Università di Genova | **DIBRIS** DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

# Exploiting JADE as a Multi-Agent simulator of the Immune System

SANCHAYAN BHUNIA

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università di Genova

MSc Computer Science
Data Science and Engineering Curriculum

# Exploiting JADE as a Multi-Agent simulator of the Immune System

SANCHAYAN BHUNIA

Advisor: Prof. Angelo Ferrando
Prof. Viviana Mascardi
Prof. Chiara Vitale

Examiner: Prof. Gianna Reggio

December, 2022

## Abstract

The immune system is one of the most complex biological systems in an organism and consists of millions – if not billions – of cells of different nature. These cells interact amongst themselves to keep the organism safe from external pathogens such as viruses and bacteria. Based upon their behaviours and instance of activation, the cells of the immune system can be further categorized as a part of either Innate or Adaptive Immune System, which we have modelled in this thesis.

There has been a huge deal of research in the field of immunology to understand the nature of the immune system that works under the hood to protect the body against these pathogens. In this thesis, we have explored another possible way to model different actors of the Immune system using software agents. Here, we model these actors as software agents and their interactions via agent communication language. This approach not only helps us to simulate various aspects of the immune system (*e.g.*, Adaptive and Innate Immune System), but also, to understand many immunological conditions related to different types of infections and the immune response in case of a re-infection.

In this thesis, we present the initial design and development of an agent-based simulation of the immune system using a well-known agent framework, JADE. We present the engineering choices we made and the instantiation of some steps of the secondary immune system response. We discuss the implementation in JADE, and we present some experimental results.

**Keywords**: Immune System, JADE, ABMS, Multiagent System, Modelling Immune System with a Multiagent System, Modelling the Immune System, Modelling Adaptive Immune System, Modelling Innate Immune System.

# Table of Contents

# Listings

# Chapter 1

# Introduction

The immune system of a multi-cellular organism is a very complex system with many different immune cells playing important roles to keep an organism free from infections. These infections can be cause by any pathogen like a Bacteria, a virus or a fungi. For this thesis we have focused our research towards an infection caused by viruses. The goal of a virus is to hijack a cell and use its resources to make a copy of itself (replication) and to spread those replicas into other parts of the body. The immune system helps an organism to fight back the infection by searching for any virus signature moving from one cell to another in general [Sim18].

The immune system in general consists of multiple different cells with different behaviours: some of them immediately activate and contribute to eradicate the infection by different and complementary mechanisms, such as Phagocytes, T-cells, B-cells; some of them remember the signature of an infection, such as B-cells and Memory T-cells, and are essential to prevent similar infection in the future. Depending upon these behaviours the immune cells can be categorized in to Innate and Adaptive Immune Cells [Sig16].

The Innate Immunity, that is present at birth and lasts an organism's entire life, is the first response of the body's immune system to a harmful foreign substance. When foreign substances, such as bacteria or viruses, enter the body, certain cells in the immune system can quickly respond and try to destroy them. Some of these cells include, Phagocytes like Macrophages and Neutrophils, Dendritic Cells, Natural Killer Cells [MWWK18], and many more. Innate immunity also includes barriers, such as skin, mucous membranes, tears, and stomach acid, that help keep harmful substances from entering the body.

The Adaptive Immunity is a type of immunity that develops when a person's immune system responds to a foreign substance or microorganism, such as after an infection or vaccination. This type of immunity involves specialized immune cells that can target and eliminate foreign intruders in the future by remembering what those substances genetically look like. Adaptive immunity may last for a few weeks or months or for a longer time, sometimes for an organism's entire life. Very similar to the Innate Immune System, there are many cells involved in this type of immunity, for example Lymphocytes like B-Cells, T-Cells (Both CD8 T-Cells and CD4+ T-Cells) and so on [MWWK18].

The goal of this thesis is to model some of the components of the immune system as interacting agents using JADE[1] [BCG07] and simulate a situation when the immune system is under attack from a virus. Agent-Based Model & Simulation (ABMS) is a well-known research area focused on simulating systems following a bottom-up approach, where agents are used to describe the actors of the simulation. There are several reasons for choosing JADE as *the* modelling framework:

1) The complexity of the individual actors can easily be managed with JADE agents and their behaviours which can be deployed as independent or dependent of environmental changes.

2) The possibility of decentralization of the computation and memory allocation of individual agents allows us to design even complex behaviours for comparatively large pool of agents.

3) The usability of JADE's native agent mobilization features allows the agents in the model to persistently move the computations and memory over a network of computers.

4) The developer-friendly API of JADE, since it is based on Java.

From our preliminary research [SBV22], we have figured out the importance CD8 T-Cell and Macrophages in the Innate Immune Response and the importance of Dendritic Cells and CD4+ T-Cells in the Adaptive Immune Response. In our model we have implemented these actors as a part of the overall immune system with the help of software agents (mobile agents) following the AOSE methodology. Other than that, we have also modelled the Somatic Cells and the Lymph Vessel cells as immobile software agents. The virus is also modelled using JADE software agent but – unlike the other simulated agents – it is generated by a class that, given some features, generates a virus compliant with them. These features include the gene of the virus, the replication factor and the resilience towards interacting with the somatic cells in the system.

Chapter 2 of this thesis digs deeper into the background knowledge that is required to understand our research project. Especially, in that chapter we discuss about the definitions and implications of numerous concepts related to Software Agents, their behaviours and Agent Based Software Engineering. That chapter also includes a brief discussion about JADE and the features JADE that were used in this thesis.

[1]**JADE**: Java Agent Development Framework.

In Chapter 3 we discuss some of the works that are related to Agent Based Modelling. Although we did not find any explicit related work regarding the implementation of the Immune System using the AOSE[2] paradigm, and the JADE framework in particular, still these studies have helped us to discover the key modelling features.

In Chapter 4 we discuss about the requirements of our model and why does JADE fulfill these criteria.

Then we move ahead in Chapter 5 with the design part of our model. In thst chapter we discuss about different types of immune responses and design structure of our model.

In Chapter 6 we discuss how we have implemented the features of our model using JADE. In that chapter we discuss about different mobile and immobile agents, their behaviours and communication methodologies in details.

Chapter 7 describes some experiments to support the consistency our software model with actual biological scenarios. In that chapter we discuss about behaviours of healthy and unhealthy immune systems from both biological and our model point views. There we also discuss about a re-infection scenario to solidify our claim about our model being able to produce an Adaptive Immune Response.

Ultimately, Chapter 8 shades light upon a general discussion about the thesis and future implications of our model.

[2]**AOSE**: Agent Oriented Software Engineering.

# Chapter 2

# Background

In this section we are going to cover the very basic concepts of Agents and Agent Oriented Software Engineering (*a.k.a.* AOSE) along with some details about how the JADE Framework works.

## 2.1 Agents

An agent, also called a software agent or an intelligent agent, is a piece of autonomous software, the words intelligent and agent describe some of its characteristic features. Intelligent is used because the software can have certain types of behavior ("Intelligent behavior is the selection of actions based on knowledge"), and the term agent tells something about the purpose of the software. An agent is "one who is authorized to act for or in the place of another" (Merriam Webster's Dictionary) [Tve01].

Examples of software Agents:

- Personal Assistant like Amazon Alexa, Google Assistant and Siri.
- Computer Viruses can be a type of software agents.
- Artificial Players in computer games [Tve01].
- Web Crawlers used by Search Engines like Google are a type of software agents [GOL].
- A self-driving car could use software agents [OOM19].

A common classification scheme of agents is the weak and strong notion of agency [WJ95]. In the weak notion of agency, agents have their own will (autonomy), they are able to interact with each other (social ability), they respond to stimulus (reactivity), and they take initiative (pro-activity). In the strong notion of agency the weak notions of agency are preserved, in addition agents can move around (mobility), they are truthful (veracity), they do what they're told to do (benevolence), and they will perform in an optimal manner to achieve goals (rationality) [Tve01].

In terms of its properties, any software which has the following properties can be classified as a software agent.

- **Situatedness** — The agent has to be situated in an environment and receiving some form of sensory input from its environment. The agent also can perform some actions with its actuators to change its environment.
- **Autonomy** — The agent can act independently without a direct intervention from a human or other software agents. Moreover, it has to have a control over its behaviours in order to achieve its goals.
- **Adaptivity** — The agent is capable of (1) reacting flexibly to changes in its environment; (2) taking goal-directed initiative (i.e., is pro-active), when appropriate; and (3) learning from its own experience, its environment, and interactions with others.

Apart form these characteristics, an agent can be conceptualized by the following anthropomorphic approach.

- **Mentalistic Notations** — Believes, Desires, Intentions, Commitments and etc.
- **Emotional Notations** — Friendliness, Trust, Untrust and etc.

### 2.1.1 Agents Coupled with their Environment

Agents are autonomous and capable of taking decisions. Thus an agent is a computer system that can anonymously act in an environment in order to achieve some goals. An agent can be thought of as an interactive collection of Sense, Decide and React. so, based upon how it behaves on changes in its environment, the behaviours can be further be classified in terms of Reactiveness and Pro-activeness.

#### 2.1.1.1 Reactive Property

If an agent is "situated" in an environment which is dynamic in nature, deploying pre-written codes in order to achieve a goal is not viable. Rather, the agent equipped with sensors should have the ability to react to the ever-changing environment in order to modify its behaviours and at on it accordingly.

#### 2.1.1.2 Pro-Active Property

Building a reactive agent is easy by making a look-up table, *e.g.*

$$\text{“}stimulus\text{”} \longrightarrow \text{“}responserules\text{”}$$

But the agent needs to achieve goals as well. Hence, we can model the goal in terms of guided behaviours. Pro-activeness refers to the ability to generate plans in order to achieve goals which might or might not be driven by the events captured by the sensors.

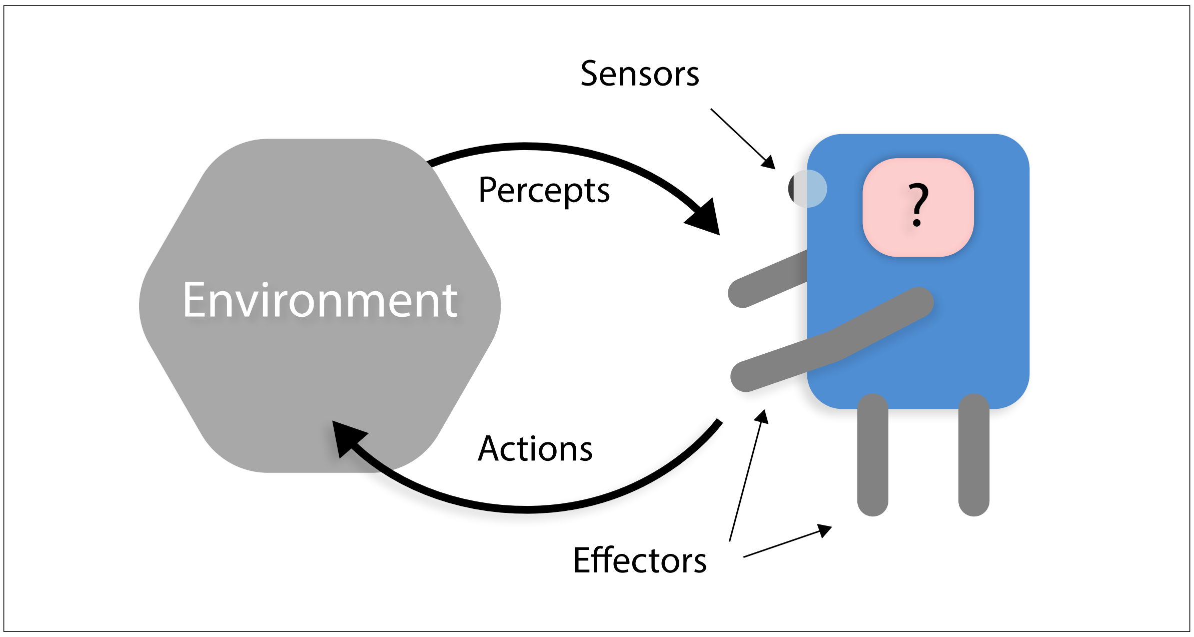


Figure 2.1: An Agent Interacting with its Environment [Tut].

### 2.1.2 Multiple Agent System

A system which is composed of multiple agents where each agent has partial information about the environment and can solve some part of the given problem. Thus the multiple Agents need to work together in order to achieve a task. In such typical scenario there is no global control over the behaviour of the system. The data is decentralized and the computation is asynchronous.

### Social Ability of Agents

In a multiagent setting, the environment is also composed of other agent which might or might not have similar goals. So, the social ability of an agent allows it to interact with other agents through *Agent Communication Languages* and perhaps coordinate with them in order to achieve goals.

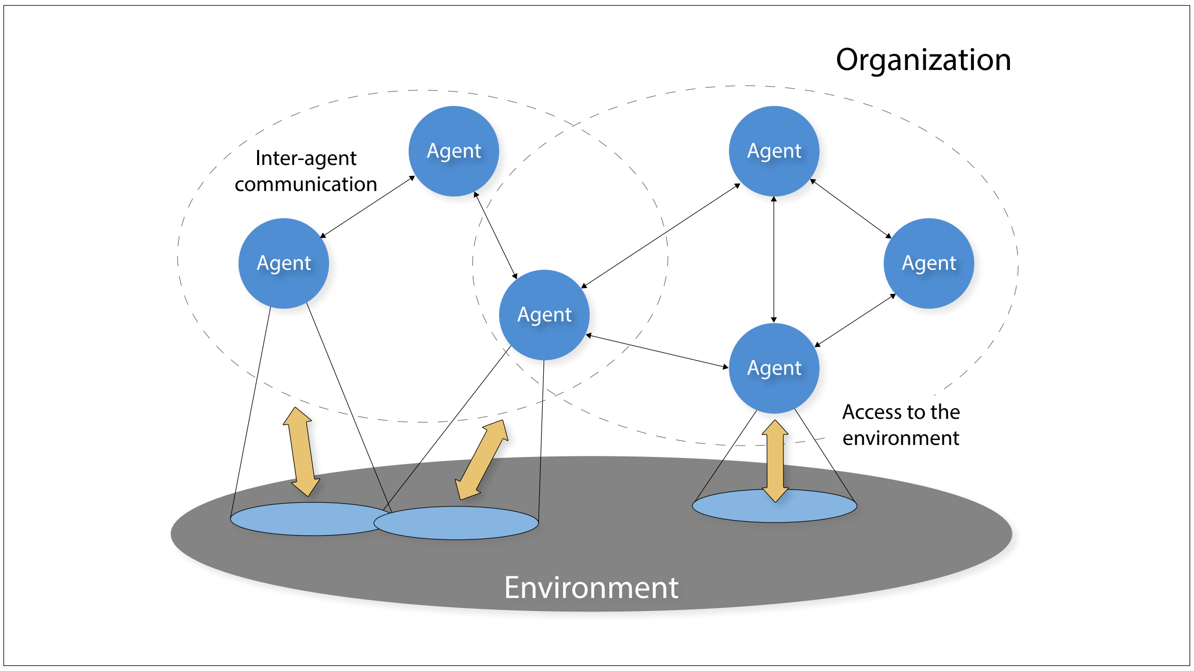


Figure 2.2: A Diagram of an interactive Multiagent system in an Environment [AZR+17].

## 2.1.3 Agent Architecture

The fundamental principles that underlie the autonomous elements that facilitate effective behavior in real-world, dynamic, and open contexts are known as agent architectures. In reality, early efforts in the field of agent-based computing were concentrated on the creation of intelligent agent architectures, and these years saw the emergence of several enduring architectural styles. These range from purely reactive (or behavioral) architectures that only respond to stimuli in a simple stimulus-response manner, like those based on Brooks's (1991) subsumption architecture, to more deliberative architectures that consider the implications of their actions, like those based on Rao and Georgeff's (1995) belief desire intention (BDI) model [GPP+99]. Between the two are layered architectures (hybrid mixtures of both) that try to combine both reaction and contemplation in an effort to adopt the best features of each strategy. Logic-based, reactive, deliberative, and layered architectures are the four major categories into which agent architectures can be subdivided [BCG07].

#### 2.1.3.1 Logic-based Architectures

In Logic-based system the environment is symbolically represented and manipulated using reasoning mechanisms from traditional logic-based systems. And since the human knowledge is also symbolic in nature, Logic-based reasoning is easier for human to understand. For the same reason it is difficult to translate the real world into an accurate, adequate symbolic description, and that symbolic representation and manipulation can take considerable time to execute with results are often available too late to be useful [BCG07].

#### 2.1.3.2 Reactive Agent Architecture

Reactive architectures implement decision-making as a direct mapping of situation to action and are based on a stimulus-response mechanism triggered by sensor data. The representation of the world in reactive architecture is very simple and there is a tight coupling between the perception and the action of an agent. The agents are modelled based upon their behavioral paradigm and “Intelligence” occurs as a result of interaction between the agent and its environment. A good example is *Swarm Intelligence* where swarms are modelled as biordoid agents. One agent might not be very intelligent in nature but the Swarm as a group coupled with the environment shows intelligent behaviours [CBD20].

The best-known reactive architecture is Brooks’s subsumption architecture (Brooks, 1991). The key ideas on which Brooks realized this architecture are that an intelligent behaviour can be generated without explicit representations and abstract reasoning provided by symbolic artificial intelligence techniques and that intelligence is an emergent property of certain complex systems. The subsumption architecture defines layers of finite state machines that are connected to sensors that transmit real-time information (an example of subsumption architecture is shown in Figure 2.3) [BCG07].

#### 2.1.3.3 Deliberative Agent Architecture

According to the definition by Wooldridge in “Software agents: an overview”, a deliberative agent is ”one that possesses an explicitly represented, symbolic model of the world, and in which decisions (for example about what actions to perform) are made via symbolic reasoning” [Nwa96]. Opposite to the Reactive agents, a deliberate agent’s internal process is more complex. The Deliberate agents keeps explicit symbolic model of the world it inhabits. And then the decision making is done via logical reasoning based on pattern

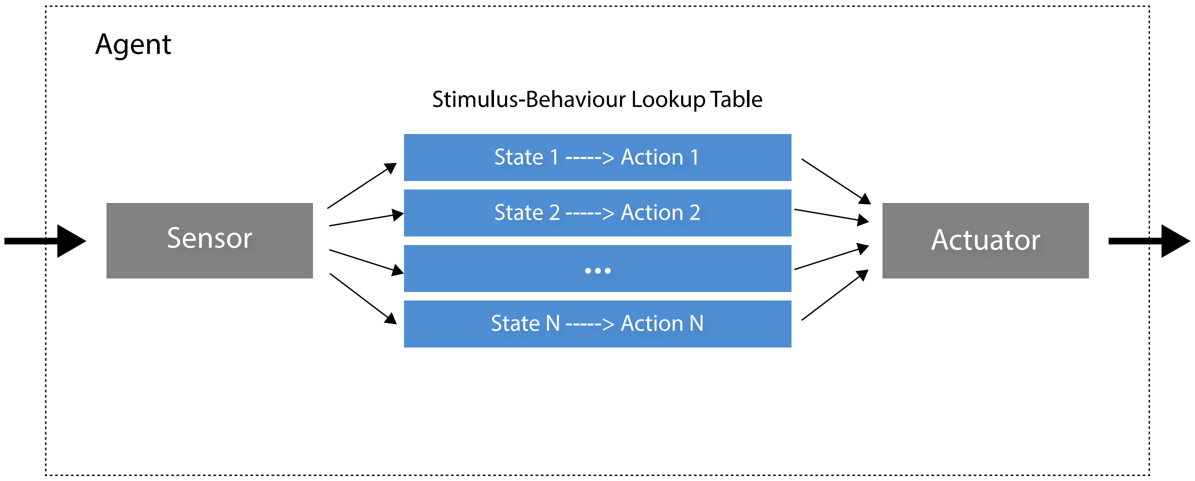


Figure 2.3: Architecture of a Reactive Agent [Nwa96].

matching and symbolic manipulation. The classic AI problem solving paradigm sense, plan and act is used for decision making and action taking. BDI architecture if one of the widely used type of Deliberate Agent Architecture.

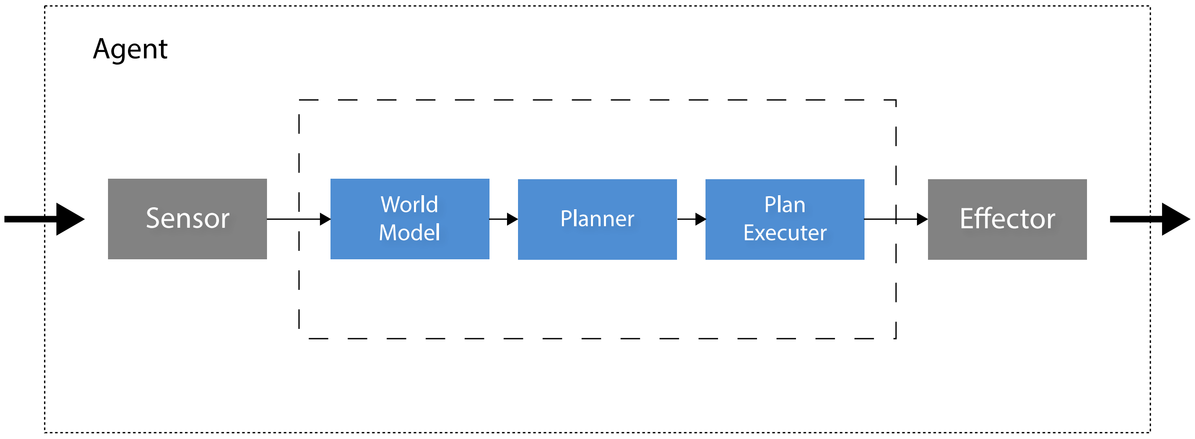


Figure 2.4: Architecture of Deliberate Agent [Nwa96].

### BDI Agents

Originally developed by Michael E. Bratman in his book "Intentions, Plans, and Practical Reason", (1987), Belief-Desire-Intention (BDI) architecture is model where an agent's beliefs about the world which represents it view of the world model, desires (goal) and

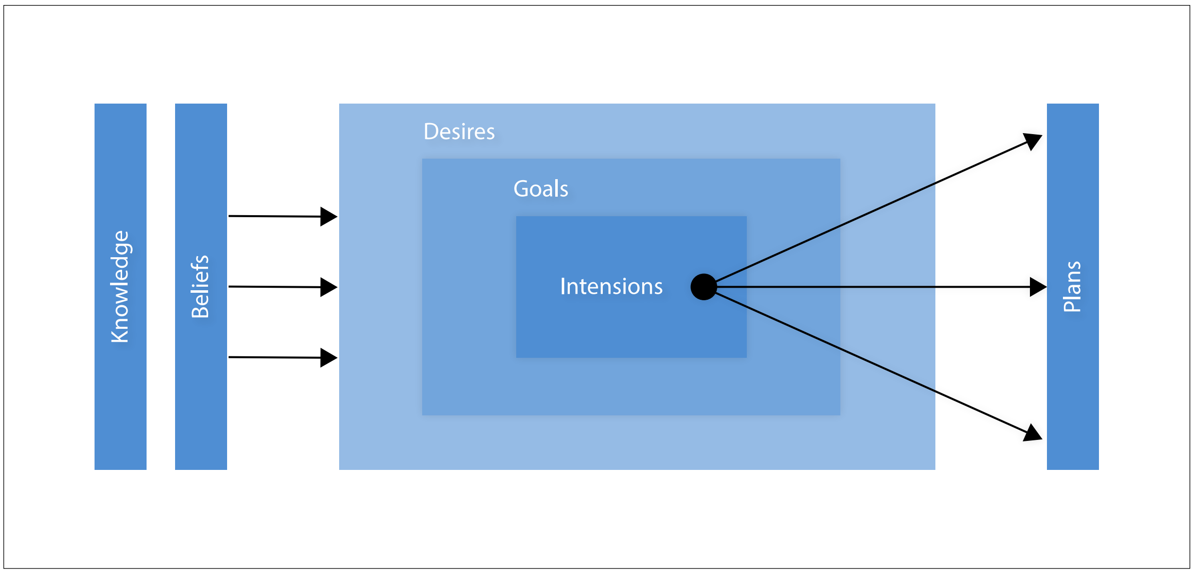


Figure 2.5: A diagram to show relation among Belief, Desire and Intention [LBGVDR17].

intentions are internally represented and first-order logic[1] is applied to figure out most suitable action to take at that point of time.

