breadth of the trail, and evaporation rate controls how long the trail persists. Users can experiment with these controls and make observations about how these properties affect the ability of the colony to exploit food resources. Such observations may lead to hypotheses on ant food-seeking behavior that can be tested experimentally.

The following screenshot shows the simulation after 148 cycles. Note that the

food resources appear to be consumed in order of their distance from the ant nest, even though the individual ants know nothing about these distances or sorting.

CHAPTER 2

Scala and Akka support for Agent Based System

2.1 Approaches to build concurrent systems

Concurrent Systems are those in which several computations execute simultaneously and may interact with each other.

Agent based systems require the construction of high performance, user-friendly, distributed and concurrent simulation frameworks [10]. Since concurrent systems require interaction between executing components, several issues arise while constructing concurrent systems.

Computations in concurrent systems can be executed by several execution paths. One of the obvious and standard approaches suggests the use of multi-threading and shared memory to implement individual computations and their respective interactions. An object oriented concurrent application has multiple stateful objects running in their own specific threads. These threads have access to shared memory. Also, states of these objects can be altered by various parts of the application [12] depending on the application logic. Such applications use locks to access shared resources, which creates an additional load on the programmer and slows down the application. Also, this approach comes with problems like race conditions, deadlocks and indeterminate output. There are several models for modeling and reasoning about concurrent systems. An example of one is the Actor Model, which has been adopted by Scala to execute concurrent operations.

2.2 Actor Concurrency Model

The basics of an Actor Model involve using ‘‘Actors’’ as primitive objects of concurrency. Actors can also be viewed as replacements for threads in traditional multi-threaded applications. An actor contains some variables that constitute its state. These actor objects are capable of interacting with each other using message-passing concepts. Messages can be exchanged in an asynchronous way. Because message passing is used instead of shared memory communication, the problems associated with synchronizing access to shared resources are eliminated. An actor can react to the messages received in the following ways:

1. An actor can create a finite number of new actors.

2. An actor can send a finite number of messages to other actors.

3. An actor can manage its internal state by updating its mutable attributes which will take effect when the next message is received and handled. This means an actor's internal computation is based on its current state and this current state is applied to all the messages received currently [36].

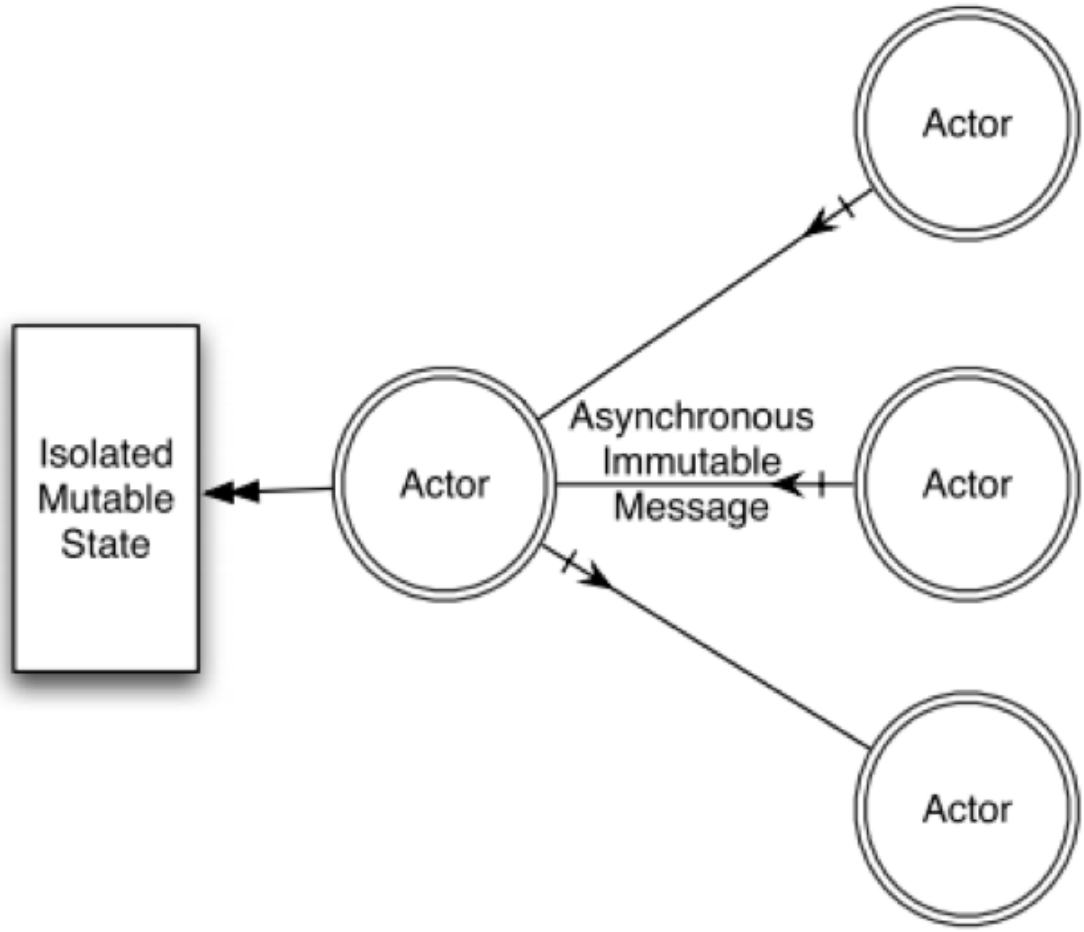

Figure 3: Actor Model for Concurrency

The above diagram explains the concurrent system implementing asynchronous messages. Actors have an isolated mutable state which means this state is inaccessible to outer world directly and is only accessible through an actor bearing this state. Actors responds to incoming messages based on its designated behavior.

The Actor Model has several important rules. First and most crucial, actors must not share their states. As stated above, an actor has an isolated mutable state that cannot be accessed by any actor other than itself. This restricts the actors from sending references, pointers, or any other shared data in messages to other actors. An actor is referred by a name, which is the address of an actor. An actor can send

immutable data and addresses (‘‘names’’) in a message to other actors.

Messages received by an actor arrive in the actor's mailbox. An actor's mailbox acts like a queue for storing pending messages. Accessing an actor's mailbox is an atomic operation. This means multiple actors can send messages to an actor at the same time, but an actor can receive only a single message at a time. Thus, the actor model makes no guarantee of the order of the messages sent from different actors to an actor at the same time, as an actor's mailbox can be accessed atomically by an incoming message. Race conditions in the mailbox are avoided by implementing queuing and de-queuing of messages as atomic operations. An actor processes the messages received in its mailbox sequentially with the aforementioned ways to react. An actor can update its mutable state upon receiving a message. The new state of an actor is applied after the current message has been processed. All the messages received by an actor in the future will perceive this updated current internal state. Thus, each message handling is processed safely by an actor.

An actor can define the way of handling messages using pattern matching. An actor can define different behaviors and can have different semantics and mailboxes for each implemented behavior, which will allow the actor to respond to specific messages in a particular behavior. When exhibiting a particular behavior, an actor receives messages specific to this behavior's mailbox and temporarily ignores other messages received. These messages are received in another mailbox. An actor can switch its behavior.

From an actor's perspective, sending messages is a non-blocking operation, whereas receiving messages is a blocking operation. Receiving messages is a blocking operation, as other actors might be trying to put messages in this actor's mailbox at the same time. Sending messages is non-blocking operation, since messages are sent in an

asynchronous way.

At a low level, the runtime platform allocates the vast number of actors to underlying machine CPU cores and resources and schedules the execution of the pending messages. Most systems follow the lock free method implementation, as actors only require atomic blocking operation while receiving messages. The internal scheduler groups actors in the form of a process queue and works in a preemptive manner. For example, when a scheduler picks an actor for processing after a certain number of messages handled, the scheduler picks the next actor that is ready to run. This boosts liveliness and ensures that no particular actor blocks the CPU for an extended time. Message handling is executed in a single threaded manner and supports various yield points.

It is relatively easy to maintain fault tolerance in concurrent systems that are developed using the Actor Model (as compared to a conventional thread based concurrent systems). In traditional thread-based concurrency, unpredictable and non-deterministic scheduling, along with the shared state, makes replication of states and taking a snapshot of the system difficult. In the Actor Model, isolated states and messages in the mailbox together are similar to taking snapshots and collecting logs. Messages are collected in a mailbox with timestamps, and, when applied from the beginning, can replicate an actor state. Thus, messages in the mailbox are logs for taking a backup of the system.

The actor model advocates the ‘‘let it crash’’ methodology. This means that, if an actor crashes, it will not affect other actors due to its non-sharable mutable state. For handling fault tolerance, actors are arranged in the hierarchical structure to provide supervision facility. For example, upon dying, an actor notifies its supervisor (parent). The supervisor then follows the supervising strategy, which might require restarting

the child actor. Supervising strategy creates a dependency relation between an actor and its subordinates.

Conceptually, each actor is a lightweight thread, as actors run over a lesser number of threads than the number of actors. Thus, many actors run on the same thread. The Actor model is designed to perform intelligent scheduling of threads among these actors. Actors have isolated states that cannot be accessed by other actors. This obviates the need to use locks in programming for synchronization. Concurrency issues like deadlock and race conditions can arise due to incorrect application design. For example, a deadlock condition arises when two actors are in cyclic dependency with each other by waiting to get a message from each other at the same time. In practice, deadlocks can be prevented by using timeouts. This is similar to practices used in traditional thread programming.

2.3 Scala

Scala is a new language developed in 2003[10]. It is object-oriented in nature and supports functional programming constructs. Scala interoperates with Java, and Scala programs can be compiled to Java byte code. This feature provides support for Java libraries and Java code to the Scala language as well.

Scala supports static type checking and has the advantage of the Hotspot Java Virtual Machine. Scala programs run at the same speed as Java programs. Also, Scala is a well-documented language, and its documentation provides many examples.

Scala stands for ‘‘scalable language’’. It is designed to grow with the demands of users. It is suitable for wide range of programming tasks, from writing scripts to building large systems and frameworks. Scala's support for object oriented programming and functional programming makes it highly scalable.

Scala implements multithreading through the Actor Model, which simplifies the concurrency programming style and reduces synchronization overhead. Scala's Actor Model implementation is a standard library feature in Scala.

It supports two types of concurrency:

1. **Thread Based Actors-** In this implementation, each actor is backed by a dedicated thread. This limits the scalability and requires synchronous blocking and suspension of threads when waiting for new messages.

2. **Event Driven Actors-**This implementation does not couple an actor to a specific thread; instead, it allows allocating a pool of threads to any number of actors. This implementation follows an asynchronous and non-blocking event driven strategy, which does not bind an actor with a dedicated thread. Instead, a thread handles a number of events. These events are queued. A thread executes its event loop, which, one by one, executes queued events. This implementation is highly scalable.

Event driven actors are widely used for building concurrent systems, as this implementation is highly scalable. Scala thus is an optimal choice for building concurrent and scalable applications. Since actors are individual objects, actors can be used to represent agents in agent based systems.

Messages received by an actor are considered to be immutable values. A special type of wrapper classes- ''case'' classes is used to support message handling by an actor.

Scala was supported by its own Actor Library until Scala version 2.10. Since version 2.11 of Scala, the Actor library has been replaced by the Akka library.

2.4 Akka

Akka is the library for concurrency in Scala. It is a framework toolkit and runtime used for developing scalable, distributed, concurrent and fault tolerant applications on a Java Virtual Machine. Akka is written in Scala. Akka also has support for the Java language.

Akka adopts the Actor Model to handle concurrency. The complexity of creating and scheduling threads, receiving and dispatching messages, and handling race conditions and synchronization is delegated to the Akka framework. Akka supports:

1. Scale up (Concurrency)
2. Scale out (Remoting)
3. Fault Tolerance

2.4.1 Reactive Manifesto

Akka raises the level of abstraction in building reactive systems. A reactive system is a system that is driven by external events. The ideal reactive system has the following properties:

1. **Responsive-** A system that responds in a timely manner to external events is said to be responsive. A responsive system aims to provide a quick and consistent response time, and thus helps in building systems that guarantee the quality of service. Such kinds of systems can help with the quick detection of problems and enables us to deal with problems effectively.
2. **Resilient -** Resiliency ensures responsiveness of the system, even at the time of failure. Resilience is ensured by replication, isolation, delegation, and con-

tainment of components. Components are designed in an isolated manner; thus, failure of a component does not affect other components. In the case of failure, recovery is delegated to another component. Replication ensures high availability.

3. **Elastic -**Elasticity ensures responsiveness, even at times of varying workload. Such a system can react to changes in input load

4. **Message Driven-** Reactive systems depend on asynchronous message passing techniques to communicate. This establishes location transparency, isolation, and a loose coupling between communicating components.