- **Beliefs** — Agent's view of the environment/world. If their are multiple beliefs, they can be stored in a dataset, which is called a beliefset.
- **Desires** — Desires represent the motivational state of an agent that follows from the Belief. A desire can be unrealistic from the point of view of the agent's behaviours and can also change with time.

  A subset of these desires can be realistic and consistent over time these are called Goals and they are later evaluated for further processing.
- **Intentions** — When an agent decides to commit a goal it is called an Intention and sequence of actions are needed to achieve it. These sequence of actions are called plans to achieve that specific goal.

#### 2.1.3.4 Layered Architecture

In a hybrid or Layered Architecture the agent is built out of two or more subsystems, e.g. the Reactive and Deliberate. Over the years many researchers have argued that this

[1]First-order logic is symbolized reasoning in which each sentence, or statement, is broken down into a subject and a predicate. Then the predicate modifies or defines the properties of the subject.

is the best way to design software agents since, we get best of the both worlds. In a hybrid architecture, an agent's control subsystems are arranged into a higherarchy where the higher layers deal with information at increasing levels of abstraction [Cas22].

One of the most difficult task of designing these architectures is to decide the layering of the subsystems. There are two different layering techniques that can be used, firstly, a horizontal layering where each layer is directly connected to the sensory input and the action output, alternatively, a Vertical layering where these input and outputs are connected to at most one layers each.

Examples:

- The TOURINGMACHINES architecture consists of perception and action subsystems, which interface directly with the agent's environment, and three control layers, embedded in a control framework, which mediates between the layers [Cas22].
- The Agent Architecture InteRRaP, which is a vertically layered, two-pass architecture where only one level interacts with the environment [MP93].

## 2.2 Agent Oriented Software Engineering

In the last few years, together with the increasing acceptance of agent-based computing as a novel software engineering paradigm, there has been a great deal of research related to the identification and definition of suitable models and techniques to support the development of complex software systems in terms of MASs. This research can be grouped under the umbrella term *Agent-Oriented Software Engineering (AOSE).* From a design and development point of view, the features of agent-based systems are well suited to tackle the complexity of developing software in modern scenarios. Some of these requirements which falls in line with AOSE are the following.

- The autonomy of application components reflects the intrinsically decentralised nature of modern distributed systems and can be considered as the natural extension to the notions of modularity and encapsulation for systems that are owned by different stakeholders.
- The flexible way in which agents operate and interact (both with each other and with the environment) is suited to the dynamic and unpredictable scenarios where software is expected to operate.
- The concept of agency provides for a unified view of AI results and achievements, by making agents and MASs act as sound and manageable repositories of intelligent behaviours.

The main purposes of Agent-Oriented Software Engineering are to create methodologies and tools that enables inexpensive development and maintenance of agent-based software. In addition, the software should be flexible, easy-to-use, scalable and of high quality.[Tve01] In other words, this sounds quite similar to the research issues of other branches of software engineering, for example, object-oriented software engineering.

### 2.2.1 OOP and AOP

Object Oriented Programming (OOP) can is the successor of structured programming where the main entity is the object. An object is a logical combination of data structures and corresponding functions or methods to manipulate those data. Objects are very successful to create abstraction over passive entities (e.g. an employee) in real world and the agents are regarded as a possible successor of objects since they have the enhanced capability to even create abstractions of active entities. Agents are similar to objects, but they also support structures for representing mental components, like beliefs, goals and commitments. Moreover, an agent is capable of more high level communication with another agent based on the "speech act" theory as opposed to ad-hoc messages frequently used between objects. Example of such languages are FIPA ACL and KQML [Kib13].

Another more important difference between objects and agents is the fact that objects are controlled from the outside, whereas the agents have autonomous behaviours which can't directly be controlled from outside. In other words, an agent can say "no" to certain set of instruction if it is not in its belief set [Tve01].

### 2.2.2 Applications of AOSE Concepts

The Agent Oriented concepts can be applied in various phases while Engineering any Software starting from the high level design and architecture to actual coding level implementation. Even one can choose between the imperative and declarative programming paradigm. Where on one hand there is JASON, on the other hand we have JADE where it is implemented in Java. Now we will discuss about recent developments in some of the areas where AOSE can be used.

#### 2.2.2.1 Agent-Oriented Architectures

Agent-oriented architecture is formed based on the fact 'agent', which has the capability of autonomy in decision making, team work, work passively and being goal oriented. These characteristics form the software operate dynamically and make appropriate decisions based on common interaction with each other in case of each event and then take appropriate reaction [DRN12a]. Below there are some examples about some of the case studies where Agent-Oriented Architectures have been used.

- In the "article Agent-Oriented Enterprise Architecture: new approach for Enterprise Architecture" [DRN12b], there is a proposed solution to redesign EA (Enterprise Architecture) programs using agent oriented architecture since this kind of architecture is well suited to handle complex information systems. The authors have also discussed about the current problems of the EA and how the Agent-oriented paradigm solves the problem.
- In the article "An agent-based service-oriented integration architecture for collaborative intelligent manufacturing" [SHW+07] the authors have proposes an agent based approach for a network of virtual enterprises to leverage manufacturing scheduling service. They have also built a software prototype system to share manufacturing resources among enterprises using agent-based web services. This is a Multiagent System (MAS) and the agents can negotiate about sharing the resources by communicating with each-other through HTTP protocol.

#### 2.2.2.2 Agent-Oriented Programming Languages

Agent-Oriented programming, a.k.a. AOP is a programming paradigm where the construction of software is centered around the concept of software agents [wik22]. An AOP will include three primary components [Sho91]:

- A restricted formal language with clear syntax and schematic for describing mental state of an agent.
- An interpreted programming language with primitive commands like REQUEST and INFORM to program the agents.
- An "agentifier" for converting neutral devices into programmable agents.

Most of these programming languages are imperative in nature and usually require first order logic to express the mental states (believes, intentions), behaviours and actions of an agent. some of the examples include, JASON, GOAL etc.

#### 2.2.2.3 Agent-based Modelling and Simulation

An agent-based model (ABM) is a computational model for simulating the actions and interactions of autonomous agents (both individual or collective entities such as organizations or groups) in order to understand the behavior of a system and what governs its outcomes [wik20]. In an ABMS usually the agents are simple reactive agents interacting among themselves and with the environment where each agent can only sense some part of the environment with its sensor and react accordingly. The intelligence then arises as a result of their interaction with each-other. This is also argued as “the third way of doing science” by experts. There are several reasons for this claim and they are the following.

- ABS takes place in an artificial world so, the aspects of the real world which are not physically accessible can be modelled here for experiments.
- Experimenting in the real system sometimes might have undesired interference from other sources.
- The time scale of the system behaviour might not be adequate to make an observation.
- The Original system might have been altered or have vanished from the existence by the time of the experiment.
- There might be cost related concern in performing the experiment live.

An example of ABM is swarm Intelligence and NetLogo[2], developed by CCL research group from Northwestern University is widely used for simulating these models. In the paper, “Survival chances of a prey swarm: how the cooperative interaction range affects the outcome” [CBD20] I have used MATLAB to simulate a swarm of interacting prays and predators.

Although this thesis falls under the category of Multiple Agent Based Modelling, the agents are not simple reactive agents, rather, they exhibits complex behaviours and interacts with the environment and other agents. Here we have used JADE Framework to implement the agents and their environment.

#### 2.2.2.4 MAS Infrastructures & FIPA Compliance

Software infrastructures are very important for developing Multiagent Systems, tools and software and throughout the years various tools have been proposed to implement MAS specifications into actual agent code, and many middle-ware infrastructures have been developed that provide support for implementing and developing distributed Multiagent Systems. But the main issue comes in the form of interoperability of these infrastructures. In order to tackle that FIPA has proposed abstract architecture to design and develop a standard MAS Infrastructure. Now lets discuss in details about what is FIPA and what makes a platform FIPA compliant.

[2]https://ccl.northwestern.edu/netlogo

**FIPA**

The Foundation for Intelligent Physical Agents (FIPA) is an organization that has defined a set of standards for Systems with Multiple Agents. These definitions allows the interoperability between agents and facilitate their development. In particular the FIPA Agent Platform defines the structure of an agent system model, it is composed of the following [BCG07]:

- **Agents**, the fundamental actor inside the system.
- **Agent Management System (AMS)**, has control over access to the system by offering a white pages service to other agents.
- **Directory Facilitator (DF)**, it is an optional component, provides the yellow pages services to other agents.
- **Message Transport Service (MTS)**, the communication method exploited by the agents.
- **Agent Platform (AP)**, the physical infrastructure in which the agents are deployed.

Apart from these components, the FIPA also defines a series of Agent Communication Language, a message schema that each agent has to follow in order to let the system to be extensible and allow the integration of other agents. But, for the scope of the thesis only architectural specifications have been followed [BCG07].

## 2.3 Java Agent Development Framework

Java Agent Development framework a.k.a JADE, developed by Telecom Italia in late 1998 in order to make a Platform for developing Interactive Agents following the FIPA specification. It is open source and distributed under the LGPL (Library Gnu Public Licence). This software provides middle-ware layer functionalities independent of any application composed of software agents. Another significant benefit of this platform is that it implements the agent paradigm over a well-known object-oriented language, Java which provides a developer-friendly API. The platform also provides the following design choices following the agent abstraction.

- **Autonomous and Proactive Agents** — Every agent has its own thread of execution where the entire life-cycle of the agent is controlled and the agent can autonomously decide when to perform which action.

- **Agents are loosely-coupled** — The communication between two agents happens through asynchronous message passing. An agent which is a sender of such message, should know the ID of the receiver agent. And since there is no temporal dependency between the sender and the receiver, the receiver might not receive any message at all.

- **Peer-to-Peer** — Each agent is identified globally by a unique Agent ID (AID) and can join or leave the host platform at its own will. So, the discovery of an agent by other agent or a human has been implemented by both white-page and yellow-page[3] services.

With the adoption of these design choices JADE becomes a fully FIPA compliant platform for implementing and developing agent abstraction based software solutions.

## 2.3.1 JADE Architecture

Agents live in a container which are Java processes that provides the JADE run-time and all the services needed for hosting and executing agents and a platform may have several containers. On the other hand, all containers might not be in the same machine. In other words, we have one JVM[4] per machine and one thread per agent. So in short, the JADE platform is composed of containers and Agents. The Figure 2.6 shows the UML diagram of the relationship among the main architectural elements. There there are some special agents like the Agent Management System Agent, a.k.a. AMS Agent, Directory Facilitator Agent, a.k.a. DF Agent to provide utility by managing the white-page and the yellow-page services.

### 2.3.1.1 JADE Containers

Containers are the fundamental building block for the JADE platform and can be distributed over a network. The programmer identifies containers by simply using a logical name, by default the main container is named 'Main Container' while the others are named 'Container-1', 'Container-2', etc.[BCG07] A container can have none to multiple number of agents. A container has a `ContainerController` class which upon instantiation creates a `containerController` `object` which can be used to get the `Controller` of an agent living inside the container by invoking `getAgent()` `method`. A new agent can be created inside the container by using the `method` `createNewAgent()`. There are `method`s

[3]white-page and yellow-page services allow the dynamic discovery of the hosted agents and the services they offer.[Tom15]

[4]JVM: Java Virtual Machine.

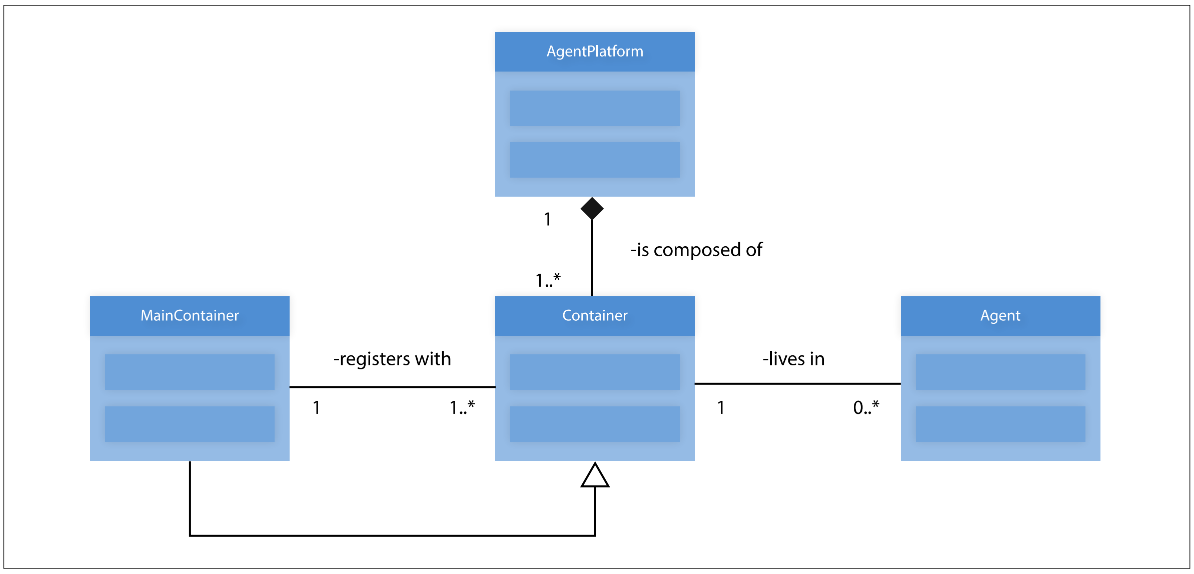


Figure 2.6: UML diagram of Relationship between the main architectural elements [BCG07].

available in this `class` to accept or reject any migrating agent as well. All containers regardless of whether it a Main Container or not shares these specifications.

### 2.3.1.2 Main Container

The main container is a special container that represents a bootstrap point of a platform. This is the container which is launched first by the JADE run-time and all other containers should register with it. By default, the main container hosts the AMS and the DF which are used to manage agents in the platform. There is a registry called global agent descriptor table managed by the Main Container which holds the information regarding the status and the location of all agents. The main container also holds a registry of all other containers in the platform and their transport addresses.

### 2.3.1.3 JADE Agents

The name of an agent is a global unique identifier that JADE constructs by concatenating a user defined local name to the name of the platform. The agent addresses are transport addresses, where each platform address corresponds to an MTP (Message Transport Protocol) end point where FIPA-compliant messages can be sent and received. And finally, the agent identity is composed of the agent name and its address and contained within an Agent Identifier (AID) [BCG07].

The base `class Agent` is the `superclass` used to build JADE agents. Generally, each agent records several services which can be implemented by one or more behaviours. Once instantiated, the `setup()` method is used to modify the data registered with the AMS, pass in attributes to the agent, register with one or more domains (DFs). Using this method it is also possible to add `behaviours` of the agents. In order to stop the execution of an agent the `doDelete()` method is used at any time. Agents can communicate with each other by exchanging ACL Messages which can be created by instantiating message objects from `ACLMessage class`. The message can then be sent to other agent by `send()` method from the `Agent class`. similarly, an agent can receieve a message by `receive()` method from the same `class`. In order to add or remove behaviours [2.3.2], the `Agent class` also provides `addBehaviour()` and `removeBehaviour()` methods. A life-cycle of an agent is then composed of various states invoked by all of these methods. The states in the life-cycle of the agent are the following.

- **Initiated** — At this state the agent is built, but it can not perform any actions since it is not registered to the AMS yet.
- **Active** — The agent is registered to the AMS and can access all JADE features. It can start executing its behaviours.
- **Waiting** — At this state the agent thread is blocked and the agent is expecting something to happen. For example, waiting for a message.
- **Suspended** — The agent is stopped and can not execute its behaviours.
- **Transit** — The agent has started migrating from one container to another. This state persists until the migration process ends.
- **Dead** — The agent is dead and deregistered from the AMS registry.

Different states of the life-cycle can be represented with the help of schematic diagram [2.7].

### 2.3.2 Behaviour of Agents

By definition, agents are autonomous entities, therefore they should act independently and concurrently with respect to one another. A behaviour is a collection of activities performed in order to achieve a goal and started by the agent autonomously. JADE implements behaviours as Java `objects`, which are executed concurrently by a single thread by using a non-primitive scheduling policy [2.8].

Any behaviour can be instantiated from `jade.core.behaviours.Behaviour class`. Every behaviour has an `action()` method which is used to define the actions that should be

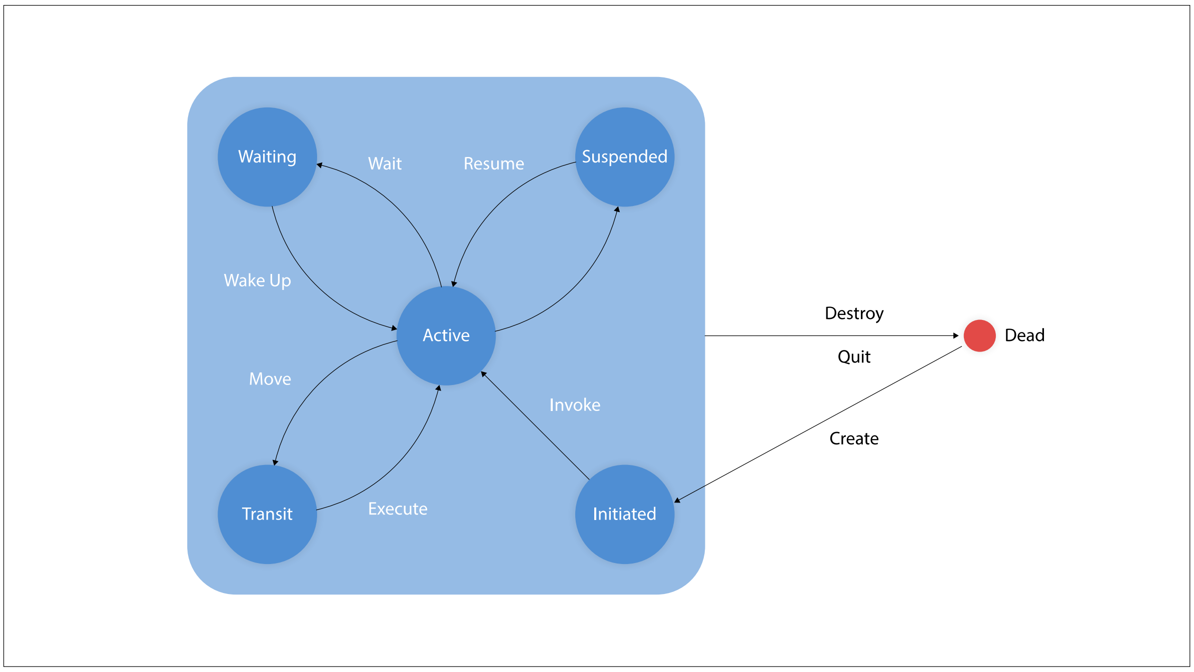


Figure 2.7: States in Agent's Life-cycle [Igu19].

performed once the agent decides to adopt a certain behaviour. Other than that `done()` method is there to check the state of a behaviour execution process. A `block()` method provides the option to block, `restart()` method to restart and a `reset()` method to reset are also provided for handling the execution of a behaviour at run-time. A `protected myAgent` object is also available within a behaviour setup in order to add or remove this or another behaviour at from the behaviour implementation itself. This allows a programmer to design Composite Behaviours by nesting multiple behaviours.

#### 2.3.2.1 One-shot Behaviour

A `OneShotBehaviour` is a behaviour where the action(s) that is to be taken by the agent is executed only once in a lifecycle. The `done()` method always returns `true` in such behaviours. By extending `jade.core.behaviours.OneShotBehaviour class` any custom one-shot behaviour can be created. For example, if an agent wants to inform another agent about some information without expecting any reply from the target, this action can be incorporated in an OneShotBehaviour.

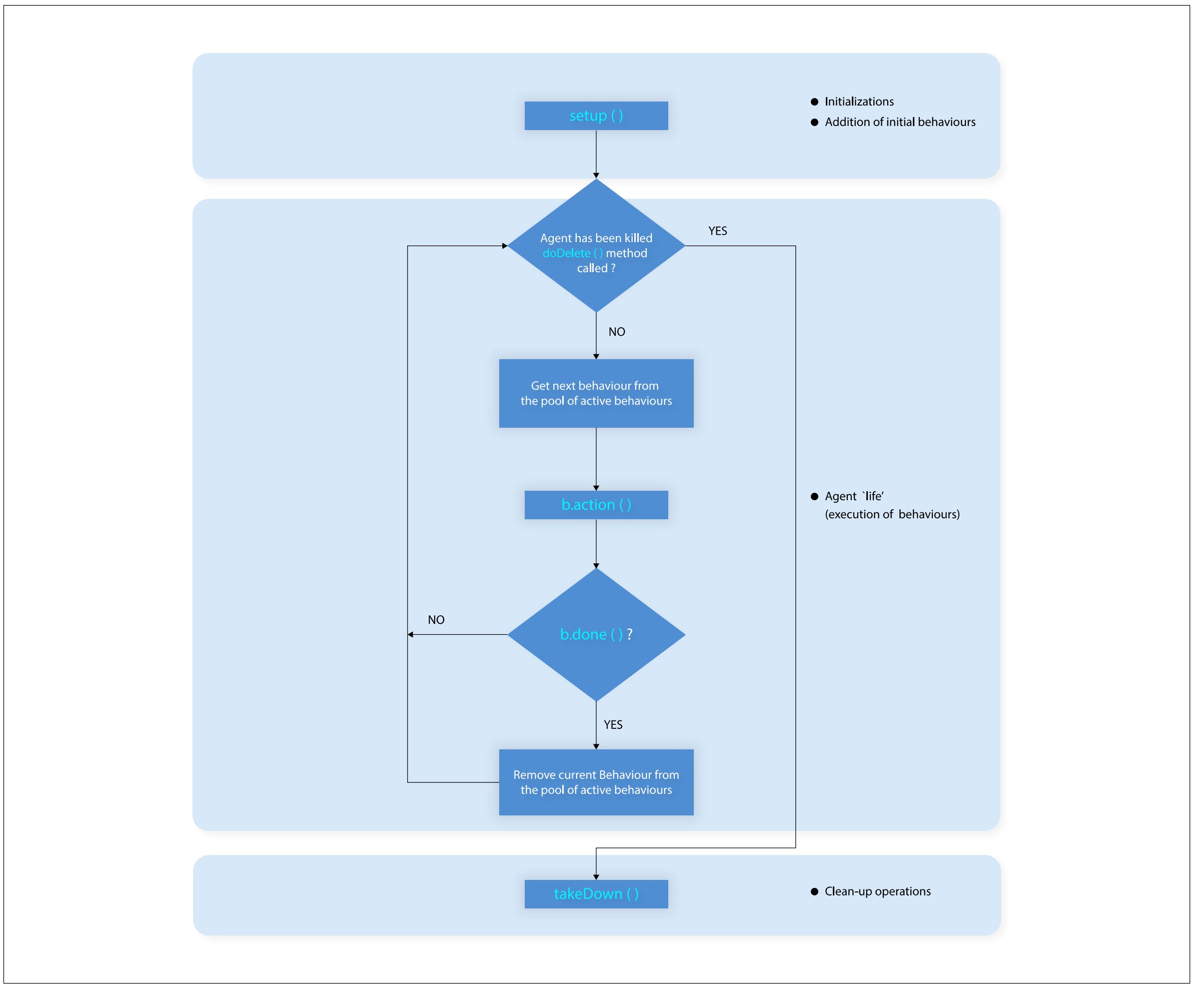

setup ()
Initializations
Addition of initial behaviours
Agent has been killed doDelete () method called ?
YES
NO
Get next behaviour from the pool of active behaviours
b.action ()
Agent `life' (execution of behaviours)
NO
b.done () ?
YES
Remove current Behaviour from the pool of active behaviours
takeDown ()
Clean-up operations

Figure 2.8: Agent behaviour scheduling policy in JADE [BCG07]. The methods are highlighted in neon blue are the methods that the programmers have to implement.

#### 2.3.2.2 Cyclic Behaviour

On the other hand, a `CyclicBehaviour` is a never-ending behaviour which keeps being iterated unless being explicitly blocked by using the `block()` method. The `done()` method always returns `false` in such behaviours. Any custom cyclic behaviour can be extended from `jade.core.behaviours.CyclicBehaviour` `class` . An example could be a situation when an agent is expecting a message from another agent, the action can be implemented using a CyclicBehaviour.

There are also other behaviours in JADE like the `SequentialBehaviour`, `TickerBehaviour` etc. but those can be easily be implemented by using combinations of `OneShotBehaviour` and `CyclicBehaviour`.

### 2.3.3 Mobility of Agents

According to standard definitions, mobile agents are everything that a non-mobile agent is (i.e. autonomous, reactive, proactive and social), but in addition they are also moveable. These agents can migrate between platforms containers in order to accomplish assigned tasks.

From the distributed systems point of view, a mobile agent is a program with a unique identity that can move its code, data and state between machines in a network. To achieve this, mobile agents are able to suspend their execution at any time and to continue once resident in another location [BCG07]. In JADE, the mobility can happen in two ways, the first one is by moving and second one is by cloning.

#### 2.3.3.1 Mobility by Moving

The Mobility is managed by a platform service called Agent Mobility Service. This provides software agents the ability to move between containers. The `doMove()` method is used by the agent to move from one container to another. This method requires the `Location` of the destination container as its input.

When invoked, this method initiates the process of moving the agent to the specified destination container. The majority of the code is located in the `jade.core.mobility` package. Migration not only includes the transmission of the code and the data but also the state of the agent. There are two other methods, `beforeMove()` and `afterMove()` to trigger operations right before or after the migration process.

#### 2.3.3.2 Mobility by Cloning

Similar to moving the cloning of an agent to a different container other than the local one is also counted as migration in JADE and very similar to the method related to moving, the `doClone()` `method` is used to clone an agent to a new container. Since two agents can not have a similar local name in a platform, a new parameter `newName` has to be specified along with the `Location` of the destination container in order to successfully execute this process.

### 2.3.4 Agent Communication Model

Communication and interactivity between agents are the fundamental characteristics of any multiagent system. In jade this is accomplished by agents through exchanging ad-hoc messages which each of the parties can understand. JADE follows FIPA standards so that ideally, JADE agents could interact with each-other even on remote platforms. The messaging standard set by FIPA is called FIPA ACL, where ACL stands for Agent Communication Language.

#### 2.3.4.1 ACL Message Structure

A FIPA ACL message contains a set of one or more message parameters. Precisely which parameters are needed for effective agent communication will vary according to the situation. For example, an ACL message can have a `conversation_Id`, `sender`, `receiver`, and ACL `performative` and sometimes a `content` as well. In JADE the sender and the receiver is identified by their unique AID. The content can be any `String` or a `Serializable`.

There are four different layers of an ACL message, an envelop layer which contains the transport information, a payload layer which contains encoded message, a message with all message parameters and a content layer which contains the actual message. FIPA defines three specific encodings: String (EBNF notation), XML and Bit-Efficient.

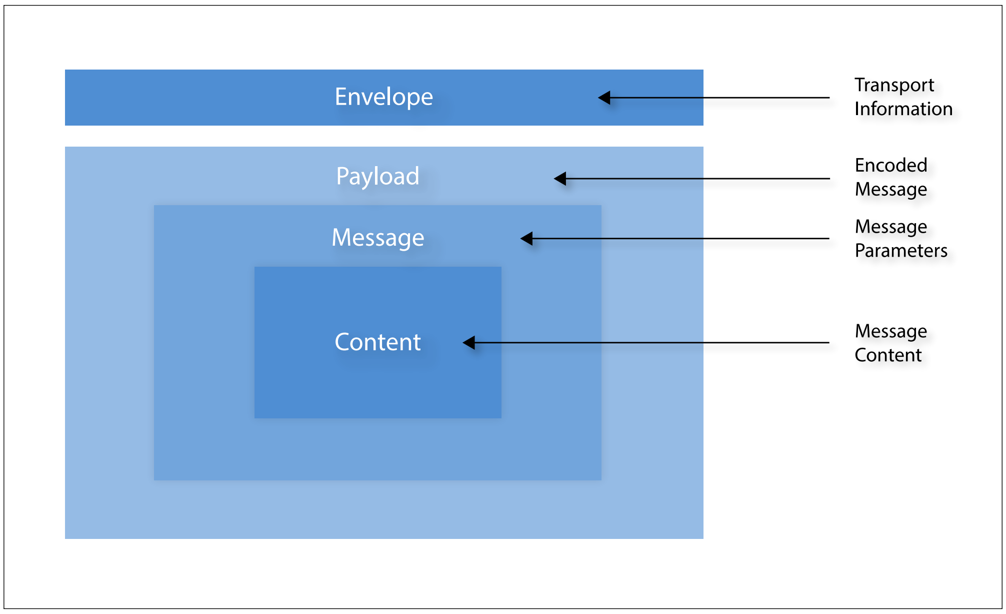


Figure 2.9: Structure of a FIPA ACLMessage.

#### 2.3.4.2 ACL Performatives

The FIPA-ACL defines communication in terms of a function or action, called the communicative act or CA, performed by the act of communicating. These functions are detailed in the FIPA CA Library specification, but examples include interrogatives which query for information, exercitives which ask for an action to be performed, referentials which share assertions about the world environment, phatics which establish, prolong or interrupt communication, paralinguistics which relate a message to other messages, and expressives which express attitudes, intentions or beliefs. If an agent does not recognize or is unable to process one or more of the elements or element values, it can reply with the appropriate *not-understood* [RAGBSM20].

Based upon the type of actions a performative can take it can be categorized as the following.[BCG07]

- **For Requesting Information** - subscribe, query-if, query-ref
- **For Passing Information** - inform, inform-ref, confirm, disconfirm
- **For Negotiation** - cfp (call-for-proposal), propose, accept-proposal, reject-proposal
- **Performing Actions** - request, request-when, request-whenever, agree, cancel, refuse

#### 2.3.4.3 JADE Communication API

in JADE an ACL Message object is created by instantiating `jade.lang.acl.ACLMessage` `class` which requires a ACL Performative [2.3.4.2] as an input parameter. While sending a, the receiver agent can be set by the `method` `addReceiver()` which requires the AID of the agent. A conversationId can be set by using `setConversationId()` `method`. The content of the message can be sent by using `setContent()` (for any `String` content) or `setContentObject()` (for any `Serializable` object content) `method` and finally, the message can be sent by using the the agent's native `send()` `method`.

On the receiving side, the agent have to listen to a conversationId in order to receive a message. A `MessageTemplate object` is first created by instantiating `jade.lang.acl.MessageTemplate` `class`. Then, the `MatchConversationId()` `method` is used to configure the MessageTemplate `object` to receive a reply with that specific conversationId. With the help of Agent's native `receive()` method, the `ACLMessage` can then be received. Once received, the agent can check the sender by using the `getSender()` `method` on the message `object`. The content of the message can then be retrieved by invoking the `getContent()` (for any `String` content) or `getContentObject()` (for any `Serializable` object content).

# Chapter 3

# Related Work

Naturally, one cannot talk about agent-based simulation without citing Netlogo[1]. Indeed, also in the context of simulating the immune system, we may find many applications based upon Netlogo (thorough review on the topic [CCP$^+$14]). When considering such applications, the most relevant difference w.r.t. our model is scalability. Netlogo is mainly an academic centralised simulator, and it is hard to decentralise (only features to perform concurrent executions exist), which makes it not suitable when the number of agents to simulate grows too much. Netlogo is good for designing simple agents and lacks its usability while designing Agents with very complex behaviours.

## 3.1 Category - I

One first category of works which are related to the one presented in this paper are the ones which belong to the area of Artificial Immune Systems (AIS).

- In [SS02], the authors present an artificial immune system based intelligent multi-agent model, named AISIMAM. In [SS02], AISIMAM is not used to simulate how the immune system works, but as it happens in other disciplines, such as genetic programming, AISIMAM aims at exploiting mechanics which efficiently work in nature. Specifically, it applies them to a mine detection scenario.
- A similar work can be found in [BF10], where a MAS is designed to mimic the human immune system behaviour. Also here, the resulting model is applied to a certain domain of interest, which is power system reconfiguration and restoration.

[1] `http://www.netlogoweb.org`

- AIS have also been applied in the robotic scenario [DGTM08], where autonomous mobile robots emulate natural behaviours of cells and molecules to realise their group behaviours (*i.e.*, cooperation). Similarly, in [HKR08], AIS are also used to engineer the organisational layer in a MAS, when exploited with simulated robot soccer.

- Another scenario where AIS have found application in MAS is the flexible job shop scheduling problem (FJSP). In [XF18], the authors analyse similarities between the FJSP and humoral immunity, which is one of the immune responses. Based on the similarities, a new immune multi-agent scheduling system (NIMASS) to solve the FJSP with the objective of minimizing the maximal completion time is presented.

- Agent-based Artificial Immune System (AbAIS) [OOW13] is a framework which uses a hybrid architecture where heterogeneous agents evolve over a cellular automata environment and are modelled following a genetic approach. AbAIs is applied to intrusion detection systems (IDS).

- In the paper Multi-agent-based modelling and simulation of high-speed train [KFS20] the authors have proposed a multi-agent-based modelling and simulation (MABMS) technology that can simulate the relationship between the individual behaviour and overall performance of the high-speed train's components. And by using this MABMS methodologies they were able to develop a simulation framework and platform that successfully integrates multi-disciplinary and heterogeneous simulation models with agent middle-ware and can drive an adaptive simulation of each agent. They have also noted that their method bypasses the need to set in advance the simulation parameters required in High Level Architecture which, significantly reduces the simulation time and makes the simulation more intelligent.

- ActoDatA (Actor Data Analysis) is an actor-based software library for the development of distributed data mining applications [LFM$^+$19]. It provides a multi-agent architecture with a set of predefined and configurable agents performing the typical tasks of data mining applications. In particular, its architecture can manage different users' applications; it maintains a high level of execution quality by distributing the agents of the applications on a dynamic set of computational nodes. Moreover, it provides reports about the analysis results and the collected data, which can be accessed through either a web browser or a dedicated mobile APP.

- Finally, we can find AIS implemented in JADE as well. Also in such works, the aim is not to simulate the immune system, but to take inspiration from it to solve different tasks. We may find [SS20], where an AIS is developed in JADE and used in a process for gas purification from acidic components, or [HEE$^+$19], where is used to minimize the microgrids operational cost and maximize the real-time response in grid-connected microgrids, or [HGY13], where a mobile agent-based system architecture is proposed in JADE for machine condition monitoring by imitating human immune system, or finally [DPG11], where the AIS in JADE is used to handle the disruption management in production systems monitoring and control.

- One other research article we have studied is related to HPC[2] and simulates spread of COVID-19 in human population using ABMS [PLCP22] methodologies. In this paper, authors claim that they were able to run the simulation on a large-scale system and were able to produce high resolution simulation results with the help of their own simulator. Using this simulator they were also able to model a use-case: the spread of COVID-19 across two Italian regions (Lombardy and Emilia-Romagna).

- The simulator *ActoDemic* is from the authors of the above article and has been published in a separate paper [PLM+21]. Structurally, the simulator is very similar to the one that we have designed for this thesis and uses JADE as the development framework. The authors have used this simulator to simulate the model of the COVID-19 spread that we have mentioned above. They have also used this simulator in the field of evolutionary computation and data analysis other than just the ABMS. *ActoDemic* can also be used for developing distributed applications. In fact, depending on the complexity of the application and on the availability of computing and communication resources, an application can involve one or more computational nodes. Here, each computational node maintains an actor space that acts as a "container" for a subset of the actors of the application and provides them with the services necessary for their execution.

Differently from our model, all these works are not interested on designing, engineering and developing an agent-based simulation of the immune system; instead, they are examples of how to take inspiration from the immune system to solve general MAS problems.

## 3.2 Category - II

The second category of works which are related to our contribution are the ones presenting agent-based simulations of the immune system. There are numerous existing works on agent-based immune system models. CAFISS [TJ05] models cell to cell interactions in a grid with each cell denoted through a bit string. In [TJ05], the authors describes a Java-based implementation of a framework for modeling the immune system, particularly Human Immunodeficiency Virus (or HIV) attack, using a Complex Adaptive Systems (CAS) model. The only downside of such solution is in its not being much scalable, indeed it has a large overhead caused by the use of separate threads for each cell. ImmSim [PKSC02, BC01] is a simulator based on cellular automata developed in APL2 [Fal91]; the framework

[2]High-Performance Computing

exploits task parallelism on distributed computers and, because of this, it is able to reduce execution time and it supports large-scale simulations. ImmSim is also the base on which ImmunoGrid [PHR$^+$09], and Sentinel [RG05], two other immune system simulators, have been built.

Swarms [JLL04] is a 3-dimensional model of the human immune system and its response to first and second viral antigen exposure (implemented in BREVE [SKPF05], a physics-based ABMS engine).

In the article *GPU acceleration and data fitting: Agent-based models of viral infections* [FD22] the authors have presented, a two dimensional biological system (very similar to our model of the grid system) is simulated with a mathematical model. The system is a culture dish of a mono-layer of cells with virus diffusing over the cells. The model is a combination of an agent based model (ABM) and a partial differential equation model (PDEM) where the cells are represented with an ABMS and virus diffusion is represented by a PDEM. This allows for simulation results within hours, but with the necessary level of detail to capture individual cell effects, and allows for parameterizing the model quickly. Here, the authors also claim that the model accurately replicates the diffusion of a virus, the stages of infection of individual cells, and can be fitted to data within hours.

ABM-DSGE is a framework made with an agent based model (ABM) and a Dynamic Stochastic General Equilibrium (DSGE) [KMWS22]. This model has been developed by the authors of the article to simulate and study the effects vaccination in human population in order to reach a “herd immunity”. The authors have also made a claim that they were able to reproduce a scenario where the vaccines and after-recovery immunity do change the dynamics of a contagion or reduce the adverse effects of pandemics on an economic scale.

For a further readings on existing ABM techniques to simulate the immune system, a thorough review can be found in [SK18].

To the best of our knowledge no agent-based immune system simulation has never been proposed in JADE before.

# Chapter 4

# Requirement Analysis

The immune system is our second most complex system, after the brain. It involves many different actors (*i.e.*, cells of our body's tissues, pathogens like bacteria and viruses), each one with different features, capabilities and objectives. Depending on the scenario, and the current state of our organism, we may find ourselves with a complete different set of cells involved in defending us, and that carry out different tasks.

## 4.1 The Secondary Immune Response Scenario

In this section we consider a simple – but correct – scenario, that will be used as the case study for checking the requirements of the model.

1. Some time ago, Alice got an infection from a virus $V$, which infected cells of tissue $TIS$ (target cells). The infection fired a reaction from the immune system that, besides successfully fighting $V$, also activated the generation of CD8 T-cells specific for $V$, that we name $CD8T(V)$. The infection also led to the generation of CD4 specific T cells. CD8 T-cells are a major cell population of the adaptive immune system and their primary function is augmented immune response once the pathogen that initially caused their generation ($V$ in this scenario) attacks the body again. CD8 and CD4 T-cells may 'wander' in the blood and in tissues close to those that had been infected by the pathogen, even months after the first infection. Importantly some memory CD4 and CD8 T cells will remain also in Lymph Nodes.
2. Alice comes into contact with $V$ again.
3. $V$ enters her body via her mouth and tries to infect cells of tissue $TIS$ again. It may

succeed in this attempt with some infection success rate $ISR$. Let us suppose that one cell $C(TIS)_a$ becomes infected. Two events take place:

(a) $C(TIS)_a$ becomes a virus-making factory: $V$ can in fact use it to move to 'infectable neighbouring cells', and infect them. The amount and density of infectable neighbouring cells depends on $TIS$, on the point the virus entered the body, and on other factors.

(b) $C(TIS)_a$ can be recognized as infected thanks to a peptide (an antigen) on its membrane surface, acting as a manifesto for its health state: when a cell is infected with a virus, it has pieces of antigens of that virus – the virus 'signature' – on its surface, not present when the cell is healthy.

4. Macrophages $MFG$ comes into this picture at this point in time although they are present in the tissue $C(TIS)_a$ looking for any pathogen even if there is no infection. Macrophages are general purpose killers, and they contribute in three ways:

   1) becoming infected and trying to eliminate the virus that they sense inside their selves,
   2) by phagocyting other dying infected cells that contain the virus, thus eliminating the potential reservoir of the virus,
   3) by producing soluble molecules ( called cytokines) that overall improve the immune response.

   But there is a limit on their ability to control infection. After this limit $MFG$ goes into a *exhausted* ($MFG_{exh}$) state where it exhibits less mobility.

5. One CD8 T-cell trained to recognize $V$, $VD8(V)_b$, that was wandering close to $C(TIS)_a$, perceives pieces of $V$'s antigen on the cell's membrane[1]. After this, $VD8(V)_b$ kills the virus.

6. On the other hand there are Dendritic cells($DEN$) moving around in the tissue and they capture any virus($V$) if it encounters one. It then disintegrates the virus and copies the genetic material of the virus $DNA(V_a)$ and synthesizes MHC Class - II molecules on the cell membrane we can denote it with $DEN_a$.

7. Afterwards, $DEN_a$ moves into the lymphatic system and heads towards the Lymph node.

8. At the Lymph Node, there are both naive and memory CD4 and CD8 T cells. Every memory T-Cell is slightly different that the other in terms of the type of virus $V_!$ it can encounter. $DEN_a$ finds for a match from this huge pool of naive T-Cells and activates it. This process spawns activated memory T cells targeted towards $V_a$.

[1]CD8 T-cells $CD8T(V)$ recognize *only* antigens of $V$, since they are virus-specific.

9. $CD4 + T_a$ then moves through the lymph vessel towards the site of infection whew it meets the exhausted Macrophages $MFG_{exh}$ and activates them back to normal $MFG$.

10. $MFG$s will recover their function and contribute with CD8 T memory cell to eradicate the secondary infection.

11. Depending on $TIS$, cells may be re-generated (cellular replication) to cope with the fact that some amount of $C(TIS)$ infected cells were killed; replication may accelerate to re-establish some stable number of cells, or may not take place at all, depending on $TIS$. For example, cells of the nervous systems cannot be replicated (perennial cells), hepatic cells can, but only as a response of a serious damage and up to some extent (stable cells), blood cells are continuously replicated (labile cells). If the damage is too vast, even tissues with labile cells may not be able to restore at the healthy situation holding before the virus infection.

## 4.2 Requirements

The first requirement is that the system has to be scalable to accommodate a large number of agents. Centralized systems as we have discussed earlier is not enough for this simulation. The next more important requirement is the fact that cells and all other actors in the immune systems are autonomous entities with very specific goals. That is the reason why an agent based development tool is suitable for the development. All immune cells are mobile, so the platform that to be chosen has to support agent mobility. For the virus we need a feature of the platform where any agent can be cloned at any given state. And finally, since the entities can interact with each-other in biology through sending chemical signals, we need our software counterpart to deliberately communicate with each-other in order to share information.

## 4.3 why choosing JADE for Modelling the Scenario

Given these requirements there are several reason for choosing JADE as the framework for developing this model.

- JADE[2] [BCG07] is a widely used agent framework. By being based upon Java, it is easy to learn and to integrate to existing solutions.

[2]`https://jade.tilab.com`

- JADE is structured around the idea of agents, behaviours, and containers. Agents follow the standard meaning, as main actors of the system.
- Each agent runs on a different thread, and it communicates with the other agents through FIPA[3] compliant messages.
- Behaviours denote how the agents act, and how they achieve their goals. Containers represent the abstract environment where the agents live, and can be distributed over multiple machines.
- It is important to note, that agents can move amongst different containers. This is going to be exploited in the project to simulate the passage of viruses and lymphocyte amongst different cells.
- Note that, since the containers can be deployed on different machines, the agents not only move, but their computations move as well. This aspect is of paramount importance in case the system to model becomes too big to be handled by a single machine; as it would happen for a realistic immune system comprised of millions of cells.

[3]`http://www.fipa.org`

# Chapter 5

# Design

The model has been designed to be a highly scalable and distributed agent-based Immune System Simulator. Even though in this thesis we only focus on couple of simple sub-systems of the overall immune system's response, we tackle all the fundamental aspects of its full engineering. Specifically, we address how to represent the main immune system's components inside JADE. There are two distinct parts of the design from the point of view of the simulation, a *physical* part which consists of actual biology entities (represented in JADE as agents) and a *meta-physical part* which consists of Agents that are present in the simulation for helping the biological agents with initiation and computation.

The physical part consists of a grid made with Auxiliary containers which forms the platform for agents representing the biology (Cells, T-Cells, Macrophages, Virus, Dendritic Cell and Lymph Vessel Agent) to interact with each other depicted in the Figure 5.1. Moreover, some of these agents can move along the grid following the movement rules set for the simulation, more about this will be discussed in the following sections.

However, there exists a meta-physical part of the model outside the grid. Agents like Initiator Agent, CD4TCell Manager etc. belong to such part and are discussed in the Implementation Chapter [6]. They are usually there to provide utility to the agents living in the grid.

In this chapter we discuss about the design choices we made while modeling the Grid System, the Innate Immune System, the Adaptive Immune System, and the Virus.