Finite state machines are reactive systems, as a finite state machine is capable of reacting to external events by changing state from one phase to another.

The aforementioned aspects of reactive systems are interconnected. Each aspect strengthens the other ones. A responsive application makes sure to react on time to all events. Thus, in the case of failure, it can react swiftly. A message driven application ensures asynchronous communication between various components irrespective of their location. This makes an application reactive by swiftly acting through messages. An elastic application makes an application resilient, as it can handle varying workloads consistently.

This can be demonstrated with the help of a diagram. These features constitute the ideal characteristics of a concurrent system.

2.4.2 Akka Actors

Actors are active objects that have specific state and behavior. They form the smallest unit when building an Akka application. Actors in Akka communicate

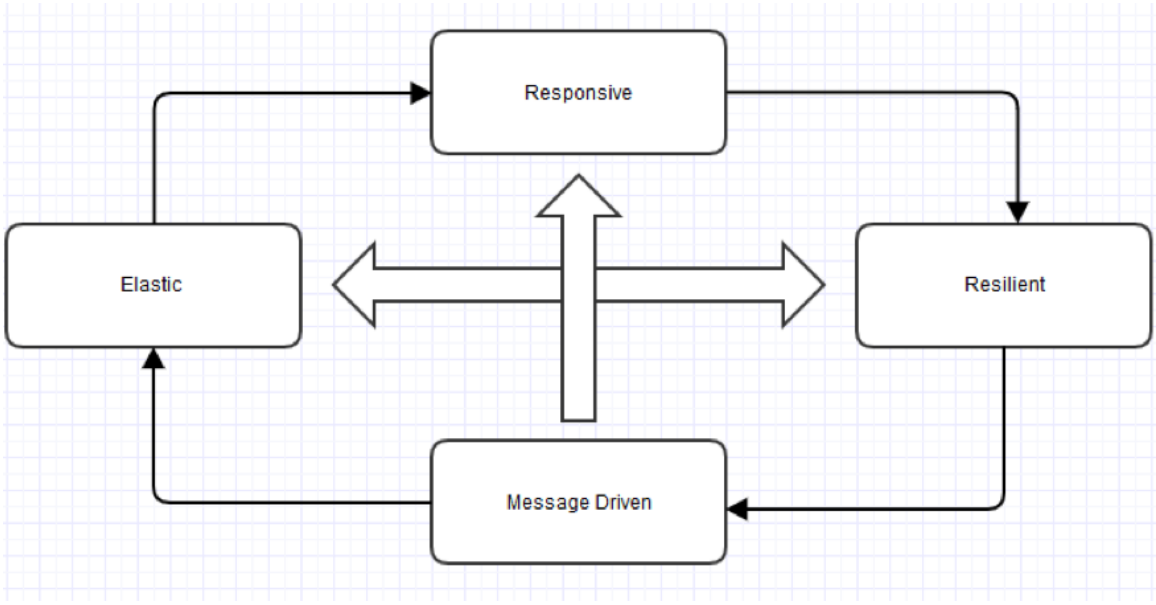

Figure 4: Characteristics of Reactive Systems

through a message passing mechanism. An actor is a container for the following:

1. **Actor Reference-** Actors are represented to the outside world using an abstraction called an actor reference. An actor reference is an object that refers to an actor and can be passed freely. Actor references are needed to provide support for sending messages to other actors [37].

2. **State-** An actor object contains variables that represent the current state of an actor. In Akka, an actor state is isolated from other actors' states. Thus, conceptually, an actor runs in its own lightweight thread. This obviates the need to write synchronization access using lock code while writing actor logic. In practice, Akka allocates a set of real threads to a set of actor objects. Thus, many actor objects share a single thread. Akka ensures that this implementation does not affect the single-headedness handling of actor's state.

3. **Behavior-** 1. An actor possesses a behavior. Behavior means a function that

defines the reaction of an actor upon receiving a particular message. A message received by an actor is processed by matching it against its behavior. Behavior can change over time, and Akka implements this by allowing an actor object to exist in different states by swapping the function that manages its behavior.

4. **Mailbox-** An actor receives a message from other actors in its mailbox. An actor has only one mailbox. It is implemented by FIFO logic, meaning an actor processes messages in the same order in which they are enqueued. Messages are enqueued according to the order of send operations. This means that messages sent from different actors might not maintain the order due to the randomness of distribution of actors to real threads. But different messages sent from the same actor maintain the same order in the mailbox of recipient actor.

5. **Children -**If an actor creates new actors to delegate subtasks to, then it acts as a supervisor to all the child actors it has created. Supervisor actors maintain a list of all their child actors and have access to them.

6. **Supervisor Strategy-** An actor maintains a supervisor strategy to handle the faults of child actors. Fault handling is handled transparently by Akka. It cannot be changed after an actor is created.

When an actor terminates (i.e., fails) such that it cannot be restarted, or when it is stopped by itself or its supervisor, it then frees all of its resources and drains all of its pending messages to the system ‘‘dead letter mailbox’’. Thus, an actor mailbox is replaced by the default system mailbox, which is for dead letters. This helps decipher test failures quickly and efficiently.

2.4.3 Supervision and Monitoring

As stated above, a supervisor delegates tasks to subordinate actors and must respond to their failures as well. Upon encountering a failure, a subordinate actor suspends itself and all its subordinate actors and sends a message to its supervisor by informing the supervisor of a failure. A supervisor can respond to a failure message in the following ways:

1. Resume the subordinate actors by preserving their internal state.
2. Restart the subordinate actors by clearing out their internal state.
3. Stop all subordinate actors permanently.
4. Stop itself, thus escalating the failure.

Since a supervisor actor is also a subordinate to a supervisor above it, this means that the actor system (explained in the topic below) forms a supervision hierarchy.

2.4.4 Akka Actors System

An actor system is considered to be a hierarchical structure of actors that have common configurations like dispatchers, deployments, remote capabilities, and addresses (explained below). It acts as an entry point for creating and looking for actors. It is a structure that allocates a number of threads to an application. Multiple actors share the same thread.

The Akka Actor system has following features-

1. An actor system follows a hierarchical structure for creating and supervising actors. A top level actor is created by the library itself.

2. If an actor creates another actor, then it acts like a supervisor of the child actor. An actor has at most one supervisor.

3. A supervisor might want to split a task into smaller tasks and delegate them to child actors. Child actors are capable of handling the task assigned to them. If a child actor is incapable of handling a task, it sends a failure message to the supervisor.

4. A supervisor is responsible for handling the failure messages of its children. A supervisor can respond to a failure message in the following ways:

    - Resume the child actor by backing up its internal state. Restarting an actor restarts all its subordinate actors.
    - Restart the child actor by clearing its internal state. Resuming an actor resumes all its subordinate actors.
    - Stop the child actor permanently. Stopping an actor will stop all the subordinate actors of an actor.
    - Support the failure by failing itself.

2.4.5 Actor Hierarchy

Actor system in Akka follows the following hierarchy.

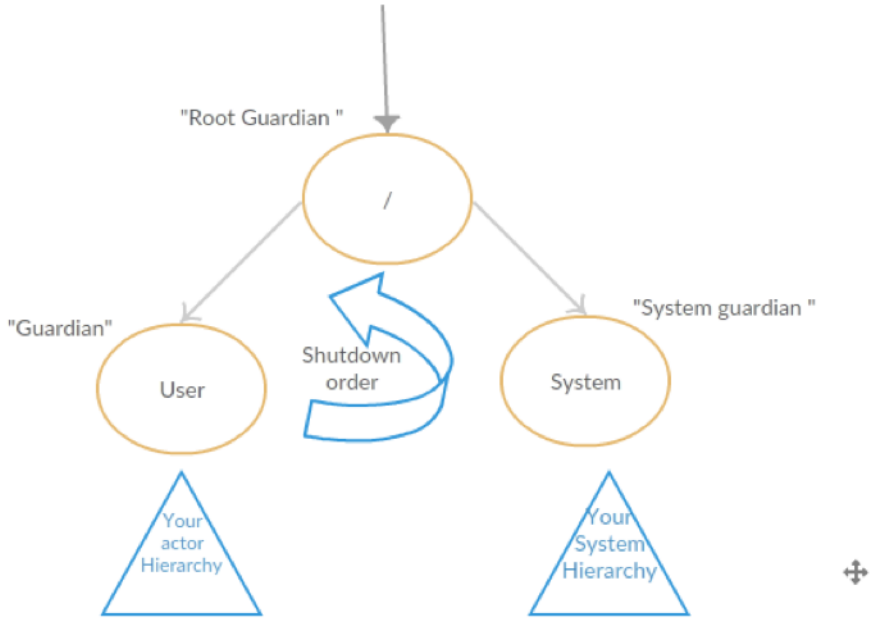

Figure 5: Actor Hierarchy in Akka

An actor system creates and starts at least three actors

1. /**user-The guardian actor-** This actor is the parent of all user-created actors. This means that when the guardian terminates, all of the child actors terminate too. The guardian supervisor strategy can decide the guardian's response to the failures of subordinate actors. If the guardian escalates a failure, then the root guardian response will terminate the guardian actor and shut down whole actor system.

2. /**system-The system actor-** This special actor will watch the user guardian actor and initiate its own shutdown upon receiving a termination message. It

is introduced to ensure the orderly shutdown of the actor system when all the actors in the system have left.

3. /**- The root guardian-** The root guardian is the grandparent of the so-called ‘‘top level’’ and special actors in the actor system. The root guardian stops child actor upon any Exception.

2.4.6 Actor Reference and Actor Path

Actor Reference

An Actor Reference represents an actor in Akka. An Actor Reference's purpose is to support the sending of messages to the actor it represents [35]. An actor can refer to its own reference by using a ‘‘self’’ field. When sending a message, an actor's reference is included in the ‘‘sender’’ field. Thus, when processing a message, an actor can reference the sender by accessing ‘‘sender’’ method. We cannot create an actor reference without creating an actor. The lifecycle of an actor reference can be associated with the actor's lifecycle.

Actor Path

An actor path is a unique hierarchical sequence of actor names given recursively using the links between child and parent actors, going to the root of the actor system. An Actor Path is a symbolic link to the actor. An actor path is just a name of an actor and does not have a lifecycle. It can never become invalid. We can create an actor path without creating an actor. An actor path cannot be associated with the lifecycle of an actor, as we can delete an actor and then recreate a new actor using the same path. The newly created actor is the new incarnation of the previous actor and has a different lifecycle.

Creating Actors

Actors are created by extending the base Actor trait. Creating an actor requires the implementation of a receive method. A receive method is a Partial Function ([Any, Unit]. A partial function can only give a solution to specific types of arguments. This means that it can query for different types of messages, which can be handled by an actor's mailbox. A receive method has to be defined in terms of a series of ''case'' statements. These ''case'' statements define the messages an actor can handle and use pattern matching to match the specific message. This acts as an actor mailbox that is responsible for receiving messages of ''Any'' type. Thus, the outcome of the receive method is an object that can be of any type, which constitutes the initial behavior of an actor. We can manage different actor behaviors using ''become/unbecome'' functionality (explained in below topics).

A typical actor implementation is as follows:

```
import akka.actor.Actor
import akka.actor.Props
class MyActor extends Actor {
val messages=‘‘’’‘‘’’
def receive = {
case ‘‘Hi’’ => sender! ‘‘hi!How are you?’’
case _     =>sender! ‘‘received unknown message’’
}
}
```

The above example defines an actor ''MyActor'' which can receive a ''hi'' message and can respond to the sender with a ''Hi!How are you'' message. An actor sends a

message by using the ‘‘!’’ symbol followed by a message.

Actor systems are created beneath the guardian actor using the ‘‘ActorSystem.actorOf’’ method and then using the ‘‘ActorContext.actorOf’’ method from the created actor to create the actor hierarchy. These methods return the reference created by the actor. A reference of an actor is sent within a message to other actors, which enable actors to reply to the sender actors using the ‘‘sender’’ keyword. Also, an actor can self-reference using the ‘‘self’’ keyword, which allows an actor to send a message to itself.

Props

Props is an immutable and shareable configuration class to specify various options for creating actors. Props is a recipe for creating actors in Akka. For example:

```
import akka.actor.Props
val propsforMyActor = Props[MyActor]
```

In the above example, propsforMyActor is the configuration class to create an actor of type ‘‘MyActor’’ (defined in previous example).

Creating actors using Props

Actors are created using the ‘‘actorOf’’ factory method and passing the props configuration class instance to it. This ’’actorOf’’ method is available under ActorSystem or ActorContext.

```
import akka.actor.ActorSystemÂă
// ActorSystem is a heavy object: create only one per application
val system = ActorSystem(‘‘mySystem’’)
val myActor = system.actorOf(propsforMyActor)
```

In above example, configuration class ‘‘propsforMyActor’’ is used for creating an actor ‘‘myActor’’ of type ‘‘MyActor’’ under ‘‘system’’ actor.