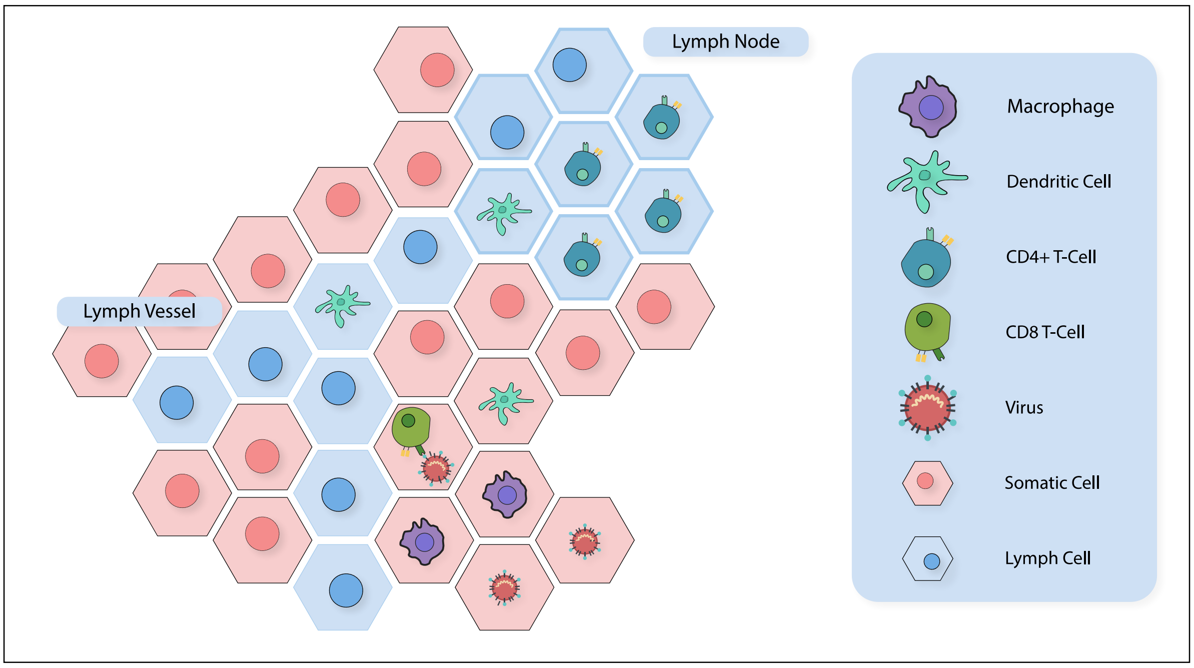


Figure 5.1: The Biological Model.

## 5.1 The Grid System

Each cell in the model does not live in an isolated island, rather is connected to other cells. This aspect of neighbourhood has been mapped into the model by implementing the concept of a grid-based system. A grid at it's unit level consists of a JADE container and a cell agent living inside that container. The total number of unit grids define the size of the Universe. In a typical 2-dimensional square-grid system the size of the universe will be the square of the number of unit grids present on any side. This is a model parameter and can tweaked to generate a grid with desired size. For example, in a $5 \times 5$ grid-system, the total number of grids will be 25, where each side of the grid has 5 unit grids depicted in the Figure 5.2.

There are multiple options that cone can choose from while designing the Grid. One of these options is to choose a grid where all of the Cell Agents live inside the same machine sharing the same pool memory. In this choice, all cells are created inside a single container and thus there is a centralized record about the positions of the cells. This is a good choice for a simple design but the problem arises when we increase the size of the universe by increasing the number of Cell Agents in it. This possibility has been explored by us in this article. [SBV22]

On the other hand, in a decentralized approach, each Cell Agent can be created in a single container. The choice of JADE as a Framework[1] for developments allows us encapsulate these Cell Agents in a JADE Container. The Cell Agent will then use the available computational resources in an isolated environment. The JADE Platform[2] also allows us to have these containers distributed in remote machines or in the local machine itself providing a higher degree of scalability. Unlike the centralized design, this design choice there does not limit the size to the universe since it is not limited by the computation power of a single machine. But poses other challenges like keeping the location information which is necessary to establish the notion of direction in the grid and neighbourhood for a cell since there is no centralized record maintaining this information. We can always define a centralized record and store it in a machine but that defeats the very purpose of the decentralization. In our model we have approached this problem by storing the neighbouring information of a cell agent locally in its own memory. We will discuss more about this next.

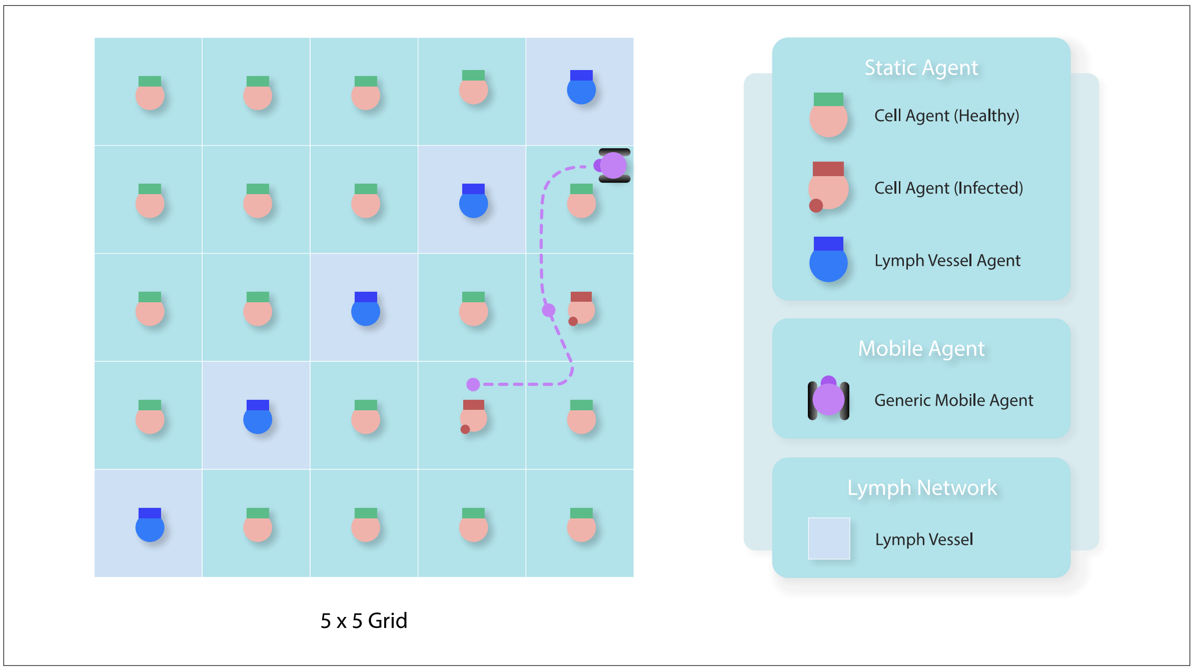


Figure 5.2: Representation of a Grid with Cells, Lymph Network, Mobile and Immobile Agents.

[1]An abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software.

[2]A platform is a set of hardware and software components that provide a space for developers to build and run applications.

### 5.1.1 Modelling the Cell

Cells are amongst the most important and fundamental building blocks of our model. They do not only provide a platform for various agents to communicate and mobilize across containers, but also work as an entry point for a pathogen in the simulation. Since cells are a part of the overall complex biological system, they have to keep track of infections, so that the immune response can be triggered to fight off the intruder.

Each cell in the model will interact with different actors present in the model like the Dendritic Cell, the Phagocyte, various T-Cells and the virus [Fel72]. Depending upon the type of agent it is interacting to, the interaction time will change which is a biological phenomena. The Cells in our model can be tweaked to exert this by manipulating the associated parameters.

Every Cell in an organism contains nearly similar genetic signature. This genetic signature which is represented in terms of genes which are usually collections of DNA base pairs. They vary in size from a few hundred DNA bases to more than 2 million bases in a human [GKQ16]. These genes dictates the behaviour of the cell throughout its lifetime. Thus changing the gene will lead to changes in the original behaviour of the cell. In our model, the genetic signature is carried by a sequence of binaries and stored locally in the memory of the Cell Agent itself [6.3.2].

As discussed here in 5.1, our model is decentralized. In this situation if an Agent need to move from one Cell to another in the grid, it has to to know the location of the next cell it is moving to. But, since there is no centralized record of the location of Cell Agents, the Agent in doubt can not make a query to any centralized entity to get the next location. In order to tackle this problem, at the initiation stage (which we will discuss in the implementation section, 6.3.1) the information about the neighbouring location of a current cell is stored locally in the memory of the Cell Agent in question. This information not only provides a sense of direction for the Mobile Agents [6.2] but also helps various cells to communicate among themselves.

According to what happens at the biological level, depending on the state of the infection, every cell adapts to a different behaviour at run-time.

#### Healthy

At the beginning of the simulation, each Cell Agent is healthy and has not been infected by any virus, yet. In this state, the Cell Agent listens for communications from any possible agents that are local to that container, including (possibly) a Virus Agent. In this phase, a Virus Agent can clone itself into the Cell Agent's container. Afterwards, the Virus Agent performs modifications in the Cell's genetics triggering the infection.

**Infected**

When a Cell is infected by a Virus Agent, it isolates itself and can not interact with other Virus Agents any longer. If Macrophages, or CD8 T-Cells (designed to detect the pathogen), are located inside the Cell Agent's container, they can kill the pathogen. When this happens, they can signal a repair command for the damaged genetic signature of the Cell, which upon taken an action, revives the cell back to it's healthy status. By that point, the process of infection can restart.

**Regenerating**

After the Cell has been killed by the T-Cell Agent, the Cell Agent can start a process of rebirth (induced by neighbour cells). Hence, a new Cell Agent is created to substitute the previously killed one. After that, the cell is considered healthy again and the process can restart.

We have to point out that the act of regenerating a new cell is not a trivial process, and is not always possible in real life scenarios. Indeed, not all cells in the human body can be regenerated; an example are brain and heart cells, which once they are lost, for any reason, are not regenerated by the body (causing the affected organ to function in a more limited way, *i.e.*, never fully recovering). In this model, we are currently focusing on cells that can be regenerated, according to some regeneration factor (which of course would depend on the typology of the cell).

Being a part of the grid-system the Cell Agent also holds the information about whether a lymph Vessel Agent is present locally or otherwise. This information is later on communicated with a Dendritic cell or a CD4+ T-Cell to let them either enter or exit the lymph vessel.

The presence of any exhausted Macrophage is also recorded by the Cell Agent. This information is then shared with any CD4+ T-Cell willing to stimulate a Macrophage. Once this stimulation proceeds, the state is updated by the Cell Agent itself to accommodate any further exhausted Macrophages.

### 5.1.2 Modelling the Lymphatic System

Lymphatic system is one of the most important part of the immune system that is composed of lymph nodes and lymph vessels. Lymph vessels work as the high connectivity lane for the lymphocytes to travel to and from the lymph nodes to the site of infection. The motion of the lymphocytes in these vessels are typically directional in nature. For example, if a Dendritic Cell wants to get to the lymph node while it is at the infection site it will look for a near-by lymph vessel and then follow it down to the lymph node which not only saves crucial time to reach the lymph node but also provides a path for the Activated CD4+ T-Cell to traverse from the node to the infection site very quickly.

### Lymph Vessel

The Lymph Vessels are introduced by creating the Lymph Vessel Agents that live locally with the Cell Agents in a specific container in such a way that they form a network. Every Lymph Vessel Agent keeps the information about the next Lymph vessel location, and so on. It also keeps information about any neighbouring lymph node and once it is asked by a Dendritic or a CD4+ T-Cell, it transmits both of these information to the later. This allows Dendritic (resp. CD4+ T-Cell) to enter the Lymphatic System. The movement time of these agents in the lymph vessel is quite low due to the directional nature of the lymph-movement. The direction itself is decided by the vessel agent. The network of these vessels typically spans across the entire grid making sure the least amount of untouched regions.

The choice of such a network is also very important since it dictates the overall reaction time of the Adaptive Immune Response [5.3]. There are several ways to imprint this network of vessel Agents onto the grid. One can simply specify the location of the containers containing Vessel Agents and that will make the container a lymph containing container. But then, these containers have to be connected in order to make it a network which requires one record to be maintained centrally. But, since our model is decentralized in nature, we have oped for a solution where we implement the lymph network by following a mathematical equation in the initiation phage. And since, the model is in 2-D realm, any bivariate[3] mathematical equation will be a suitable choice.

For example,

$$x = y \tag{5.1}$$

*i.e.*, a cross-diagonal path can be chosen as a Lymph Vessel path; where coordinates of each and every individual Vessel Cell will follow (5.1) relationship.

### Lymph Node

The end-point of a Lymph network is typically the Lymph Node, which is the destination for the Dendritic Cells after they have collected virus signature from the infection site. In

[3]Equation with two variables

our previous example, Lymph Network (5.1), a Lymph Node can be set-up at coordinate $[8, 8]$ in a grid [5.1] with size $9 \times 9$.

A Lymph node holds an arsenal (billions in number) of native T-Cells with the ability to neutralize a wide variety of pathogens. Keeping this enormous number of Agents, even though are inactive, is both computationally and memory-wise expensive. In order to tackle this problem, we have designed a probabilistic solution with the help of a CD4 T-Cell Manager Agent. Typically, there is always a chance that a certain match of native T-Cell could be found (we model that with a parameter that can be tweaked to simulate desired scenario).

### 5.1.3 Mobility across the Grid

As we have seen, the model consists of multiple agents with the characteristics of being mobile or immobile. For example, the Cell Agents and Lymph Vessel Agents are immobile agents. Indeed, they do not move from one Container to another. On the other hand, T-Cell, Macrophages, and Dendritic Cell Agents are mobile agents. The Virus Agents are not mobile, in the typical sense, but they can move amongst adjacent Containers by cloning themselves.

Since the grid is decentralized, the movement across the grid is settled at a local level. The movement is possible in the grid with 4-degrees of freedom (North-South, East-West and 2 cross-diagonals). There are two types of movements that are present in all of the mobile agents in the model: a directed movement and a direction-less movement. In both of these types of movements, a request has to be made to the local Cell or Lymph Vessel Agent by a mobile agent to retrieve the neighbouring location of the cells or the lymph vessel.

#### Directional Movement

As the name suggests a directional moment is a movement which is targeted and performed by an agent in order to move towards a specific location in the grid. Since our model is decentralized and distributed, the movement to the next location from current location has to be guided in order to achieve any directional mobilization. In our model this kind of movement is only possible inside the lymph network and are directed towards the Lymph Nodes. Any Agent (CD4+ T-Cell or Dendritic Cell) that wants to get to the lymph nodes are guided by the vessel agent present in the Vessel Containers (building block of the Lymph Network).

**Direction-less Movement**

On the other hand, the direction-less movement as the name suggest does not have any specific direction of movement. This behaviour is exhibited by certain Agents while searching for something in the grid. For example, while the Macrophages and CD4+ T-Cells search for virus, the Dendritic Cells search for both virus and Lymph Vessels, the CD8 T-Cell search for exhausted Macrophages in the grid. All of the mentioned movements are random in nature. Which means, these agents while in this behaviour, can choose the next location of movement at random from all 8 possible directions. And since the local cell agents already contain the information about all 8 of its neighbours (discussed earlier in 5.1.1), the agent in question can just query the local Cell Agent in order to retrieve these locations.

## 5.2 The Innate Immune System

The primary job of the Innate Immune response is to quickly detect any pathogen and neutralize it in order to completely eliminate or reduce the spread of the infection. There are multiple actors involved in this type of immune response, such as Phagocytes like Macrophages and Neutrophils, Dendritic Cells, Natural Killer Cells. One thing to note here is that the Dendritic cells belong to innate immunity but strictly interact with CD4 and CD8 T cells which belong to Adaptive immunity, thus, to keep the cohesion of documentation CD8 T-Cell is included in this section and Dendritic Cell is included in the Adaptive Immune System section [5.3.1].

The goal of creating this model has always been to simulate various scenarios where we closely replicate the corresponding biological phenomenon with an acceptable level of accuracy. The underlying biological model in question is very complex in nature and have more than 8 (with major contribution and many more with minor contribution) different immune cells namely, Macrophage, Neutrophil, Eosinophil, Basophil, Mast Cell, T-Cells, Monocytes and Natural Killer Cells to name a few. First of all, implementing all of which is time consuming and the complexity of their behaviours are beyond the scope of this thesis. Rather, we have focus on the usability of JADE as Framework to model some part of the vast immunology while not diverting far from our primary goal mentioned earlier. One important point to be noted here is the fact that other than the Macrophages, T-Cells and Mast Cells, all of the immune cells are short lived and lasts at most a month while the lifetime of these three in the time frame of years. On the other hand, the concentration of the Mast Cells is very low in number ($< 1\%$) in adult human [MWWK18].

Keeping this in mind and after discussing the importance of these two agents in the entire Innate Immune Response with with Professor Chiara Vitale from the Department of Experimental Medicine, University of Genova, that co-supervises this thesis and brings her experience as a biologist, we have focused mainly on implementing Macrophages and CD8 T-Cells in our model as a part of Innate Immune System. This choice does not only give us a very close resemblance of the overall behaviour of an Innate Immune System, but also simplifies such complex system enough to be able to be encoded with software agents.

### 5.2.1 Macrophages

The Macrophage Agents are agents born to control virus spreading and to improve immune response. They keep on moving in the tissue unless they find a Virus or dying virus infected cells " it would be a little more correct and you could anyhow associated these concepts to virus unit cell killing whether it is instrumental for your system at the moment. The immunity obtained from Macrophages is typically not related to any specific type of pathogen, these are general purpose killers. In terms of biological behaviour, there is a limit on how many Virus units a Macrophage can kill before it gets exhausted and stops killing. This limit exists in order to keep the collateral damage caused by the massacre low.

Based upon these two scenarios the behaviour of the Macrophages can be categorized in two different states, an Active State and an Exhausted state.

#### 5.2.1.1 Active State

In an active state the Macrophage Agent is mobile in nature, although the direction of motion is random. Upon moving to a new Cell, it looks for any Virus Agent present in the container by asking the Cell Agent for its *genetic signature*. The Cell Agent, as we have mentioned earlier, changes its genetic signature when infected [5.1.1]. This action by the Cell Agent allows the Macrophage Agent to possibly detect a local infection in the Cell Agent (*i.e.*, in its container). If that is the case, the Macrophage Agent makes an attempt to kill the virus by engulfing it. Although it is the case most of the time, sometimes random mutation in the Cell Agent's genetic signature can trigger this behaviour. This phenomenon is very common in nature which is generally due to the environmental reasons like ultra violet radiations or chemicals that the organism has been exposed to. Usually, some minor part of a cell's genetic material (DNA or RNA) is altered by these factors. Most of the time this is harmful and quickly been repaired by cellular proteins as a part of cell's internal mechanism. But not very often, these repairs are omitted and when enough of these accumulate, it can cause *cancer* [TV15].

#### 5.2.1.2 Exhausted State

In this state the Macrophage Agent needs a stimulation from the CD4+ T-Cell in order to revive its killing behaviour once again. This phenomenon has been modeled in the simulation by implementing a limit on the kill stats of the Macrophage Agent. This limit is a model parameter and can be tweaked to simulate a variety of immune conditions. Once this limit is reached, the Macrophage Agent shades its mobility and settles down to its last kill site to enhance the chance of being revived by a CD4+ T-Cell Agent afterwards. At this point, it informs the local Cell Agent about its presence in that container as discussed earlier in [5.1.1] and waits for a stimulation signal. Once it is stimulated, it updates the cell about this and moves on into the grid to continue the killing [5.2.1.1].

The concentration of these white blood cells (*i.e.*, Macrophages) varies widely in a human (due to age, sex, genetics, any underlying medical conditions, drug usage, nutrition and stress level) which impacts his/her immunity strength against any pathogen, in general [MWWK18]. In our model, the amount of Macrophages is identified by a parameter which can be tweaked to simulate a specific immunity in the system. Like in biology, the distribution of these Cells in the model is uniform across the grid in the beginning of the simulation.

### 5.2.2 CD8+ and CD4+ T-Cells

CD8 T-Cells are results from an adaptive immune response from a previous infection. After an infection is over an organism keeps some of the CD4+ T-Cells in their body in order to fight similar type of infection in future, which makes the organism immune to that type of pathogen. This type of immunity can be artificially achieved by vaccination as well. But depending upon any immune condition, the concentration of these cells may vary. Since they result from a previous infection, every CD8 T-Cell can only target a specific type of future infection and since an organism can have multiple types of infections from multiple pathogens in the course of its lifetime, there are a huge variety of these cells that are present in that organism [Sig16].

The complexity only increases when we include the time scale into the picture. The immune response in a present infection by the same pathogen will be much intense in case of an recent infection and less intense or even none in case of an infection happened years ago. In other words, the concentration of each variety of CD8 T-Cell is non-uniform in nature and in some cases there might not even be any CD8 T-Cell present in the system at all to fight off the current infection [Sig16].

This complexity has been modelled by a parameter which decides the percentage of the specific type of CD8 T-Cells present in the simulation to fight off the current infection. This parameter can be tweaked accordingly in consecutive simulation runs in order to simulate a variety of situations including one where the host is vaccinated prior to the infection.

Prior to the simulation run, the CD8 T-Cells are initialized with the type of pathogen it can encounter at run-time depending upon the parameter discussed earlier, and are uniformly distributed in the grid system [5.1]. Once the simulation starts a CD8 T-Cell moves around in the grid communicating with the local cell to check if the genetic signature has been altered by any pathogen. If it has been altered, it then tries to evaluate its ability to encounter the specific pathogen. Once it is confident about the signature match, it attempts to encounter the said pathogen. Upon failure, it moves on and repeats the said behaviour.

## 5.3 The Adaptive Immune System

Adaptive Immunity is also known as Acquired Immunity or Specific Immunity which is exclusive to vertebrates and is a subset of the Immune System. It's primary job is to create a long time immunity from past infections in order to quickly fight-off a new infection from a similar type of pathogen. Very similar to the Innate Immune System, there are many actors involved in this type of immunity. For example, Lymphocytes like B-Cells, T-Cells (Both CD8 [5.2.2] and CD4+), Dendritic Cells, and so on [MWWK18].

Although classified as two systems of immunity, the innate and the adaptive immune system are very much interconnected through the behaviours of their actors. which means, one Agent from one type of immunity triggers some behavioural changes or activates some other agent from the other type of immunity. In the Innate Immune System [5.2] discussed earlier, we have narrowed down our focused on the Phagocytes and the CD4+ T-Cells. This has helped us to narrow down our focus on implementing the Dendritic Cells and CD4+ T-Cells in our model which are the ones that are been mostly affected by their counter parts in Innate Immune System [MWWK18]. This choice of these actors not only gives us a very close resemblance with the overall behaviour of an Adaptive Immune System but also simplifies this complex system enough to be able to be encoded with the help of software agents.

### 5.3.1 Dendritic Cells

Dendritic Cells work as the intelligence office of the Immune System. While the Macrophage and CD8 T-cells are busy fighting the infection, the Dendritic Cell moves around and gathers information about the infection. The primary goal of a Dendritic Cell after detecting an infection is to recruit more CD4 T cell and induce their activation [5.3.2]. The CD4+ T-Cells can only be activated against a specific type of pathogen and we will discuss more about this in the Section 5.3.2. For this reason, the dendritic cell needs to carry the specific signature of the pathogen to the Lymph Node [5.1.2].

The behaviour of this agent changes drastically depending on its state. Below we will discuss about these states and the behaviours of the Dendritic Cell associated with these states.

#### 5.3.1.1 Virus Detection

The Dendritic Cell moves from one Container to another while in communication with the local Cell in that Container. If it finds any discrepancy it tries to decode the virus signature from the genetic signature of the Cell Agent.

In an actual biological system, the Dendritic Cell synthesizes and express a MHC Class-II molecule outside their cell membrane very specific to the type of pathogen it has encountered. To put it in other words, there is a one-to-one mapping between the signature of the pathogen and the type of MHC Class-II molecule being synthesized. In our model the Dendritic Cell Agent directly stores the viral signature.

#### 5.3.1.2 Mobility towards Lymph Vessel

Once the Virus signature has been captured, the Dendritic Cell starts roaming around looking for a nearby Lymph Vessel Cell. Once it stumble upon such a Container with a Lymph Vessel Agent present locally, it asks the Cell Agent in that container to let it enter the Lymph Network [5.1.2].

#### 5.3.1.3 Destination Lymph Node

Since the mobility inside the Lymph Network is directional and managed by the Lymph Vessel Agent itself, upon entering the Lymph Network the Dendritic Cell is guided towards the nearest Lymph Node. On the other hand, the Dendritic Cell keeps a record of the path it has taken to get to the Lymph node [5.1.2]. This information is necessary for the CD4+ T-Cells to get to the site of infection in record time.

#### 5.3.1.4 Activation of CD4+ T-Cell

Once at the Lymph node, the Dendritic Cell keeps looking for a naive T-Cell with the matching MHC Class-II receptors on its cell membrane. As we have discussed earlier in section 5.1.2, it is not computationally efficient, if not virtually impossible to keep billions of T-Cell. For this reason, the CD4TCell Manager Agent has been modeled. This Agent communicates with the Dendritic Cell and generates multiple copies of CD4+ T-Cell Agents with the Virus Signature and the information about the path to reach the site of infection obtained from the latter. Once these T-Cells are created, the goal of the Dendritic Cell is complete and it stays inside the Lymph Nodes until the end of the simulation.

Depending upon the immune condition of an organism, the concentration of the Dendritic Cells can vary. In order to cope with this situation, a parameter has been introduced in our model to tweak the amount of Dendritic Cells present in the grid system.

### 5.3.2 CD4+ T-Cells

The primary job of a CD4+ T-Cell is to re-activate the Innate Immune System [5.2] by stimulating the Macrophage s [5.2.1.2] at the site of infection.

There are two main behaviours that has been implemented in the model depending upon whether it is trying to get to the site of infection or trying to stimulate the Macrophage.

#### 5.3.2.1 Movement Towards the site of Infection

Once a native T-Cell has been activated by the Dendritic Cell [5.3.1.4] it begins its journey to get to the site of infection with the path-information collected from the latter. This movement is a directed movement till it leaves the Lymph Network. At the junction, where the Lymph Vessel ends according to its path, it communicates with the local Cell Agent to let it enter the tissue. The motion of this agent inside the tissue is directionless [5.1.3].

#### 5.3.2.2 Stimulating Macrophages

Outside the Lymph Vessel the CD4+ T-Cell Agent communicates with the Cell to gather information about any available exhausted Macrophage Agent [5.2.1.2] locally. If that is not the case, it continues its search. On the other hand, if it stumbles upon a Cell with an exhausted Macrophage parked in, it communicates with the latter. This act of communication stimulates the Macrophage which flips its state to *Active* [5.2.1.1]. Once the stimulation process is complete, the T-Cell sends a state update signal to the local Cell Agent.

The number of copies of CD4+ T-Cells are controlled with a parameter in the model which can be tweaked to simulate various immunological condition in the simulation.

## 5.4 Model of Virus

The Virus Agent mimics a virus in real life whose goal is to replicate itself exploiting the resources of the host cells. Differently from the Macrophages and T-Cells, the Virus Agent is not a mobile agent. In fact, it can only move by replicating itself to the adjacent cells. Like it happens in nature, the Virus Agent has a replication factor, which defines how many copies the Virus Agent can make of itself before destroying the host cell. Such replication factor is an input parameter of our model, and it can be customised to simulate different replication scenarios. The behaviours of the Virus agents can be summarized into three distinct natural actions.

### 5.4.1 Infecting a Cell

The Virus Agent is spawn in a random container in the grid at the beginning of the simulation (naturally multiple viruses can be spawn at the same time). Subsequently, the Virus Agent asks the cell in such container to change its own antigen peptide sequence by marking that the virus is now present in that cell.

### 5.4.2 Replication

Once the infection of the cell is completed, the Virus Agent starts replicating to the neighbour cells. This is obtained by spawning a random number (well within the replication factor) of instantiations of the Virus Agent in a random set of neighbour cells. Once a new Virus Agent is created inside a neighbour cell, then the same process described above is reiterated (*i.e.*, infection and replication, in this order). Since a cell could have been previously infected by another instance of the same virus, the Virus Agent also checks if the Cell Agent can, or not, be infected. In case, this is not possible, then the Virus Agent stops replicating and dies. Otherwise, it infects the Cell Agent and continues its replication.

### 5.4.3 Terminating The Cell

After a certain amount of time (which can be parameterised in the model), the Virus Agent kills the host Cell Agent and thereby commits suicide. Hence, once the resources of a cell have been completely consumed by the virus, the virus dies (*i.e.*, the corresponding Virus Agent is killed).

# Chapter 6

# Implementation

## 6.1 The Universe

The `Universe class` provides an abstraction for the simulation. A `universe object` is created from the same in every simulation with a given size. The replication factor of a virus that has to be introduced in the simulation and the strength of immunity against that specific virus is as the other two input parameters. Once the Universe class is instantiated, the `start() method` can be invoked. This method both initiates and starts the simulation.

Once the `start() method` is invoked, the simulator creates the Grid with the dimension provided. Afterwards, it also creates all of the `HashMaps` [1] [*e.g.* 6.1.1.1, 6.1.1.2, 6.1.1.3 and 6.1.1.4]. Then with the help of the Container Controller HashMap and Controller Grid Map the Cell Agents are placed inside the Auxiliary Containers and with the help of the Lymph Coordinate Map the Lymph Vessel Agents are placed in the Grid.

Since the model is distributed, no centralized record of locations like the `HashMaps` mentioned above can be used by any Agent after the initialization phase is over. For this reason an Initiator Agent [6.3.1] is created; it initiates the Cell Agents [6.3.2] and Lymph Vessel Agents by introducing them to their neighbouring Cells or Lymph Vessels to store the neighbouring locations locally which enables mobility across the grid for the Mobile Agents [6.2] once they are active.

Once the initiation phase is complete, all of the Mobile Agents are created in the Auxiliary Containers and the CD4TCell Manager is created inside the Primary Container. The Virus Agent needs to be Created with the help of the Virus Generator [6.1.2.3] before Creating the CD8 T-Cell [6.2.4] since it requires the generated Virus Signature to be able to create

[1] A HashMap stores information in a (key, value) pairs. Once stored, any specific value can be accessed by its key.

the immunity against the generated virus. The number of these Agents decide the strength of immunity against that specific virus in a simulation and taken as a parameter input while instantiating the `Universe class` .

The simulation, while running, can be safely terminated at any stage by applying the `stop()` method on the `universe object` .

### 6.1.1 Various Maps in the Universe

At the beginning of the simulation in order to properly organize the structure of the Grid System [5.1] and the Lymph Network [5.1.2], some `HashMap` s are created.

#### 6.1.1.1 Container Controller Hash Map

A Container Controller Hash Map is a `HashMap` of container names assigned by the universe and the controller of Auxiliary Containers (*a.k.a.*, JADE Container [2.3.1.1]) that form the base of the grid-system [5.1].

#### 6.1.1.2 Controller Grid Map

A Controller Grid Map is a `HashMap` of Auxiliary Container Controllers from the grid and their corresponding coordinate [5.1.3] which is assigned by the universe following the possible Movements [6.1.2.1] in the Grid [5.1].

#### 6.1.1.3 Lymph Coordinate Map

Lymph Coordinate Map is a `HashMap` containing the coordinate of a Lymph Vessel assigned by the universe following the Lymph Map [6.1.2.2] and its name which allows the Initiator Agent [6.3.1] to setup the Lymph Network [5.1.2].

#### 6.1.1.4 Lymph Path Map

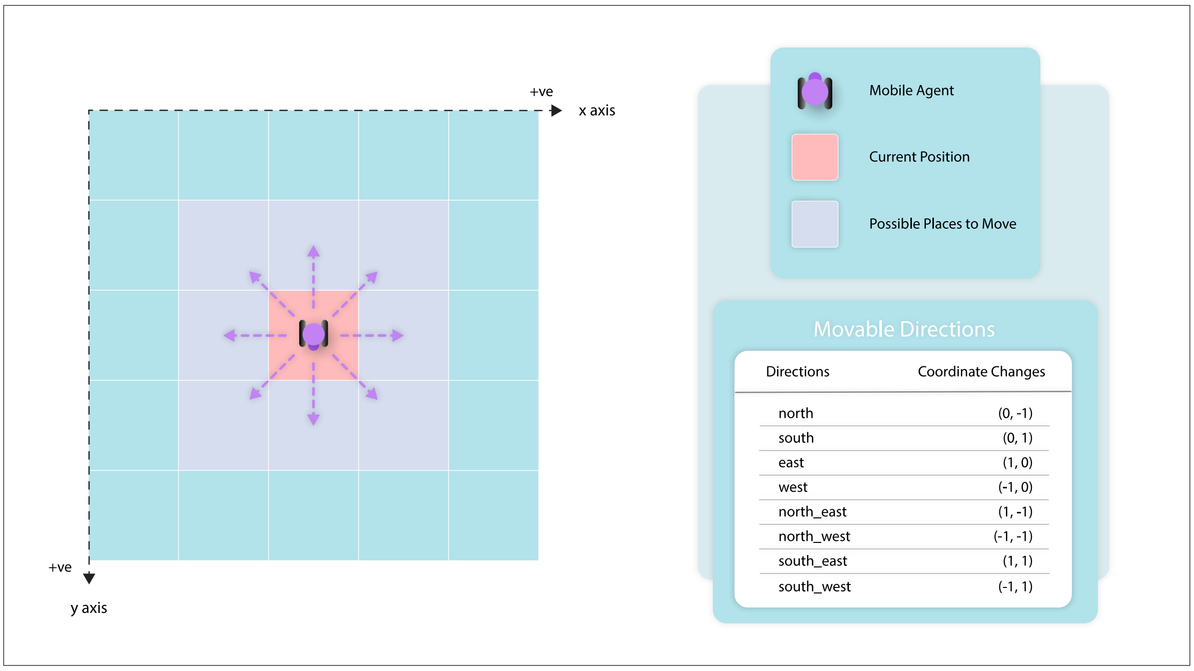
Lymph Path Map is a `HashMap` of the Auxiliary Container names and their Container Controller which is used by the Initiator Agent to form the Lymph Network.

### 6.1.2 Laws of the Universe

#### 6.1.2.1 Movement

The Movement class dictates how the mobility is implemented in the grid. The movement directions are represented according to the coordinate changes if moved in a certain direction. Let us discuss about it using the Figure 6.1.

In the grid, any unit step at any direction leads to a new unit grid. If a mobile agent is stationed in an unit grid (represented by Orange), it has 8 possible unit grids (represented by Purple) as options where it can move. Any unit move in east direction increases the x-coordinate of the mobile agent by one unit, and any move in south direction increases the y-coordinate by one unit. Similarly, any move in the west direction decreases the x-coordinate of the agent by one unit, and any move in south direction decreases the y-coordinate by one unit. These are the basis movements. Any other movement for example, south-east is equivalent to moving east by one unit and then moving south by one unit, is basically a linear combination of all basis movements.



| Directions | Coordinate Changes |
|---|---|
| north | (0, -1) |
| south | (0, 1) |
| east | (1, 0) |
| west | (-1, 0) |
| north_east | (1, -1) |
| north_west | (-1, -1) |
| south_east | (1, 1) |
| south_west | (-1, 1) |

Figure 6.1: Movement in the Grid.

Once the movements are defined, a `public` method `getAdjacentContainerControllers()` can be called with the current Container Controller as a parameter which computes the possible movements and returns an `ArrayList` of neighbouring Container Controllers in all possible directions.

#### 6.1.2.2 Lymph Map

The Lymph Map `class` dictates the blue-print of the Lymph Network [5.1.2] in the grid. Given a coordinate it returns the coordinate of the next Lymph location. This information is then used by the Initiator Agent to set-up the Network of Lymph Vessel Cells and Nodes [6.3.1]. As discussed in Lymph Vessel Design [5.1.2], any suitable bivariate mathematical equation can be used to design the network. Here is the representation of the equation used in our model.

```
/* vessel coordinate Equation x = y */
int dx = 1;
int dy = 1;

int nextVessel_x = current_x + dx;
int nextVessel_y = current_y + dy;

int[] nextCoordinate = new int[] { nextVessel_x, nextVessel_y };
```

Listing 6.1: Representation of a Mathematical Equation.

#### 6.1.2.3 Virus Generator

With the help of Virus Generator it is possible to generate a variety of Viruses with different features that determine how a virus behaves in a simulation. The behaviour of a virus depends on multiple parameters which includes the replication factor of the virus, how much time it takes to replicate, how efficient is the virus infecting its host Cell Agent, and the time it takes to exhaust the host's resources (thus killing it). All of these parameters associated with the Genetic Signature [5.4] of the virus, to put it in other words, there is an one-to-one map between the values of these parameters and the genetic signature of the Virus Agent.

```
int virus_replication_factor;
int virus_cell_communication_time = 2000 * Constants.SIMULATION_TIME_SCALE;
int virus_replication_time = 20000 * Constants.SIMULATION_TIME_SCALE;
int time_to_kill_the_cell = 100000 * Constants.SIMULATION_TIME_SCALE;
```

Listing 6.2: Parameters of the Virus.

Given all of these parameters and an Auxiliary Container Controller, a Virus Agent can be generated inside the grid by calling a `public` method `generateVirus()` on a `virusGenerator` object which returns the Genetic Signature of the generated Virus.

After a Virus Agent is generated, it is in an inactive state, which means all of its behaviours are not active yet. In order to activate the Virus, the `activateVirus()` method has to be called which internally invokes a `start()` method (provided by the JADE Framework) on the `virusAgentController` which activates the virus behaviours.

```
public void activateVirus() {
    try {
        virusAgentController.start();
    } catch (StaleProxyException e) {
        e.printStackTrace();
    }
}
```

Listing 6.3: Method to Activate the Virus.

#### 6.1.2.4 Constants

The Constant `class` stores different constant parameters that do not change in a single simulation like the original Genetic Signature (the DNA) of the Cell Agent before it has been modified by the Virus Agent, the time it takes to communicate among various Agents, concentration of different agents like the Macrophages, Dendritic Cells, T-Cells etc. Some of these parameters can be found below.

```
public static int SIMULATION_TIME_SCALE = 1;

public static final int MACROPHAGE_SLEEP_TIME = 3000 * SIMULATION_TIME_SCALE; //Seconds
public static final int MACROPHAGE_CELL_COMMUNICATION_TIME = 100 * SIMULATION_TIME_SCALE;

public static final int[] CELL_IDENTIFYING_DNA = new int[] {0, 1, 1, 0, 1, 0, 1, 0, 0, 1,
    1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1,
     0, 0, 1, 0, 1, 0};

public static final int CELL_MUTATION_PERIOD = 5000 * SIMULATION_TIME_SCALE; //Seconds
public static final int CELL_REGENERATION_TIME_AVG = 4000 * SIMULATION_TIME_SCALE;

/* Virus Constants */
public static final int[] GENERIC_VIRUS_SIGNATURE = new int[] {10, 25, 20, 22, 15};
public static final int VIRUS_SIGNATURE_LENGTH = CELL_IDENTIFYING_DNA.length * 30 / 100;
    // 30% of Cell DNA Length
```

Listing 6.4: A few constant Parameters.

One more important thing to note here is the fact that each codon of the DNA of the Cell is represented in terms of a binary rather than a base four system found in nature, where each codon can be one of $A$, $T$, $G$ or $C$. But this does not impact the behaviour of the model since a pair of binaries could represent all of these four states of an unit codon.

For example,

$$\begin{array}{cc} \{0,0\} \implies A & \{1,0\} \implies T \\ \{0,1\} \implies G & \{1,1\} \implies C \end{array}$$

On the other hand, the the Genetic signature of the virus implies the positions at Cell's DNA it introduces a change. For example, if a virus signature is $\{3, 8, 12\}$, it will flip the binaries at the 3rd, 8th and 12th position of the Cell DNA. Generally the length of a Virus Signature is 30% of the length of the Cell's DNA which means, once a virus infects a Cell Agent, 30% of its DNA has changed.

All of the parameters related to the concentration of different agents are represented in percentage of total number of cells present in the model at the beginning of the simulation, which is equal to the size of the Grid.

The parameter `SIMULATION_TIME_SCALE` dictates how slow the simulation is been performed. If the value of this parameter is set to 2, all of the communication between the Agents, the movements of the Mobile Agents and the replication of the virus will take twice as much time to complete.

## 6.2 Mobile Agents

Mobile Agents as the name suggests can traverse across the grid with the help of the Cell Agents [6.3.2] outside the Lymph Vessel or with the help of Lymph Vessel Agents [6.3.3] while inside the Lymph Network [5.1.2]. One such movement is commenced by applying the native `doMove()` method provided by the JADE Framework with the `location` of the destination passed as an argument.

### 6.2.1 Macrophage Agent

At the beginning of the simulation the Macrophages are created randomly inside the grid. After that the goal of this agent is to search and kill a virus. But as discussed earlier in 5.2.1 there is a limit on how many kills it can make before it gets exhausted and that is stored in a variable within the `Macrophage class`. Depending upon this value the behaviour of the Macrophage Changes.

#### 6.2.1.1 Searching for a Virus

Search for a virus in the Grid requires the Macrophage to perform multiple OneShot-Behaviours[2] in sequence. First, it has to move to a neighbouring Container by using the `doMove() method` then it has to check if the Check if the Cell Agent is alive in the Container with the help of `isCellAlive() method` from the `AuxiliaryContainer class` or it is already been killed by the virus. If the Cell is alive, then it has to communicates with the local Cell Agent through `Signature_Verification_Channel` by ex-

[2]OneShotBehaviour is a behaviour that is executed once in an Agent's Life Cycle.

changing `ACLMessage`. As a reply the Cell Agent then sends its DNA as an `Array` of `Integers` which is then stored by the Macrophage Agent. After that it has to trigger its `DetectingAndKillingVirus` behaviour.

#### 6.2.1.2 Detecting and Killing a Virus

Once this OneShotBehaviour is triggered, the Macrophage looks for changes in the DNA of the Cell by matching it with the ideal DNA sequence of the cell discussed 6.1.2.4. If an anomaly is found in the genetic code, which essentially due to the presence of a Virus agent. In that case the Macrophage carries on with killing the virus Agent. In order to do that the Macrophage retrieves the `agentController` of the Virus Agent and calls the native `kill() method` on in. At this point the kill record is updated by one kill and its virus searching behaviour is triggered back again.

#### 6.2.1.3 Switching to Exhausted State

Once the kill limit of the Macrophage has been reached, this behaviour is triggered. In this behaviour the Macrophage loses its mobility. The local Cell Agent is then informed by the through `exhausted_macrophage_cell_connection_channel` by sending an `ACLMessage`.

#### 6.2.1.4 Waiting for a Stimulation

Once the Exhausted state has been triggered, the Macrophage waits for a stimulation from a CD4+ T-Cell Agent. This behaviour is a `OneShotBehaviour`. Here, the Macrophage keeps listening to `cd4t_macrophage_communication_channel` unless a message from the CD4+ T-Cell Agent is received. Once a message is received and the message content is `stimulate_macrophage`, the Macrophage is stimulated and resumes its behaviour of searching and Killing the virus.

### 6.2.2 Dendritic Cell Agent

Similar to the Macrophage Agent, the Dendritic cells are also created randomly inside the grid at the beginning of the simulation. The Goal of the Dendritic Cell is to present the virus signature to the CD4+ Cell Manager inside the Lymph Node. This is achieved by two sequences of `OneShotBehaviours`. The job of the first sequence of behaviours is to Search, Detect and extract the virus *genetic signature* while the job of the final sequence is to search a Lymph Vessel, enter the Lymph Network, reach the Lymph Node and sate up a communication with the CD4+ T-Cell Manager.

#### 6.2.2.1 Searching for a Virus

The searching for the virus behaviour is similar to that of the Macrophage Agent discussed in 6.2.1.1. But instead of Killing the Virus, Detecting and extracting behaviour is triggered.

#### 6.2.2.2 Detecting and Extracting Virus Signature

Once a Virus Agent has been detected, the Dendritic Cell scans through the Cell's DNA looking for the places where the change has been made by the virus. From our previous discussion in 6.1.2.4 we know that the virus imprints its signature by changing the cell's DNA sequence at specific location. Thus recording these changes during the scan extracts the virus signature. The Macrophage then stores this signature in an `Array` and triggers a new behaviour of searching for the nearby Lymph Vessel.

#### 6.2.2.3 Searching for a Lymph Vessel

Once this behaviour is triggered, the Dendritic Cell starts moving to neighbouring Cells using the native `doMove()` `method`. But this time instead looking for a virus, it contacts the local Cell Agent through `vessel_confirmation_channel` by exchanging `ACLMessage` if there is any Lymph Vessel Agent present locally in the current container. If that is not the case, it moves to a new container and repeats the same behaviour. But if it stumbles upon a container with a Lymph Vessel present inside, a new behaviour (`ContactVesselInCell`) is triggered.

#### 6.2.2.4 Entering into Lymph Vessel Network

The Dendritic Cell enters the Lymph Vessel Network by contacting the Vessel Agent present inside the local Container `Vessel_Dendritic_Communication_Channel` by exchanging `ACLMessage`. The reply is then sent by the Vessel Agent with the information about the next vessel in the Lymph Network. This `OneShotBehaviour` terminates once the next destination is set which triggers the behaviour for moving towards the Lymph Node.

#### 6.2.2.5 Reaching The Lymph Node

This Behaviour is achieved by repeating a sequence of two `OneShotBehaviour`s. The first Behaviour is checking if the current location is a Lymph Node or Otherwise. If it the case, then the state value is change as the Lymph Node has been reached and the next behaviour is triggered. Otherwise, The local Vessel Agent is asked about the next Vessel Location and the Dendritic Cell is moved to that next location using `doMove()` `method` and the behaviour repeats until it reaches the Lymph Node. During this process the Dendritic Cell keeps a track of its path towards the lymph node by storing these locations in an `ArrayList` of `Locations`.

#### 6.2.2.6 Communicating with the CD4TCell Manager

Once it reaches the Lymph Node, the CD4+ T-Cell Manager in the MainContainer [**??**] is contacted through `dendritic_cell_c4t_communication_channel`. In this communication, the Dendritic Cell sends both the Virus Signature and the path it has taken from the site of infection to reach the Lymph Node is sent. But since only `serializable` can be sent through `ACLMessage`, the information has been be serialized by implementing `java.io.Serializable`. The example is presented below.

```java
import java.io.Serializable;
import java.util.ArrayList;
import jade.core.Location;

public class DendriticCellInformation implements Serializable{
    public int[] virus_signature;
    public ArrayList<Location> path;

    public DendriticCellInformation(int[] virus_signature, ArrayList<Location> path) {
        this.virus_signature = virus_signature;
        this.path = path;
    }
}
```

Listing 6.5: Serialization of Information for ACLMessage.

### 6.2.3 CD4+ T-Cell Agent

The CD4+ T-Cells are created inside the `MainContainer` by the CD4 T-Cell Manager Agent once it is been contacted by the Dendritic Cell. The Main goal of these agents are to get to the point of infection as quickly as possible in order to stimulate the exhausted Macrophage Agents. During the creation the *virus signature*, the `Location` of the Lymph Node and the path to reach infection site are stored inside them.

#### 6.2.3.1 Moving to Lymph Node

The first thing it does after activation is to move to the Lymph Node using the native doMove() method with the Location provided during the initiation. This behaviour is a OneShotBehaviour. Once at the Lymph Node the behaviour of travelling to the infection site is triggered.