Actors that are created under ActorSystem are top-level actors and are supervised by the guardian actor provided by the system. Actors that are created under ActorContext are called child actors.

Looking for actors using concrete path

Actor references can be searched by using ‘‘ActorSystem.actorSelection’’ method. This can be used for communicating with the selected actor.

Top level scopes for Actor Path (Special Actors)

1. ‘‘/user’’- It is the guardian actor for all the user created actors.
2. ‘‘/system’’- It is the guardian actor for all the system created actors.
3. ‘‘/deadletters’’-It is the dead letter actor where messages to all the dead actors are sent.
4. ‘‘/temp’’- It is the guardian actor for all short lived system created actors.
5. ‘‘/remote’’- It is the artificial path below which all the remote actors reside.

Messages

An actor uses immutable case classes to define the type of messages it can receive. Messages received by an actor have to be immutable.

```
// define the case class
case class Register(user: User)
// create a new case class message
```

```
val message = Register(user)
```

Sending Messages

Messages can be sent using the following methods:

1. **‘‘Fire and Forget-!’’-** 1. This method sends a message and returns immediately. It is a non-blocking and asynchronous operation.
2. **‘‘Ask-?’’-** This method sends a message and returns a âĂIJFutureâĂİ which is a possible reply.
3. **‘‘Forward message’’-** 1. This method is used to forward messages from one actor to another.

Replying to messages

An actor can reply to sender using its reference by accessing a sender keyword and sending a message by ‘‘!’’. An actor can also save the sender actor reference to reply to a message later.

Client Server Example

To demonstrate Akka in action, a small client server application is implemented. This calculator-type application has three Actors: two clients and one server. The client and server perform the following task:

1. **Server** Receives the commands from the clients and calculates the answers.

   Server looks like:

   ```
   case class add(num1:Int,num2:Int)
   ```

```
case class mul(num1:Int, num2:Int)


class server extends Actor{



  def receive={
    case _=>{
      println(‘‘Unsupported message’’)
    }
    case add(num1:Int,num2:Int)=>{
      var num3=num1+num2
      println(‘‘Add result= %s’’.format(num3))
      sender!addanswer(num3)
    }
    case mul(num1:Int,num2:Int)=>{
      var num3=num1*num2
      println("Multiply result= %s’’.format(num3))
      sender!mulanswer(num3)
    }
  }
}
```

Servers can accept multiplication and addition requests from clients. Messages sent to the server can be handled by the server mailbox. Server, upon receiving an add and multiply message, calculates the answer and sends the response back to the client. Server can access the client through a ‘‘sender’’ reference.

2. **Client** - Sends addition and multiplication commands to server.

   Client looks like:

```
case class mulanswer(ans:Int)
case class addanswer(ans:Int)
class client extends Actor{


  sender!add(2,3)
  sender!mul(2,3)
  def receive={
     case mulanswer(ans:Int)=>{
      println(‘‘multiplication answer is %s’’.format(ans))
    }
     case addanswer(ans:Int)=>{
      println("addition answer is %s’’.format(ans))
    }
    case _=>{
      println("Unsupported message")
    }
  }
}
```

   Client can send addition and multiplication requests to server by asynchronously sending commands like add (num1, num2) or mul (num1,num2). Server calculates the answer and sends the reply back to clients. Client is capable of accepting answers from server and displaying it to user.

   Output:

Addition answer is 5 Multiplication answer is 6

Receive Timeout

An actor can have inactivity timeout which is defined by ‘‘ActorContext setReceiveTmeout’’.

Stopping an actor

Actors can be stopped by invoking the stop method on ActorSystem or ActorContext. Termination of an actor occurs asynchronously, and the stop method can return to an actor before it has been stopped. Actor context is used to stop child actors or the actor itself. Actor will continue the execution of the current message before stopping. Additional message will not be processed, and messages to this actor are received by ‘‘deadletters’’ which is a special system actor.

Upon receiving a stop message, an actor will leave its mailbox processing and will send a stop command to all its child actors. Actor waits for a termination notification from all of its children and exits when it receives a notification from the last one.

Posion Pill

A posion pill message will also stop an actor, but it is enqueued as normal message in a mailbox. It will be processed after handling messages that are already enqueued in the mailbox.

Graceful Stop

Graceful stop is used when application logic needs to define the order of termination of various actors.

Become/Unbecome

Akka allows hot swapping of an actor's behavior at runtime. An actor's behavior is defined in terms of a message loop. The context.become method swaps the actor's message loop with a different message loop. The hot-swapped code is kept in a stack and can be pushed and popped according to application logic.

```
class HappyActor extends Actor {
  import context._
  def angry: Receive = {
    case "foo" => sender() ! "I am already angry?"
    case "bar" => become(happy)
  }
  def happy: Receive = {
    case "bar" => sender() ! "I am already happy :-)"
    case "foo" => become(angry)
  }
  def receive = {
    case "foo" => become(angry)
    case "bar" => become(happy)
  }
}
```

The above example shows a simple example of hot swapping the message loop at runtime. A ‘‘HappyActor’’ can receive a ‘‘foo’’ message and can become ‘‘angry’’. This message swaps this actor's message loop to a specific ‘‘angry’’ behavior's message loop. When this actor receives the message ‘‘bar,’’ it switches its behavior to ‘‘happy’’.

This Akka functionality allows actors to maintain states and exhibit different behaviors in different states. Thus, state machines with diverse behaviors can be

implemented using become/unbecome behavior.

Killing an actor

An actor can be killed by a ‘‘Kill’’ message. The actor will suspend all its operations. The supervisor of an actor then decides how to tackle this actor's failure. The supervisor might resume an actor or restart it.

/ kill the 'victim' actor victim ! Kill

CHAPTER 3

Hive- An Agent Based Modeling Framework

3.1 Hive

Hive is an agent based modeling framework programmed using Scala and Akka. It enables programmers with intermediate knowledge of Scala and Akka to develop agent based models and provides a command line interface to execute the model.

3.1.1 Hive Architecture

Since Hive is created using Scala and Akka, agents are represented by Akka actors.

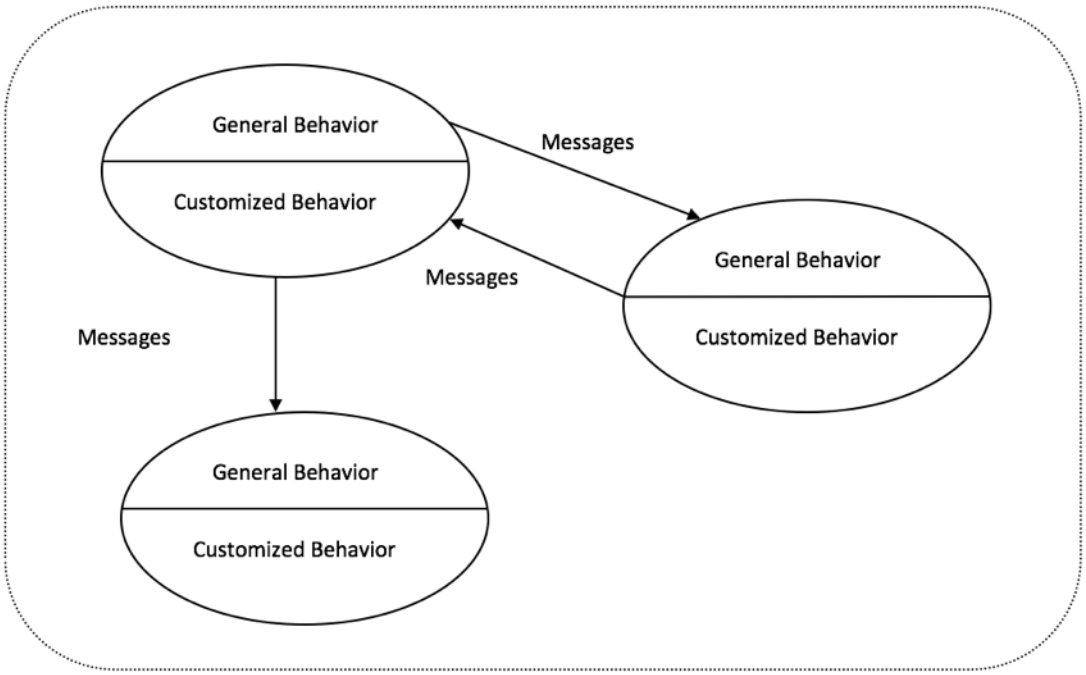

Figure 6: Hive agents interactions and implementation in specific models

Agent interactions use the message passing mechanism between Akka actors. An agent based model implemented using Hive will extend the general behavior of Hive agents to create model specific agents.

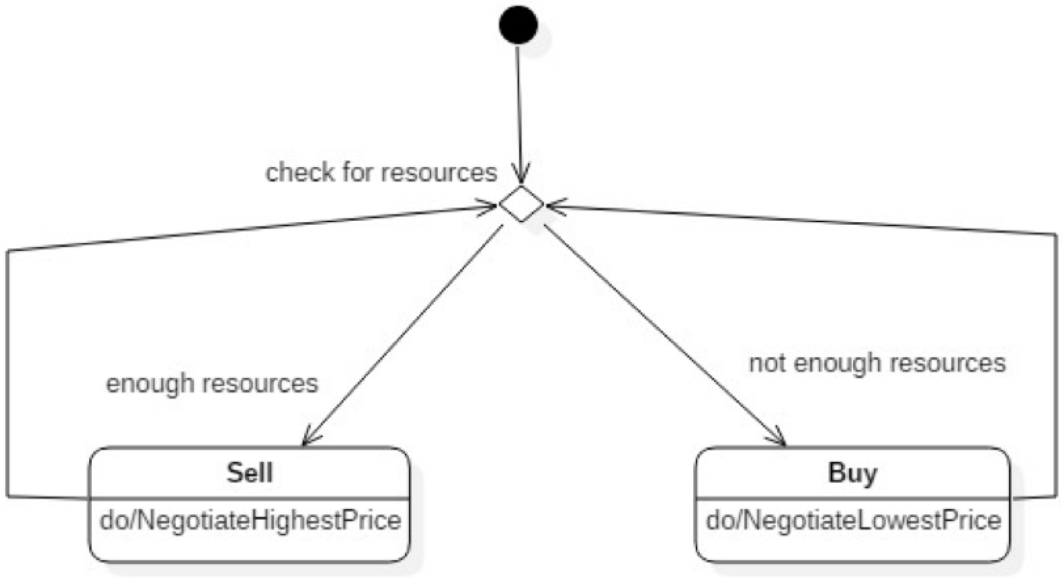

Figure 7: State Diagram depicting state transition of an agent in a market

In Hive, agents execute simple rules and exhibit simple behavior. An agent's behavior is expressed in terms of its states. Agents interact with each other by passing asynchronous messages. An agent needs to compute information before sending a message. This may involve checking its internal state. Upon receiving a message, an agent can change its internal state by updating itself according to its behavior. An agent's typical behavior, which is programmed according to the conceptual model, will decide the agent's reaction upon receiving a message. For example, an agent based model depicting a market will have agents that can exhibit two states ''buy'' and ''sell''. Upon getting a ''buy'' request from a neighboring agent, an agent will switch its state to the ''sell '' state. In this state, an agent will execute the behavior of a

seller and negotiate with the interacting agents for highest price. Also when an agent does not have enough resources (such as products to sell in the market), it switches its state to the ‘‘buy’’ state and sends a request to other sellers to buy resources.

The cumulative effect of message exchanges between numbers of agents may lead to interesting patterns in the model. In the above example, an observer might be surprised to see the average prices paid by buyers in a market reach an equilibrium, even though agents may only know the prices paid for products by nearby agents. Thus, individual interactions between agents at the micro level can lead to macro behavior like ‘‘recession’’ and ‘‘inflation’’ in economic agent based models.

3.2 Hive Components

Hive consist of three salient components.

1. Hive agents
2. Environment
3. Initiator

3.2.1 Hive Agents

Hive agents are actors that represent interacting people, robots, birds, etc. in a population. Abstractly, an agent has the following characteristics:

1. An agent has a mutable state and has a specific behavior associated with a state. An agent can change its state upon meeting a condition for transition.
2. An agent is reactive and can react to developments in the environment.

3. An agent has attributes, which are mutable variables that can be updated by various operations.

4. An agent has specific behavior (actions taken by an agent as a reaction upon receiving a message). The behavior can change over the time.

5. An agent has a ‘‘name’’,‘‘attribute’’, ‘‘collaborator’’ and ‘‘message’’.

    - **Name-** Name is name of the agent, which is provided at the time of initialization.
    - **Attributes-** Attributes are the specific properties of an agent. These properties are programmed according to the Agent Based Conceptual Model. They are considered deciding factors in making various decisions for interaction. The values of these attributes cumulatively define the state of an agent.
    - **Collaborator-** Collaborator is an agent with whom another agent is interacting at a particular moment. A collaborator for an agent can change from cycle to cycle in a simulation. An agent might be initiating interaction with a collaborator or receiving a message from a collaborator requesting interaction.
    - **Message -**Message is a type of message an agent has sent or received while interacting with a Collaborator.