#### 6.2.3.2 Travelling to the Site of Infection

Travelling to the site of infection is achieved by performing a CyclicBehaviour[3] called MoveToNextVessel. This behaviour uses the path provided during the initiation of this agent to get to the end of the vessel and once it has reached there, the behaviour is removed by native removeBehaviour() method. At this stage an exhausted Macrophage can be right next resting into the local Container so it triggers a check on the local Cell Agent about the presence of the Macrophage. The implementation of this behaviour is presented below.

```java
private class MoveToNextVessel extends CyclicBehaviour {

@Override
public void action() {
    if (path_to_site.size() > 0) {
        doWait(Constants.CD4TCell_SLEEP_TIME);
        doMove(path_to_site.get(0));
        path_to_site = new ArrayList<>(path_to_site.subList(1, path_to_site.size()));
    } else {
        removeBehaviour(this);
        addBehaviour(new CheckIfExhaustedMacrophagePresent());
    }
}
```

Listing 6.6: Code snippet of how does the CD4+ T-Cell move inside the Lymph Vessel Network.

#### 6.2.3.3 Searching for an Exhausted Macrophage

The checking behaviour is a OneShotBehaviour performed by communicating with the local Cell Agent through asking_cell_exhausted_macrophage_connection_channel by exchanging ACLMessage . If the reply is negative, the behaviour terminates triggering a new search behaviour similar to the one discussed in 6.2.1.1. But instead of looking for a Virus, it looks for an exhausted Macrophage by travelling across the grid until it stumbles upon one such Agent. But if the reply is affirmative, it also receives the AID[4] of the tar-

[3]A behaviour which is cyclically executed by the JADE agent (until is not removed).

[4]Instance of the class that represents a JADE Agent Identifier

get Macrophage. In that scenario this behaviour terminates triggering a new Macrophage stimulation behaviour.

#### 6.2.3.4 Stimulating the Macrophage

This behaviour is composed of a sequence of two `OneShotBehaviour`. The first of those is the one which stimulates the exhausted Macrophages waiting to be stimulated[6.2.1.4] while listening to `cd4t_macrophage_communication_channel`. This agent then sends a `stimulate_macrophage` signal in that communication channel which stimulates the Macrophage. The implementation of this behaviour is presented below.

```java
private class StimulatingMacrophage extends OneShotBehaviour {

    String conversationID = "cd4t_macrophage_communication_channel";
    String instruction = "stimulate_macrophage";

    @Override
    public void action() {
        ACLMessage message = new ACLMessage(ACLMessage.INFORM);
        message.setConversationId(conversationID);
        message.addReceiver(exMacrophageAID);
        message.setContent(instruction);
        send(message);

        addBehaviour(new TellingCellSimulatedAboutMacrophage());
    }

}
```

Listing 6.7: Code snippet of CD4+ T-Cell stimulating a Macrophage.

The other behaviour which gets triggered at this point is the one that sends an update signal to the local Cell Agent about this stimulation event using the same `ACLMessage` protocol on the channel, `telling_cell_macrophage_channel_cd4t`.

### 6.2.4 CD8 T-Cell Agent

CD8 T-Cells are the the targeted killers which are designed to execute a specific type of Virus Agent identified by their *virus signature*. This virus signature if fed into these agents in the initiation stage as discussed in 6.1. In any simulation not all of these agents are designed to kill the target Virus Agent, hence the number of those which can, defines the immunity against the target virus. At the beginning of the simulation these agents are uniformly distributed among the grid, and once activated they starts searching for the virus in the grid.

#### 6.2.4.1 Searching For a Virus

This Behaviour of searching for the virus is very similar to that of the Macrophage discussed here in 6.2.1.1. Once an anomaly has been detected in the DNA of the Cell Agent, this behaviour is terminated triggering a new virus signature checking behaviour.

#### 6.2.4.2 Virus Signature Check

This is a `OneShotBehaviour` and once triggered it scans through the Cell's DNA looking for the places where the change has been made by the virus. From our previous discussion in 6.1.2.4 we know that the virus imprints its signature by changing the cell's DNA sequence at specific location. Thus recording these changes during the scan extracts the virus signature. The CD8 T-Cell Agent then checks through the virus signature to find out if it is capable of neutralizing the virus by comparing this signature with the one that is stored in it. upon receiving an affirmative result it calls the `protected killTheVirus()` method to attempt a kill, otherwise it terminates this behaviour and moves to a neighbouring container repeating 6.2.4.1. Some implementation of this behaviour is presented below.

```java
if (!Arrays.equals(cellDNAToBeVerified, Constants.CELL_IDENTIFYING_DNA)) {
    ArrayList<Integer> differenceList = new ArrayList<>();
    for (int index = 0; index < cellDNAToBeVerified.length; index++) {
        if (cellDNAToBeVerified[index] != Constants.CELL_IDENTIFYING_DNA[index]) {
            differenceList.add(index);
        }
    }

    int[] differences = differenceList.stream().mapToInt(i -> i).toArray();

    if (Arrays.equals(differences, virus_signature)) {
        try {
            killTheVirus(myAgent);
            myAgent.addBehaviour(new MovingToNewCell());
        } catch (ControllerException blocked) {
        }
    }
}
myAgent.addBehaviour(new MovingToNewCell());
```

Listing 6.8: Code snippet showing the CD8 T-Cell checking for a Virus signature.

#### 6.2.4.3 Killing the Virus if Possible

Once the killing method is invoked, the agent retrieves the `AgentController` of the Virus Agent with a series of operations. First, the `ContainerController` of the current Container is retrieved by using native `getContainerController() method`. And Finally, by applying the native `getAgent() method` on `currentContainerController object` the `AgentController` is retreaved. Once available without any `ControllerException`[5], the native `kill() method` can be called on the `agentController object` to terminate the Virus.

## 6.3 Immobile Agents

As the name suggests, these type of agents are immobile in the Grid. They can either live inside the Auxiliary Container (*a.k.a.*, JADE Container [2.3.1.1]) providing the structure and the navigation to the Grid enabling the Mobile Agents [6.2] to traverse or in the Main Container (*a.k.a.*, Primary Container [2.3.1.2]) providing utility to the other Agents. Here, the Virus Agent is an exception, while it is immobile in nature but can spread across the Auxiliary Containers in the grid by replicating itself to the neighbouring containers.

### 6.3.1 Initiator Agent

Initiator Agent is the agent that lives in the meta-physical Main Container and initialized Cell Agent and the Lymph Vessel Agents in the beginning of the simulation, and once the initiation is complete, it sits idle. This agent is created with Controller Grid Map [6.1.1.2], Lymph Path Map [6.1.1.4] and Lymph Coordinate Map [6.1.1.3] from the Universe [6.1]. Now let us discuss the behaviours of this agent.

#### 6.3.1.1 Initiating the Cell

With this `CyclicBehaviour` this agent collects the `Location` of the Cell Agents present in the grid. The collection process starts by iterating over the Controller Grid Map to get the `ContainerController`. Next, the Cell Agent present in that container is contacted by sending an `ACLMessage` on a specific channel where the Cell Agents are listening to. But in order to do that the Initiator Agent has to activate the Cell Agents by invoking `start() method` since they are inactive at the beginning of the simulation. Once Activated, the Cell Agents can then send their location as a response to the query by this Agent. The Location is then stored in an `ArrayList` of `Location objects`. Once the locations of all of the Cell Agents have been collected, this CyclicBehaviour is terminated and a new behaviour is triggered for assigning neighbours to the Cell Agents.

[5]ControllerException class is thrown when an operation fails on any of the agent controller methods.

#### 6.3.1.2 Assigning Neighbours to the Cell

The neighbouring Cell Locations are assigned to the a Cell with this `OneShotBehaviour`, which uses the `findNeighbourLocation()` `method` with the `ContainerController` of the Cell Agent in order to get an `ArrayList` of `Location` `objects`. This `method` in tern instantiates the `Movement` `class` [6.1.2.1] and invokes the `method` to discover the `ArrayList` of neighbouring `ContainerControllers`. Then the `ArrayList` of neighbouring Containers are retreaved from the `ArrayList` of `Location` `objects` stored in while Initiating the Cell Agent. Once this Location List is ready, it is then sent to the Cell agent through `neighbour_allocation_channel`. Now this process is repeated for all of the Cell Agents in the grid by iterating over the Controller Grid Map. The implementation of `findNeighbourLocation()` `method` is presented below.

```
private ArrayList<Location> findNeighbourLocation(ContainerController containerController)
throws ControllerException {
    ArrayList<Location> neighbourLocationList = new ArrayList<>();
    ArrayList<ContainerController> neighbouringContainerController;

    Movement movement = new Movement();

    neighbouringContainerControllers = movement
                    .getAdjacentContainerControllers(containerController);

    for (ContainerController eachNeighbourController : neighbouringContainerControllers) {
        neighbourLocationList.add(containerControllerLocationHashMap.get(
            eachNeighbourController));
    }

    return neighbourLocationList;
}
```

Listing 6.9: Implementation of findNeighbourLocation() method.

While iterating, it also triggers a new `OneShotBehaviour` to inform the concerned cell about the presence of a vessel agent locally.

#### 6.3.1.3 Telling the Cell Agents About a Local Lymph Vessel

This behaviour checks if the concerned Cell Agent is in the Lymph Path Map with the help of `checkIfCellisaVessel()` `method`. The result is then shared with the Cell Agent through `vessel_cell_connection_channel`.

#### 6.3.1.4 Initializing the Lymph Vessel Agents

Initializing the Lymph Vessel Agents is very similar to initializing the cell agents [6.3.1.1] in terms of retrieving the `Location` of the Lymph Vessels by using the Lymph Path Map.

But instead of putting them in an `ArrayList`, the `Location` `objects` are put in a `HashMap` with the vessel name as `key` and the vessel location as `value`, and stored in memory. Once all of the vessel locations from the Lymph Path Map is stored, the behaviour terminates with triggering a new behaviour to Assign the Next Vessel Location to each of the Vessel Agents.

#### 6.3.1.5 Assigning Next Vessel Locations

Very similar to the process discussed in 6.3.1.2, this behaviour is also a `OneShotBehaviour` and a very similar `method` `findNextVessels()` is used to find an `ArrayList` of the next vessel locations, given a vessel's coordinate. Once it is retreaved, the `ArrayList` is then sent to the corresponding Vessel Agent. The process is then repeated for all of the Vessel Agents in the Lymph Network. The implementation of `findNeighbourLocation()` `method` is presented below.

```java
private ArrayList<Location> findNextVessels(String vesselName) {
    ArrayList<Location> nextVesselLocations = new ArrayList<>();

    int[] currentVesselCoordinate = lymphVesselCoordinateMap.get(vesselName);
    LymphMap lymphMap = new LymphMap();
    ArrayList<int[]> coordinatesOfNextVessels = lymphMap.getNextVessels(
        currentVesselCoordinate);

    for (int[] coordinate : coordinatesOfNextVessels) {

        String nextVesselName = null;
        for (String vessel : lymphVesselCoordinateMap.keySet()) {

            if (Arrays.equals(lymphVesselCoordinateMap.get(vessel), coordinate)) {
                nextVesselName = vessel;
            }
        }

        if (nextVesselName != null) {
            Location nextLocation = vesselLocationHashmap.get(nextVesselName);
            nextVesselLocations.add(nextLocation);
        }
    }

    return nextVesselLocations;
}
```

Listing 6.10: Implementation of findNextVessels() method.

### 6.3.2 Cell Agent

As we have already discussed in 5.1.1, the Cell Agents acts as the functional structure of the Grid System providing the sense of direction in the grid and thus enabling the mobility in it. In the beginning of the simulation, every unit grid represented by an auxiliary container, contains one Cell Agent inside it. At this time, although they are present, the Cell Agents are in inactive state (meaning, none of their behaviours are in action yet) and do not have any information about their neighbouring Cells' location, and are waiting to be activated by the Initiator Agent.

#### 6.3.2.1 Cell's Initiation Phase

After the creation, the initiation phase begins. In this phase the Initiator Agent gets the `AgentController` of these Agents with the help of Container Controller Grid Map and invokes the native `start()` `method` on individual `AgentController` in order to start them, which allows any further Communications to be made in between them. The primary goals at this phase includes first, to send own location to the Initiator Agent, second, storing the neighbour's locations, and third, to check if a Vessel Agent is present locally (which is possible only after the initiation of the Lymph Network). These goals can be achieved by the Cell Agent by realizing three `CyclicBehaviour`s consecutively.

*Firstly,* the `SendingOwnLocationToInitiator` behaviour will perform a communication between the Cell Agent and the Initiator Agent through `tell_initiator_about_location` channel, essentially informing it about the Cell's Location retrieved by applying native `here()` method.

*Secondly,* the `SettingNeighboursLocationWithInitiator` behaviour will perform a communication between the same through `neighbour_allocation_channel` essentially receiving and storing the `ArrayList` of neighbouring locations.

*Thirdly,* the `CheckingIFaVessel` behaviour will perform a communication between the Cell Agent and the local Lymph Vessel through `vessel_cell_connection_channel` after it has been activated by the Initiator Agent, to set the status about presence of a Lymph Vessel which information is important for the Dendritic and CD4+ T-Cell agents to figure out an enter or an exit point into the Lymph Network.

#### 6.3.2.2 In a Healthy Phase

In the beginning of the simulation all of the Cell Agents are healthy and the Cell's DNA is unmodified. In this phase all of the normal behaviours of the Cell Agent are active along with the behaviour to spawn a Virus agent called `SpawningVirus`. when this behaviour is active, a virus Agent (As a part of it's Cloning behaviour) from a neighbouring Cell can replicate into this cell by sending its genetic information through `Spawn_A_New_Virus_Channel`.The Cell Agent can then communicate with the Virus Agent through `Update_DNA_Message_From_Virus` communication channel with `ListenToVirus` behaviour. The Virus can then alter the behaviour Cell Agent by asking it to change its DNA sequence according to the *virus signature* sent through this communication channel. Upon receiving this instruction, the Cell Agent invokes a `updateDNA()` method that performs an update on the DNA. This Altered behaviour flags an infection.

```
private void updateDNA(int[] flipLocations) {

    for (int flipLocation : flipLocations) {
        if (myDNA[flipLocation] == 0) {
            this.myDNA[flipLocation] = 1;
        } else if (myDNA[flipLocation] == 1) {
            this.myDNA[flipLocation] = 0;
        }
    }

}
```

Listing 6.11: Implementation of updateDNA() method.

#### 6.3.2.3 In an Infected Phase

Once the Cell Agent is infected, it can no longer communicate with any new Virus Agent, since it removes the `ListenToVirus` and `SpawningVirus` behaviours from its list of active behaviours. At this state, it waits for the Macrophage or the appropriate CD8 T-Cell Agent to terminate the Virus before it is completely exhausted off its resources, and by that point it is killed by the Virus Agent. On the other hand, If one of these Agents Are present in the local Container and they have terminated the virus, the cell keeps listening to a `DNA_Repair_Channel` as a part of its `DNARepairBehaviour` in order to receive a signal to repair its DNA, in that case it repairs its DNA and puts `ListenToVirus` and `SpawningVirus` behaviours back to it's list of active behaviours which flags a healthy phase.

#### 6.3.2.4 Common Behaviours in all Phases

Other than all of the behaviours discussed in earlier there are come Behaviours that do not change depending upon the state of infection. Some of them provides mobility for Example once a mobile Agent want to traverse into the grid it will ask the local Cell Agent to send the neighbouring locations through `Tell_About_Neighbours` communication channel, for this reason, the cell agent keeps listening to that channel as a part of its `TellNeighboursLocation` behaviour. On the Other hand if a Dendritic Cell, after detecting a *virus signature*, wants to get to a Lymph Vessel, it will communicate the Cell Agent through `vessel_confirmation_channel` so the Cell Agent keeps a `TellDendriticCellAboutLymphVessel` behaviour in its active state of behaviours in order to perform this communication. Similarly, for a situation when a Macrophage, CD8

T-Cell and a Dendritic Cell Agent wants communicate with the it to detect an infection and asking for its DNA, A `SignatureVerificationBehaviour` is there in the active state to handle this query.

Rest of the behaviours are to tackle the information about an exhausted Macrophage Agent. As we have already discussed in 6.2.1, the Macrophage Agent, after reaching its kill limit, prevents itself from moving to a new container, in that situation the local Cell Agent needs to be updated, so that when a CD4+ T-Cell arrives in the container, it can be informed about this situation. For this reason the Cell Agent Has three more `CyclicBehaviours`. The First one is `SettingExhaustedMacrophagePresence` which updates the presence of an exhausted Macrophage when contacted by the Macrophage through `exhausted_macrophage_cell_connection_channel`. Second behaviour communicates the presence or the absence of such a Macrophage with the CD4+ T-Cell Manager through `asking_cell_exhausted_macrophage_connection_channel`. And the final one is to update the presence of such an Agent when it is stimulated by the CD4+ T-Cell Agent which communicates with through `telling_cell_macrophage_channel_cd4t` by sending `ACLMessages`.

To summarize, all of the behaviours of the of the Cell Agent Other than the ones required for the initiation, can be presented with the help of 6.1. Here we find that every behaviours other than DNARepairBehaviour is active when the cell is healthy. On the other hand, only the ListenToVirus, and the SpawningVirus behaviour is inactive when the cell is infected.

### 6.3.3 Lymph Vessel Agent

Lymph Vessel agents are the agents that form the Lymph Network. At the beginning of the simulation these are created inside the grid following the Lymph Coordinate Map but are not activated immediately since they do not contain the directional information of the Lymph Vessel Network. They need the help of the Initiator Agent [6.3.1.4] in order to be initiated and eventually activated. The behaviours of this agent are the following.

#### 6.3.3.1 Sending Own Location to Initiator

With this `Cyclic Behaviour`, this agent first retrieves it own location by invoking native `here()` `method` and then immediately sends it to the Initiator Agent when asked through `tell_initiator_about_location` this communication channel by sending an `ACLMessage`. It has an `ArrayList` of `Locations` which is initially empty. This behaviour keeps repeating until the `ArrayList` is no longer empty which implies to the fact that it has been contacted by an Initiator Agent. Once that is the case, this cyclic behaviour terminates.

| All Behaviours | Healthy | Infected |
|---|---|---|
| SignatureVerificationBehaviour | o | o |
| TellNeighboursLocation | o | o |
| TellDendriticCellAboutLymphVessel | o | o |
| ListenToVirus | o | × |
| SpawningVirus | o | × |
| SettingExhaustedMacrophagePresence | o | o |
| TellingCD4TCellAboutExhaustedMacrophage | o | o |
| ListeningAboutMacrophageStimulationFromCD4T | o | o |
| DNARepairBehaviour | × | o |

Table 6.1: Cell Behaviours After Initiation Depending Upon its Phase.

#### 6.3.3.2 Setting Next Vessel Locations

With this behaviour, which is a `Cyclic Behaviour`, Lymph Vessel Agent first listens to the `next_vessel_allocation_channel`, for assigning the next Lymph Location in the Lymph Network. Once a message with an `ArrayList` of neighbouring locations is received from the Initiator Agent it stores the it in the memory. If the received `ArrayList` is empty, that represent this is an end of the lymph network thus a assigning itself as a Lymph Node. Similar to 6.3.3.1, once the `ArrayList` of `Locations` is not empty, this behaviour terminates.

#### 6.3.3.3 Telling if Self is a Lymph Node

This `CyclicBehaviour` is used to inform a Dendritic Cell weather this agent is a Lymph Node or otherwise. The Agent keeps listening to the `lymph_node_verification_channel` for a message from the Dendritic Cell and when queried with `tell_if_lymph_node`, returns a reply to the concerned agent.

#### 6.3.3.4 Telling Next Vessel Location to Cell

Very similar to the previous behaviour, the purpose of this `CyclicBehaviour` is to inform the Dendritic Cell About the next Vessel Location(s). The Agent keeps listening to the `Vessel_Dendritic_Communication_Channel` for a message from the Dendritic Cell and when queried with `give_me_next_vessel`, returns a reply to the concerned agent with the next Vessel Location(s).

### 6.3.4 Virus

The virus Agent is created by the virus generator in a randomly selected Auxiliary Container in the at the beginning of the simulation. That Virus Agent has a specific *genetic signature*, replication factor, time it takes to replicate and the time it takes to kill the host, which assigned by the Generator. The virus behaviour can be grouped together to understand it's broader goals.

#### 6.3.4.1 Infecting a Cell

This is achieved by sequentially performing two `OneShotBehaviours`. Firstly, upon introduction to a new container, the virus needs to check if the Cell Agent is alive in that container or otherwise. If that is the case, the virus agent then contacts the Cell Agent inside the container through `Update_DNA_Message_From_Virus` channel and sends its *genetic signature*, essentially forcing the Cell to change its DNA sequence. Which triggers the cell to change its behaviour. After this, the Virus starts the preparation for cloning to the new neighbouring container.

#### 6.3.4.2 Cloning

The Cloning behaviour is also a sequence of two `OneShotBehaviours` which includes asking the Cell Agent about its neighbouring locations which is necessary for the cloning process. In order to retrieve this information, The virus sends a query (“*neighbour_list*”) to the Cell Agent through `Tell_About_Neighbours` channel and waits for a reply from the Cell. Once the Cell replies with its neighbouring location, the `ArrayList` is stored and this behaviour is terminated with triggering a new behaviour to start the cloning process.

The Virus Can not clone itself, it uses the resources from the cell to clone. In order to incorporate that the virus in this model contacts a target Cell Agent, asking it to spawn a virus agent identical to the current one. The virus also sends all of its parameters as a serializable object by instantiating an object of VirusInformation class. The example is presented below.

```
import java.io.Serializable;

public class VirusInformation implements Serializable {
    public int[] virus_signature;
    public int virus_replication_factor;
    public int virus_cell_communication_time;
    public int virus_replication_time;
    public int time_to_kill_the_cell;
}
```

Listing 6.12: Serializing the parameters of the virus.

A few of these neighbours are then randomly chosen for cloning following the replication factor of the virus. Once the cloning process is complete, and all of the resource of the cell is been exhausted, the Virus starts preparing for the execution of the host Cell Agent.

#### 6.3.4.3 Killing the Host Cell

This is a OneShotBehaviour performed by invoking the native kill() method on the AgentController of the of the native Cell Agent. By performing this action the virus also kills itself since without the cell in that container the virus can not survive. This phenomenon is introduced in the model by implementing the native doDelete() method. The sample code is the following.

```
VirusInformation virusInformation;
ContainerController currentContainerController = myAgent.getContainerController();

String targetCell = "cell.".concat(myAgent.getContainerController().getContainerName());
AgentController targetAgentController = currentContainerController.getAgent(targetCell);

targetAgentController.kill();
myAgent.doDelete();
```

Listing 6.13: Code snippet of a Virus killing its host Cell.

### 6.3.5 CD4TCell Manager Agent

The CD4 T-Cell Manager Agent is an metaphysical agent that substitutes the computationally heavy and memory intensive task of keeping a billion naive T-Cells to find a match for the *virus signature* captured by the Dendritic Cell Agent and brought back to the Lymph Node.

### 6.3.5.1 Communicating with a Dendritic Cell

At the beginning of the simulation it is created inside the Main Container and upon being contacted by a Dendritic Cell through `dendritic_cell_c4t_communication_channel`, it receives the `Location` of the Lymph Node and the path to get to the site of infection and stores them in the memory. This is a CyclicBehaviour and once it is contacted by the Dendritic Cell a new behaviour is triggered to generate the CD4+ T-Cell Agents.

### 6.3.5.2 Finding a CD4 T-Cell Match

With this `OneShotBehaviour` the CD4 T-Cell Manager Agent creates the CD4+ T-Cell Agents inside the Main Container with the communicated information (*virus signature*, Lymph Location and the path to get to the site of infection) from the Dendritic Cell and activates the agents by invoking the native `start()` `method` on the `AgentController`. The number of these generated T-Cells depends upon the concentration of CD4+ T-Cell agents defined in `Universe.Constants` [6.1.2.4]. Below is a snippet of the CD4+ T-Cell generation.

```
String agentName = "CD4TCell-".concat(String.valueOf(id));
AgentController cd4TCellController = mainContainerController.createNewAgent(
        agentName,
        "universe.agents.CD4TCellAgent",
        new Object[] {
                virus_signature,
                lymphLocation,
                path_to_site
        });

cd4TCellController.start();
```

Listing 6.14: Snippet code for Generating a CD4+ T-Cell.

# Chapter 7

# Experiments

The final prototype of this project can be found as a publicly available GitHub repository[1]. In the current state, the tool can be customised through a list of parameters. Each parameter influences a different aspect of the simulation, and can be used to model different scenarios. Some of the available parameters are the following:

- *GRID_SIZE*: the size of the grid of cells.
- *PATHOGENS_REPLICATION_FACTOR*: the factor by which the virus multiplies.
- *RE_INFECTION_IMMUNITY_STRENGTH_PERCENTAGE*: The strength of immunity against a specific pathogen on consecutive infections.
- *CELL_IDENTIFYING_PEPTIDE*: The initial peptide of a healthy cell.
- *PERCENTAGE_OF_MACROPHAGE*: The number of Macrophages as a percentage of total number of cells in the system.
- *NUMBER_OF_VIRUS_MACROPHAGE_CAN_KILL*: The number of individual viruses an individual Macrophage kills before it gets exhausted and seeks for a stimulation from the CD4+ T-Cells.
- *PERCENTAGE_OF_CD4TCell*: Dictates the number of CD4+ T-Cells as a percentage of total number of cells in the system.
- *PERCENTAGE_OF_DENDRITIC_CELLS*: The number of Dendriric Cells as a percentage of total number of cells in the system.
- *PERCENTAGE_OF_CD8T_CELLS*: Dictates the number of CD8 T-Cells as a percentage of total number of cells in the system.

[1] `https://github.com/sanchayan721/Multi_agent_Immune_System`

We also created a `Monitor class` to dynamically keep track of not only the number of infected and dead cells but also the virus count. Every 50 milliseconds, such agent collects the data and stores them into a *.csv* file. These log files are then used to plot the so obtained results. Note that, every simulation is stopped when there are no virus or cells left in the system. We also have to keep in mind that the system is a *stochastic* in nature, which means, any of the outcomes of the simulations can not exactly be reproduced. But we can definitely observe a trend if we perform consecutive simulations with identical parameters.

## Experimental Scenarios

Various experimental scenarios can be modelled using the list of parameters discussed above. First we will run some simulations to observe the behavior of just the Innate Immune System depending upon the size of the grid and the replication factor of the pathogen. Then we will simulate some cases where a person is healthy and has a strong immunity and a person whose immune system is compromised. Finally, we will also run some experiments to simulate a situation where there is a re-infection after a primary infection by a single pathogen.

## 7.1 Experiments with Grid-size and *R-factor*

Although this set of experiments has less biological importance attached, nevertheless, it is very important from the system's performance point of view. Here we are interested in inspecting both the infection and death rate of the cells in the course of the simulation. Every consecutive simulation is created with an increasing number of containers (*i.e.*, cells), one Virus Agent and one Macrophage Agent.

- **Grid Size** — Given an assumption of a square-shaped universe, the grid size represents the length of a side in unit grid.
- **Replication factor** — The replication factor of the virus (e.g., if the replication factor is $n$, then the virus can make at-most $n$ copies of itself to the adjacent containers in each cycle).

At the beginning of each simulation, the Virus Agent is dropped in a random container and starts replicating itself according to its replication factor parameter, while the Macrophage Agent, already present in the system since the beginning of the simulation, searches for the virus from one container to another.

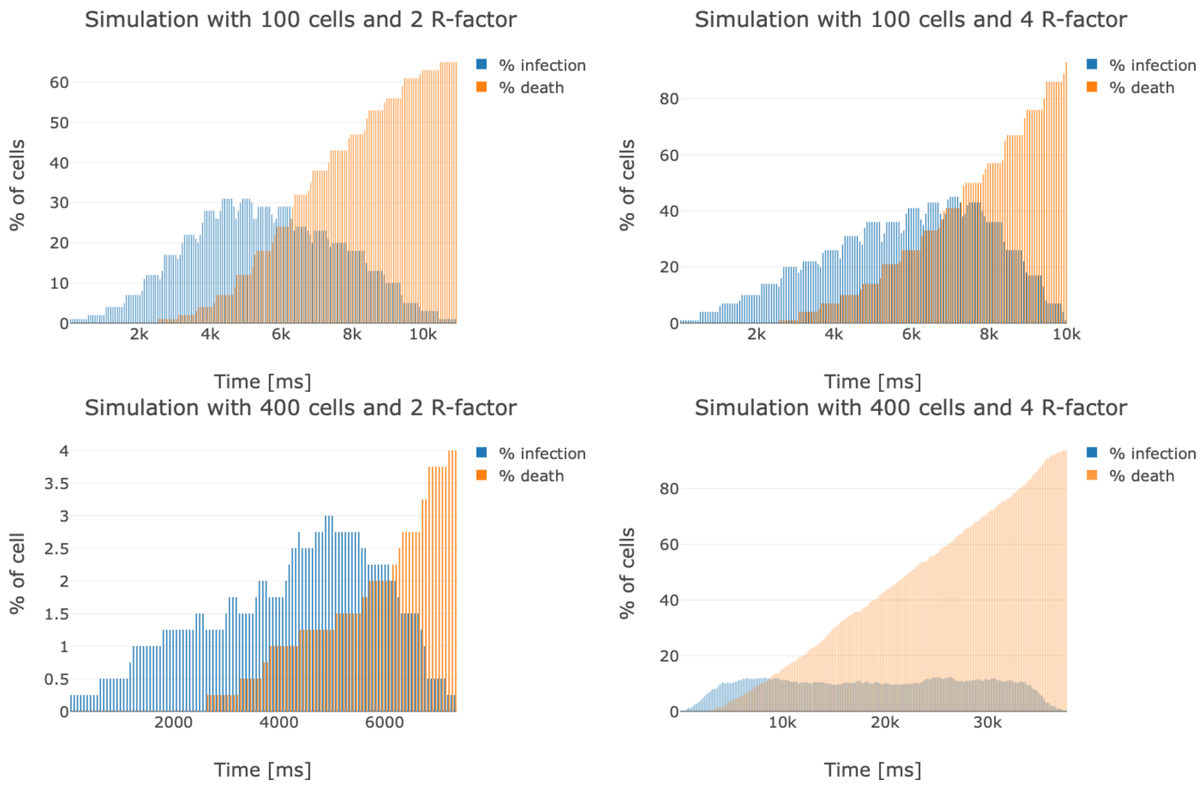


Figure 7.1: Results obtained through our experiments.

Figure 7.1 reports four different scenarios that we have experimented. The first (top left) is a scenario with 100 cells (*i.e.*, 10 × 10 grid) and replication factor set to 2, the second (top right) with 100 cells and replication factor 4, the third (bottom left) with 400 cells (*i.e.*, 20 × 20 grid) and replication factor 2, and finally, the fourth (bottom right) with 400 cells and replication factor 4. In each scenario, we can observe how there is a delay between the percentage of infected cells, and the percentage of dead cells. This derives by the intrinsic nature of infections, where a virus first infects a cell and then, after depriving the latter of all its resources, kills it. In most of the reported experiments (3 out of 4), the virus ends up killing most of the cells before being killed by the Macrophage, or dying from starvation. Indeed, in all but the third scenario (bottom left), the percentage of dead cells reaches 60% (top left), 80% (top right), and 90% (bottom right) of the total population of cells. In the third case, we have an example of simulation where the Macrophage is capable of detecting the virus soon enough to stop its replication; in fact, in such simulation, the percentage of dead cells reaches only 4% of the entire population. Another interesting aspect to note is the behaviour we obtain in the fourth scenario. There, by having a large matrix (200 × 200 containers) and a high replication factor, the virus succeeds in replicating and is the scenario in fact where we obtain the highest percentage of dead cells (since the fast replication is not successfully fought by the Macrophage in such a large area.

## 7.2 Strong Immunity

In this section we are going to simulate different scenarios, of a healthy person with a strong immune system.

### 7.2.1 Experiment Background

In a biological system a stronger immunity is obtained by both stronger Adaptive and Innate Immune responses against a pathogen. These experiments model a patch of the tissue where the pathogen is encountered so, here we define the size of the grid as our window to observe the state of infection and the concentration of various immune cells present in our observation window represents higher or lower immune strength.

#### 7.2.1.1 Innate Immune Response

In the design section [5.2] we have discussed the role of Macrophages and CD8 T-Cells in Innate Immune System. In this experiment we tweak some parameters to obtain various degrees strengths of the Innate Immune Response.

- **Concentration of Macrophages** — The concentration of macrophages present inside the observation window. This is represented as percentage of Macrophage Agents with respect to the number of cell agents present inside the window.
- **Number of Viruses a Macrophages kills before exhaustion** — The total number of viruses one Macrophage can kill before it goes into an "exhausted" state. At this state a Macrophage no longer kills the virus and becomes immobile waiting to be "activated by" the CD4+ T-Cells.
- **Concentration of CD8 T-Cells** — Similarly, the concentration of CD8 T-Cells is the percentage of CD8 T-Cell Agents present within the observation window. This percentage is also calculated with respect to the total number of Cell agents in that window.

#### 7.2.1.2 Adaptive Immune Response

In the design section [5.3] we have also discussed the role of Dendritic Cells and CD4+ T-Cells in the Adaptive immune system. Very similar to the Innate Immune Response, here we also have some parameters that translate to the overall strength of the Adaptive Immune Response.

- **Concentration of Dendritic Cells** — The percentage of the Dendritic Cells Agents present in the observation window calculated with respect to the total number of Cell Agents present in that window.
- **Concentration of CD4+ T-Cells** — Similarly, this parameter represents the percentage of CD4+ T-Cells present in the observation window and calculated w.r.t. the number of Cell Agents in that window.

#### 7.2.1.3 Pathogen

As we have discussed earlier in this section, the pathogen (e.g. virus) replicates itself in order to spread the infection to the adjacent cells inside our observation window. In order to be consistent with the results for these experiments we will not tweak the replication factor of the pathogen and set it to, "$R - factor = 3$". Which means, the virus can make at-most 3 copies of itself to infect the nearby cells in a simulation cycle.

### 7.2.2 Setup of the Experiment

All of the simulations corresponds to a window with 100 Cells (*i.e.*, $10 \times 10$ grid), and we are interested in inspecting the number of cells that are alive in each epoch of the simulation. We are also interested in observing the number count of viruses in those epochs. According to our primary assumption, the immune response is strong which translates to higher concentration of Macrophages, CD8 T-Cells, Dendritic cells and CD4+ T-Cells inside the window. Similar to the previous experiment, [7.1] the monitor collects the data in 50 milliseconds time interval and stores in a ".csv" file which is then plotted to get the figures ([7.2] and [7.3]). In both of these experiments each Macrophage individually can kill 5 viruses before it goes to an "exhausted" state.

### 7.2.3 Results of the Experiment

Both of these experiments' results, ([7.2] and [7.3]) depict a strong immune response, meaning, there are minimal cell deaths due to the infection.

#### 7.2.3.1 Stronger Innate Immune Response

In case of the first experiment [7.2], the simulation ends after 63 seconds and the total cell death is merely about 10%. Here, the virus is unable to spread a lot since the concentration of Macrophage and CD8 T-Cell is high about 30% each and all of the virus replicas are

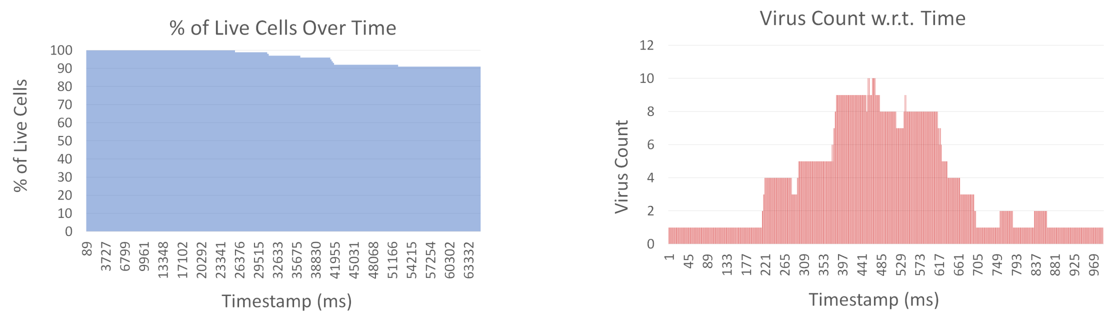


Figure 7.2: Simulation with 30% - Macrophage, 30% - CD8 T-Cell, 5% - Dendritic and 20% - CD4+ T-Cell Concentration.

killed relatively sooner before all of the Macrophages go into their "exhausted" state. This is the reason why we see a single peak here.

#### 7.2.3.2 Stronger Adaptive Immune Response

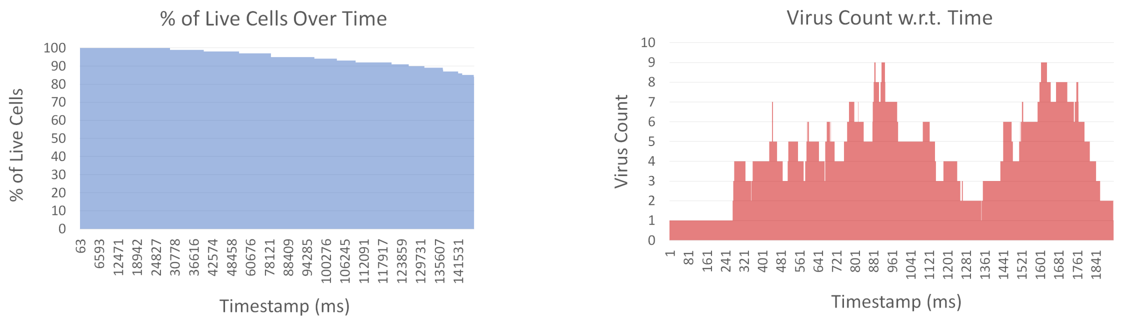


Figure 7.3: Simulation with 25% - Macrophage, 25% - CD8 T-Cell, 10% - Dendritic and 30% - CD4+ T-Cell Concentration.

On the other hand, in the second scenario [7.3], due to relatively lower concentration of the Macrophage and CD8 T-Cells the overall cell death is more than 15%, relatively higher than previous experiment [7.2]. If we analyze the virus count plot, we see two peaks. The first peak corresponds to the Innate Immune Response, after which the virus count goes down quite a lot but after a certain point the Macrophages are "exhausted" and stop killing it, giving a chance to the virus to grow in number again, which results in a second peak. After a while, the CD4+ T-Cells "activate" the Macrophages and then they together with the CD8 T-Cells, keep the infection under control which results in a sharp decline in the virus count.

## 7.3 Weak Immunity

In this section we are going to simulate different scenarios, modelling a person with a compromised immune system. We have to keep in mind the fact that a weak immunity does not always refer to the fact that the weaker immunity is caused by a previous infection from a virus. Although this could be a case, but it could also be caused by a genetic disorder or some foreign substance in the body for example, some chemicals or due to an immune suppressant medication (generally prescribed to a patient with an organ transplantation or with autoimmune diseases).

### 7.3.1 Background of the Experiment

In a biological system a weaker immunity can be caused by either or both week Adaptive and Innate Immune Responses. Like in the previous set of experiments [7.2], these further experiments model a patch of tissue where the pathogen is encountered, which is defined by the size of the grid and represents our observation window.

#### 7.3.1.1 Simulation Parameters

All of the parameters used in this set of experiments are similar to those of the Strong Immunity [7.2] but their values are different, much essentially lower than the values used in case of the Strong Immunity. In order to make the results comparable with the previous set of experiments we will keep the "Number of Viruses a Macrophages kills before exhaustion" parameter along with the "R-factor" of the virus similar. Which means, in these experiments the Macrophages individually kills 5 viruses before it is "exhausted" and the "$R - factor = 3$" for the virus that we introduce into the system.

### 7.3.2 Setup of the Experiment

Similar to the previous set of experiments [7.2], these set of simulations also corresponds to a window of 100 Cells (*i.e.*, $10 \times 10$ grid) and we are interested in inspecting the number of cells that are alive in each epoch of the simulation. Like the previous experiments, we are also interested in observing the number count of the viruses in these epochs. But unlike the previous experiments, here our assumption is a weaker immune system, thus the concentration of Macrophages, CD8 T-Cells, Dendritic Cells and CD4+ T-Cells will be comparatively lower inside the window. Similar to that of the previous set of experiments [7.2], the monitor collects the data in 50 milliseconds time interval and stores the data in a ".csv" file which is then plotted in order to obtain the figures ([7.4] and [7.5]).

### 7.3.3 Results of the Experiment

Both of these experiments' results ([7.4] and [7.5]) depict a weak Immune Response, meaning there is a large number of cell deaths due to infection.

#### 7.3.3.1 Weaker Innate Immune Response

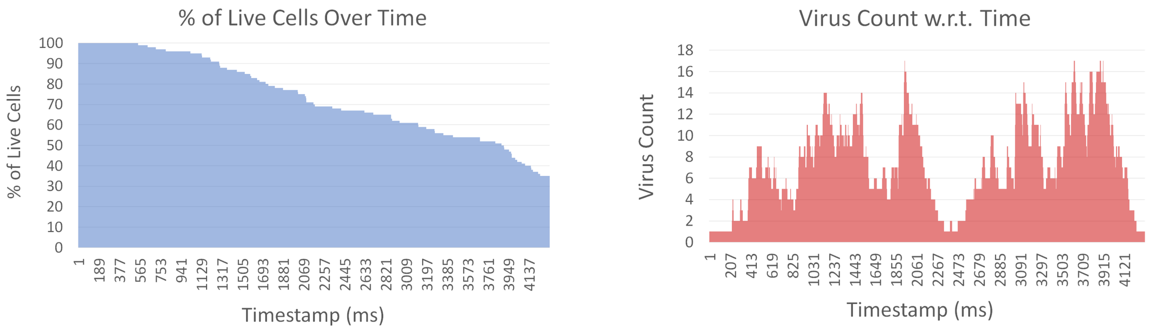


Figure 7.4: Simulation with 10% - Macrophage, 10% - CD8 T-Cell, 10% - Dendritic and 20% - CD4+ T-Cell Concentration.

In case of the first experiment [7.4], at the end of the simulation more than 60% of the cells are dead representing a situation where the organism is struggling to survive. Here, the percentage of the Macrophages and CD8 T-Cells are very low – about 10% each – which results in this massacre of cells, but at least the number of Dendritic cells and CD4+ T-Cells are relatively higher which results in an "activation" of the "exhausted" Macrophages that is effective: the infection is stopped before more cells die. This hypothesis is backed up by the second graph (*i.e.*, Virus Count w.r.t. Time). In this graph we can clearly see the two major global peaks and further local ones, which suggests that once most of the Macrophages are "exhausted", there is a sharp rise in the number of virus counts and once they are "activated", we see a decline in those numbers.

#### 7.3.3.2 Weaker Adaptive Immune Response

In case of the second simulation [7.5], although the number of Macrophages and CD8 T-Cells is higher (20% each), due to the lack of "activation" of "exhausted" Macrophages, more than 85% of the Cells die. This situation is caused by the very low concentration of the Dendritic Cells (5%) and the CD4+ T-Cells (10%). These numbers are too few for the "activation" of "exhausted" Macrophages to be effective, which is also backed up by the second graph (*i.e.*, Virus Count w.r.t. Time). Here, we do not see two proper global peaks but rather multiple small peaks. This situation represents a very dangerous one for the survival of organism due to huge cell loss.

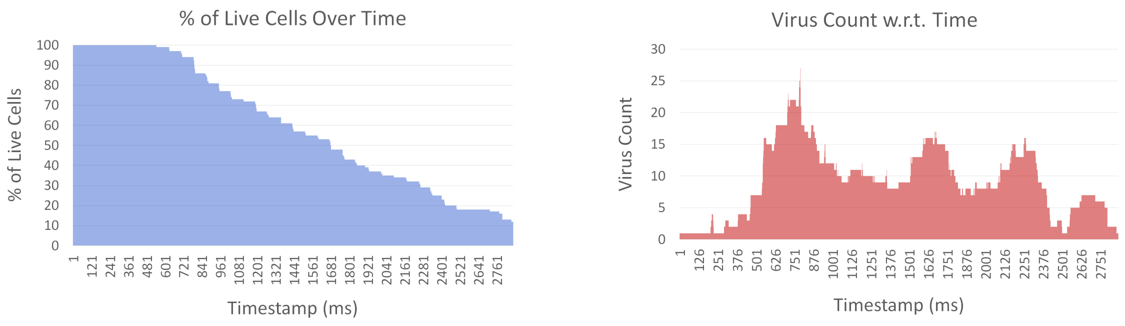


Figure 7.5: Simulation with 20% - Macrophage, 20% - CD8 T-Cell, 5% - Dendritic and 10% - CD4+ T-Cell Concentration.

## 7.4 Re-infection Scenario

In this scenario, the goal is to model a case where an Organism has once been infected by a pathogen and survived the infection, developing an immunity against the same pathogen. Now later in its life the pathogen is being encountered again. Based upon what we have discussed so far in this thesis, our hypothesis is that the immune response should be able to handle the infection well this time, at least better than how it had been handled for the very first time.

### 7.4.1 Background of the Experiment

#### 7.4.1.1 Pathogen

The pathogen in the experiment can be uniquely identified using its genetic signature. Also, by detecting this signature the CD8 T-Cell identifies the specific strain of the virus. This specific virus can have any replication factor but for the sake of this experiment we will set this parameter to be “$R - factor = 2$”.

- **Genetic Signature** - An array of zeros and ones that uniquely identifies viruses of same strain.

This experiment is composed of two co-related simulations and for both of these simulations the strain of the virus is identical.

#### 7.4.1.2 First Infection

In the first simulation (first infection), the immune system does not recognize the pathogen, which means, we do not have any CD8 T-Cells, because the immune system has to have encountered the pathogen in a previous infection in order to produce any CD8 T-Cell that can target that specific pathogen. There could be CD8 T-Cells in the system but they can not kill the pathogen at this time. But since the Macrophage is non-targeted, it can kill any virus strain in the simulation. The Dendritic Cell engulfs the virus (pathogen) and stores this signature by producing "MHC Class-II" molecules. These Dendritic cells then travel into the lymph nodes and activate the naive CD4 T-Cells against this virus. These activated CD4+ T-Cells can then travel to the site of infection and "activate" the "exhausted" Macrophages. At this time some of the CD4+ T-Cells become memory T-Cells and later can activate the CD8 T-Cells [SMAD+22].

#### 7.4.1.3 Next re-infection

This simulation inherits the activated CD8 T-Cells from the previous simulation [7.4.1.2]. So, there is a significant amount of targeted CD8 T-Cell to encounter the specific strain of the virus once it is introduced into the system. Alongside this, there are Macrophages present in the system that will encounter the virus regardless of its strain. The Dendritic and The CD4+ T-Cells are also present in the simulation for the "activation" of "exhausted" Macrophages in successive simulation epochs.