6. It can invoke its own methods to send messages to itself or to other agents via Facilitator (see below).

7. It is capable of killing itself upon reaching a terminating condition.

At the heart of every Hive simulation is a special agent called the facilitator. The facilitator has the following characteristics:

1. It creates other agents.
2. Its duty is to start all the other agents by sending a message which signals start.
3. Other agents communicate with each other through the facilitator.
4. The facilitator manages and maintains the environment.
5. The facilitator additionally maintains records of all the agents that exist in the system.
6. The facilitator may execute additional agent requests. .

3.2.2 Environment

The Hive environment includes all of the active agents. It may also include other entities such as stores, products, and devices. The environment provides atomic update and query methods for these entities.

3.2.3 Initiator

An initiator creates a Facilitator agent. It is considered the point of entry in the simulation. The initiator is not an Akka actor and is implemented as a Scala object.

3.3 Hive agent structure and state machine behavior

Hive defines an agent behavior using state machines. A hive agent has different states that allow an agent to maintain behavior specific to a state. A hive agent

receives and sends specific messages in a particular state. This ensures a consistent reaction to messages received by an agent in a specific state.

Hive adopts a simple form of state machine behavior in which changing of a state is based on satisfying a condition that leads to an action, which leads to state transition.

The following table shows three types of state transitions:

```
(State1, message) => (Action, State2)
(State1, Condition Check) => (Action, State2)
(State1, message) => (Action, State1)
```

1. In the first example, an agent in state1 receives a message. It performs the indicated action and transitions to state2. For example, in the market model, when it receives a message from a buyer who is willing to buy services, an idle agent updates its interactor attribute to that agent and transitions to sell state.

2. In the second example, an agent in state1 checks to see if its attributes satisfy some condition. If they do, the agent performs some action and transitions to state2.For example, in the market model, upon gathering enough resources, an agent updates its repository and transitions its state from ‘‘buy’’ to ‘‘sell’’

3. In the third example, an agent in state1 receives a message. It performs the indicated action and remains in state1. For example, in the market model, an agent in ‘‘sell’’ state updates its resource repository after every transaction message from a buyer agent. If it still has enough resources to sell, then it continues to be in ‘‘sell’’ state.

3.3.1 State Machine Behavior for Agent

Hive agents have attributes that allow them to change their behavior between messages and processes. This switchable behavior is the most salient feature and provides fundamental capabilities to build finite state machines. This is implemented by the become/unbecome functionality of Akka actors.

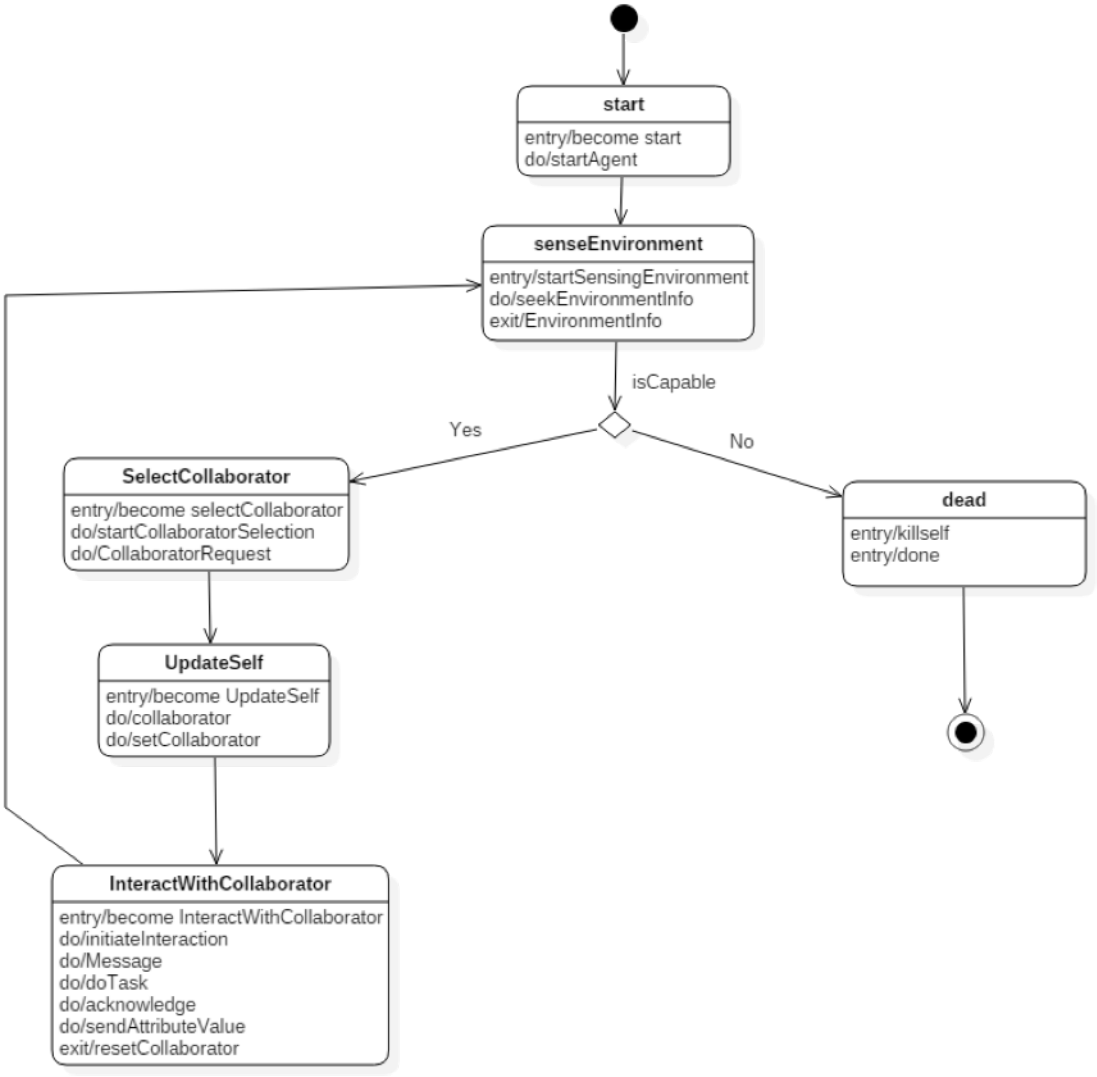

Figure 8: State Behavior of Hive Agents

1. Upon creation, an agent enters into a ‘‘start’’ state. This state allows an agent to start itself and initialize its attributes.

2. After initializing its attributes, an agent will sense the environmental status and transition into ‘‘sense environment’’ state. This state allows an agent to judge its internal state by checking attributes. If an agent is capable of carrying

out interactions, it will enter into ‘‘select collaborator’’ state. If an agent is not capable of interacting, then it will kill itself and ask the facilitator to remove it from the environment.

3. In the ‘‘select collaborator’’ state, an agent will send a request for a collaborator to the Facilitator. The facilitator will select a collaborator for an agent either randomly or based on some strategy. After sending a request to the facilitator, the agent will transition itself to ‘‘UpdateSelf’’ state.

4. An agent will receive a collaborator in ‘‘UpdateSelf’’ state and set its collaborator to the reference of the received collaborator.

5. After setting its collaborator, an agent will transition to the ‘‘initiate interaction’’ state and will initiate the interaction. An agent will ask the Facilitator to notify the chosen collaborator about the interaction interest from this particular agent. On the other hand, if collaborator agent is busy interacting with another agent, it can send a busy message to the facilitator, signaling the facilitator to find a different agent to be a collaborator.

6. After initiating interaction, an agent waits for the interaction to complete. A successful interaction will return acknowledgment from an agent to the facilitator stating completion of the interaction. For example, in the market model, upon buying a service from an agent in sell state, an agent in buy state sends an acknowledgement stating the completion of transaction. Also, an agent will update the facilitator with its attribute values so that the facilitator can have statistical information about the environment. Upon receiving acknowledgment, the facilitator will signal the interacting agents to reset their collaborators and prepare for other interactions. This is needed to consistently complete the

current round of interaction.

7. The agent will now sense the environment again, move to step one, and repeat the process.

3.3.2 State Machine behavior for Facilitator-Agent

The Facilitator agent is designed with a simple state behavior. It has a single state, which is the ‘‘available’’ state. The facilitator handles all types of requests in this state.

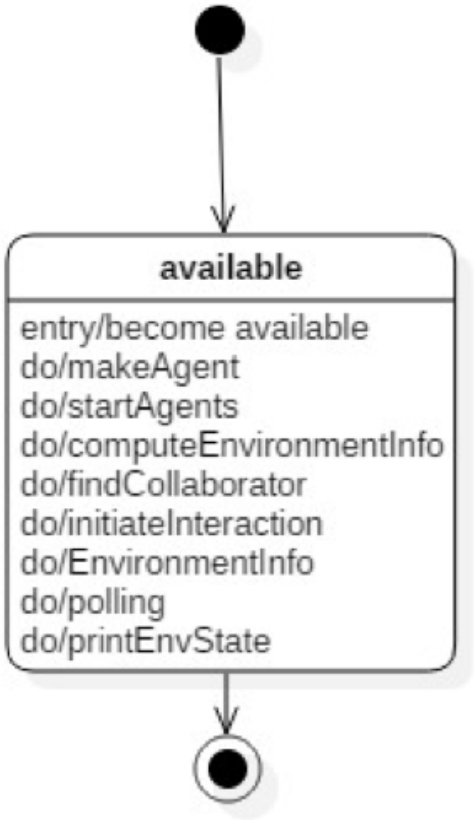

Figure 9: State behavior of a facilitator

3.4 Hive Implementation

Hive framework has following package structure.

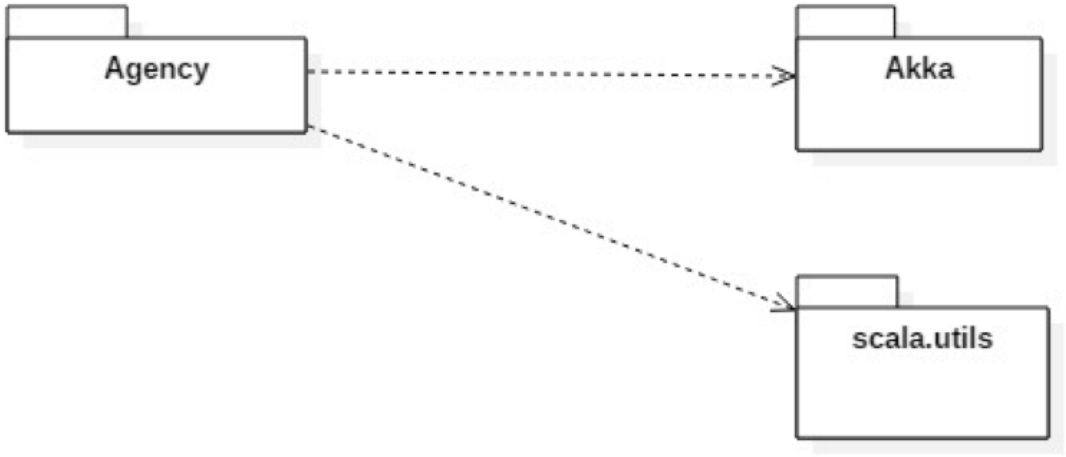

Figure 10: Hive Framework package diagram

Agency Package- The agency package contains the Hive framework code. This package must be imported in your simulation's Scala source code.

This contains three Scala files.

1. Agent.scala- This contains the state machine behavior for Agent Actor.
2. Facilitator.scala-This contains the facilitator behavior.
3. Initiator.scala- An initiator will create Facilitator actor.