### 7.4.2 Setup of the Experiment

Similar to the previous set of experiments [7.3], this set of simulations also corresponds to a window of 100 Cells (*i.e.*, $10 \times 10$ grid) and we are interested in inspecting the number of cells that are alive in each epoch of the simulation. Like for the previous experiments, we are also interested in observing the number count of the viruses in these epochs.

### 7.4.3 Results of the Experiment

The first point to be noted here is that viruses in both simulations ([7.6] and [7.7]) are of similar strain since the second graphs (*i.e.*, Virus Count w.r.t. Time) in both of the simulation show a very similar pattern (a smaller first peak in the beginning and a large second peak at the end). Although they are not identical due to the fact that in the second simulation [7.7], there are both Macrophages and CD8 T-Cells that kill it, unlike the first simulation [7.6] where only the Macrophages can kill, which is the reason the peaks are less severe in the following simulation.

#### 7.4.3.1 First Infection Results

In the first simulation [7.6], the immune system has never encountered the specific strain of the virus so the "*Re-immunity Strength* = 0". Which means, there are no CD8 T-Cells present that can encounter the specific strain of the virus. But since the organism was healthy (1 with 20% - Macrophage, 20% - CD8 T-Cell, 20% - Dendritic and 30% - CD4+ T-Cell Concentration), the immune system was able to handle the infection within 2 seconds of simulation time with a cell loss of about 40% referring to the fact that the organism has survived.

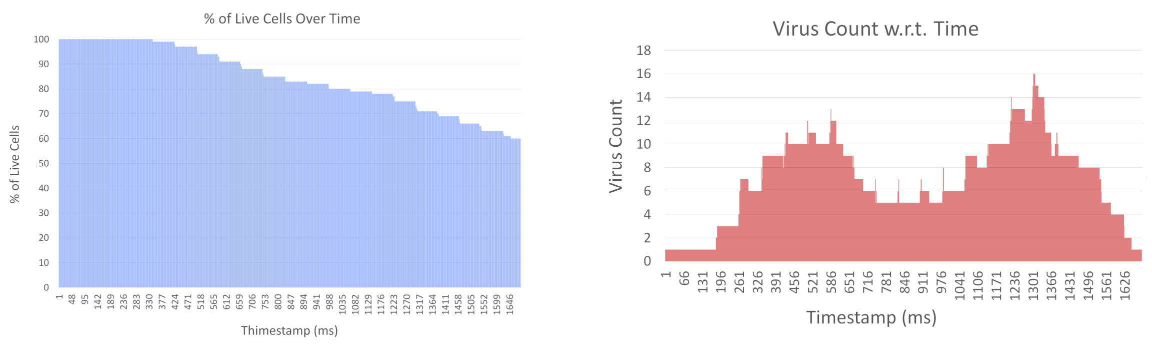


Figure 7.6: Simulation - 1 with 20% - Macrophage, 20% - CD8 T-Cell, 20% - Dendritic and 30% - CD4+ T-Cell Concentration.

#### 7.4.3.2 Next Re-infection Results

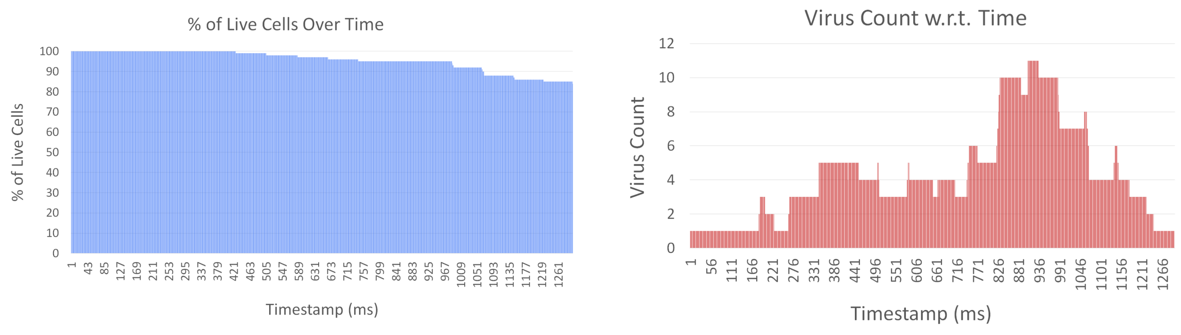


Figure 7.7: Simulation - 2 with 20% - Macrophage, 20% - CD8 T-Cell, 5% - Dendritic and 10% - CD4+ T-Cell Concentration.

In the second consecutive simulation [7.7], the immunity is being carried from the first simulation [7.4.1.2] and we get a strong immune response this time which is being represented by "*Re-immunity Strength* = 80". Which means, there is a chance that 80% of all of the CD8 T-Cells present can encounter the specific strain of the virus. Now if we assume that the system parameters are unchanged, (*i.e.*, 1 with 20% - Macrophage, 20% - CD8 T-Cell, 20% - Dendritic and 30% - CD4+ T-Cell Concentration), the immune system is now able to handle the infection within 1.3 seconds of simulation time with a cell loss of about 15%. Which means, the infection is now kept under control faster, in 35% less time and with 25% less cell damage.

# Chapter 8

# Conclusion and Future Work

In this thesis, we presented the building blocks of the an agent-based simulator built in JADE to reproduce the human body's immune system in a scalable and distributed manner. We are well aware that the actual immune system is extremely complex and that the model is only tackling a very specific aspect of it. Nonetheless, the current work allowed us to focus on the initial engineering steps required to map the immune system in JADE. In fact, we showed how some of the main organic actors (cells, Macrophages, T-Cells, and so on), can be mapped into their corresponding agent representation in JADE.

Using our first two sets of simulations we were also able to reproduce some of the biological scenarios like the immune response of a healthy person and an immune compromised person. Using the same model we have also simulated a situation where there is a second infection (or re-infection) by an identical virus strain and were able to produce results about the dynamics of the re-infection which explicitly explain the underlying biological phenomena. All of the experimental results point towards the direction that our model is in sync with the actual biological mode, although more test cases are required to verify our claim.

One problem that we have faced while comparing the results of experiments is the fact that collecting the biological data is a huge challenge. To give an example, let say, we want to collect the data about how does a viral infection spreads in a biological system without the interference of any immune response from the immune system of the organism. This may sound very simple in theory but the experiment itself is very challenging due to the fact that biological systems are not isolated systems, meaning, there are numerous external factors that are in play like the body-temperature of the organism (which is directly related to the reproductive capability of the pathogen), the $pH$[1] factor of the environment (also

[1]$pH$ - A figure expressing the acidity or alkalinity of a solution on a logarithmic scale on which 7 is neutral, lower values are more acid and higher values more alkaline.

impacts the the reproductive capability of the pathogen) etc. If we (or, better, some expert in biological systems and in immune systems in particular) wanted to collect the data, the difficulties increases exponentially: either we have to inject the virus into a live target (*e.g.*, like a lab animal) or we have to produce a culture in a perti-dish, in that case it is even more difficult if not impossible to reproduce the immune system with all of its components.

In order to understand these complex biological systems it is necessary to perform simulation like this which has the capability to be produced in an isolated environment and in a case by case basis. That is one of the reasons why implications of simulations of biological phenomena are so important: the possibilities in a simulated environment are only limited by the computational power of the base system that is being used to simulate. And this brings us to one of the key features of this models, the scalability. Our system architecture is distributed and can be deployed over a network of computers. Although for the thesis we did not run the model on a cluster, it is definitely possible in practice to deploy this model on a large cluster with multiple nodes.

For the very same reasons mentioned above, drug testing is one of the fields of research where models like the one we developed can help to understand the dynamics. In that case we would need to add some more agents with the behaviours of the drug (how does the drug interact with the different actors of the immune system and various cells) to observe any long term effect on the body.

Other than this, our model can be updated likewise to study the propagation and proliferation of cancer producing malignant tumours from the cellular level. In that case, the cell agents needs to be programmed in such way that they replicate regardless of the biological constraints imposed upon them by their genetics.

One other biological phenomenon that we have just touched the surface in one of our simulations is the situation where there is the possibility of introducing an immune suppressant medication in order to prevent the rejection of an organ transplantation by the immune system, or to control autoimmune diseases. In that case, the immune response is intentionally suppressed by medications. Using our model it is also possible to simulate the long term implication of such medication. Especially, in a situation where the patient has been infected by a pathogen after the implantation or after the autoimmune disease showed up.

As future directions, we are planning to build on top of the model, by adding more agents and by enriching the currently available ones. Moreover, we plan to build an open source platform where other developers around the globe can contribute by developing parts of the immune system. Once online, that global platform will be made available with an *open access* to the researchers who will be able to exploit it for their research purposes, and to drug manufacturers not only to test and discover new drugs, but also to discover new ways to deliver the drugs.

# Bibliography

[AZR+17] Norkhushaini Awang, Nurul Hidayah Ahmad Zukri, Nor Aimuni Md Rashid, Zuhri Arafah Zulkifli, and Nor Afifah Mohd Nazri. Multi-agent integrated password management (MIPM) application secured with encryption. In *AIP Conference Proceedings*. Author(s), 2017. doi:10.1063/1.5005364.

[BC01] Massimo Bernaschi and Filippo Castiglione. Design and implementation of an immune system simulator. *Comput. Biol. Medicine*, 31(5):303–331, 2001. doi:10.1016/S0010-4825(01)00011-7.

[BCG07] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[BF10] Rabie Belkacemi and Ali Feliachi. Multi-agent design for power distribution system reconfiguration based on the artificial immune system algorithm. In *International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France*, pages 3461–3464. IEEE, 2010. doi: 10.1109/ISCAS.2010.5537841.

[Cas22] Ana Casali. Extension of the bdi agent model: Representing and reasoning using graded attitudes. 11 2022.

[CBD20] Dipanjan Chakraborty, Sanchayan Bhunia, and Rumi De. Survival chances of a prey swarm: how the cooperative interaction range affects the outcome. *Scientific Reports*, 10(1), May 2020. doi:10.1038/s41598-020-64084-3.

[CCP+14] Filippo Castiglione, Ferdinando Chiacchio, Marzio Pennisi, Giulia Russo, Santo Motta, and Francesco Pappalardo. Agent-based modeling of the immune system: Netlogo, a promising framework. *BioMed Research International*, 2014:907171, 2014. doi:10.1155/2014/907171.

[DGTM08] Mamady Dioubate, Tan Guanzheng, and Lamine Toure Mohamed. An artificial immune system based multi-agent model and its application to robot cooperation problem. In *2008 7th World Congress on Intelligent Control and Automation*, pages 3033–3039, 2008. doi:10.1109/WCICA.2008.4593405.

[DPG11] Saber Darmoul, Henri Pierreval, and Sonia HAJRI Gabouj. An immune inspired multi agent system to handle disruptions in manufacturing production systems. In *International Conference on Industrial Engineering and Systems Management, IESM*, volume 2011, 2011.

[DRN12a] Babak Darvish Rouhani and Fatemeh Nikpay. Agent-oriented enterprise architecture: new approach for enterprise architecture. *IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 6, No 1, November 2012*, 9:331–334, 11 2012.

[DRN12b] Babak Darvish Rouhani and Fatemeh Nikpay. Agent-oriented enterprise architecture: new approach for enterprise architecture. *IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 6, No 1, November 2012*, 9:331–334, 11 2012.

[Fal91] Adin D. Falkoff. The IBM family of APL systems. *IBM Syst. J.*, 30(4):416–432, 1991. doi:10.1147/sj.304.0416.

[FD22] Baylor G. Fain and Hana M. Dobrovolny. GPU acceleration and data fitting: Agent-based models of viral infections can now be parameterized in hours. *Journal of Computational Science*, 61:101662, May 2022. doi:10.1016/j.jocs.2022.101662.

[Fel72] M Feldmann. Cell interactions in the immune response in vitro. v. specific collaboration via complexes of antigen and thymus-derived cell immunoglobulin. *J. Exp. Med.*, 136(4):737–760, October 1972.

[GKQ16] Charles Gawad, Winston Koh, and Stephen R. Quake. Single-cell genome sequencing: current state of the science. *Nature Reviews Genetics*, 17(3):175–188, Mar 2016. doi:10.1038/nrg.2015.16.

[GOL] Overview of google crawlers (user agents). https://developers.google.com/search/docs/crawling-indexing/overview-google-crawlers. Accessed: 2022-11-02.

[GPP$^+$99] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In *Intelligent Agents V: Agents Theories, Architectures, and Languages*, pages 1–10. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-49057-4_1.

[HEE$^{+}$19] Fatima Zahra Harmouch, Ahmed F. Ebrahim, Mohammad Mahmoudian Esfahani, Nissrine Krami, Nabil Hmina, and Osama A. Mohammed. An optimal energy management system for real-time operation of multiagent-based microgrids using a t-cell algorithm. *Energies*, 12(15), 2019. URL: `https://www.mdpi.com/1996-1073/12/15/3004`, `doi:10.3390/en12153004`.

[HGY13] Xue-Liang Hua, Iqbal Gondal, and Farrukh Yaqub. Mobile agent based artificial immune system for machine condition monitoring. In *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*, pages 108–113. IEEE, 2013.

[HKR08] Vincent Hilaire, Abder Koukam, and Sebastian Rodriguez. An adaptative agent architecture for holonic multi-agent systems. *ACM Trans. Auton. Adapt. Syst.*, 3(1):2:1–2:24, 2008. `doi:10.1145/1342171.1342173`.

[Igu19] Kingsley Theophilus and Igulu. Intelligent road emergency response system using GAIA and JADE. *International Journal of Engineering Research and*, V8(04), April 2019. `doi:10.17577/ijertv8is040075`.

[JLL04] Christian Jacob, Julius Litorco, and Leo Lee. Immunity through swarms: Agent-based simulations of the human immune system. In Giuseppe Nicosia, Vincenzo Cutello, Peter J. Bentley, and Jon Timmis, editors, *Artificial Immune Systems, Third International Conference, ICARIS 2004, Catania, Sicily, Italy, September 13-16, 2004*, volume 3239 of *Lecture Notes in Computer Science*, pages 400–412. Springer, 2004. `doi:10.1007/978-3-540-30220-9\_32`.

[KFS20] Li Kou, Wenhui Fan, and Shuang Song. Multi-agent-based modelling and simulation of high-speed train. *Computers & Electrical Engineering*, 86:106744, September 2020. `doi:10.1016/j.compeleceng.2020.106744`.

[Kib13] Rodger Kibble. 9. speech act theory and intelligent software agents. In *Pragmatics of Speech Actions*, pages 313–338. DE GRUYTER, July 2013. `doi:10.1515/9783110214383.313`.

[KMWS22] Jagoda Kaszowska-Mojsa, Przemysław Włodarczyk, and Agata Szymańska. Immunity in the ABM-DSGE framework for preventing and controlling epidemics—validation of results. *Entropy*, 24(1):126, January 2022. `doi:10.3390/e24010126`.

[LBGVDR17] Donfeng Liu, Luis Barba-Guamán, Priscila Valdiviezo-Díaz, and Guido Riofrio. Intelligent tutoring module for a 3dgame-based science e-learning platform. *Inteligencia Artificial*, 20(60):1–19, February 2017. `doi:10.4114/intartif.vol20iss60pp1-19`.

[LFM$^+$19] Gianfranco Lombardo, Paolo Fornacciari, Monica Mordonini, Michele Tomaiuolo, and Agostino Poggi. A multi-agent architecture for data analysis. *Future Internet*, 11(2):49, February 2019. `doi:10.3390/fi11020049`.

[MP93] Jörg Müller and Markus Pischel. The agent architecture interrap: Concept and application. 07 1993.

[MWWK18] Jean S. Marshall, Richard Warrington, Wade Watson, and Harold L. Kim. An introduction to immunology and immunopathology. *Allergy, Asthma & Clinical Immunology*, 14(S2), September 2018. `doi:10.1186/s13223-018-0278-1`.

[Nwa96] Hyacinth S. Nwana. Software agents: an overview. *The Knowledge Engineering Review*, 11(3):205–244, September 1996. `doi:10.1017/s026988890000789x`.

[OOM19] James Imende Obuhuma, Henry Okora Okoyo, and Sylvester Okoth McOyowo. A software agent for vehicle driver modeling. In *2019 IEEE AFRICON*, pages 1–8, 2019. `doi:10.1109/AFRICON46755.2019.9134033`.

[OOW13] Chung-Ming Ou, CR Ou, and Yao-Tien Wang. Agent-based artificial immune systems (abais) for intrusion detections: inspiration from danger theory. In *Agent and Multi-Agent Systems in Distributed Systems-Digital Economy and E-Commerce*, pages 67–94. Springer, 2013.

[PHR$^+$09] Francesco Pappalardo, Mark D. Halling-Brown, Nicolas Rapin, Ping Zhang, Davide Alemani, Andrew Emerson, Paola Paci, Patrice Duroux, Marzio Pennisi, Arianna Palladini, Olivo Miotto, Daniel Churchill, Elda Rossi, Adrian J. Shepherd, David S. Moss, Filippo Castiglione, Massimo Bernaschi, Marie-Paule Lefranc, Søren Brunak, Santo Motta, Pierluigi Lollini, Kaye E. Basford, and Vladimir Brusic. Immunogrid, an integrative environment for large-scale simulation of the immune system for vaccine discovery, design and optimization. *Briefings Bioinform.*, 10(3):330–340, 2009. `doi:10.1093/bib/bbp014`.

[PKSC02] Roberto Puzone, B. Kohler, Philip Seiden, and Franco Celada. Immsim, a flexible model for in machina experiments on immune system responses. *Future Gener. Comput. Syst.*, 18(7):961–972, 2002. `doi:10.1016/S0167-739X(02)00075-4`.

[PLCP22] Mattia Pellegrino, Gianfranco Lombardo, Stefano Cagnoni, and Agostino Poggi. High-performance computing and ABMS for high-resolution COVID-19 spreading simulation. *Future Internet*, 14(3):83, March 2022. `doi:10.3390/fi14030083`.

[PLM+21] Mattia Pellegrino, Gianfranco Lombardo, Monica Mordonini, Michele Tomaiuolo, Stefano Cagnoni, and Agostino Poggi. Actodemic: A distributed framework for fine-grained spreading modeling and simulation in large scale scenarios. In *WOA*, pages 194–209, 2021.

[RAGBSM20] Alejandro Rodríguez-Arias, Bertha Guijarro-Berdiñas, and Noelia Sánchez-Maroño. A fipa-acl based communication utility for unity. In *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 9–13, 2020. `doi:10.1109/WETICE49692.2020.00010`.

[RG05] M. J. Robbins and Simon M. Garrett. Evaluating theories of immunological memory using large-scale simulations. In Christian Jacob, Marcin L. Pilat, Peter J. Bentley, and Jonathan Timmis, editors, *Artificial Immune Systems: 4th International Conference, ICARIS 2005, Banff, Alberta, Canada, August 14-17, 2005, Proceedings*, volume 3627 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2005. `doi:10.1007/11536444\_15`.

[SBV22] Viviana Mascardi Sanchayan Bhunia, Angelo Ferrando and Chiara Vitale. Mais: Exploiting jade as a multi-agent simulator of the immune system. 2022. URL: `https://emas.in.tu-clausthal.de/2022/papers/paper13`.

[Sho91] Yoav Shoham. Agent0: A simple agent language and its interpreter. In *AAAI*, 1991.

[SHW+07] Weiming Shen, Qi Hao, Shuying Wang, Yinsheng Li, and Hamada Ghenniwa. An agent-based service-oriented integration architecture for collaborative intelligent manufacturing. *Robotics and Computer-Integrated Manufacturing*, 23(3):315–325, 2007. International Manufacturing Leaders Forum 2005: Global Competitive Manufacturing. URL: `https://www.sciencedirect.com/science/article/pii/S0736584506000226`, `doi:https://doi.org/10.1016/j.rcim.2006.02.009`.

[Sig16] Luis J. Sigal. Activation of CD8 t lymphocytes during viral infections. In *Encyclopedia of Immunobiology*, pages 286–290. Elsevier, 2016. `doi:10.1016/b978-0-12-374279-7.14009-3`.

[Sim18] Julia Simundza. Infection and immunity: insights and therapeutic strategies through genomic analysis of the host, pathogen, and host–pathogen interaction. *Genome Medicine*, 10(1), September 2018. `doi:10.1186/s13073-018-0583-9`.

[SK18] Snehal B Shinde and Manish P Kurhekar. Review of the systems biology of the immune system using agent-based models. *IET Systems Biology*, 12(3):83–92, 2018.

[SKPF05] Lee Spector, Jon Klein, Chris Perry, and Mark Feinstein. Emergence of collective behavior in evolving populations of flying agents. *Genet. Program. Evolvable Mach.*, 6(1):111–125, 2005. `doi:10.1007/s10710-005-7620-3`.

[SMAD+22] Kazuki Sasaki, Mouhamad Al Moussawy, Khodor I. Abou-Daya, Camila Macedo, Amira Hosni-Ahmed, Silvia Liu, Mariam Juya, Alan F. Zahorchak, Diana M. Metes, Angus W. Thomson, Fadi G. Lakkis, and Hossam A. Abdelsamed. Activated-memory t cells influence naïve t cell fate: a noncytotoxic function of human CD8 t cells. *Communications Biology*, 5(1), jun 2022. URL: `https://doi.org/10.1038%2Fs42003-022-03596-2`, `doi:10.1038/s42003-022-03596-2`.

[SS02] S. Sathyanath and F. Sahin. Application of artificial immune system based intelligent multi agent model to a mine detection problem. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 6 pp. vol.3–, 2002. `doi:10.1109/ICSMC.2002.1176015`.

[SS20] Galina A. Samigulina and Zarina I. Samigulina. Design of technology for prediction and control system based on artificial immune systems and the multi-agent platform JADE. In Gordan Jezic, Yun-Heh Jessica Chen-Burger, Mario Kusek, Roman Sperka, Robert J. Howlett, and Lakhmi C. Jain, editors, *Agents and Multi-Agent Systems: Technologies and Applications 2020, 14th KES International Conference, KES-AMSTA 2020, June 2020 Proceedings*, pages 143–153. Springer, 2020. `doi:10.1007/978-981-15-5764-4\_13`.

[TJ05] Joc Cing Tay and Atul Jhavar. CAFISS: a complex adaptive framework for immune system simulation. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, pages 158–164. ACM, 2005. `doi:10.1145/1066677.1066716`.

[Tom15] Michele Tomaiuolo. Service composition in open agent societies, Aug 2015. URL: `https://www.academia.edu/14833173/Service_Composition_in_Open_Agent_Societies`.

[Tut] TutorialsPoint. AI – Agents & Environments – tutorialspoint.com. `https://www.tutorialspoint.com/artificial_intelligence/`

artificial_intelligence_agents_and_environments.htm. [Accessed 21-Oct-2022].

[TV15] Cristian Tomasetti and Bert Vogelstein. Variation in cancer risk among tissues can be explained by the number of stem cell divisions. *Science*, 347(6217):78–81, January 2015. doi:10.1126/science.1260825.

[Tve01] Amund Tveit. A survey of agent-oriented software engineering. 08 2001.

[wik20] Agent-based model, Oct 2020. URL: https://en.wikipedia.org/wiki/Agent-based_model.

[wik22] Agent-oriented programming, Sep 2022. URL: https://en.wikipedia.org/wiki/Agent-oriented_programming.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, June 1995. doi:10.1017/s0269888900008122.

[XF18] Wei Xiong and Dongmei Fu. A new immune multi-agent system for the flexible job shop scheduling problem. *J. Intell. Manuf.*, 29(4):857–873, 2018. doi:10.1007/s10845-015-1137-2.