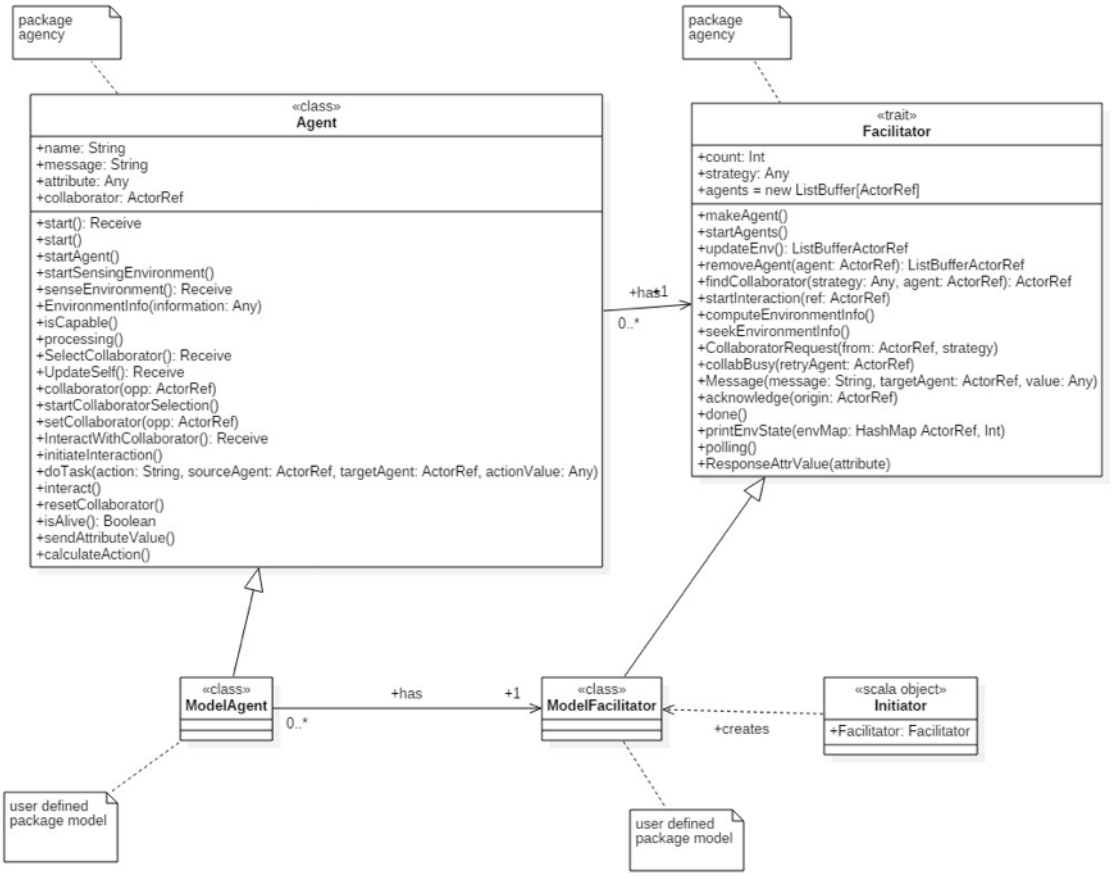

Figure 11: The above class diagram shows how Hive is customized by extending the Agent and Facilitator classes

Initiator- Initiator will create Facilitator actor.

Environment- Facilitator maintains the environment, which is an updated list of agents in the system. It has the following utility methods:

1. **updateEnv():ListBuffer[ActorRef]**= This method updates current list of agents maintained by facilitator.

2. **removeAgent(agent:ActorRef):ListBuffer[ActorRef]**= This method removes a particular agent from the environment.

Facilitator- The Facilitator trait is a factory for creating agents. Facilitator uses the abstract factory method design pattern to create model agents. It maintains the environment, which is the list of agents created. It has methods to facilitate agents with these requests:

1. **makeAgent()** Instantiation of a specific Hive agents is managed by factory method ‘‘makeAgent()’’ of Facilitator sub class. This method makes agents of type general Hive Agent or specific agent implementing Hive Agent. While implementing the Hive framework, the user needs to override this abstract factory method to create specific agents according to model.

2. **startAgents()** This method starts all the agents created.

3. **findCollaborator(strategy:Any,agent:ActorRef):ActorRef** This method takes an actor reference as an argument and strategy for finding a collaborator. It then finds a collaborator for this actor agent. This method can be overridden to find a collaborator according to some specific strategy.

4. **startInteraction()** After getting a collaborator, this method will signal an agent to start interacting with the collaborator.

5. **computeEnvironmentInfo()** Upon request, this method will send environment information to agents, which contains list of agents that are currently active in the system.

6. **available: Receive ()** This method implements the Facilitator agentâĂŹs message loop/mailbox in available state. A facilitator can receive following messages in available state:

- **seekEnvironmentInfo()-** This message is sent from an agent who is seeking environment information to initiate an interaction. The Facilitator will call computeEnvironmentInfo() upon receiving this message.
- **collaboratorRequest(from:ActorRef, strategy:Any)-** This message is sent from an agent that needs a collaborator to interact with. The Facilitator will call findCollaborator() upon receiving this message.
- **collabBusy(retryAgent:ActorRef)** If the collaborator selected for an agent is busy interacting with another agent, then that agent will send a collabBusy message to the facilitator instructing the facilitator to find a new a collaborator for the agent.
- **Message(message,targetAgent,Value)-** This message comes from an agent who wants to send a message to particular agent. The Facilitator forwards this message to the destination agent. The sending agent includes the value of the action for the target agent in this message.
- **Acknowdledge()-** This message is sent from an agent who has successfully completed an interaction with a collaborator. After receiving this message, the Facilitator will send a resetCollaborator() message to both source and destination agents.
- **Polling()-** This method will send a ‘‘sendAttributeValue()’’ message to all the agents.
- **sendAttributeValue()-** This message will direct the agents to send their attribute values to the facilitator to create overall statistics of the environment.
- **printEnvState()-** This method will print the statistical information of the environment in regular intervals of ten seconds.

- **Done()-** This message is sent from an agent that is dying. The Facilitator will remove this agent from environment by calling the removeAgent method.

Agent The agent class contains the following attributes:

1. **Name** This is a string that denotes the name of an agent.
2. **Message** This is a string that represents a message that an agent has just received or sent.
3. **Collaborator** This is an actor reference that represents the collaborator with which an agent wants to interact.
4. **Facilitator** This is an actor reference that represents the facilitator or the parent of an agent.
5. **Attribute** This attribute/attributes decides various decisions of an agent's life.

The agent class implements the following methods-

1. **start:Receive()=** This method represents the start state of an agent. It is a message loop implementation for agentâĂŹs start behavior. An agent, when instantiated, transitions into start state. The agent can receive a start message from the Facilitator, which will direct it to call the startAgent() method and calls startSensingEnvironment() with a timeout of 10 milliseconds.
2. **startAgent()=** This method initializes an agent's attributes. This can be overridden according to the model's requirements.

3. **startSensingEnvironment()**= This method transitions the agent into senseEnvironment state and sends seekEnvironmentInfo() to the Facilitator.

4. **senseEnvironment:Receive()**= This represents the behavior of an agent in senseEnvironment state. An agent can receive EnvironmentInfo(information:Any) from the Facilitator. This start will call ''isCapable()'' method.

5. **isCapable()**= This method will check the ability of an agent to interact. It checks an attribute value. If this value is above the threshold value of a model, then it calls the processing() method.

6. **isAlive()**= This method will check if an agent is alive by checking attribute values. If an agent is alive, it returns true. If an agent is incapable of surviving, then the agent will kill itself, send a done() message to facilitator and return false.

7. **killself()**= This method stops an agent by calling stop (inbuilt Akka method).

8. **Processing()**= This method will transition the agent to SelectCollaborator state and send itself a startCollaboratorSelection() message.

9. **sendDone()**= This method sends the Facilitator a done() message, signaling that it will now exit the system.

10. **SelectCollaborator:Receive()**= This state represents collaborator selection state. In this state, an agent can receive a startCollaboratorSelection() message.

11. **startCollaboratorSelection()**= Upon receiving this message, the agent will send a CollaboratorRequest() message to the facilitator and transition itself to UpdateSelf state.

12. **UpdateSelf:Receive()**= In this state, an agent can receive a collaborator from the facilitator. The agent then sets this collaborator reference to its collaborator attribute and transitions into InteractWithCollaborator state

13. **setCollaborator(co:ActorRef)**= This method will check if the agent is free to interact. If the agent is available, then agent will update its collaborator reference and proceed to interact. If the agent is not available, then the agent will send a collabBusy message to the facilitator, instructing it to find a different collaborator.

14. **InteractWithCollaborator:Receive()**= In this state, an agent will receive intiateInteraction() message from the facilitator and send a message to the facilitator for the collaborator. Also, an agent can receive a doTask(action:String, orgin:ActorRef,destination:ActorRef,value) message from the facilitator.

15. **initiateInteraction()**= This method will send a message to a collaborator and send the action value to the facilitator.

16. **calculateAction()**= This method will calculate the action value that an agent wants to send to a collaborator. The agent will calculate the action value according to its attributes. This method can be overridden by the user.

17. **doTask(action:String, orgin:ActorRef,destination:ActorRef,actionValue)**= This message performs the interaction action and calls the interact() method with action value. Upon completing interaction, it will send an acknowledge message to the facilitator, reset its collaborator, and transition into senseEnvironment state.

18. **sendAttributeValue()** This message is sent by the facilitator requesting that an agent send its attribute values to generate statistical information about

environment. An agent, in return, will send the ResponseAttrValue(attribute) to the facilitator.

19. **resetCollaborator()**= This method will reset the current collaborator of an agent to null.

20. **Interact(actionValue)**= This method performs an interaction action. It comes with an action value, which can modify an agent's attribute. This method can be overridden according to the model's implementation.

A possible extension of Hive will use this agency package and extend the Agent class and Facilitator trait. The user can create a model package and import an agency package in it. A model-specific agent can be created by extending the Agent class, and a model-specific Facilitator can be created by extending the Facilitator trait. The next chapter will explain the exact implementation details of Hive using two agent based models as examples.

CHAPTER 4

Hive Implementation and Demonstration

For the scope of this project, we have implemented two Agent Based Models: Fight Club and Wealth Market.

4.1 Trends in real world Agent Based Model Simulations

Simple and predictable interactions between agents can lead to interesting global patterns. These global patterns can depict evolutionary trends in a population of agents, such as survival of the fittest, diffusion of information, social patterns, attracting mates, market crashes, competing businesses, equilibrium between demand and supply in a market, and many more.

It is difficult to create and program a complex system that is capable of observing a global pattern such as survival of the fittest. But it is easy to model the simple behavior of an agent by defining simple rules, rather than modelling the entire system. For example, programming a model that can create agents that fight with random neighbors is easier than programming a system that can predict the ‘‘last surviving agent’’.

4.2 Fight Club

A fight club is an agent based model in which fighter agents fight with each other until only one survives. All of the fighters are initialized with good health (health=100). During a fight, a fighter will give a blow to an opponent fighter. This will decrease the opponent's health. If a fighter is in bad health (health=0), then it will kill itself and exit the fight club. The agent that survives till until the end

will demonstrate the survival of the fittest theory. As the simulation proceeds, Hive output will give the following statistical information of the fighters' health at various times:

1. Number of fighters having health >100.
2. Number of fighters having health between 70 and 100.
3. Number of fighters having health between 50 and 70.
4. Number of fighters having health between 20 and 50.
5. Number of dead fighters (health =0).

This model can be the basis for much more complex social models wherein fighters can look for the most desirable available fighter to mate with. A fighter couple can also pass their fighting skills and strategies to their offspring, thus producing a better and smarter fighter population. As the simulation proceeds, we can learn which fighting strategies are the most successful.

4.2.1 Fighter Agents

A fighter agent can be implemented as extensions of Hive agents. A fighter agent will have a health attribute, which is the driving factor for a fighter when making various decisions. A fighter is capable in a fight only if it has good health. In good health, a fighter can ask the Facilitator to find an opponent. A fighter can calculate its blow and then fight with the opponent by giving it a blow, which will decrement the health of the opponent. Giving a blow decrements the fighter's health as well. Once done fighting, the fighter will again check if it has good health (i.e., health > 0) and look for another opponent. If a fighter is in bad health (i.e., health <= 0) then it

dies and leaves the fight club. It notifies the Facilitator Fighter about its departure and kills itself.

4.2.2 Facilitator Agent

This agent creates, starts, and monitors the fighter agents.

4.2.3 Customization and required changes

The Hive framework is designed by following the ‘‘Open-Closed’’ principle, which states that systems should be open to extension but closed to modification. In this case, we want to be able to extend the Hive framework to Fight Club without modifying the Hive code. To accomplish this, users of the framework are required to override some methods.

The following steps are required by the programmer to customize the Hive Framework-

1. Installing Scala and Akka
2. Importing the agency package into the project.
3. Extending Agent class to a model specific agent class. For example, in Fight Club Fighter extends Agent.
    - Overriding isCapable, interact and calaculateAction methods
    - The Fighter's isCapable method determines if the fighter is capable of delivering a blow. Its capability is determined by its health. If a fighter is capable, then it will continue its processing; otherwise, it will kill itself.
    - The Fighter's interact method will perform the interaction. This method

comes with the âĂIJblowâĂİ value from a fighter who is the opponent of this fighter. The health of the fighter is decremented by the blow value. If this leads to insufficient health (health<=0), then the fighter kills itself.

- The calculateAction method will calculate the value of the action, which is the strength of the blow. This blow value will be included in the message sent to the opponent.

4. Extending the Facilitator trait by implementing a model specific facilitator. For example, in Fight Club, Facilitator extends Facilitator.

   - The makeAgent method, which is the abstract factory method for creating agents, will be overridden to create fighters. The method will prompt the user to enter the number of fighters it wants to create in its simulation.

4.2.4 Implementation Details of Fight Club

In this section we will sketch the Fight Club extensions described above.

1. FighterFacilitator.scala

```
class FighterFacilitator extends Facilitator with Actor{



override  def makeAgent()={


println(‘‘Enter the number of fighters you want to create’’)
     val num=readInt()
     this.count=num
```

```
      println("CREATING AGENTS")
       var i:Int=0
       for(i<-1 to num)
       {
         val Agent= context.actorOf(Props(new
             Fighter()),name=‘‘Fighter’’+i)
         agents.+=:(Agent)

       }

       this.updateEnv()
  }
}
```

The Fighter Facilitator will create fighters by calling the constructor for Fighter agent. The Facilitator needs to maintain the created fighters in the list to maintain the environment. Also, after adding the fighters to the list, the Facilitator needs to update the environment.

2. Fighter.scala

```
package fighter
import agency._

class Fighter extends Agent{
  import context._
  name=self.path.name
  var health:Int=100
```

```
 var blow:Int=0


 override def isCapable()={
     //this will check the health
     if(isAlive()==true)
     {
     health= health-2
     processing()
     }
     else
     {


       killself()
     }
   }
override def calculateAction():Int={
  if(health>0)
  {
   blow = r.nextInt(High-Low) + Low;
   return blow
  }


  else return 0
}
 override def interact(blow:Int)={
   if(isAlive()==true)
     {
```

```
    //receiving a blow from other fighter
    if(health>blow)
    {
    this.health=health-blow
    if(health<0)
    {

      killself()

    }
    }
    else
    {

      killself()
    }
  }
  }
}
```

The isCapable() method will check if the fighter is alive or not. If alive, the fighter will decrement its health before continuing its processing. A fighter's health will decrease when it indulges in a fight. A fighter will call processing method, which will switch its state to select collaborator state, and then ask the facilitator to find an opponent for it. If a fighter is in bad health (attribute<=0), then it kills itself.

Also, while initiating its interaction in InteractWithCollaborator state, the

fighter will calculate the blow through calculateAction() method, which it can give to its opponent. this blow value will be included in the message Message(message,collaborator,value) being sent to the facilitator for the opponent. The facilitator will forward this message to the opponent through a doTask(‘‘Interact’’,sourceAgent,targetAgent,actionValue) message.

On the other hand, when an opponent receives a doTask(‘‘Interact’’,sourceAgent,targetAgent,actionValue) message, the facilitator will call the interact method and opponent will receive the blow and decrement its health by the blow value. If a fighter is in bad health during the interaction, then it kills itself.

3. Initiating launching the simulation through Initiator.scala

```
package fighter
import agency._

object Initiator extends App{
 val system = ActorSystem(‘‘system’’)
      println(‘‘Starting facilitator’’)
      var facilitator : ActorRef =
          system.actorOf(Props[FighterFacilitator],name =
          ‘‘facilitator’’)

}
```

Initiator object will create the Actor System and create a facilitator fighter within the system. As discussed above, the facilitator will create a bunch of fighters.

Output Observation-

Upon running Fight Club, the user can enter the number of fighters to be created. This example demonstrates the creation of 40 fighters. Fighters, when created, will be initialized in good health (health=100).

```
Starting facilitator
Enter the number of fighters you want to create
40
CREATING AGENTS
Fighter39 starts to Act
Fighter34 starts to Act
Fighter37 starts to Act
Fighter38 starts to Act
.............................
Time = 1
total    is : 5000
Agents    0< Values <20   are 0
Agents   20< Values <50   are 0
Agents   50< Values <70   are 0
Agents   70< Values <100  are 50
Agents       Values >100  are 0
Agents       dead         are 0
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 2
total    is : 4900
Agents    0< Values <20   are 0
Agents   20< Values <50   are 0
Agents   50< Values <70   are 0
Agents   70< Values <100  are 50
Agents       Values >100  are 0
Agents       dead         are 0
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 19
total    is : 2632
Agents    0< Values <20   are 0
Agents   20< Values <50   are 16
Agents   50< Values <70   are 0
Agents   70< Values <100  are 20
```

```
Agents       Values >100  are 0
Agents       dead         are 14
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 26
total    is : 2036
Agents    0< Values <20   are 6
Agents   20< Values <50   are 8
Agents   50< Values <70   are 4
Agents   70< Values <100  are 15
Agents       Values >100  are 0
Agents       dead         are 17
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 31
total    is : 1603
Agents    0< Values <20   are 6
Agents   20< Values <50   are 10
Agents   50< Values <70   are 3
Agents   70< Values <100  are 11
Agents       Values >100  are 0
Agents       dead         are 20
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 38
total    is : 1289
Agents    0< Values <20   are 4
Agents   20< Values <50   are 6
Agents   50< Values <70   are 2
Agents   70< Values <100  are 10
Agents       Values >100  are 0
Agents       dead         are 28
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 43
total    is : 885
Agents    0< Values <20   are 4
Agents   20< Values <50   are 8
Agents   50< Values <70   are 2
Agents   70< Values <100  are 5
Agents       Values >100  are 0
Agents       dead         are 31
```

```
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 50
total    is : 496
Agents    0< Values <20   are 3
Agents   20< Values <50   are 3
Agents   50< Values <70   are 2
Agents   70< Values <100  are 3
Agents       Values >100  are 0
Agents       dead         are 39
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 62
total    is : 45
Agents    0< Values <20   are 1
Agents   20< Values <50   are 1
Agents   50< Values <70   are 0
Agents   70< Values <100  are 0
Agents       Values >100  are 0
Agents       dead         are 48
-----------------------------------------------------------------
```

A typical output shows the statistics of fighters' health at regular intervals.

Observation

The output shows that the majority of the fighters initially show a steep decrease in their health. Since initially all of the fighters are in good health (100), the blow values for the fighters are high. This leads to substantial decrease in the health of the fighters (at time=19). We can observe a steep decrease in health for more than half of the agent population. The following graph depicts the health distribution at the start, middle, and end of the model.

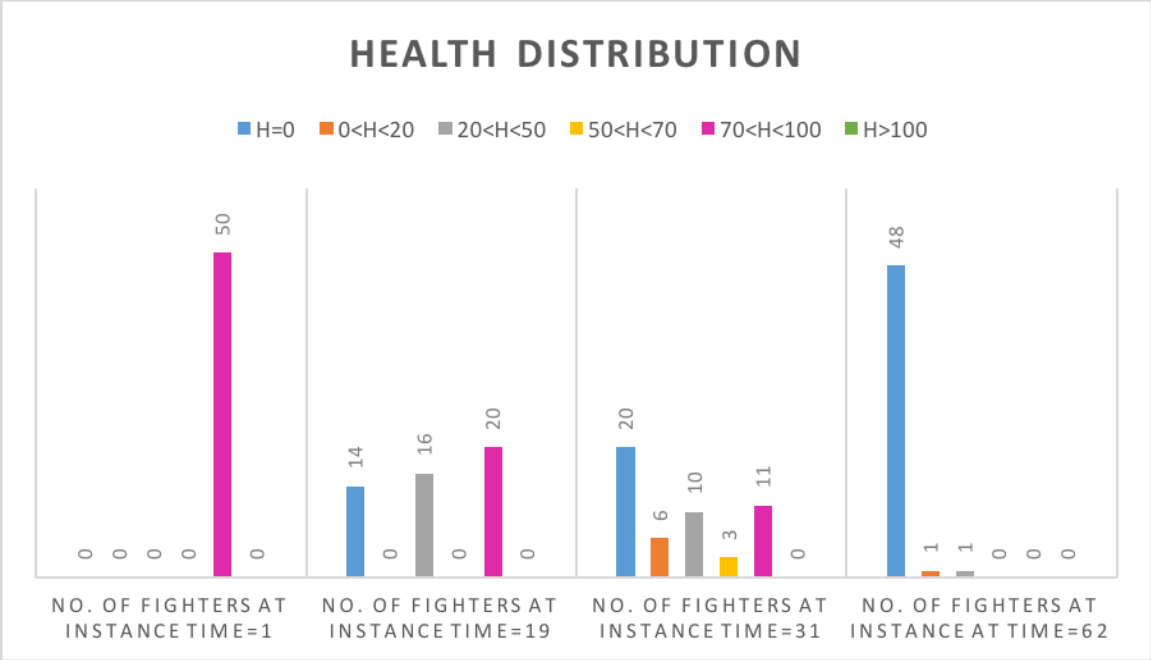

Figure 12: Fight Club Output

Fight Club is a relatively simple example of much more complex agent based models. This model can be extended by giving the fighters the ability to choose their fight strategy. For example, a fighter can choose to give light blows to some fighters and can choose to give more powerful blows to others. Also, users can change the way a fighter calculates its blow value or change the amount by which it decrements its own health and observe the output change.

4.3 Wealth Market

In a wealth market, customer agents purchase services from each other. The price of a service is fixed at $5. Initially all customer agents have $100. (This is an attribute called wealth.) As the simulation proceeds, the Hive output displays the numbers of low, middle, and high-income customers. One might expect that the wealth of a customer agent remains relatively constant, since a customer agent is giving and

receiving dollars. Unfortunately, even in a simple market, we eventually see an inverse power distribution of wealth, with many poor agents and a few super-rich agents.

4.3.1 Customer Agents

Customer agents can be implemented as Hive agents. In a wealth market, the wealth attribute of the customer agent is the driving force for staying in the market. A customer agent is capable of buying services only if it has enough wealth. When not broke, a customer can ask the Facilitator to find it a collaborator customer agent to interact with. During a transaction, a customer agent will lose some wealth and the collaborator customer agent will gain some wealth. Once finished interacting, the customer agent will again look for another collaborator. If a customer agent is broke (wealth=0), it notifies the Facilitator about its departure from the market and kills itself.

4.3.2 Facilitator Agent

This agent creates, starts, and monitors the customer agents.

4.3.3 Customization and required Changes

Following customization is required by the programmer to use Hive Framework-

1. Installing Scala and Akka
2. Importing the agency package into the project.
3. Extending Agent class to a model specific agent class. For example, in services market Customer extends Agent.
    - Overriding isCapable, interact and calaculateAction methods.

- The Customer 's isCapable method determines if the customer is capable of doing a transaction. Its capability is determined by its wealth. If a customer is capable then it will continue its processing otherwise it will kill itself.
- The Customer's interact method will perform the interaction. This method comes with the âĂIJtransactionâĂİ value for a collaborator customer. Wealth of the collaborator customer is incremented by the transaction value.

4. The calculateAction method will calculate the value of the transaction, that is $5. This transaction value will be included in the message sent to collaborator.
5. Extending the Facilitator trait by implementing model specific facilitator.
6. The makeAgent method which is the abstract factory method for creating agents will be overridden to create customers. The method will prompt user to enter the number of customers it wants to create in its simulation.

4.3.4 Implementation details of Wealth market

Services market model will be implemented as following three Scala classes.

1. Customer.scala

```
package wealthExample
import agency._
class Customer extends Agent{
var wealth:Int = 100
```

```
import context._
  name=self.path.name
  override def isCapable()={
      //this will check the wealth
  if(isAlive()==true)
    {
      processing()
    }
     else
      {
        killself()
      }
    }
  override def calculateAction():Int={
    var trsction:Int=5
    wealth=wealth-5
    return trsction
 }


  override def interact(transaction:Int)={
    if(isAlive()==true)
      {
    wealth=wealth+transaction
     }
  }
}
```

The isCapable() method will check if the customer is broke or not. If not broke, it continues its processing. A customer will call its processing method, which will switch its state to select collaborator state. It then asks the facilitator to find a collaborator for it. If the customer is broke (wealth attribute<=0), then it kills itself. Also, the customer, while initiating its interaction in InteractWithCollaborator state, will calculate the transaction through calculateAction() method, which it can give to its collaborator. This transaction value will be included in the message Message(message,collaborator,value) that is sent to facilitator for the collaborator. The facilitator will forward this message to the collaborator through a doTask (''Interact'', sourceAgent,targetAgent,actionValue) message.

On the other hand, when a collaborator receives a doTask("Interact", sourceAgent,targetAgent,actionValue) message from the facilitator, it will call the interact method. The collaborator will then receive the transaction amount and increment its wealth by the transaction amount.

2. WealthFacilitator.scala

```
package wealthExample
import agency._

class WealthFacilitator extends Facilitator with Actor{
override  def makeAgent()={

     println("Enter the number of agents you want to create ")
     val num=readInt()
     this.count=num
     println("CREATING AGENTS")
```

```
        var i:Int=0
        for(i<-1 to num)
        {
          val Agent= context.actorOf(Props(new
              Customer()),name="Agent"+i)
          //context.watch(Agent)
          agents.+=:(Agent)

        }

        this.updateEnv()

      }
}
```

The facilitator will create fighters by calling constructor for Customer agent. The facilitator needs to maintain the created customers in the list to maintain the environment. Also, after adding the customers to the list, the facilitator needs to update the environment.

3. Initiatior.scala

```
package wealthExample
import agency._

object Initiator extends App{
 val system = ActorSystem("system")
      println("Starting facilitator")
```

```
        var facilitator : ActorRef =
            system.actorOf(Props[WealthFacilitator],name = "facilitator")

    }
```

The Initiator object will create the Actor System and create the Facilitator within the system. As discussed above, the facilitator will create a bunch of customers.

Output- Upon running Wealth Market, the user can enter the number of agents to be created. The following simulation runs for 50 customers.

```
  Starting facilitator
Enter the number of agents you want to create
50
CREATING Customers
Customer40 starts to Act
Customer50 starts to Act
Customer48 starts to Act
Customer46 starts to Act
Customer45 starts to Act
-----------------------------------------------------------------
Time = 1
total    is : 5000
Agents    0< Values <20   are 0
Agents   20< Values <50   are 0
Agents   50< Values <70   are 0
Agents   70< Values <100  are 50
Agents       Values >100  are 0
Agents       dead         are 0
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 2
total    is : 5000
Agents    0< Values <20   are 0
Agents   20< Values <50   are 0
Agents   50< Values <70   are 0
Agents   70< Values <100  are 50
```

```
Agents       Values >100  are 0
Agents       dead         are 0
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 588
total    is : 5000
Agents    0< Values <20   are 0
Agents   20< Values <50   are 1
Agents   50< Values <70   are 2
Agents   70< Values <100  are 24
Agents       Values >100  are 23
Agents       dead         are 0
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 641
total    is : 4995
Agents    0< Values <20   are 0
Agents   20< Values <50   are 1
Agents   50< Values <70   are 2
Agents   70< Values <100  are 25
Agents       Values >100  are 22
Agents       dead         are 0
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 1725
total    is : 4860
Agents    0< Values <20   are 2
Agents   20< Values <50   are 7
Agents   50< Values <70   are 3
Agents   70< Values <100  are 16
Agents       Values >100  are 22
Agents       dead         are 0
-------------------------------------------------------------------
-------------------------------------------------------------------
Time = 2190
total    is : 4850
Agents    0< Values <20   are 4
Agents   20< Values <50   are 4
Agents   50< Values <70   are 6
Agents   70< Values <100  are 13
Agents       Values >100  are 23
Agents       dead         are 0
```

```
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 3336
total    is : 4870
Agents    0< Values <20   are 6
Agents   20< Values <50   are 8
Agents   50< Values <70   are 3
Agents   70< Values <100  are 12
Agents       Values >100  are 21
Agents       dead         are 0
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 4964
total    is : 4865
Agents    0< Values <20   are 18
Agents   20< Values <50   are 2
Agents   50< Values <70   are 1
Agents   70< Values <100  are 8
Agents       Values >100  are 21
Agents       dead         are 0
-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 5710
total    is : 4845
Agents    0< Values <20   are 22
Agents   20< Values <50   are 4
Agents   50< Values <70   are 2
Agents   70< Values <100  are 4
Agents       Values >100  are 18
Agents       dead         are 0

-----------------------------------------------------------------
-----------------------------------------------------------------
Time = 6064
total    is : 4840
Agents    0< Values <20   are 24
Agents   20< Values <50   are 2
Agents   50< Values <70   are 1
Agents   70< Values <100  are 2
Agents       Values >100  are 14
Agents       dead         are 0
-----------------------------------------------------------------
```

The above output can be depicted using below graph.

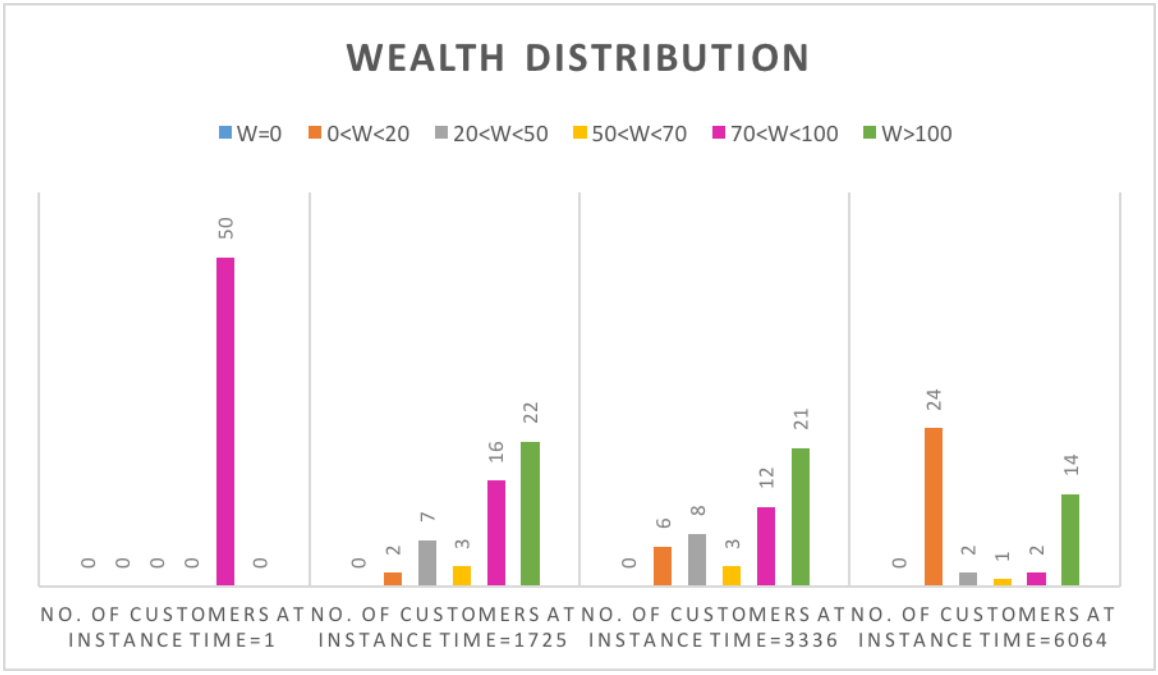

Figure 13: Wealth Market Output

Output Observation:

Upon running the wealth market model, the user might expect the customer's wealth to be constant, as the customer is both buying and receiving services at the same price. But the output shows that after some time, a few customers get rich, while most get very poor. (i.e. At time =6064, number of customers with wealth between $0 and $20 is 24, and the number of customers with wealth greater than $100 is 14). While all customer agents are buying services, some of the customer agents are lucky enough to be randomly picked to receive money from many customers' multiple times. The Wealth Market Model can be extended by changing the way a customer calculates

the transaction amount or by changing the way a customer selects a collaborator customer. For example, the rules may be altered so that a customer can only look for rich customer collaborators. The user can then observe the difference in the output.

CHAPTER 5

Future Work and Enhancements

5.1 Heterogeneous Agent Societies Support

In the future, this framework can be enhanced to support the interactions between Heterogeneous types of agents. This could be the next step in moving closer to Net Logo features [3], which allow interactions between different breeds of agents that have different behaviors, such as ‘‘turtles’’ with ‘‘patches’’ [4]. (Turtles are mobile agents, whereas patches are stationary). This feature would allow the users to implement different agent societies that have different rules for communicating with each other. Heterogeneous agent communication support will allow latitude for the implementation more complex agent based models using the Hive Framework.

5.2 Graphical User Interface Support

The Hive Framework needs a graphical user interface. This feature will allow the observer to visually perceive the developments in an agent based model.

CHAPTER 6

Conclusion

6.1 Conclusion

In this thesis, we designed an Agent Based Modeling Framework that can simulate an agent based model in a variety of domains. We also explored a new technology, the Akka library an Actor support in Scala language. Our framework uses the Actor Model to implement agents and their interactions. By employing this model, we have avoided various synchronization problems that arise when building concurrent systems. Our framework has been successfully implemented by two agent based model examples - Fight Club and Wealth Market. Previous work in this field, including NetLogo, has been written in Scala and Java using Scala Actors. Scala actors are deprecated, and since Hive is written in Scala and Akka, this could be a step towards building a Netlogo type agent based modeling platform with the support of a newer and better Actor library. Also, the user only needs intermediate knowledge of the Scala programming language and does not require knowledge of a domain specific language in order to develop simulations through Hive. The Hive Framework has evolved and developed while keeping extensibility in mind. It can be easily extended to a specific Agent Based Model, giving leverage to programmers to add more details to the simulated agent based model.

LIST OF REFERENCES

[1] William Rand,Uri Wilensky,Retrieved December 11, 2015 *An Introduction to Agent-Based Modeling Modeling Natural, Social, and Engineered Complex Systems with NetLogo*

[2] Luck, M., Ashri, R., *Agent-based software development. Boston: Artech House (2004).*

[3] Wilensky, U ,NetLogo Home Page. Retrieved November 11, 2015, from `https://ccl.northwestern.edu/netlogo/`

[4] Tisue, S., Wilensky, U., NetLogo: A Simple Environment for Modeling Complexity (p. 10). Boston.

[5] Programming Guide. (n.d.). Retrieved November 11, 2015, from `https://ccl.northwestern.edu/netlogo/docs/programming.html`

[6] Wilensky, U. (1997) NetLogo Ants model `http://ccl.northwestern.edu/netlogo/models/Ants` . Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[7] Wilensky, U. (1999). NetLogo. `http://ccl.northwestern.edu/netlogo/` Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[8] NetLogo Web. (n.d.). Retrieved November 11, 2015, from `http://www.netlogoweb.org/launch#http://www.netlogoweb.org/assets/modelslib/SampleModels/Biology/Ants.nlogo`

[9] Isaac, A. (n.d.). NetLogo A Basic Introduction. Retrieved November 11, 2015, from `https://subversion.american.edu/aisaac/notes/netlogo-basics.xhtml`

[10] Todd, A., Keller, A., Lewis, M., Kelly, M. (n.d.). Multi-agent System Simulation in Scala: An Evaluation of Actors for Parallel Simulation. Retrieved November 11, 2015, from `http://www.worldcomp-proceedings.com/proc/p2011/PDP4783.pdf`

[11] Scala Actors: A Short Tutorial. (n.d.). Retrieved November 11, 2015, from `http://www.scala-lang.org/old/node/242`

[12] Westheide, D. (n.d.). Daniel Westheide. Retrieved November 11, 2015, from `http://danielwestheide.com/blog/2013/02/27/the-neophytes-guide-to-scala-part-14-the-actor-approach-to-concurrency.html`

[13] Software framework. (n.d.). Retrieved November 11, 2015, from https://en.wikipedia.org/wiki/Software_framework

[14] Actors. (n.d.). Retrieved November 11, 2015, from http://doc.akka.io/docs/akka/2.4.0/scala/actors.html#Become_Unbecome

[15] Integrating Play for Java and Akka. (n.d.). Retrieved November 11, 2015, from http://www.informit.com/articles/article.aspx?p=2228804

[16] Actor model. (n.d.). Retrieved November 11, 2015, from https://en.wikipedia.org/wiki/Actor_model

[17] Akka and the Java Memory Model. (n.d.). Retrieved November 11, 2015, from http://doc.akka.io/docs/akka/snapshot/general/jmm.html

[18] Agent Based Modelling and Simulation using State Machines. (n.d.). Retrieved November 11, 2015, from http://users.uom.gr/~iliass/projects/NetLogo/TSTATES/papers/ABMSUsingStateMachines.pdf

[19] Petabridge. (n.d.). Retrieved November 11, 2015, from https://petabridge.com/blog/akka-actors-finite-state-machines-switchable-behavior/

[20] Actors. (n.d.). Retrieved November 11, 2015, from http://doc.akka.io/docs/akka/2.4.0/scala/actors.html

[21] Dispatchers. (n.d.). Retrieved November 11, 2015, from http://doc.akka.io/docs/akka/2.4.0/scala/dispatchers.html

[22] Complex Systems Modeling: Using Metaphors From Nature in Simulation and Scientific Models. (n.d.). Retrieved January 22, 2016, from http://www.informatics.indiana.edu/rocha/complex/csm.html

[23] About Complex Systems. (n.d.). Retrieved January 15, 2016, from http://necsi.edu/guide/

[24] Jennings, N. R. (2001). AN AGENT-BASED APPROACH FOR BUILDING COMPLEX SOFTWARE SYSTEMS.ÂăCommunications Of The ACM,Âă44(4), 35-41. doi:10.1145/367211.367250

[25] About Complex Systems. (n.d.). Retrieved January 15, 2016, from http://necsi.edu/guide/

[26] Multi-agent system. (n.d.). Retrieved January 10, 2016, from https://en.wikipedia.org/wiki/Multi-agent_system

[27] Dr. Roman ÅăPERKA, PhD Multi Agent System and Agent Based Model. (n.d.). Retrieved January 12, 2016, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.4592&rep=rep1&type=pdf

[28]] Multi Agent System and Agent Based Model. (n.d.). Retrieved January 12, 2016, from http://kes2015.kesinternational.org/cms/userfiles/is13.pdf

[29] Scientific Model. (n.d.). Retrieved January 13, 2016, from https://www.ifa.hawaii.edu/users/iroussev/Lecture03_110f2009.pdf

[30] Danny Weyns,Michael Schumacher,Alessandro Ricci,Mirko Viroli,Tom Holvoet, Environments for Multiagent Systems Report AgentLink Technical Forum Group,Ljubljana, February 2005 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.4592&rep=rep1&type=pdf

[31] Multi-Agent Systems. (n.d.). Retrieved November 6, 2015, from http://cormas.cirad.fr/en/demarch/sma.htm

[32] Kundu, A., Guha, S., Pal, A., Sarkar, T., Mandal, S., Dattagupta, R., Mukhopadhyay, D. (2008). Fuzzy Based Multi-Agent System Offering Cost Effective Corporate Environment. TOAUTOCJ The Open Automation and Control Syst

[33] Actor-based Concurrency. (n.d.). Retrieved February 22, 2016, from http://berb.github.io/diploma-thesis/original/054_actors.html

[34] The Reactive Manifesto. (n.d.). Retrieved February 25, 2016, from http://www.reactivemanifesto.org/

[35] Actor References, Paths and Addresses. (n.d.). Retrieved March 22, 2016, from http://doc.akka.io/docs/akka/2.4.0/general/addressing.html

IMAGE REFERENCES

[36] Actors. Digital image. Retrieved on 22 Nov. 2015 from `Https://ljcbookclub.files.wordpress.com/`

[37] Actors [Actors Lifecycle].Retrieved on 2 Jan. 2016 from `http://ptgmedia.pearsoncmg.com/`

[38] Reactive Manifesto [Reactive Manifesto Paradigms] Retrieved on 25 Dec. 2015 from `http://datacentricmanifesto.org`

[39] Akka Actor Supervision [Digital image]

APPENDIX

Appendix 1. *Following is the code snippet for Agent.scala*

1. Agent.scala

```
package agency


case class start()
case class EnvironmentInfo(information:Any)
case class startCollaboratorSelection()
case class collaborator(opp:ActorRef)
case class initiateInteraction()
case class acknowledge(collaorator:ActorRef)
case class
    doTask(action:String,orgin:ActorRef,destination:ActorRef,actionValue:Int)
case class resetCollaborator()
case class sendAttributeValue()
class Agent extends Actor{
import context._


 var name:String=self.path.name
 var message:String=null
 var attribute:Int=100
 var collaborator:ActorRef=null
 var facilitator:ActorRef=parent
 var property=(name,attribute,collaborator)
 become(start)
```

```
def senseEnvironment: Receive = {
   case EnvironmentInfo(information:Any) =>
     {
     if(isAlive()==true)
    {
     sender!ResponseAttrValue(attribute)
     isCapable()


     }
     }
     case
         doTask(action:String,sourceAgent:ActorRef,targetAgent:ActorRef,actionValue:Int)=>
      if(isAlive()==true)
     {
        if(message=="Interact")
         {
           setCollaborator(sourceAgent)
           interact(actionValue)
           sender!acknowledge(sourceAgent:ActorRef)
           resetCollaborator()


           startSensingEnvironment(10.milliseconds)
         }
         else if(message=="nothing"){
           println("Not interacting %s".format(name))
         }
 }
```

```
   }

}

def SelectCollaborator: Receive={

  case startCollaboratorSelection()=>
    {
      if(isAlive()==true)
    {

      parent! CollaboratorRequest(self,this.attribute)
      become(UpdateSelf)
    }
    }
    case
        doTask(action:String,sourceAgent:ActorRef,targetAgent:ActorRef,actionValue:Int)=>
     if(isAlive()==true)
    {
       if(message=="Interact")
        {
          setCollaborator(sourceAgent)
          interact(actionValue)
          sender!acknowledge(sourceAgent:ActorRef)
          resetCollaborator()

          startSensingEnvironment(10.milliseconds)
```

```
        }
        else if(message=="nothing"){
          println("Not interacting %s".format(name))
        }
}
   }
}

def UpdateSelf:Receive={
  case collaborator(opp:ActorRef)=>
    {
      if(isAlive()==true)
    {
      setCollaborator(opp)

      become(InteractWithCollaborator)
    }

    }
    case
        doTask(action:String,sourceAgent:ActorRef,targetAgent:ActorRef,actionValue:Int)=>
     if(isAlive()==true)
    {
       if(message=="Interact")
        {
          setCollaborator(sourceAgent)
          interact(actionValue)
```

```
          sender!acknowledge(sourceAgent:ActorRef)
          resetCollaborator()

          startSensingEnvironment(10.milliseconds)
        }
        else if(message=="nothing"){
          println("Not interacting %s".format(name))
        }
}
   }
}

 def InteractWithCollaborator:Receive={

   case initiateInteraction()=>
     {
       if(isAlive()==true)
    {
      message="Interact"
      var value:Int=0
      value=calculateAction()
      if(attribute==0 && value==0)
      {
        killself()
      }
      else
      {
```

```
      facilitator!Message(message,collaborator,value)
      }

    }

     }

   case
       doTask(action:String,sourceAgent:ActorRef,targetAgent:ActorRef,actionValue:Int)=>{
     if(isAlive()==true)
    {
       if(message=="Interact")
        {
          setCollaborator(sourceAgent)
          interact(actionValue)
          sender!acknowledge(sourceAgent:ActorRef)
          resetCollaborator()

          startSensingEnvironment(10.milliseconds)
        }
        else if(message=="nothing"){
          println("Not interacting %s".format(name))
        }
}
   }
   case sendAttributeValue()=>{
    if(attribute>0)
```

```
   {
     sender!ResponseAttrValue(attribute)
     Thread.sleep(5000)
   }
   else
   {
      sender!ResponseAttrValue(0)
   }


  }

  case resetCollaborator()=>{
    if(isAlive()==true)
   {
   resetCollaborator()
   startSensingEnvironment(10.milliseconds)
   }
  }
}

def start:Receive={
  case start()=>
    {
      if(isAlive()==true)
   {
       startAgent() // put activation code here
```

```
   }
    }

}

def receive={
  case "hi"=>"hello"
}



 def startAgent()={
   if(isAlive()==true)
   {
   println("%s starts to Act".format(name))
   startSensingEnvironment(10.milliseconds)
   }
 }
def processing()={

  sender!ResponseAttrValue(attribute)
  become(SelectCollaborator)
  self!startCollaboratorSelection()

}

def startSensingEnvironment(duration: FiniteDuration): Unit={
   if(isAlive()==true)
```

```
    {
    become(senseEnvironment)
    system.scheduler.scheduleOnce(5.seconds,sender,seekEnvironmentInfo())
    }
}

def setCollaborator(co:ActorRef)={
  if(isAlive()==true)
    {
      if(collaborator==null)
      {
        collaborator=co
      }
      else
      {
            sender!collabBusy(co)
      }
    }
}

def resetCollaborator()={
  if(isAlive()==true)
    {
    collaborator=null
    }

}
```

```
def interact(actionValue:Int)={
 if(isAlive()==true)
   {
  this.attribute=this.attribute-actionValue
  if(this.attribute<0)
  {
   this.attribute=0
   sender!ResponseAttrValue(attribute)
  }
  resetCollaborator()
  }



}

def calculateAction():Int={
   return 0
}



 def isCapable()={
   //this will check the attribute
    if(isAlive()==true)
   {


   processing()
```

```
  }

}

def isAlive():Boolean={
  if(this.attribute==0)
  {
    sender!ResponseAttrValue(attribute)
    facilitator!done()
    killself()
    return false
  }
  else
  {

    return true

  }

}
def killself()={
  attribute=0
  sender!ResponseAttrValue(attribute)
  context stop self
}
```

```
}
```

Appendix 2. *Following is the code snippet for Facilitator.scala*

1. Facilitator.scala

```
package agency


import akka.actor.Actor


case class CollaboratorRequest(requester:ActorRef,strategy:Any)
case class Message(message:String,targetAgent:ActorRef,actionValue:Int)
case class seekEnvironmentInfo()
case class acknowledge(origin:ActorRef)
case class update(sender:ActorRef)
case class done()
case class collabBusy(busyCollab:ActorRef)
case class ResponseAttrValue(attribute:Int)
trait Facilitator extends Actor{
import context._
var count:Int=0
var strategy:Any= null
var time=0
var agents = new ListBuffer[ActorRef]()
var envMap=new HashMap[ActorRef, Int] with
           SynchronizedMap[ActorRef, Int]
this. makeAgent()
```

```
this.startAgents()
system.scheduler.schedule(10 seconds,10 seconds)(printEnvState(envMap))
become(available)

    def makeAgent()=
    {
     println("Enter the number of agents you want to create ")
     val num=readInt()
     println(num)
     this.count=num
     println("CREATING AGENTS")
      var i:Int=0
      var agents = new ListBuffer[ActorRef]()
      this.updateEnv()

    }

  def updateEnv() :ListBuffer[ActorRef]={
   this.agents=agents
   return agents
 }

  def startAgents()={
    agents.foreach(_ !start())
  }
```

```
  def removeAgent(agent:ActorRef):ListBuffer[ActorRef]={
  if(agents.length>1)
    {
     agents=agents-agent
     return agents
    }
  else{
    return null
  }

}

  def findCollaborator(strategy:Any,agent:ActorRef):ActorRef={
   if(agents.length>1)
   {
          val random:Random = new Random()
          val randomInt: Int= random.nextInt(agents.length)
          var selctedcollab : ActorRef= agents.apply(randomInt)


          var x=agent.path compareTo selctedcollab.path
          if(x==0)
        {

          findCollaborator(strategy,agent)
        }
        else
```

```
        {
           return selctedcollab
        }

    }
    else
    {
     println("Winner is %s".format(sender.path.name))
     return null
    }
}

  def startInteraction(ref:ActorRef)={
   ref!initiateInteraction()
 }

   def computeEnvironmentInfo()={
     sender!EnvironmentInfo(agents)
   }


 def available: Receive={

    case seekEnvironmentInfo()=>{
      computeEnvironmentInfo()
    }
```

```
case CollaboratorRequest(from:ActorRef, strategy)=>
  {

    var colbrtr:ActorRef=findCollaborator(strategy,sender)
    sender!collaborator(colbrtr)
    startInteraction(sender)

  }

case collabBusy(retryAgent:ActorRef)=>{
  findCollaborator(strategy,retryAgent)
}
case Message(message,targetAgent,actionValue)=>
 {
    var sourceAgent:ActorRef=sender
    targetAgent!doTask("Interact",sourceAgent,targetAgent,actionValue)
  }

case acknowledge(origin:ActorRef)=>{
   sender!resetCollaborator()
   origin!resetCollaborator()
   polling()
}

 case done()=>{
  removeAgent(sender)
```

```
    }

     case ResponseAttrValue(attribute)=>
       {

          envMap += (sender -> attribute)

       }

}

  def printEnvState(envMap:HashMap[ActorRef, Int])=
    {
       var map2=ListMap(envMap.toSeq.sortWith(_._2 > _._2):_*)
       var c1=0
       var c2=0
       var c3=0
       var c4=0
       var c5=0
       var c6=0
       println("-----------------------------------------------------------------
       time=time+1
       println("Time = %s ". format(time))
       for ((k,v) <- map2)
       {
         if(v>0 && v<=20)
```

```
 {
   c1=c1+1
 }
 else
   if(v>20 && v<=50)
   {
     c2=c2+1
   }
   else if(v>50 && v<70)
   {
     c3=c3+1
   }
  else if(v>=70 && v<=100)
   {
     c6=c6+1
   }
   else if(v>100)
   {
     c5=c5+1
   }
   else if(v==0)
   {
     c4=c4+1
   }

}
 println("Agents  0< Values <20  are %s  ".format(c1))
```

```
        println("Agents 20< Values <50 are %s  ".format(c2))
        println("Agents 50< Values <70 are %s  ".format(c3))
        println("Agents 70< Values <100 are %s ".format(c6))
        println("Agents     Values >100 are %s  ".format(c5))
        println("Agents     dead        are %s  ".format(c4))
       println("-----------------------------------------------------------------
    }

   def polling()=
     {
      agents.foreach(_ !sendAttributeValue())
     }

    def receive={

      case "default"=>println("default case")
    }
}
```